IEEE-ISTO
Industry Standards and Technology Organization
affiliated with the IEEE and the IEEE Standards Association

The Nexus 5001 Forum™
Standard for a Global Embedded Processor Debug Interface

15 December 1999

IEEE Networking the World™
Industry Standards and Technology Organization (IEEE-ISTO)
445 Hoes Lane • P.O. Box 1331 • Piscataway, NJ 08855-1331, USA
Phone +1.732.981.3434 • Fax +1.732.562.1571 • http://www.ieee-isto.org/

The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface

Recognized by the IEEE Vehicular Technology Society

History: The Global Embedded Processor Debug Interface Standard (GEPDIS) Consortium was formed in April 1998 to define and develop a much-needed embedded processor debug interface standard for embedded control applications. On 23 September 1999, the GEPDIS Consortium chose the IEEE Industry Standards and Technology Organization (IEEE-ISTO) as the operational and legal forum in which to continue its efforts. During the transition, the group also changed its name to the Nexus 5001 Forum™ to reflect the submission of Version 1.0 of their standard to the IEEE-ISTO for publication, distribution and future management as IEEE-ISTO 5001™ - 1999, The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface.

Abstract: A general-purpose specification that addresses the rigorous challenges for debug interfaces is outlined. Auxiliary pin functions, transfer protocols and standard development features are defined.

Keywords: Application Programming Interface (API), auxiliary port, Boolean, breakpoint, bit, client, compliance classification, debug interface, embedded processor, emulator, full-duplex, half-duplex, Hardware Abstraction Layer (HAL), high-speed input/output (HSIO), low-speed input/output (LSIO), Nexus, pin, register, Target Abstraction Layer (TAL), watchpoint.

About the IEEE-ISTO

The IEEE-ISTO is a not-for-profit corporation offering industry groups an innovative and flexible operational forum and support services. The IEEE-ISTO provides a forum not only to develop standards, but also to facilitate activities that support the implementation and acceptance of standards in the marketplace. The organization is affiliated with the IEEE (http://www.ieee.org/) and the IEEE Standards Association (http://standards.ieee.org/).

For additional information regarding the IEEE-ISTO and its industry programs visit http://www.ieee-isto.org.

About the Nexus 5001 Forum™

The Nexus 5001 Forum™ (formerly known as the Global Embedded Processor Debug Interface Consortium) is chartered to advance the development, dissemination and implementation of the Global Embedded Processor Debug Interface Standard. The Nexus 5001 Forum™ is open to all interested parties.

For additional information (membership, procedures, articles, news releases, etc.) regarding the Nexus 5001 Forum™, visit http://www.ieee-isto.org/Nexus5001/.

Feedback

Comments and questions may be submitted to the Nexus 5001 Forum™ through the IEEE-ISTO:

Program Administrator, IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08855-1331 USA
Telephone: +1.732.981.3434
Fax: +1.732.562.1571
Email: ieee-isto@ieee.org

IEEE Vehicular Technology Society Recognition

VEHICULAR TECHNOLOGY SOCIETY

October 25,1999

Mr. Andrew Salem, President and CEO
IEEE Industry Standards and Technology Organization
445 Hoes Lane
Piscataway, NJ 08855-1331

Dear Mr. Salem:

In response to your letter of September 30, 1999 regarding the publication of the IEEE-ISTO 5001™-1999, I brought this matter before the IEEE Vehicular Technology Society (VTS) at their Board of Governors (BOG) meeting on October 8, 1999. The VTS BOG was very supportive of the request from the IEEE-ISTO. More specifically, the IEEE VTS has taken note of the publication of IEEE-ISTO 5001™-1999, and recognizes it as a significant contribution to the needs of the Automotive industry. Should the Nexus 5001 Forum, a program of the IEEE-ISTO desire to advance this standard as an IEEE Standard, the VTS wishes to offer its services as the sponsor for this effort. The IEEE-ISTO may so note our recognition of the standard and our offer of sponsorship on the standard itself.

Sincerely,

Dennis Bodson

Dennis Bodson
Chairman
VTS Standards Committee

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Preface

The Nexus 5001 Forum™ (formerly known as the Global Embedded Processor Debug Interface Consortium) is chartered to advance the development, dissemination and implementation of IEEE-ISTO 5001™-1999, the Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface.

The industry group was formed in April 1998 to define and develop a much-needed embedded processor debug interface standard for embedded control applications. As advances in semiconductor and system design continue, embedded applications are using higher-performance embedded processors. Efficient use of these embedded processors requires software and hardware development tools that can easily access critical processor functionality.

However, the lack of a unifying standard among the various embedded processors on the market at the time of publication has impeded this accessibility, preventing tool vendors from creating standard tools with consistent functionality across a broad range of processors. This, in turn, has become a gating factor for chipmakers, tool providers and developers. Ultimately customers are forced to pursue costly custom solutions to meet their tool needs.

IEEE-ISTO 5001-1999, the Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface is an open industry standard that provides a general-purpose interface for the software development and debug of embedded processors. Standardization on this interface benefits customers' reuse of their Nexus 5001™ compliant development tools on compliant processor architectures. A future method and a process by which Nexus 5001 compliance can be validated will further global implementation of the standard.

Although the initial focus of the effort was based on the stringent requirements of the automotive powertrain applications, a general purpose standard has been developed, aimed to also benefit data communications and computer peripherals, wireless systems and other embedded control applications industries.

The Nexus 5001 Forum is open to all interested companies. Members of the Forum represent all aspects of the technologies required for embedded control applications: embedded processor suppliers, independent tools providers, semiconductor and hardware development tools, and software tools (emulators, compilers, simulators, debuggers, RTOS's, etc.).

Additional information regarding the Nexus 5001 Forum can be found on the IEEE-ISTO's website at http://www.ieee-isto.org/.

Contents

SECTION 1
Introduction

The Nexus 5001 Forum™ (http://www.ieee-isto.org/Nexus5001/), previously referred to as the Global Embedded Processor Debug Interface Standard Consortium (GEPDISC), was formed to develop a much-needed embedded processor debug interface standard for embedded control applications. The internal name for this standard is “Nexus,” which is used throughout this document only.

The goal is a general-purpose specification that addresses the rigorous challenges for debug interfaces. Applications that may benefit from this standard interface include automotive powertrain, data communications, computer peripherals, wireless systems and other control applications.

As advances in semiconductor and system design continue, embedded applications are using higher-performance embedded processors. Efficient use of these embedded processors requires software and hardware development tools that can easily access critical processor functionality. The lack of a unifying standard among the various embedded processors on the market has impeded this accessibility, preventing tool vendors from creating standard tools with consistent functionality across a broad range of processors. Ultimately, system developers are forced to pursue costly custom solutions to meet their tool needs.

To provide the best opportunity for achieving a world-wide development interface standard, it is prudent to leverage off an accepted pin interface that exists today—the IEEE 1149.1 standard.[a] The Nexus standard defines an extensible auxiliary port (AUX) that may either be used with the IEEE 1149.1 port or as a stand-alone development port. The Nexus standard defines the auxiliary pin functions, transfer protocols and standard development features.

a. For information on references, see APPENDIX E.

1.1 Terms and Definitions

Table 1-1 lists terms and definitions used in this standard.

Table 1-1 Terms and Definitions

Term	Definitions
Address	The term is used to indicate logical address. If there is no address translation in an application, then it also refers to the physical address.
Application Programming Interface (API)	API abstracts the semantics of APPENDIX B so that a tool can perform a common set of operations on any target, irrespective of hardware registers implemented on the target.
Auxiliary Port (AUX)	Refers to the Nexus auxiliary port. Used as auxiliary port to the IEEE 1149.1 interface, or as a stand-alone development port.
Branch Trace Messaging (BTM)	Visibility of addresses for taken branches and exceptions, and the number of sequential instruction units executed between each taken branch.
Data Breakpoint	Processor is halted at an appropriate instruction boundary after a trigger is set at a data valid time. The trigger is set when the data address and/or value matches a pre-selected address and/or value.
Calibration Constants	Performance-related constants which must be tuned for automotive powertrain and disk drive applications.
Calibration Variables	Intermediate calculations which must be visible during the calibration or tuning process to enable accurate tuning of calibration constants.
Client	A functional block on an embedded processor which will require development visibility and controllability. Examples are a central processing unit (CPU) and an intelligent peripheral.
Data Acquisition Messaging (DQM)	Visibility of related data parameters stored in internal resources, e.g. related calibration variables for automotive applications.
Data Read Messaging (DRM)	Visibility of data reads to internal memory-mapped resources, e.g. on-chip random access memory (RAM).
Data Write Messaging (DWM)	Visibility of data writes to internal memory-mapped resources, e.g. on-chip RAM.
Data Trace Messaging (DTM)	Visibility of how data flows through the embedded system. May include DRM and DWM. Refer to 2.1 Required Development Features for Embedded Processors on Page 5 for more information on data trace requirements.
Full-duplex	Messages can be transmitted in both directions between tool and target simultaneously.
Global Embedded Processor Debug Interface Standard Consortium (GEPDISC)	GEPDISC, renamed the Nexus 5001 Forum™ (http://www.ieee-isto.org/Nexus5001/), was formed to develop a much-needed embedded processor debug interface standard for embedded control applications. The internal name for this standard is “Nexus,” which is used throughout this document only.
Half-duplex	Messages can be transmitted in only one direction at a time between tool and target.
Hardware Breakpoint	Typically a hardware comparator used to halt the processor at an appropriate instruction boundary after an address or data value matches a pre-selected address or data value.

Table 1-1 Terms and Definitions (Continued)

Term	Definitions
High Speed Input/Output (HSIO)	The term *HSIO,* as used in the Nexus standard, is intended to refer to an external bus of the embedded processor. Assertion and negation timing is critical to system integrity.
Instruction Breakpoint	Processor is halted when all previous instructions are retired and just prior to when any architectural state is changed by the instruction associated with a pre-selected address.
IEEE 1149.1 Instruction Register (IR) and Data Register (DR) Sequence	IEEE 1149.1 IR scan to load an opcode value for selecting a development register. The selected development register is then accessed via an IEEE 1149.1 DR scan.
Low Speed Input/Output (LSIO)	LSIO pin functions are typically implemented on microcomputer units (MCUs), e.g. an output pin to set system configuration. Assertion and negation timing is not critical to system integrity.
Nexus	Internal code name for this standard.
Nexus API	API required by the Nexus standard.
Ownership Trace Messaging (OTM)	Visibility of the process/function that is currently executing.
Public Messages	Messages on the auxiliary pins for accomplishing common visibility and controllability requirements, e.g. DRM and DWM.
Read-Only Memory (ROM)	Read-only memory, such as non-volatile flash.
Standard	The phrase “according to the Nexus standard” is used to indicate “according to the Nexus standard contained in this document.”
Target	Generally refers to an end application or evaluation board, containing one or more embedded processors, which is connected to a development tool.
Transfer Code (TCODE)	Message header that identifies the number and/or size of packets to be transferred, and how to interpret each of the packets.
Memory Substitution Messaging (MSM)	Messaging for a memory substitution access in which internal accesses are re-directed through the auxiliary pins defined in the Nexus standard.
Nexus Recommended Register (NRR)	NRRs are defined in APPENDIX B.
Watchpoint	A data or instruction breakpoint that does not cause the processor to halt. Instead a pin is used to signal that the condition occurred.

1.2 Conventions

This document uses the following notational conventions:

ACTIVE_HIGH Names for signals that are active high are shown in uppercase text without an overbar. Signals that are active high are referred to as asserted when they are high and negated when they are low.

$\overline{\text{ACTIVE\_LOW}}$	A bar over a signal name indicates that the signal is active low. Active-low signals are referred to as asserted (active) when they are low, and negated when they are high.
0x0F	Hexadecimal numbers
0b0011	Binary numbers
LSB	Means least significant bit. The LSB is the lowest bit number, e.g. bit 0
MSB	Means most significant bit. The MSB is the highest bit number, e.g. bit 31
Set bit	To set a bit (or bits) means to establish logic level one on the bit (or bits), i.e. the voltage that corresponds to Boolean true (1) state.
Clear bit	To clear a bit (or bits) means to establish logic level zero on the bit (or bits), i.e. the voltage that corresponds to Boolean false (0) state.

SECTION 2
Basic Development Needs for Embedded Processors

Embedded developers need to accomplish specific functions using their development tools in order to accomplish their jobs. For logic analysis the basic needs are:

- To access instruction trace information with acceptable impact to the system under development. The developer needs to be able to interrogate and correlate instruction flow to real-world interactions.
- To retrieve information on how data flows through the system with acceptable impact to the system under development, and to understand what system resource(s) are creating and accessing data.
- To assess whether embedded software is meeting the required performance level with acceptable impact to the system under development.

For run control the basic needs are:

- To query and modify when the processor is halted, showing all locations available in the processor's supervisor map.
- To support breakpoint/watchpoint features in debuggers, either as hardware (HW) or software (SW) breakpoints depending on the architecture. Configuration of breakpoint/watchpoint features may be performed when the processor is halted.

2.1 Required Development Features for Embedded Processors

The evolution of high-performance microprocessor units (MPUs) and highly integrated MCUs has had an impact on development processes and tools. High-performance on-chip caches, flash and RAM, and other changes have eliminated the internal visibility needed for instruction and data trace. Thus there are specific features the Nexus standard should address, as listed below:

1. Program trace visibility is needed for development tools with acceptable impact to the system under development. With high-performance on-chip instruction cache and flash, visibility needed for program trace is restricted. In some applications the external bus is used for a secondary function such as general-purpose I/O, or is not available.

2. Data trace visibility is needed for development tools with acceptable impact to the system under development. With on-chip high performance data cache and RAM the visibility needed for data trace is restricted. There are 2 types of data visibility needed:

    a. Which process (i.e. which instruction address) wrote which data parameter and what new value was written?

    b. For a chosen data parameter which process(es) accessed it?

3. A standard development methodology and tool set is needed for embedded applications. Since embedded processor vendors generally do not support the same development interface/methodology, development methodology and tools are not compatible.

4. A development pin interface standard is needed to support development with multiple clients (processor cores or intelligent peripherals) on the embedded processor. The development pin interface comprises basic visibility and controllability of each processor independently.

5. An independant processor development pin interface standard is needed to support development for all mainstream processor architectures.

6. An embedded development pin interface standard is needed to allow for connection to multiple development tools. Tool arbitration may be needed if multiple development tool boxes are connected to the same target. Arbitration among tools is not addressed in this standard.

7. Multiplexing of development pin functions should be performed in a manner so that undue constraints are not placed on the embedded system developer. MCU vendors occasionally multiplex on the same pins development functions and general purpose I/O (GPIO). Guidelines should be given to eliminate improper multiplexing, especially out of reset, which can lead to unpredictable behaviors and anomalies in development tools. Development pins should be configurable to be in development mode out of reset.

8. A scalable development pin interface standard, which will work for different price targets of embedded MCUs/MPUs, is needed.

9. An embedded development pin interface standard is needed for cost-effective tools.

2.2 Additional Needs for Automotive Powertrain and Disk Drive Development

The development cycle for mechanical and electro-mechanical control applications includes additional needs for calibration of mechanical performance-related constants that are tuned for specific loads. The calibration process is performed during runtime. For calibration the basic needs for development tools are:

1. To acquire during mechanical operation (e.g. running an engine), rotational position synchronous data relating to calibration factors as they are being used or modified during high-speed transient events. This should be accomplished with minimal impact to the system under development.

2. To acquire during mechanical operation, time synchronous data relating to calibration factors as they are being used or modified during high-speed transient events. This should be accomplished with minimal impact to the system under development.

3. To coherently modify table(s) of calibration constants during mechanical operation.

Refer to the white paper, *The Evolution of Powertrain Microcontrollers and its Impact on Development Processes and Tools,* for more information on automotive powertrain development needs. A copy of the paper can be found on the Nexus web site, http://www.ieee-isto.org/Nexus5001/microcontrollers_evolution.pdf.

SECTION 3
Compliance and Performance Classifications

The capability of Nexus-compliant development ports shall comprise two basic designations: the development features supported by the port and the performance capability for downloading and uploading via the port. All development features described in the Nexus standard are assigned to at least one compliance classification: class 1, class 2, class 3 or class 4. Performance capability is designated by full- or half-duplex capability, and transfer bandwidth in megabits per second for both downloads to the embedded processor and uploads from the embedded processor. Thus Nexus-compliant development ports implemented by embedded processor integrated circuit (IC) vendors shall be designated as follows:

- Class 1, 2, 3 or 4 compliant
- Download rate (megabits per second)
- Upload rate (megabits per second)
- Full- or half-duplex

3.1 Compliance Classification

Complying embedded processors shall be designated as class 1, 2, 3 or 4, or as an approved compliance sub-class. Class 1 compliant devices implement the fewest Nexus development features. Class 4 compliant devices implement the most Nexus development features. Thus class 4 devices offer the most development capability and standardization. Classes 1, 2 and 3 devices offer a graduated subset of the Nexus development features, which may be appropriately suited for some applications.

Table 3-1 and **Table 3-2** show the minimum features for the four compliance classifications.

Table 3-1 Compliance Classification for Static Development Features

Development Feature	Class 1	Class 2	Class 3	Class 4	Nexus Feature
STATIC DEVELOPMENT FEATURES					
Read/write user registers in debug mode	V	V	V	V	Refer to SECTION 5
Read/write user memory in debug mode	A	A	A	A	Read/Write Access
Enter a debug mode from reset	A	A	A	A	Development Control and Status
Enter a debug mode from user mode	A	A	A	A	
Exit a debug mode to user mode	A	A	A	A	
Single step instruction in user mode and re-enter debug mode	A	A	A	A	
Stop program execution on instruction/ data breakpoint and enter debug mode (minimum 2 breakpoints)	A	A	A	A	Breakpoints/ Watchpoints

Note:
"A" indicates a required development feature that must be implemented via the Nexus API.
"V" indicates a required vendor-defined development feature implemented in the Nexus API.

3.1.1 Compliance Sub-Class for Application-Specific Development Needs

To comply with application-specific development needs, compliance sub-classes for specific applications shall be approved by a standards developing organization. Sub-classes are allowed when standardized support of application-specific development features are needed.

3.2 Performance Classification

Complying embedded processors shall be designated by a performance classification. The embedded processors shall be designated by full- or half-duplex capability, and transfer bandwidth in megabits per second for both downloads to the embedded processor and uploads from the embedded processor.

Full- and half-duplex capability is related to compliance classification as described in **Table 3-3**. Refer to APPENDIX A, which contains the connector options defined at the time this standard was released. Other connector options are expected as this standard evolves.

Table 3-2 Compliance Classification for Dynamic Development Features

Development Feature	Class 1	Class 2	Class 3	Class 4	Nexus Feature
DYNAMIC DEVELOPMENT FEATURES					
Ability to set breakpoint or watchpoint	A	A	A	A	Breakpoints/ Watchpoints
Device Identification	A	A and P	A and P	A and P	Device ID Message (see SECTION 6)
Ability to send out an event occurrence when watchpoint matches	P[1]	P	P	P	Watchpoint Message (see SECTION 6)
Monitor process ownership while processor runs in real-time	—	P	P	P	Ownership Trace
Monitor program flow while processor runs in real-time (logical address)	—	P	P	P	Program Trace
Monitor data writes while processor runs in real-time	—	—	P	P	Data Trace (Writes only)
Read/write memory locations while program runs in real-time	—	—	A and P	A and P	Read/Write Access
Program execution (instruction/data) from Nexus port for reset or exceptions	—	—	—	P	Memory Substitution
Ability to start ownership, program or data trace upon watchpoint occurrence	—	—	—	A	Development Control and Status
Ability to start memory substitution upon watchpoint occurrence or upon program access of device-specific address	—	—	—	O	Development Control and Status
Monitor data reads while processor runs in real-time	—	—	O	O	Data Trace (Reads and Writes)
LSIO port replacement and HSIO port sharing	—	O	O	O	Port Replacement/ Sharing
Transmit data values for acquisition by tool	—	—	O	O	Data Acquisition
Note: “A” indicates a required development feature that must be implemented via the Nexus API. “P” indicates a required development feature that must be implemented via the Nexus development port as a Public Message or with a Nexus port pin (as appropriate). “O” indicates an optional development feature as defined by the Nexus standard.					

1. Since no auxiliary port is required for class 1, the event occurrence should be provided via an $\overline{\text{EVTO}}$ pin defined in SECTION 7 Auxiliary Port Signals, or via a Message Out mechanism defined in SECTION 9.

Table 3-3 Performance Interface Options

Development Feature	Class 1	Class 2	Class 3	Class 4	Nexus Feature
IC DEVELOPMENT INTERFACE OPTIONS					
IEEE 1149.1 port	X	—	—	—	Half-duplex
IEEE 1149.1 or auxiliary input port with an auxiliary output port	—	X	X	X	Full-duplex

3.2.1 Interpreting Performance Classification

System developers of embedded processors should interpret the performance classification properly in order to assess the capability needed for their application. To do so, a basic knowledge is needed of Nexus features, the developer's code characteristics and visibility needs.

The transfer bandwidth for downloads can be thought of as the sustainable input bandwidth required to the device. Conversely, the transfer bandwidth for uploads can be thought of as the sustainable output bandwidth required from the device. Bandwidth requirements are typically determined by what Nexus development features are needed during runtime. Bandwidth requirements are compensated for by AUX size and clock rate.

The Nexus AUX is used to fulfill the output bandwidth requirements. In calculating the average output bandwidth requirements for an application, factors that may be considered are:

- Frequency of taken direct and indirect changes of flow
- Frequency and size of internal data reads/writes that must be visible
- Frequency and size of data that must be read from device

The Nexus AUX or the IEEE 1149.1 port is used to fulfill the input bandwidth requirements. In calculating the average input bandwidth requirements for an application, a factor that may be considered is the frequency and size of data that must be written to the device.

For this and other relevant factors, convert to bandwidth estimates. Refer to SECTION 5 Application Programming Interface (API) for more information on converting factors to bandwidth estimates.

3.3 Other Terminology within the Nexus Standard

The following list describes Public Messages and Vendor Defined Messages. These transfer operations occur via the Nexus AUX or the IEEE 1149.1 interface, between the development tool and complying embedded processors.

- Public Messages are defined for the AUX and the IEEE 1149.1 interface. These messages must be used for designated functions when these functions are implemented. Public Messages are specified pin protocols for accomplishing common configuration, status and visibility.

- Vendor Defined Messages are allowed via the AUX and the IEEE 1149.1 interface for development features that may be specific to each vendor. These messages must follow the protocol defined for the AUX.

The terms *vendor-defined extensions*, *vendor-defined operations* and *vendor-defined information* are used to indicate API extensions that may be implemented as needed for a vendor's device.

The term *vendor-defined bit fields* is used to indicate bit fields that may be defined as needed for the vendor's device.

The term *device-specific* is used to indicate allowances in the Nexus standard to match characteristics of a vendor's device. Device-specific packets may be of zero length (not implemented). For a tool to interpret message contents, it must determine from the device ID whether device-specific packets exist in each type of message. Some device-specific packets have their length fixed by this specification. Other device-specific packets are target processor dependent and have a fixed size determined by the processor vendor.

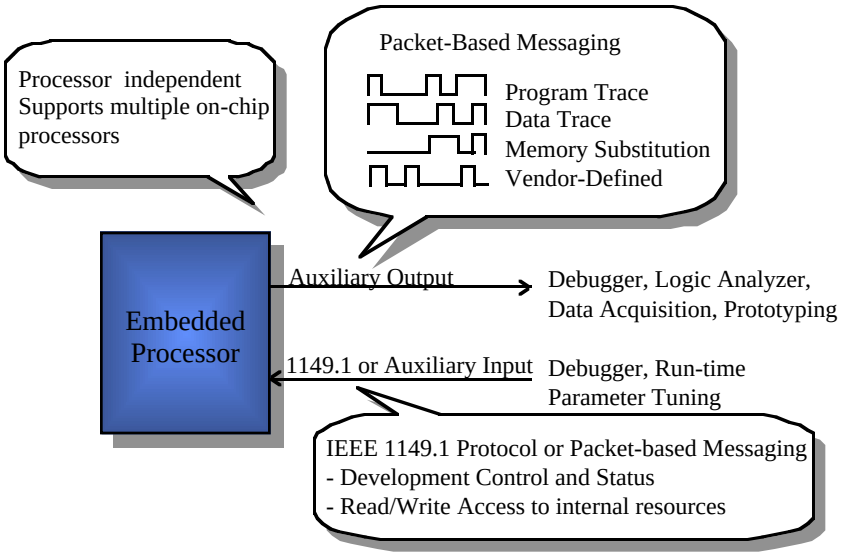
SECTION 4
Development Interface and Features

The development interface shall be implemented by Nexus classes 1, 2, 3 and 4 compliant embedded processors as described in **Table 3-3**. The development features shall be implemented by Nexus classes 1, 2, 3 and 4 compliant embedded processors as described in **Table 3-1** and **Table 3-2**.

4.1 Development Interface

Embedded processors complying to class 1 shall implement the IEEE 1149.1 standard for access to the minimum development features of compliance class 1. Embedded processors complying to classes 2, 3 and 4 shall implement a Nexus pin interface according to the Nexus standard, for external visibility required for the minimum development features of compliance classes 2, 3 and 4. Additionally, embedded processors complying to classes 2, 3 and 4 shall implement either an IEEE 1149.1 standard or Nexus pin interface according to the Nexus standard for access to the minimum development features of compliance classes 2, 3 and 4.

Figure 4-1 illustrates Nexus development interface options for a class 2, 3 or 4 embedded processor.



Note: The auxiliary input port is input only. Although the IEEE 1149.1 interface is bi-directional, for simplicity it is illustrated as input only.

Figure 4-1 Illustration of IEEE 1149.1/Nexus Development Interface

For implementation of the IEEE 1149.1 interface options for classes 2, 3 and 4 embedded processors, as referenced in **Figure 4-1**, only 3 auxiliary pins[b] are required for compliance. The performance classification, however, would also be minimal, and may only meet the transfer bandwidth requirements for low-end applications or for lower compliance classifications. If faster downloads to the embedded processor are required than is possible via the IEEE 1149.1 interface, an auxiliary input port should be implemented.

The Nexus standard allows for greater performance capability in either or both of the following ways: with a scalable auxiliary pin interface to transfer more bits on each clock and/or a faster transfer clock to transfer more bits per unit time. **Table 4-1** shows recommendations (not requirements) for AUX type.

Table 4-1 Recommendations for Auxiliary Port Type

Compliance Class and Port type	Number of Device Data Pins				
	1	2	4	8	16
Class 2 input port	X	X	—	—	—
Class 2 output port	X	X	—	—	—
Class 3 or 4 input port	X	X	X	—	—
Class 3 or 4 output port	—	—	X	X	X

4.1.1 IEEE 1149.1 Interface

The IEEE 1149.1 standard defines the required protocol for access to the minimum development features of compliance class 1. Additionally, the IEEE 1149.1 standard defines the required protocol for access to the minimum development features of compliance classes 2, 3 and 4, if the Nexus input interface option is not selected by the embedded processor IC developer.

The IEEE 1149.1 interface shall provide the following capability:

- IEEE 1149.1 sequences for access to processor identification, development control and status information according to the Nexus standard, e.g. configuring a breakpoint via API
- IEEE 1149.1 sequences for access to user memory-mapped registers during halt or runtime according to the Nexus standard
- IEEE 1149.1 sequences for access to development messages according to the Nexus standard, e.g. ownership trace
- IEEE 1149.1 sequences for access to all device-specific development features, e.g. user registers when processor is halted

b. Pins include TCK, TMS, TDI, TDO and $\overline{\text{TRST}}$.

4.1.2 Nexus Auxiliary Pin Interface

The Nexus pin interface shall be implemented according to the Nexus standard for external visibility required for the minimum development features of compliance classes 2, 3 and 4. Additionally, the Nexus pin interface shall be implemented according to the Nexus standard for access to the minimum development features of compliance classes 2, 3 and 4, if the IEEE 1149.1 interface option is not selected by the embedded processor IC developer.

The auxiliary interface shall provide the following external visibility according to the Nexus standard:[c]

- Trace of operating system software execution via OTM
- Program trace via BTM
- Data trace via DTM
- Signal watchpoint and breakpoint events
- Runtime system memory substitution via MSM
- Other high-bandwidth information transfer (vendor defined)

Additionally, the auxiliary interface shall provide the following access according to the Nexus standard, if the IEEE 1149.1 option is not selected by the embedded processor IC developer:

- Access to processor identification, development control and status information
- Access to user memory-mapped registers when halted or during runtime
- Access to all device-specific development features, e.g. user registers when processor is halted
- Provide optional access according to standard support for compliant development tools to implement port replacement of development port
- Provide optional access according to standard ability for embedded processor to transmit data values for acquisition by development tool

4.2 Development Features

The development features are described in 4.2.1 Application Programming Interface (API) on Page 16 through 4.2.10 Data Acquisition on Page 26.

c. Refer to 1.1 Terms and Definitions on Page 2 for definitions of all new terms in the list.

4.2.1 Application Programming Interface (API)

Embedded processors complying to all classes shall provide an API according to the Nexus standard.

Application: The Nexus API allows tool vendors to use a common “low-level semantic” API to abstract the low-level implementation details of each Nexus-compliant embedded processor. Since the Nexus standard does not mandate that the NRRs defined in APPENDIX B are implemented, embedded processor vendors are free to implement a different set of development registers. The required feature set for each class, however, must be available (refer to **Table 3-1** and **Table 3-2**). The API allows embedded processors not implementing the NRRs to be accessed in a standard manner.

The Nexus API is suitable for use by tools (debuggers, etc.), and also for the Nexus validation suite. It is designed to capture the low-level semantics of the Nexus features, so that it can be used to implement the bottom layers of a tool vendor’s own target debug API.

Tool vendors support a large number of different host platforms, operating systems and compilation systems. Emulator vendors similarly support a multitude of systems. The Nexus API is suitable for use in a wide variety of systems. It has been designed to not rely on any platform-specific, real-time operating-system-specific (RTOS-specific) or compiler-specific features.

Description: The Nexus API abstracts the semantics of the NRRs, so that a tools can perform a common set of operations on any target, irrespective of its class or its underlying register set.

The Nexus API is divided into two sections, which are described in SECTION 5:

- Emulator Hardware Abstraction Layer (HAL)
- Target Abstraction Layer (TAL)

4.2.2 Development Control and Status

Embedded processors complying to class 1, 2, 3, or 4 shall provide the development control and status required by the Nexus API.

Application: Standardized development control and status, and standardized access to it offer a significant degree of commonality. This can be leveraged by development tool vendors for creating standard tools with consistent functionality across a broad range of processors. Ultimately, system developers will benefit with more effective tools to meet their tool needs.

Description: For development control and status, embedded processors may implement the NRRs described in APPENDIX B, or a set of device-specific development control and status registers that implement the requirements specified by the Nexus API.

Nexus recommended development registers comprise the following control and status registers:

- Device ID Register
- Client Select Register
- Development Control Register
- Development Status Register
- User Base Address Register (memory map base address for user access of development features)
- Read/Write Access Registers
- Watchpoint Trigger Register
- Data Trace Attribute Registers (minimum of 2)
- Breakpoint/Watchpoint Control Registers (minimum of 2)[d]

Development control and status registers (Nexus recommended or device-specific) shall be accessed via the IEEE 1149.1 interface or the auxiliary interface according to the Nexus standard. A sequence for each interface is recommended in the Nexus standard.

4.2.3 Read/Write Access

Embedded processors complying to classes 3 and 4 shall provide read/write access to user memory-mapped resources according to the Nexus standard, either via the IEEE 1149.1 interface or the auxiliary pin interface. The capability to perform read/write access shall be provided when the processor is halted or running.

Application: The Read/Write Access feature supports runtime development visibility needed for real-time embedded applications. This feature also supports program tuning needs of automotive powertrain and disk drive applications.

d. Optionally, the 2 Breakpoint/Watchpoint Control Registers may be combined with the 2 Data Trace Attribute Registers so that a total of 2 registers may be simultaneously active, i.e. 2 Breakpoint/Watchpoint Control Registers, 2 Data Trace Attribute Registers or 1 Breakpoint/Watchpoint Control Register and 1 Data Trace Attribute Register.

Description: The Read/Write Access feature provides DMA-like access to user memory-mapped resources when the client is halted or during runtime. Either of two options may be used to implement this feature on the embedded processor. The NRRs may be implemented, which provide specific registers to support this feature. Otherwise, Public Messages as described in SECTION 6 are provided to implement this feature.

4.2.4 Ownership Trace

Embedded processors complying to classes 2, 3 and 4 shall provide ownership trace visibility according to the Nexus standard.

Application: Ownership trace provides a macroscopic view, such as task flow reconstruction, when debugging software written in a high level (or object-oriented) language. It offers the highest level of abstraction for tracking operating system software execution. This is especially useful when the developer is not interested in debugging at lower levels.

Ownership trace is especially important for embedded processors with a memory management unit, in which all processes can use the same logical program and data spaces. Ownership trace offers development tools a mechanism to decipher which set of symbolics and sources are associated for lower levels of visibility and debugging.

Description: Ownership trace information is transmitted out the AUX using OTM. OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. An Ownership Trace Message is transmitted to indicate when a new process/task is activated, allowing development tools to trace ownership flow. Additionally, for embedded processors which implement virtual addressing or address translation, an Ownership Trace Message is also transmitted periodically during runtime at a minimum frequency of every 256 Program Trace or Data Trace Messages.

The Nexus standard defines an OTM Register whose user memory map location is accessed via the IEEE 1149.1 or auxiliary pin interfaces. The OTM Register is to be updated as determined by the operating system software, to provide task/ process ID information. When new information is updated in the register by the embedded processor, the information is transmitted out via the AUX. Refer to B.5 User Base Address (UBA) Register on Page 134 for more information.

4.2.5 Program Trace

Embedded processors complying to class 2, 3 and 4 shall provide program trace visibility via the AUX according to the Nexus standard.

Application: The program trace feature defines a standard protocol for program trace visibility that is processor independent. Additionally, the amount of program trace signals that must be visible external to the device is significantly reduced over conventional methods. The benefit is standard logic analysis tools with consistent functionality.

Description: The program trace feature implements a Program Flow Change Model in which program trace is synchronized at each program flow discontinuity. A program flow discontinuity occurs at taken branches and exceptions.

Development tools can interpolate what transpires between program flow discontinuities by correlating information from BTM and static source or object code files. Self-modifying code cannot be traced with the Program Flow Change Model because the source code is not static.

BTM facilitates program trace by providing several key types of visibility. The visibility comprises the following

- Messaging for taken direct branches includes how many sequential instruction units were executed since the last taken branch or exception, and an indication of which client (if more than one are present on the embedded processor) took the branch. Direct branches that are not taken are included in the count of sequential instruction units. Direct branches that are taken are not included in the count of sequential instruction units.

- Messaging for taken indirect branches and exceptions includes how many sequential instruction units were executed since the last taken branch or exception, the unique portion of the branch target address or exception vector address, and an indication of which client (if more than one are present on the embedded processor) took the branch. Indirect branches that are not taken are included in the count of sequential instruction units. Indirect branches that are taken are not included in the count of sequential instruction units.

The information regarding the number of sequential instruction units executed since the last taken branch is used to facilitate the following:

1. Trace which direct branch is taken

2. Detect which instruction may have caused an exception

The unique portion of the indirect branch target address transmitted out the AUX is relative to a prior address transmitted out the AUX.

BTM can also be triggered during runtime at the occurrence of watchpoint.

4.2.5.1 Program Trace Overrun Errors

Embedded processors complying to classes 2, 3 and 4 shall provide the capability to detect and signal according to the Nexus standard program trace overrun errors, if the condition occurs during application of the embedded processor. A program trace error shall be detected and transmitted out the AUX according to the Nexus standard when any Program Trace Message is lost and not signaled via the AUX.

Embedded processors complying to class 4 shall provide the capability to delay the processor and avoid overruns.

Application: The overrun Error Message is to be used by development tools to notify the developer that program trace information has been lost. A BTM overrun error occurs when the number of messages to be transmitted via the AUX in a given time period exceeds the bandwidth capacity of the AUX.

Description: An Error Message provides an indication of which client (if more than one are present on the embedded processor) generated an error, and what type of error was generated.

4.2.5.2 Program Trace Synchronization

Embedded processors complying to classes 2, 3 and 4 shall provide the capability to synchronize program trace according to the Nexus standard. A Program Trace Message for synchronization shall be transmitted via the AUX by the embedded processor for the following conditions:

- Initial Program Trace Message upon exit of system reset, exit of a power down state or exit of a debug mode
- Periodically during runtime at a minimum frequency of every 256 Program Trace Messages
- When program trace is enabled during normal execution of the embedded processor
- Upon assertion of an Event-In pin
- Program trace overrun error
- Upon overflow of the sequential instruction unit counter
- Optionally upon occurrence of a watchpoint

Application: Due to the nature of some processor architectures, such as reduced instruction set computer (RISC) processors, some application programs may

comprise a significant number of direct branch instructions, and very few indirect branch instructions. Since BTM for taken direct branches does not provide the target address, program trace for these application programs must be accomplished in a relative manner (possibly without branch target address information). Synchronization messages ensure that development tools fully synchronize with the program flow regularly.

Description: A synchronization message, displayed as an individual message or as part of another message, always includes an indication of which client (if more than one are present on the embedded processor) is being synchronized and the full address of a recently executed instruction.

4.2.6 Data Trace

Embedded processors complying to classes 3 and 4 shall provide data trace for write visibility via the AUX according to the Nexus standard. Embedded processors complying to classes 3 and 4 may optionally provide data trace for read visibility via the AUX according to the Nexus standard.

Application: The data trace feature defines a standard protocol for data trace visibility of accesses to device-specific internal peripheral and memory locations. Practical limitations exist which constrain the number of locations which may be traced via the AUX. In application use, limiting the number of traced locations is necessary for effective use of data trace. Additionally, excluding processor stack area from data trace is beneficial.

Description: The data trace feature provides a minimum of 2 data trace windows which include the following qualifiers:

- Start and end user address for data trace
- Trace reads, writes or both within the start/end address range

Data accesses are monitored and qualifying data accesses are then transmitted out the AUX using DTM. DTM facilitates data trace by providing several key types of visibility. The messaging for data trace includes the unique portion of the data address and the data value. The unique portion of the data address transmitted out the AUX is relative to the prior data trace address transmitted out the AUX.

4.2.6.1 Data Trace Overrun Errors

Embedded processors complying to classes 3 and 4 shall provide the capability to detect and signal according to the Nexus standard data trace overrun errors, if the condition occurs during application of the embedded processor. A data trace error shall be detected and transmitted out the AUX according to the Nexus standard when any Data Trace Message is lost and not signaled via the AUX.

Embedded processors complying to class 4 shall provide the capability to delay the processor and avoid overruns.

Application: The overrun error message is to be used by development tools to notify the developer that data trace information has been lost. A DTM overrun error occurs when the number of messages to be transmitted via the AUX in a given time period exceeds the bandwidth capacity of the AUX.

Description: An error message provides an indication of which client (if more than one was present on the device) generated an error, and what type of error was generated.

4.2.6.2 Data Trace Synchronization

Embedded processors complying to classes 3 and 4 shall provide the capability to synchronize Data Trace Messages according to the Nexus standard. A Data Trace Synchronization Message shall be transmitted out the AUX by the embedded processor for the following conditions:

- Initial Data Trace Message upon exit of system reset, exit of a power down state or exit of a debug mode
- Periodically during runtime at a minimum frequency of every 256 Data Trace Messages
- When data trace is enabled during normal execution of the embedded processor
- Upon assertion of an Event-In pin
- Optionally upon occurrence of a watchpoint
- Data trace overrun error

Application: The output bandwidth requirements for the AUX are minimized for data trace by messaging out only the unique portion of the data address (instead of the complete address). Consequently a data trace address is reconstructed relative to each prior message. Synchronization messages provide the full address and ensure that development tools fully synchronize with the data trace regularly. Synchronization messages provide a reference address for subsequent DTMs, in which only the unique portion of the data trace address is transmitted.

Description: Synchronization information includes an indication of which client (if more than one are present on the device) is being synchronized and the full address for a recent data trace.

4.2.7 Memory Substitution

Embedded processors complying to class 4 shall provide the capability to activate user memory substitution via the AUX according to the Nexus standard. Memory substitution shall be capable of being activated upon exit of reset. A class 4 processor optionally may also support memory substitution activated upon the occurrence of a watchpoint or upon the occurrence of a data access or an instruction fetch from a device-specific address range. If supported, these optional capabilities must be implemented according to the Nexus standard.

Application: Memory substitution facilitates the software development process with program execution via the AUX upon exit of reset. Instructions are fetched and data is read from the development tool. Providing this capability via the AUX eliminates the need for a second development port dedicated to the software development process. Additionally, single stepping with instruction and data fetches via the AUX can be used for a non-real-time ROM monitor.

Optionally, the feature can be activated upon the occurrence of a watchpoint. This can support runtime patching for portions of internal ROM memory, with the patch provided via the AUX. ROM patching during runtime, however, is limited by capability factors of the complying embedded processor. Some factors that may limit the embedded processor are:

- The number of watchpoints implemented (one data value patch or one instruction sequence patch per watchpoint),
- The port size and clock rate of the auxiliary pin interface implemented and
- The portion of AUX bandwidth allocated for this feature if other messaging activities are also enabled at the same time.

Another option is to activate memory substitution upon the occurrence of a data access or an instruction fetch from a device-specific address range. For a full memory emulation capability, data reads, data writes and instruction fetches continue via the AUX until the address of a data access or instruction fetch falls outside a specified address range. The address range is device-specific and not typically programmable.

Memory substitution is not intended to be used for tuning parameters during runtime, such as is required for development of automotive powertrain and disk drive applications. There may be other applications, however, that may be able to use this feature during runtime.

Description: A class 4 embedded processor shall be capable of the following three types of memory substitution operations:

- Reading data and fetching instructions via the AUX (both data and instructions substituted by tool),
- Only reading data via the AUX (only data operands substituted by tool) and
- Only fetching instructions via the AUX (only instruction operands substituted by tool).

Note that class 4 compliant embedded processors are not required to write data via the AUX.

In memory substitution, the processor will make all qualifying memory-mapped fetches (data, instructions or both) via the AUX, in a single step or normal processor mode. Operands that are not enabled for memory substitution shall be accessed normally from user memory. Qualifying memory-mapped fetches are selected by configuring control and status information via the IEEE 1149.1 port or the AUX.

A class 4 embedded processor shall be capable of activating memory substitution upon exit from reset, and optionally capable of activating it upon the occurrence of a watchpoint or upon the occurrence of a data access or an instruction fetch from a device-specific address range.

The memory substitution feature can be activated upon exit from reset by configuring control and status information via the IEEE 1149.1 port or the AUX. It can be activated on watchpoint occurrence by configuring watchpoint trigger information via the IEEE 1149.1 port or the AUX. It can be activated on occurrence of a data access or an instruction fetch from a device-specific address range. No configuration is required for the latter.

When memory substitution is activated upon exit of reset or a watchpoint occurrence, the processor will make all qualifying memory-mapped fetches via the AUX, until the development tool disables memory substitution. When memory substitution is activated upon the occurrence of a data access or an instruction fetch from a device-specific address range, the processor will make all qualifying memory-mapped fetches via the AUX, until the address of a data access or instruction fetch falls outside the device-specific address range. Once memory substitution is disabled, user memory shall be accessed normally.

MSM facilitates memory substitution by providing messages for access requests and transfers via the AUX. These comprise the following:

- Messaging for a memory substitution access request provided from the processor to an external development tool containing access attributes such as instruction/data, size, and the memory-mapped address. The full address is transmitted for each memory substitution access request.

- Messaging for a memory substitution transfer provided from the external development tool to a processor containing the instruction or data specified by access attributes.

- Messaging for the *last* memory substitution transfer provided from the external development tool to a processor containing the *last* instruction or data specified by access attributes, and a disable command for MSM. Subsequent memory-mapped accesses will be accessed normally from the internal memory-mapped resource designated by the access attributes.

For patching an ROM instruction sequence, the last memory substitution transfer may consist of a direct branch to the address following the patched instruction sequence.

4.2.8 Breakpoints/Watchpoints

Embedded processors complying to class 1, 2, 3 or 4 shall provide a minimum of 2 instruction/data hardware breakpoints.[e] Embedded processors complying to class 2, 3 or 4 shall provide according to the Nexus standard the capability to message via the AUX any occurrence of a watchpoint.

Application: The breakpoint and watchpoint features facilitate the software development process by allowing the developer to halt at a specific processor state or to signal a specific processor state. If there is an internal ROM or if a breakpoint or trap instruction does not exist in the vendor's architecture, then this feature becomes a valuable tool for development.

Description: Breakpoints and watchpoints comprise the following:

- Data breakpoint—processor is halted at an appropriate instruction boundary after a trigger is set at a data valid time. The trigger is set when the data address and/or data value matches a pre-selected address and/or value.

- Instruction breakpoint—processor is halted when all previous instructions are retired and just prior to when any architectural state is changed by the instruction associated with a pre-selected address.

- Watchpoint—a data or instruction breakpoint which does not cause the processor to halt. A watchpoint message via the AUX is used to signal that the condition occurred.

e. Optionally, the 2 Breakpoint/Watchpoint Control Registers may be combined with the 2 Data Trace Attribute Registers so that a total of 2 registers may be simultaneously active, i.e. 2 Breakpoint/Watchpoint Control Registers, 2 Data Trace Attribute Registers or 1 Breakpoint/Watchpoint Control Register and 1 Data Trace Attribute Register.

4.2.9 Port Replacement and Sharing

Embedded processors complying to class 2, 3 or 4 may optionally implement support on the auxiliary pins according to the Nexus standard for LSIO port replacement. Embedded processors complying to class 2, 3 or 4 may optionally share AUX pins according to the Nexus standard with a second HSIO port.

Application: In embedded processor applications the use of every pin is scrutinized by embedded processor developers. Inevitably there are never enough pins available on the embedded processor to meet both the application and development needs. Pins that are designated for product development are often reduced or removed to make way for other pin functions directly used in the application. Port replacement and sharing support is intended to solve this dilemma by using common embedded processor ports for a secondary development support function.

Description: Port replacement provides a mechanism for LSIO pin functions to be replaced using messages via the AUX. The standard messages between the development tool and AUX provide the necessary information for the development tool to replace the LSIO port (with additional delay).

The mechanism is enabled in a plug and play manner. When a development tool is connected to the AUX, it enables the AUX with Port Replacement Messages. When no development tool is connected, the port functions as only an LSIO port.

Port sharing allows a primary port function, such as an external bus of the embedded processor, to be shared with an auxiliary output port function. For example, an L2 cache bus function and the auxiliary output port function may share the same pins.

Most bus traffic in a typical application will be due to external bus cycles on the shared pins for accessing system resources. During external bus cycles, AUX control signals are negated and the development tool ignores the external bus information. Upon occurrence of a condition that generates development information (e.g. BTM and DTM), a corresponding message is sent out the shared pins and captured by the tool.

4.2.10 Data Acquisition

Embedded processors complying to class 2, 3 or 4 may optionally implement support for data acquisition by the development tool from the embedded processor, via the AUX, according to the Nexus standard.

Application: The feature provides a mechanism for visibility of intermediate variables calculated by the embedded processor. An application includes time-critical parameters passed to an external co-processor for rapid prototyping. The

embedded processor is required to queue up data for acquisition by the development tool.

Description: DQM provides the capability to message on the AUX internal data related to one another. Because of construction, DQM is a more efficiently packed message than DTM.

DQM facilitates data acquisition by providing several key types of visibility: display data ID tag (to specify which group of data) and all data values. The display data ID tag is typically a reference number to identify the data, e.g. 3 may represent time-critical parameters passed to an external co-processor for rapid prototyping.

As mentioned above, the embedded processor must queue up DQM. In the Nexus standard a user memory-mapped interface and protocol is recommended (not required) for the embedded processor to queue up DQM. The user memory-mapped locations are configured via the IEEE 1149.1 or auxiliary pin interface. The protocol recommended consists of writing to a designated user memory-mapped location to generate a DQM with a specific display data ID tag.

SECTION 5
Application Programming Interface (API)

5.1 Introduction

The Nexus API consists of two layers (see **Figure 5-1**).

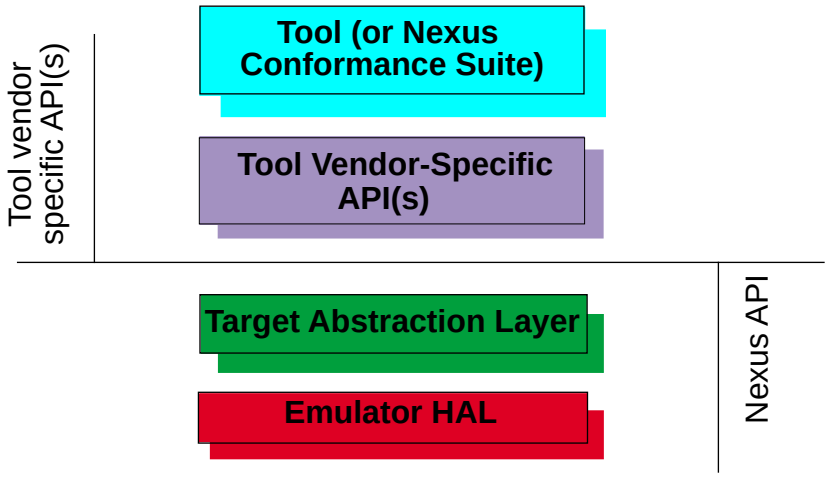


Figure 5-1 Software Layers Showing Nexus API Layers

These layers are defined by a standard set of header files, supplied as part of the Nexus API specification (see the Nexus API Header Files section on the Nexus web site, http://www.ieee-isto.org/Nexus5001/standard.html). The implementations of these layers must conform to the standard header files.

- The Target Abstraction Layer (TAL)

  The semiconductor vendor supplies the emulator vendor with the source code or binaries for the TAL. It provides an implementation of the Nexus debug semantics, using the underlying target's on-chip debug hardware. It uses the emulator HAL in order to communicate with the target system.

  Tools (and the Nexus Validation Tests) are built on top of this API. Tools will typically consist of multiple layers of APIs. The TAL API is used to implement the bottom levels of the tool vendor's existing APIs. For this reason, the TAL API only provides facilities to access the debug hardware features itself, and does not implement anything that is normally provided in the tool vendor's higher-level APIs.

- The Emulator Hardware Abstraction Layer (HAL)

  An implementation of this layer is supplied by an emulator[f] vendor for their emulator hardware. It abstracts the mechanisms by which the host machine communicates with the emulator hardware and will typically include a device driver to communicate with the underlying emulator/ host interface.

There are two typical tool/emulator scenarios:

1. **Figure 5-2** where the tool executes directly on the emulator. Thus the TAL executes on the emulator itself.

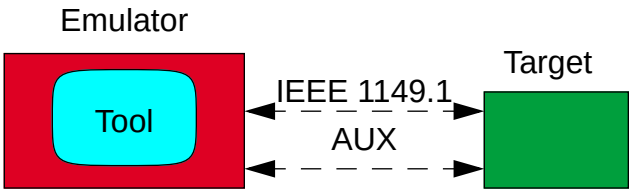


Figure 5-2 Emulator-Based Tool

2. **Figure 5-3** where the tool executes on a host computer connected to an emulator, and the host and the emulator are connected via some means (e.g. Ethernet or PCI). In this situation, a device driver interface is typically used on the host to communicate to the emulator hardware.

   Thus the “real-time” properties of the interface (e.g. reading trace packets at high speed) are dealt with by both the emulator hardware and firmware, in conjunction with a device driver running on the host.

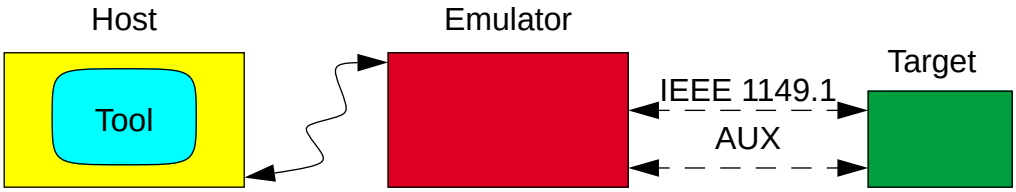


Figure 5-3 Host-Based Tool, Connected to Target via Emulator

The TAL and Emulator HAL are designed with the model described in **Figure 5-3** in mind. They assume that high-speed tool/emulator interface issues are resolved in the lower-levels of the Emulator HAL interface (such as the device driver).

f. In this context, the term *emulator* is used to describe the hardware/software layer that is used to interface between the debugging tool and the target.

5.2 Overview

Table 5-1 cross-references the development features listed in sub-section 3.1 Compliance Classification on Page 8 with the corresponding Nexus API Function.

Table 5-1 Cross-Reference of Development Features and Nexus API Functions

Development Feature	Nexus API Function
Read/write user registers in debug mode	`nx_Control()`
Read/write user memory in debug mode	`nx_WriteMem()` `nx_ReadMem()`
Enter a debug mode from reset	`nx_Control()`
Enter a debug mode from user mode	`nx_Control()`
Exit a debug mode to user mode	`nx_Control()`
Single step instruction in user mode and re-enter debug mode	`nx_SetEvent()`
Stop program execution on instruction/ data breakpoint and enter debug mode (minimum 2 breakpoints)	`nx_SetEvent()`
Ability to set breakpoint or watchpoint	`nx_SetEvent()`
Device Identification	`nx_Control()`
Ability to send out an event occurrence when watchpoint matches	`nx_SetEvent()`
Monitor process ownership while processor runs in real-time	`nx_GetEvent()`
Monitor program flow while processor runs in real-time (logical address)	`nx_GetEvent()`
Monitor data writes while processor runs in real-time	`nx_GetEvent()`
Read/write memory locations while program runs in real-time	`nx_WriteMem()` `nx_ReadMem()`
Program execution (instruction/data) from Nexus port for reset or exceptions	`nx_GetEvent()`
Ability to start ownership, program, or data trace upon watchpoint occurrence	`nx_SetEvent()`
Ability to start memory substitution upon watchpoint occurrence or upon program access of device-specific address	`nx_SetEvent()`
Monitor data reads while processor runs in real-time	`nx_GetEvent()`
LSIO port replacement and HSIO port sharing	`nx_Control()`
Transmit data values for acquisition by tool	`nx_SetEvent()`

5.3 Vendor Extensions

The Nexus standard, and thus the Nexus API, allows vendor-defined extensions in terms of:

1. Vendor-defined extensions to standard Nexus operations
2. Additional vendor-defined operations
3. Additional vendor-defined information in messages

These extensions are made using a data-driven approach (i.e. extra data structures are used with the standard Nexus API operations).

5.4 Target-Specific Issues

Nexus targets have a large amount of device-specific behavior, such as the size of physical addresses or the number and size of registers. The basic data types associated with these aspects are defined by the semiconductor vendor. However, as the emulator vendor supplies a binary instance of the API, the implementation of the Emulator HAL is the area that finally resolves these platform-specific issues.

5.5 Deliverables

1. The Nexus standard supplies the API header files.
2. Semiconductor vendors supply Emulator vendors with an implementation of the Target Abstraction Layer, in source code or binary form.
3. The Emulator vendors compile the Target Abstraction Layer, and build it with their implementation of the Emulator HAL in order to provide a binary deliverable that can be used by a tool vendor.

5.6 Concepts and Data Types

5.6.1 Naming Conventions

The Nexus API uses the following naming conventions:

- Target Abstraction Layer data types are prefixed with “nxt_”; functions are prefixed with “nx_”; constants are prefixed with “NX_”. Vendor-defined types are prefixed with “nxvt_”.
- Emulator HAL functions are prefixed with “nxhal_”.

5.6.2 Header Files

The following Nexus API header files, will be made available on the Nexus web site, http://www.ieee-isto.org/Nexus5001/.

- `nxtypes.h` defines the standard data types.
- `nxapi.h` defines the standard Target Abstraction Layer entry points.
- `nxhal.h` defines the Emulator HAL entry points.
- `nxvtypes.h` should be supplied by the semiconductor vendor. It defines the target-specific data types.

5.6.3 Status/Error Values

A common set of status/error values is defined in nxtypes.h.

```
typedef enum {
  NX_ERROR_NONE             = 0,  /* success */
  NX_ERROR_FAILED           = 1,  /* generic failure */
  NX_ERROR_NO_CAPABILITY    = 2,  /* operation not within capabilities */
  NX_ERROR_NOSPACE          = 3   /* insufficient buffer space or requested op */
} nxt_Status;
```

In addition, both the Target Abstraction Layer, and the Emulator HAL use either a callback routine or an `nx_GetLastError()` routine to provide further information on the cause of errors.

```
void (*errorCallback)(const char *)
```

This callback can be invoked to provide much more detailed error information than what is normally available from standard error return codes (e.g. “write to address 0x42 failed”). Note that setting this pointer to null will prevent callbacks.

As an alternative to using the preceding callback mechanism, a “get last error” routine can be used:

```
nxt_Status nx_GetLastError(char * lastError, int maxBytes)
```

where `lastError` is a pointer to a buffer allocated for the error string and `maxBytes` is the size of the allocated error buffer.

The number of error codes is limited as there are few actions possible when an error occurs.

5.6.3.1 Error Handling

The Nexus APIs are designed to be used as the lowest level of a vendor’s own APIs. Thus, facilities to validate memory addresses etc. are not implemented in

the Nexus APIs—they are assumed to be implemented according to the higher-level APIs' requirements. Thus, it is possible to invoke the Nexus API routines with invalid data. This can cause undefined effects (e.g. writing to invalid memory). Thus, the higher-level APIs are assumed to be checking the validity of such operations before they are performed.

As a last resort, the Nexus API connection should be capable of being closed without disturbing the host machine (i.e. without crashing, or locking a device driver connection).

5.7 Target Abstraction Layer (TAL)

Sub-sections 5.7.1 Opening a Connection—nx_Open on Page 33 through 5.7.8 Reading an Event—nx_GetEvent on Page 46 describe the TAL functions and data types. The Emulator HAL functions are described in sub-section 5.8 Emulator HAL on Page 48.

5.7.1 Opening a Connection—nx_0pen

```
nxt_Handle *nx_Open (const nxt_TargetSpec *tSpec,
                     void (*errorCallback)(const char *),
                     nxt_Status *status);
```

Note that it is the responsibility of the Target Abstraction Layer to allocate and de-allocate the memory associated with the handle.

5.7.1.1 Preconditions

`nxtypes.h` defines the following types used for opening a connection:

`nxt_Endian` is used to specify the byte ordering of the emulator and/or target.

```
typedef enum {
  NX_ENDIAN_UNKNOWN,             /* used for initial assignment */
  NX_ENDIAN_BIG,
  NX_ENDIAN_LITTLE,
  NX_ENDIAN_OTHER
} nxt_Endian;
```

`nxt_PortType` specifies either the Nexus AUX or the IEEE 1149.1 JTAG port.

```
typedef enum {
  NX_PORT_TYPE_UNAVAILABLE,      /* an unused port */
  NX_PORT_TYPE_JTAG,
  NX_PORT_TYPE_AUX,
  NX_PORT_TYPE_OTHER             /* reserved for future use */
} nxt_PortType;
```


`nxt_TargetSpec` is used when opening a connection to a target system.

```
typedef struct {
  nxt_PortType accessPort;        /* port for access/ctrl */
  nxt_PortType unsolicitedPort;   /* port for unsolicited messages */
  nxt_Endian targetEndian;
  nxvt_VendorDefinedTargetSpec VendorTargetSpec;
                                  /* see header files for vendor extensions */
} nxt_TargetSpec;
```

`nxapi.h` defines the `nx_Open` function.

The `tSpec` parameter should be set up to define the target's byte ordering and the ports on which it is connected. The available options depend upon the target and the Emulator system. The full set of connection options is defined below:

In order to connect to the IEEE 1149.1 JTAG port for access functions only:

```
tSpec.accessPort = NX_PORT_TYPE_JTAG;
tSpec.unsolicitedPort = NX_PORT_TYPE_UNAVAILABLE;
```

In order to connect to the IEEE 1149.1 JTAG port for access functions, and also for unsolicited messages (where available):

```
tSpec.accessPort = NX_PORT_TYPE_JTAG;
tSpec.unsolicitedPort = NX_PORT_TYPE_JTAG;
```

In order to connect to the IEEE 1149.1 JTAG port for access functions, and the AUX port for unsolicited messages (where available):

```
tSpec.accessPort = NX_PORT_TYPE_JTAG;
tSpec.unsolicitedPort = NX_PORT_TYPE_AUX;
```

In order to connect to the AUX port for both access and unsolicited messages (where available):

```
tSpec.accessPort = NX_PORT_TYPE_AUX;
tSpec.unsolicitedPort = NX_PORT_TYPE_AUX;
```

5.7.1.2 Postconditions

If `nx_Open` succeeds, a handle is returned and status is set to `NX_ERROR_NONE`. If it fails, `NULL` is returned and the status is set to `NX_ERROR_NO_CAPABILITY`.

`nxtypes.h` defines `nxt_Handle` as:

```
typedef struct {
  nxt_Capability cap;             /* capabilities */
  nxt_TargetSpec targetSpec;
  void *nxTALPrivatePtr;          /* private data as needed for the API */
  void *nxHALPrivatePtr;          /* private data as needed for the HAL */
} nxt_Handle;
```

The `nxTALPrivatePtr` and `nxHALPrivatePtr` allows the API and Emulator HAL implementations to reference private data (such that they can avoid keeping static state). `void *` is used so that this data is opaque to a user of the API/HAL. Use of these pointers is optional.

When `nx_Open` returns successfully, `nxt_Capability` has been set up to define the target's capabilities. The capabilities provide a fuller set of information than the Nexus Class system classifications.

5.7.1.2.1 Event Capabilities

Most of the Nexus features are abstracted as *events* (see 5.7.6 Setting an event—nx_SetEvent on Page 41). Thus, the target's capabilities are described in terms of event ID values.

Event IDs that are not available are defined by `NX_EVENTID_INVALID.`

```
#define NX_EVENTID_INVALID (0)

typedef struct {
   char *apiVersionString;         /* NUL character terminated version string */
   char *halInfo;                  /* NUL character terminated HAL-specific string */
   nxt_Endian targetEndian;        /* target byte order */
   nxt_Endian emuEndian;           /* emulator byte order */
   int deviceId;                   /* as per DID Message (see APPENDIX B) */
   int maxMemMap;
   int maxMemAccessPriority;
   int maxAccessSize;

/* event IDs for event capabilities or NX_EVENTID_INVALID if not used */

   int btmEventId;
   int dtmMinEventId;
   int dtmMaxEventId;
   int otmEventId;
   int substitutionEventId;
   int watchMinEventId;
   int watchMaxEventId;
   int breakMinEventId;
   int breakMaxEventId;
} nxt_Capability;
```

`deviceId` is extracted from the target.

`maxMemMap` and `maxMemAccessPriority` determine the maximum number of memory map and memory access priority structures that can be used in memory read/write operations (see 5.7.4 Writing Target Memory—nx_WriteMem on Page 40 and 5.7.5 Reading Target Memory—nx_ReadMem on Page 41).

`maxAccessSize` indicates the maximum allowable memory access size in bits. A value of 0 indicates a don't care selection (i.e. use default size).

If BTM is not available, `btmEventId` is equal to `NX_EVENTID_INVALID`, otherwise `btmEventId` identifies the ID to use.

Availability of OTM and memory substitution is similarly indicated using the `otmEventId` and `substitutionEventId` fields.

If DTM is not available, `dtmMinEventId` and `dtmMaxEventId` are equal to `NX_EVENTID_INVALID`. Otherwise, these fields are used to indicate the ID of the first and last DTM channels (e.g. if 2 DTM channels are available, and are allocated from event ID 4 upward, `dtmMinEventId` would be set to 4 and `dtmMaxEventId` would be set to 5).

The availability and number of watchpoints and breakpoints are denoted by `watchMinEventId, watchMaxEventId, breakMinEventId` and `breakMaxEventId.`

5.7.2 Closing a Connection—nx_Close

```
nxt_Status nx_Close (nxt_Handle *handle);
```

5.7.2.1 Preconditions

`handle` is from a successful invocation of `nx_Open.`

5.7.2.2 Postconditions

`nx_Close` does not return a status code because, in the unlikely event of a failure, nothing can be done anyway. After calling `nx_Close,` the `handle` is de-allocated in the TAL, and must not be used in subsequent operations.

It must be possible to call `nx_Close` to “abort” a connection in the event of failure; this may involve calling it from a non-sequential thread (e.g. from a signal handler context).

The higher-level APIs built on top of the Nexus APIs will typically contain time-out and error recovery mechanisms. These are not built into the Nexus API itself because doing so would make the Nexus API definitions dependent on the platform and RTOS, and thus restrict the environments on which the API could be deployed.

5.7.3 Controlling a Connection—nx_Control

```
nxt_Status nx_Control (nxt_Handle *handle, nxt_CtrlData ctrl);
```

5.7.3.1 Preconditions

`handle` is from a successful invocation of `nx_Open,` and `ctrl` specifies the control operation to apply.

`nxt_CtrlTag` specifies a control action; these correspond to non-event-based control of the target system (i.e. controls that globally apply to the target).


```
typedef enum {
  NX_CTRL_SET_CLIENT          = 0x1,
  NX_CTRL_OVERRUN_MODE        = 0x2,
  NX_CTRL_SUBSTITUTION_MODE   = 0x3,
  NX_CTRL_RESETORHALT         = 0x4,
  NX_CTRL_EVENTIN             = 0x5,
  NX_CTRL_CLIENTBREAK         = 0x6,

  NX_CTRL_RESTART_FROM_BREAKPOINT        = 0x50
    /* values from 0x100 upward are for vendor extensions */
} nxt_CtrlTag;

typedef struct {
  nxt_CtrlTag cTag;
  union {
   struct {
    int clientId;
   } setClient;              /* if cTag == NX_CTRL_SET_CLIENT */
                             /* core select = client select */
   struct {
    int delay;               /* == 0 to disable */
   } overrunMode;            /* if cTag == NX_CTRL_OVERRUN_MODE */
   struct {
    int enable;              /* if == 0 disable substitution else enable it */
    int forInstructions;     /* valid if enable != 0 */
    int forData;             /* valid if enable != 0 */
   } substitutionMode;       /* if cTag == NX_CTRL_SUBSTITUTION_MODE */
   struct {
    int performResetSequence;
    int haltState;
    } resetOrHalt;;          /* if cTag == NX_CTRL_RESETORHALT */
   struct {
     nxvt_Registers regs;
   } restartFromBreakpoint;  /* NX_CTRL_RESTART_FROM_BREAKPOINT */
   struct {
    int eventIn;
   } eventInMode;            /* if cTag == NX_CTRL_EVENTIN */
   struct {
    int clientBreakpoint;
   } clientBreakpointMode;   /* if cTag == NX_CTRL_CLIENTBREAK */
  } u;
    nxvt_VendorDefinedCtrlData vendorDefinedCtrlData;
} nxt_CtrlData;
```

The following control operations are possible (where `x` is an instance of `nxt_CtrlData`).

To select a client

`x.cTag = NX_CTRL_SET_CLIENT`

`x.u.setClient.clientId` specifies the client.

If an invalid value is given (e.g. the target does not contain multiple clients), `nx_Control` will return `NX_ERROR_NO_CAPABILITY`.

To disable, or enable, a specified trace overrun mode

`x.cTag = NX_CTRL_OVERRUN_MODE`

`x.u.overrunMode.delay` defines a specified trace overrun mode.

- When equal to 0, enables overruns.
- When equal to 1, delays for BTM.
- When equal to 2, delays for BTM and OTM.
- When equal to 3, delays for BTM, DTM and OTM.

To disable, or enable, a specified memory substitution mode out of reset

(Unless explicitly enabled, substitution out of reset is disabled by default.)

`x.cTag = NX_CTRL_SUBSTITUTION_MODE`

Assigning `x.u.substitutionMode.enable` a value of 0 disables memory substitution, and assigning a value other than 0 enables it.

When enabling memory substitution, `x.u.substitutionMode.forInstructions` and `x.u.substitutionMode.forData` are used to enable the instruction and data substitution features.

To perform a reset or halt/unhalt the target

`x.cTag = NX_CTRL_RESETORHALT`

If `x.u.resetOrHalt.performResetSequence` is not equal to 0, a reset will be applied to the target:

- If the target's IEEE 1149.1 JTAG port is being used for the Nexus control port (`nx_Open` was invoked with `tSpec.accessPort` equal to `NX_PORT_TYPE_JTAG`), reset will be applied to the IEEE 1149.1 JTAG port.
- If the target's AUX port is being used for the Nexus control port (`nx_Open` was invoked with `tSpec.accessPort` equal to `NX_PORT_TYPE_AUX`), reset will be applied to the AUX port.

If `x.u.resetOrHalt.performResetSequence` is equal to 0, a reset sequence will not be applied.

`x.u.resetOrHalt.haltState` is used to alter the halt (or "debug") state of the target.

- When equal to 0, the target will be unhalted, i.e. removed from debug mode so that it is free to continue execution (assuming it is not stopped at a breakpoint/single step event).
- When not equal to 0, the target will be halted, i.e. put into debug mode such that it will not execute until unhalted.

To define EVTI development control

`x.cTag = NX_CTRL_EVENTIN`

`x.u.eventInMode.eventIn` defines the action that will occur when the $\overline{\text{EVTI}}$ signal transitions from high to low.

- When equal to 0, use $\overline{\text{EVTI}}$ for program or data trace synchronization.
- When equal to 1, use $\overline{\text{EVTI}}$ for breakpoint generation.
- When equal to 2, $\overline{\text{EVTI}}$ transitions have no action.

To enable, or disable, global breakpoint recognition

`x.cTag = NX_CTRL_CLIENTBREAK`

`x.u.clientBreakpointMode.clientBreakpoint` enables global breakpoint breaks on the client.

- When equal to 0, break for internal breakpoints only.
- When equal to 1, break for global and internal breakpoints.

To restart from a breakpoint

`x.cTag = NX_CTRL_RESTART_FROM_BREAKPOINT`

`x.u.restartFromBreakpoint.regs` contains a device-specific data structure, which contains the state of all the target's general-purpose registers. This data structure is described in header file `nxvtypes.h`.

In order to restart from a breakpoint, a breakpoint must have previously occurred (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

Vendor-Defined control operations

`x.cTag` values from 0x100 upward can be used to allow vendors to supply additional control operations.

The `x.u.vendorDefinedCtrlData` field is used to supply vendor-defined information with respect to a vendor-defined value in `x.cTag`.

5.7.3.2 Postconditions

The selected operation or configuration is performed and `NX_ERROR_NONE` is returned, or another `nxt_Status` value is returned to indicate an error.

5.7.4 Writing Target Memory—`nx_WriteMem`

```
nxt_Status nx_WriteMem (nxt_Handle *handle,
                        const int map, const int accessPriority,
                        const nxvt_Address addr, const size_t numBytes,
                        const int accessSize, const void *bytesToWrite);
```

5.7.4.1 Preconditions

`handle` is from a successful invocation of `nx_Open`.

`map` is used to specify a memory map; not all targets support this facility. `handle->cap.maxMemMap` determines the maximum value that can be used.

`accessPriority` is used to specify a bus access priority to be used when applying the write operation. `handle->cap.maxMemAccessPriority` determines the maximum value which can be used.

`addr` is used to specify the target address to write to. Because the size of the target's address varies from target to target, the address is vendor defined.

`accessSize` is used to specify the byte access size to be used during the data transfer. A value of 0 indicates that the Target Abstraction Layer is allowed to determine the access size to be used. `handle->cap.maxAccessSize` determines the maximum value that can be used.

`numBytes` is used to specify how many bytes to write, and `bytesToWrite` contains `numBytes` of data to transfer.

5.7.4.2 Postconditions

If the operation succeeds, `NX_ERROR_NONE` is returned, otherwise `NX_ERROR_FAILED` is returned, and the implementation is free to invoke the error callback supplied with `nx_Open` to provide further information.

5.7.5 Reading Target Memory—nx_ReadMem

```
nxt_Status nx_ReadMem (nxt_Handle *handle,
                       const int map, const int accessPriority,
                       const nxvt_Address addr, const size_t numBytes,
                       const int accessSize, void *bytesRead);
```

As per `nx_WriteMem`, but reads target memory instead of writing it.

5.7.6 Setting an event—nx_SetEvent

```
nxt_Status nx_SetEvent (nxt_Handle *handle, const nxt_SetEvent *setEvent);
```

5.7.6.1 Preconditions

`handle` is from a successful invocation of `nx_Open`.

`setEvent` is of type `nxt_SetEvent`, which specifies all the data necessary when setting an event.

```
typedef struct {
  nxt_EventType eType;
  int eid;
  union {
    struct {
       nxt_RWMode rwMode;           /* trigger on read, write or both */
       nxvt_Address addr;
       nxt_BreakpointOperand op;    /* match ID address/data, or both */
       nxvt_Word data;
       nxvt_Word mask;
       nxt_EventOutputMode eoMode;

    } breakpoint;
    struct {
       nxt_RWMode rwMode;           /* trigger on read, write or both */
       nxvt_Address addr;
       nxt_WatchpointOperand op;    /* match ID address/ data, or both */
       nxvt_Word data;
       nxvt_Word mask;
       nxt_EventOutputMode eoMode;
    } watchpoint;
    struct {
       int startTriggerId;          /* disabled == 0, else = event ID */
       int endTriggerId;            /* disabled == 0, else = event ID */
    } btm;
    struct {
       nxt_RWMode rwMode;
       int startTriggerId;          /* disabled == 0, else = event ID */
       int endTriggerId;            /* disabled == 0, else = event ID */
       nxvt_Address startAddr;      /* (addr >= startAddr) && (addr <= endAddr) */
       nxvt_Address endAddr;
    } dtm;
    struct {
       int startTriggerId;       /* disabled == 0, else = event ID */
    } substitution;
```



```
    nxvt_VendorDefinedBasicSetEvent vendorDefinedBasic;
  } u;
  nxvt_VendorDefinedExtensionSetEvent vendorDefinedExtension;
} nxt_SetEvent;
```

`nxt_EventType` specifies the type of an event.

```
typedef enum {
  NX_ETYPE_STEP             = 0x1,     /* single step */
  NX_ETYPE_BREAKPOINT       = 0x10,    /* breakpoint */
  NX_ETYPE_WATCHPOINT       = 0x20,    /* watchpoint */
  NX_ETYPE_BTM              = 0x50,    /* branch trace messaging */
  NX_ETYPE_DTM              = 0x51,    /* data trace messaging */
  NX_ETYPE_OTM              = 0x52,    /* ownership trace messaging */
  NX_ETYPE_SUBSTITUTION     = 0x70     /* memory substitution */
  /* values from 0x100 upwards are for vendor extensions */
} nxt_EventType;
```

`nxt_RWMode` is used when setting breakpoint and watchpoint events, to specify the type of memory access they will match.

```
typedef enum {
  NX_RWMODE_READ,
  NX_RWMODE_WRITE,
  NX_RWMODE_READ_OR_WRITE
} nxt_RWMode;
```

Similarly, `nxt_BreakpointOperand` and `nxt_WatchpointOperand` specify the address/data to use when matching for the event.

```
typedef enum {
    NX_BREAKPOINT_DATAADDR,
    NX_BREAKPOINT_DATAVALUE,
    NX_BREAKPOINT_DATAADDR_AND_DATAVALUE,
    NX_BREAKPOINT_INSTRADDR
} nxt_BreakpointOperand;

typedef enum {
    NX_WATCHPOINT_DATAADDR,
    NX_WATCHPOINT_DATAVALUE,
    NX_WATCHPOINT_DATAADDR_AND_DATAVALUE
    NX_WATCHPOINT_INSTRADDR
} nxt_WatchpointOperand;
```

`nxt_EventOutputMode` is used when setting the mode for the Event-Out ($\overline{\text{EVTO}}$) pin.

```
typedef enum {
  NX_EVTO_NOCHANGE,          /* occurrence does not change Event-Out pin */
  NX_EVTO_TRIGGERED          /* occurrence asserts Event Output */
} nxt_EventOutputMode;
```


The following events may be set (where `x` is an instance of `nxt_SetEvent`):

Single Step Event

`x.eType` is `NX_ETYPE_STEP;`

`x.eid` is not used for this type of event.

When the step event occurs, an `NX_READ_EVENT_BREAKSTEP` will be returned from `nx_GetEvent` (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

Breakpoint/Watchpoint Event

`If x.eType` is `NX_ETYPE_BREAKPOINT`, then when a breakpoint event occurs, `NX_READ_EVENT_BREAKSTEP` will be returned (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

`If x.eType` is `NX_ETYPE_WATCHPOINT`, then when the watchpoint event occurs, `NX_READ_EVENT_MESSAGE` will be returned (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

`x.eid` must correspond to the relevant event ID value from the `nxt_Capability` value returned in `nxt_Handle` (see 5.7.1.2.1 Event Capabilities on Page 35).

`x.u.breakpoint.rwMode` or `x.u.watchpoint.rwMode` specifies whether the match will occur for a read and/or write operation; `x.u.breakpoint.op` or `x.u.watchpoint.op` specifies the type of match that will be made:

Instruction Address Breakpoint

`x.u.breakpoint.op = NX_BREAKPOINT_INSTRADDR`

`x.u.breakpoint.addr` specifies the instruction address to match against.

`x.u.breakpoint.rwMode` is not used in this case.

Data Address Breakpoint

`x.u.breakpoint.op = NX_BREAKPOINT_DATAADDR`

`x.u.breakpoint.addr` specifies the data address to match against.

`x.u.breakpoint.rwMode` specifies whether read and/or write accesses will trigger it.

Data Value Breakpoint

`x.u.breakpoint.op = NX_BREAKPOINT_DATAVALUE`

`x.u.breakpoint.rwMode` specifies whether read and/or write accesses will trigger it.

`x.u.breakpoint.data` specifies the data to match against; `x.u.breakpoint.mask` specifies the mask to use when performing the match.

Data Address and Data Value Breakpoint

`x.u.breakpoint.op = NX_BREAKPOINT_DATAADDR_AND_DATAVALUE`

`x.u.breakpoint.addr` specifies the data address to match against.

`x.u.breakpoint.data` specifies the data to match against; `x.u.breakpoint.mask` specifies the mask to use when performing the match.

Watchpoints

The previous examples are the same for watchpoint values.

Branch Trace Messaging (BTM)

`x.eType = NX_ETYPE_BTM`

`x.eid` must correspond to the `btmEventId` value from the `nxt_Capability` value returned in `nxt_Handle` (see 5.7.1.2.1 Event Capabilities on Page 35). If this is `NX_EVENTID_INVALID`, BTM is not available.

`x.u.btm.startTriggerId` specifies an event ID that can enable the specified BTM event. To make it permanently enabled, supply `NX_EVENTID_INVALID` here, otherwise use the event ID from a breakpoint/watchpoint event ID (see 5.7.1.2.1 Event Capabilities on Page 35).

Similarly `x.u.btm.endTriggerId` specifies an event ID that can disable the specified BTM event.

When the watchpoint event occurs, `NX_READ_EVENT_MESSAGE` will be returned (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

Data Trace Messaging (DTM)

`x.eType = NX_ETYPE_DTM`

`x.eid` must correspond to a value between the `dtmMinEventId` and `dtmMaxEventId` values from the `nxt_Capability` value returned in `nxt_Handle` (see 5.7.1.2.1 Event Capabilities on Page 35). If this is `NX_EVENTID_INVALID`, DTM is not available.

`x.u.dtm.startTriggerId` specifies an event ID that can enable the specified DTM event. To make it permanently enabled, supply `NX_EVENTID_INVALID` here, otherwise use the event ID from a breakpoint/watchpoint event ID (see 5.7.1.2.1 Event Capabilities on Page 35).

Similarly `x.u.dtm.endTriggerId` specifies an event ID that can disable the specified DTM event.

When the DTM event occurs, `NX_READ_EVENT_MESSAGE` will be returned (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

`x.u.dtm.rwMode` specifies whether a read and/or write access will trigger the DTM event; `x.u.dtm.startAddr` and `endAddr` specify the addresses within which accesses will trigger.

Ownership Trace Messaging (OTM)

`x.eType = NX_ETYPE_OTM`

`x.eid` must correspond to `otmEventId` from the `nxt_Capability` value returned in `nxt_Handle` (see 5.7.1.2.1 Event Capabilities on Page 35). If this is `NX_EVENTID_INVALID`, OTM is not available.

If `x.eid` is `NX_EVENTID_INVALID`, OTM will be disabled. If `x.eid` is equal to `otmEventId` and `otmEventId` is not `NX_EVENTID_INVALID`, then OTM is enabled.

Memory Substitution

`x.eType = NX_ETYPE_SUBSTITUTION`

`x.eid` must correspond to `substitutionEventId` from the `nxt_Capability` value returned in `nxt_Handle` (see 5.7.1.2.1 Event Capabilities on Page 35). If this is `NX_EVENTID_INVALID`, memory substitution is not available.

`x.u.substitution.startTriggerId` specifies an event ID that can enable the specified substitution event. To make it permanently enabled, supply `NX_EVENTID_INVALID` here, otherwise use the event ID from a breakpoint/watchpoint event ID (see 5.7.1.2.1 Event Capabilities on Page 35).

Vendor-defined events/vendor-defined event extensions

`x.eType` values from `0x100` upwards can be used to allow vendors to supply additional events.

The `x.u.vendorDefinedBasic` field is used to supply vendor-defined information with respect to a vendor-defined value in `x.eType`.

The `x.u.vendorDefinedExtension` field is used to allow vendor-defined extensions to standard events.

5.7.6.2 Postconditions

If the operation succeeds, `NX_ERROR_NONE` is returned, otherwise `NX_ERROR_FAILED` or `NX_ERROR_NO_CAPABILITY` is returned.

5.7.7 Clearing an Event—nx_ClearEvent

```
void nx_ClearEvent (nxt_Handle *handle, const int eid);
```

5.7.7.1 Preconditions

`handle` is from a successful invocation of `nx_Open`.

`eid` is the ID of an event previously set up with `nx_SetEvent`.

5.7.7.2 Postconditions

Once this function completes, the specified event will be disabled.

5.7.8 Reading an Event—nx_GetEvent

```
nxt_Status nx_GetEvent (nxt_Handle *handle, nxt_ReceivedEvent *event,
                        int maxBytes, const int block);
```

5.7.8.1 Preconditions

`handle` is from a successful invocation of `nx_Open`.

`event` is a pointer to a block of memory to receive the event.

`maxBytes` is set by the caller to equal the maximum number of bytes that can be received in an event.

`block` specifies whether this function will block until an event is available from the target.

5.7.8.2 Postconditions

The following data types are used to hold the event:

`nxt_Packet` defines a Nexus Packet.

```
typedef struct {
  int numBitsInPacket;
  void *data;                       /* stream of ((numBitsInPacket + 7)/8) bytes */
} nxt_Packet;
```

`nxt_Message` defines a Nexus Message.

```
typedef struct {
  nt numPackets;
  nxt_Packet *packets;
} nxt_Message;
```

`nxt_ReadEvent` defines the type of event.

```
typedef enum {
  NX_READ_EVENT_MESSAGE = 0x1,
  NX_READ_EVENT_BREAKSTEP = 0x2,
  NX_READ_EVENT_INPUTPIN = 0x3
} nxt_ReadEvent;
```

`nxt_ReceivedEvent` contains the actual event:

```
typedef struct {
  nxt_ReadEvent rTag;
  union {
    nxt_Message message;            /* rTag == NX_READ_EVENT_MESSAGE */
    nxvt_Registers regs;            /* rTag == NX_READ_EVENT_BREAKPOINT */
    struct {
       int level;
    } inputPin;;/* rTag == NX_READ_EVENT_INPUTPIN */
  } u;
} nxt_ReceivedEvent;
```

If block is equal to 1, `nx_GetEvent` will block until an event is available. The `nxt_ReceivedEvent *event` data structure can be modified as follows:

- If a message was read, `x.rTag` is `NX_READ_EVENT_MESSAGE`.

  `x->u.message` contains the message.

NOTE

Ultimately, it would be desirable to supply a library here that decodes a message into a C data structure. However, at this point in time, this functionality is considered out of the scope of this specification.

- If a breakpoint or single step occurred, `x->rTag` is equal to `NX_READ_EVENT_BREAKSTEP`.

  `x->u.regs` contains the target's register state. The target will not execute further until restarted using `nx_Control` with `NX_CTRL_RESTART_FROM_BREAKSTEP`. These registers are vendor defined.

- If the AUX port's Event-In pin changes state, `x->rTag` is equal to `NX_READ_EVENT_INPUTPIN`.

  `x->u.level` contains the current level of the pin.

If `block` is equal to 0, `nx_GetEvent` will return an event if one is available.

5.8 Emulator HAL

`nxhal.h` defines the entry points.

CAUTION

The Emulator HAL described here is incomplete - it provides mechanisms to connect and disconnect. IEEE 1149.1 implementations will be able to access any underlying facilities using the NRR read/write mechanism. Similarly, AUX implementations will be able to access the target using the WriteMessage and Get Event mechanisms.

However, the HAL specified here does not provide generic functionality necessary to implement get breakpoint/step event, read/write the target's purpose register state or implement with the debug exception handler to restart. These facilities are not addressed by the Nexus Hardware Specifications, and can not be readily expressed in a totally portable Emulator HAL. Therefore, they are left as target-specific features that will vary from HAL to HAL.

5.8.1 Opening a Connection—nxhal_Open

This function is used by `nx_Open` to connect to the Emulator; its parameters are identical to `nx_Open`.

`nxhal.h` defines the `nxhal_Open` function.

```
nxt_Handle *nxhal_Open (const nxt_TargetSpec *tSpec,
                        void (*errorCallback)(const char *),
                        nxt_Status *status);
```

5.8.1.1 Preconditions

`tSpec` should be set up to define the target's byte ordering, and the ports on which the target can be connected. The available options depend upon the target and the Emulator system. The full set of connection options are defined in 5.7.1 Opening a Connection—nx_Open on Page 33.

5.8.1.2 Postconditions

If `nxhal_Open` succeeds, a handle is returned and status is set to `NX_ERROR_NONE`; if it fails, `NULL` is returned, and `status` is set to `NX_ERROR_NO_CAPABILITY`.

Before `nxhal_Open` returns successfully, it must set the following fields of `nxt_Capability` to define the emulator's capabilities:

- `halInfo`. A NUL-terminated string containing a version/information number specific to the Emulator.
- `emuEndian`. The Emulator's byte ordering.

5.8.2 Closing a Connection—nxhal_Close

This function is used by `nx_Close` to close a connection to the Emulator; its parameters and behavior are identical to 5.7.2 Closing a Connection—nx_Close on Page 36.

```
void nxhal_Close (nxt_Handle *handle);
```

5.8.3 Writing to a Nexus IEEE 1149.1 Register—nxhal_WriteNRR

This function is used by various functions in the Target Abstraction Layer, whose implementation needs to write the underlying debug control registers, which are accessible via NRRs.

This function is only available for targets that are configured to perform access control via IEEE 1149.1 JTAG (i.e. `nxhal_Open` was invoked with `tSpec.accessPort` equal to `NX_PORT_TYPE_JTAG`).

```
nxt_Status nxhal_WriteNRR (nxt_Handle *handle,
                           const int index, const int numBitsInNRR,
                           const void *data);
```

5.8.3.1 Preconditions

`handle` is from a successful invocation of `nxhal_Open`.

`index` ranges from 0 through 127; it is used to identify which NRR to write.

`numBitsInNRR` specifies the number of bits in the NRR, specified by `index`.

`data` contains ((`numBitsInNRR`+7)/8) bytes of data to write to the NRR.

5.8.3.2 Postconditions

If the write operation succeeds, `nxhal_WriteNRR` will return `NX_ERROR_NONE`, or else it will return `NX_ERROR_FAILED`.

5.8.4 Reading a Nexus IEEE 1149.1 Register—nxhal_ReadNRR

This function is used by various functions in the Target Abstraction Layer, whose implementations need to read the underlying debug control registers, which are accessible via NRRs.

This function is only available for targets that are configured to perform access control via IEEE 1149.1 (i.e. `nxhal_Open` was invoked with `tSpec.accessPort` equal to `NX_PORT_TYPE_JTAG`).

```
nxt_Status nxhal_ReadNRR (nxt_Handle *handle,
                          const int index, const int numBitsInNRR,
                          void *data);
```

5.8.4.1 Preconditions

`handle` is from a successful invocation of `nxhal_Open`.

`index` ranges from 0 through 127; it is used to identify which NRR to read.

`numBitsInNRR` specifies the bit size of the NRR, specified by `index`.

5.8.4.2 Postconditions

If the read operation succeeds, `data` points to ((`numBitsInNRR`+7)/8) bytes read from the NRR and `NX_ERROR_NONE` is returned, else `NX_ERROR_FAILED` is returned.

5.8.5 Reading an Event—nxhal_GetEvent

This function is used by `nx_GetEvent` (see 5.7.8 Reading an Event—nx_GetEvent on Page 46).

```
nxt_Status nxhal_GetEvent (nxt_Handle *handle, nxt_ReceivedEvent *event,
                           int maxBytes, const int block);
```

This function behaves as per `nx_GetEvent`, except that it does not need to implement `nxt_ReceivedEvent.rTag` equal to `NX_READ_EVENT_BREAKSTEP`.

SECTION 6
Public Messages

The AUX provides a high-speed communication link between the tool and a target processor. All communication over this link uses messages in one or both directions.

The format and meaning of certain messages, called Public Messages, are defined by this specification. Other messages can be vendor defined and defined by the target processor vendor. Tools require enhanced capability to be able to support Vendor Defined Messages.

All messages start with a 6-bit TCODE which uniquely defines the type of message. Fifty-six TCODEs (values 0 to 55) indicate that the message is a Public Message defined by the Nexus standard, or reserved for future definition by the Nexus standard. Seven TCODEs (values 56 to 62) indicate that the message is a Vendor Defined Message. One TCODE (value 63) indicates that the message is a Vendor Defined Message and then a second level code designated by the vendor further identifies the specific message.

In addition to being transferred over the AUX, Public Messages can also be transferred via an IEEE 1149.1 port using the method described in SECTION 8.

6.1 Compliance Requirements for Public Messages

Embedded processors complying to class 2, 3 or 4 shall implement messaging via the AUX according to the Nexus standard. Embedded processors complying to class 1 may optionally implement messaging via the IEEE 1149.1 interface.

Embedded processors shall implement the minimum Public Messages as required per the compliance class. **Table 6-1** lists the minimum required Public Messages per compliance class. Embedded processors may optionally implement Vendor Defined Messages.

Table 6-1 Minimum Required Public Messages

Message Type	Compliance Class	Minimum Required Public Messages
Device ID	2, 3, 4	Device ID.
Ownership Trace	2, 3, 4	Ownership Trace.
Program Trace	2, 3, 4	Direct Branch, Indirect Branch, Synchronization[1], Error.
Data Trace	3, 4	Data Write, Data Write Message with Sync, Error.
Read/Write Access	3, 4	1) For embedded processors that implement the recommended registers defined in APPENDIX B: Target Ready, Read Register, Write Register, Read/ Write Response. 2) For embedded processors that implement device-specific registers: Read Target, Write Target, Read Next Target Data, Write Next Target Data, Target Response.
Watchpoint	2, 3, 4	Watchpoint Message.
Memory Substitution	4	Read Tool, Read Next Tool Data, Tool Response.

1. The Direct Branch with Sync and Indirect Branch with Sync Message may be implemented instead of the Synchronization Message.

6.2 Definitions and Terminology

The following terms relate to Public Messages:

Message: Each message starts with a 6-bit TCODE, which defines the type of information carried in the message and its format. The TCODE packet length for all Public Messages must be 6 bits. When messages are transferred via the AUX, message start/end (MSE) signaling protocol, described in SECTION 8 Auxiliary Port Message Protocol, defines the start and the end of each message.

Transmission Order: Messages are transmitted least-significant bit(s) first. Additionally, program and Data Trace Messages are transmitted in a temporal order such that the transmission of messages should correlate as closely as possible with the temporal occurrence of activity on the embedded processor.

Packet: A packet is a distinct piece of the information contained within a message and messages may contain one or more packets. A common alternative term for a packet is a *field*. When messages are transferred via the AUX, MSE signaling protocol defines the end of each variable-length packet.

Port Boundary: This relates the size of a packet to the width of the Auxiliary Input Port or Auxiliary Output Port.

Variable: When the format of a message specifies that a packet is variable-size, it means that the message must contain the packet but that its size may vary from a

minimum of 1 bit. When messages are transferred via the AUX, variable-size packets must end on a port boundary, if necessary by zero filling bit positions beyond the highest-order bit of the variable data. Since variable-size packets may be of different lengths in messages of the same type, the tool must use the MSE signaling protocol to determine the end of packet boundaries.

Device-Specific: The term *device-specific* is used to indicate allowances in the Nexus standard to match characteristics of a vendor's device. Device-specific packets may be of zero length (not implemented). For a tool to interpret message content, it must determine from the device ID whether device-specific packets exist in each type of message.

Some device-specific packets have their lengths fixed by this specification. Other device-specific packets are target processor dependent and have a fixed size determined by the processor vendor.

Sync and Non-Sync Trace Messages: Program Trace and Data Trace Messages fall into two broad categories—normal (or *non-sync*) versions and *with-sync* versions. The main difference between the two categories is that *with-sync* versions include full addresses whereas normal versions contain addresses which are relative to a previous trace message.

Next Address Generation: To minimize the size of trace messages, the address packets in the normal (*non-sync*) versions of all trace messages contain a compressed address. This compressed address, called *the unique portion of the address,* is relative to the address associated with a previous trace message of the same type. Program Trace Messages contain an address that is relative to the previous Program Trace Message; Data Trace Messages contain an address that is relative to the previous Data Trace Message.

The target processor computes the relative address by exclusive-OR-ing the current program or data address with the full address associated with the previous Program Trace Message or Data Trace Message (see **Figure 6-1**).

Number of Messages Cancelled: Several messages for program and data trace synchronization (Direct Branch Message With Sync, Indirect Branch Message With Sync, Data Write Message with Sync, Data Read Message with Sync) contain a packet for the number of messages cancelled. There are three device-specific interpretations of this field:

1. For embedded processors which transmit only valid messages, this field will have a value of 0.

2. For embedded processors that do not queue up Program/Data Trace Messages as they become back-logged, but can truncate the current message as it is being transmitted to send out a fresher message, this field will have a value of 1 if the previous message has been truncated.

3. For embedded processors that send out preliminary Program Trace Messages (e.g. speculative execution) and later correct the trace information by cancelling fully transmitted messages, this field will notify the tool of the number of fully transmitted program (or data) messages to be cancelled.

Example of how the target processor generates the address to send in a trace message:

Previous absolute address (A1) = 0x003FC01,
Absolute address associated with new trace occurrence (A2) = 0x0003F365

```
     A1 =  0000 0000 0000 0011 1111 1100 0000 0001
     A2 =  0000 0000 0000 0011 1111 0011 0110 0101
A1⊕A2 =  0000 0000 0000 0000 0000 1111 0110 0100
```

The unique portion of the address (M1), sent in the message (high-order zeros are suppressed):

```
M1 =  1111 0110 0100
```

Example of how the tool recreates the address based on its previously calculated address and the address contained in the trace message:

Previously calculated address (A1) = 0x003FC01,
Address in message (M1) = 0xF64

```
     A1 =  0000 0000 0000 0011 1111 1100 0000 0001
     M1 =  0000 0000 0000 0000 0000 1111 0110 0100
A1⊕M1 =  0000 0000 0000 0011 1111 0011 0110 0101
```

Address recreated by the tool = 0x0003F365

Figure 6-1 Next Address Generation Example

Periodic Message Counter: Because addresses contained in normal program and Data Trace Messages are relative, the loss or corruption of a trace message means that the tool will be unable to correctly recreate addresses following the corruption or loss. To minimize the effect of any such loss or corruption of a trace message, the target processor must send a with-sync version at least every 256 trace messages.

To provide this function, the target processor must maintain two periodic message counters, one for counting normal Program Trace Messages and the other for counting normal Data Trace Messages.

Program Address and Data Address Threads: On embedded processors that implement data and program trace, there will be an address thread for each type of trace - the data address thread and the instruction address thread. Messages containing a data address packet will be encoded and compressed using the data address most recently transmitted, thus creating a data address thread. Likewise, messages containing an instruction address packet will be encoded and compressed using the instruction address most recently transmitted, thus creating an instruction address thread.

Source of Message Transmission: Many of the Public Messages contain a packet that may be used to identify which client was the source of the message transmission. In embedded processors that comprise only a single client, this packet need not be transmitted. For embedded processors that comprise multiple clients, this packet must be transmitted as part of the message to identify the source of the message transmission.

6.3 Complete List of Nexus Public Messages

Table 6-2 gives a complete list of Nexus Public Messages.

Table 6-2 Nexus Public Messages

Message Name	TCODE Value	Direction
Debug Status	0	From target
Device ID	1	From target
Ownership Trace	2	From target
Program Trace, Direct Branch	3	From target
Program Trace, Indirect Branch	4	From target
Data Trace, Data Write	5	From target
Data Trace, Data Read	6	From target
Data Acquisition	7	From target
Error	8	From target
Program Trace Synchronization	9	From target
Program Trace Correction	10	From target
Program Trace, Direct Branch with Sync.	11	From target
Program Trace, Indirect Branch with Sync.	12	From target
Data Trace, Data Write with Sync.	13	From target
Data Trace, Data Read with Sync.	14	From target
Watchpoint Hit	15	From target
Target Ready	16	Both ways

Table 6-2 Nexus Public Messages (Continued)

Message Name	TCODE Value	Direction
Read Register	17	From tool
Write Register	18	From tool
Read/Write Response	19	Both ways
Port Replacement, Output	20	From target
Port Replacement, Input	21	From tool
Read Target/Tool	22	Both ways
Write Target/Tool	23	Both ways
Read Next Target/Tool Data	24	Both ways
Write Next Target/Tool Data	25	Both ways
Target/Tool Response	26	Both ways
Reserved	27–55	Both ways
Vendor Defined Message	56–62	Both ways
Vendor Defined Extension Message	63 (0x3F)	Both ways

6.4 Detailed Description of Public Messages

In this sub-section, Public Messages are grouped according to their function.

In the following Public Message descriptions, a description table lists the most significant bits (transmitted last) at the top of the table, and the least significant bits (transmitted first) at the bottom of the table (see **Table 6-3** through **Table 6-31**).

6.4.1 Debug Status

Table 6-3 Debug Status Message

Debug Status Message			Direction: from target
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	STATUS	Device-specific	Status information[1]
0	SRC	Device-specific	Client that is source of message.
6	TCODE	Fixed	Value = 0

1. For target processors that implement the NRRs described in APPENDIX B, the status packet contains the same information as the Development Status (DS) Register. For target processors that implement device-specific registers, the status packet must provide all the information required by the API.

Message Occurrence: This message is output by the target whenever there is a change of state of any of the following:

- Entry to the debug exception handler
- Exit from the debug exception handler
- Change in power-managed state of the processor
- Detection of a breakpoint

In addition to having the target processor send a Device Status Message whenever the debug status changes, the tool is able to request the current debug status at any time. For target processors that implement the NRRs described in APPENDIX B, the tool requests the current debug status by sending a Read Register Message containing the Development Status opcode. For target processors that implement device-specific registers, the API knows which device-specific register(s) to read to obtain device status.

6.4.2 Device Identity

Table 6-4 Device Identity Message

Device Identity Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	ID	Fixed	ID information. Refer to B.1 Device ID (DID) Register on Page 129.
6	TCODE	Fixed	Value = 1

Message Occurrence: If the AUX is enabled, i.e. a tool is connected, this message is output by the target processor only after the target's debug logic has been reset by the tool. The tool resets the target's debug logic by asserting and de-asserting the $\overline{\text{RSTI}}$ signal.

NOTE

A Device Identity Message is not automatically output following power-on reset, even when a tool is connected. The tool must specifically reset the target's debug logic for this message to occur.

In addition to having the target processor send a Device Identity Message following a debug logic reset, the tool is able to request the device identity at any time. For target processors that implement the NRRs described in APPENDIX B, the tool requests the device identity by sending a Read Register Message containing the device identity opcode. For target processors which implement device-specific registers, the API knows which register to read to obtain the device identity.

6.4.3 Ownership Trace

Table 6-5 Ownership Trace Message

Ownership Trace Message			Direction: from target
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	PROCESS	Device-specific	Task/process ID.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 2

Message Occurrence: There are three ways in which this message may occur:

1. For target processors in which the OTM Register is a read-only alias of a process ID register, this message is output whenever the process ID changes.

2. For target processors where the OTM Register is directly written by the operating system or application code to indicate the current process or task, this message is output whenever the operating system writes to the OTM Register.

3. For target processors using virtual memory, this message is output immediately prior to (or immediately following) a Program or Data Trace with Sync Message produced when a periodic message counter expires. This allows a tool to be regularly updated with the latest process ID.

6.4.4 Program Trace

There are six different Public Messages associated with program tracing:

- Program Trace, Direct Branch
- Program Trace, Indirect Branch
- Program Trace, Direct Branch with Sync
- Program Trace, Indirect Branch with Sync
- Program Trace Synchronization
- Program Trace Correction

Table 6-6 Program Trace, Direct Branch Message

Program Trace, Direct Branch Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	I-CNT	Variable	Number of instruction units executed since the last taken branch. Each message may contain up to eight of these packets, each one corresponding to a direct branch taken.[1]
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 3

1. In target architectures in which all instructions are the same size, this packet contains the number of instructions executed since last taken branch. If instructions are of variable size, then the number reported is the number of instruction units.

Message Occurrence: This message is output by the target processor whenever there is a change of program flow caused by a conditional branch. To conserve AUX bandwidth and trace buffer space, the target processor may queue trace information about taken direct branches and output one message containing up to eight I-CNT packets.

Table 6-7 Program Trace, Indirect Branch Message

Program Trace, Indirect Branch Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	U-ADDR	Variable	The unique portion of the branch target address for a taken indirect branch or exception.
1	I-CNT	Variable	Number of instruction units executed since the last taken branch. [1]
0	SRC	Device-specific	Client which is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 4

1. In target architectures in which all instructions are the same size, then this packet contains the number of instructions executed since last taken branch. If instructions are of variable size, then the number reported is the number of instruction units.

Message Occurrence: This message is output by the target processor whenever there is a change of program flow caused by a subroutine call, return instruction or asynchronous interrupt/trap.

Table 6-8 Program Trace, Direct Branch with Sinc Message

Program Trace, Direct Branch with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	F-ADDR	Variable	The full target address for a taken direct branch. Most significant bits that have a value of 0 may be truncated.
1	CANCEL	Variable	Number of previous Program Trace Messages that should be ignored by the tool. This packet is generated only by processors performing speculative execution where a trace message may be output before it is known whether the branch was actually taken.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 11

Message Occurrence: This message is output by the target processor when any of the following conditions occurs:

1. Upon exit from system reset. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool. This trace message follows exit from reset for target processors not capable of immediately generating a Program Trace Synchronization Message.

2. A direct branch is detected after program trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool.

4. A direct branch is detected following the processor exiting from debug mode.

5. A direct branch is detected and an overrun condition had previously occurred in which one or more branch trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an Error code (ECODE) value of 00001 or 00111 immediately prior to the Program Trace, Direct Branch with Sync Message.

6. A direct branch is detected and the periodic Program Trace Message counter has expired indicating that 255 *without-sync* versions of Program Trace Messages have been sent since the last *with-sync* version. The value of 255 is a maximum number, target processors may use a smaller value.

7. A direct branch is detected, the Event-In ($\overline{\text{EVTI}}$) pin has been asserted and the EIT field in the Development Control Register (APPENDIX B) determines that $\overline{\text{EVTI}}$ pin action is to generate program trace synchronization. This message is output by target processors not capable of immediately generating a Program Trace Synchronization Message.

8. Upon overflow of the sequential instruction unit counter. Since a limited counter size must be implemented in the embedded processor, there will likely be sequential instruction sequences (with no taken branches) which will cause the counter to overflow.

9. Upon the occurrence of a watchpoint hit and the next taken direct branch (optional). This trace message follows the watchpoint hit message for target processors not capable of immediately generating a Program Trace Synchronization Message.

Table 6-9 Program Trace, Indirect Branch with Sync Message

Program Trace, Indirect Branch with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	F-ADDR	Variable	The full target address for a taken indirect branch or exception. Most significant bits that have a value of 0 may be truncated.
1	CANCEL	Variable	Number of previous Program Trace Messages that should be ignored by the tool. This packet is generated only by processors performing speculative execution where a trace message may be output before it is known whether the branch was actually taken.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 12

Message Occurrence: This message is output by the target processor when any of the following conditions occurs:

1. Upon exit from system reset. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool. This trace message follows exit from reset for target processors not capable of immediately generating a Program Trace Synchronization Message.

2. An indirect branch is detected after program trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool.

4. An indirect branch is detected following the processor exiting from debug mode.

5. An indirect branch (a change of program flow caused by a subroutine call, return instruction or asynchronous interrupt/trap) is detected and an overrun condition had previously occurred in which one or more branch trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an ECODE value of 00001 or 00111 immediately prior to the Program Trace, Indirect Branch with Sync Message.

6. An indirect branch is detected and the periodic Program Trace Message counter has expired, indicating that 255 *without-sync* versions of Program Trace Messages have been sent since the last *with-sync* version. The value of 255 is a maximum number—target processors may use a smaller value.

7. An indirect branch is detected, a debug control register field specifies that $\overline{\text{EVTI}}$ pin action is to generate program trace synchronization and the Event-In ($\overline{\text{EVTI}}$) pin has been asserted. This message is output by target processors not capable of immediately generating a Program Trace Synchronization Message.

8. Upon overflow of the sequential instruction unit counter. Since a limited counter size must be implemented in the embedded processor, there will likely be sequential instruction sequences (with no taken branches) which will cause the counter to overflow.

9. Upon the occurrence of a watchpoint hit and the next taken indirect branch (optional). This trace message follows the watchpoint hit message for target processors not capable of immediately generating a Program Trace Synchronization Message.

Table 6-10 Program Trace Synchronization Message

Program Trace Synchronization Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	PC	Variable	The full current instruction address. Most significant bits that have a value of 0 may be truncated.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 9

Message Occurrence: This message is output by the target processor when any of the following conditions occurs:

1. Upon exit from reset. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool.

2. When program trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This is required to allow the *number of instruction units executed* packet in a subsequent Program Trace Message to be correctly interpreted by the tool.

4. Upon exiting from debug mode.

5. An overrun condition had previously occurred in which one or more branch trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an ECODE value of 00001 or 00111 immediately prior to the Program Trace Synchronization Message.

6. The periodic Program Trace Message counter has expired indicating that 255 *without-sync* versions of Program Trace Messages have been sent since the last *with-sync* version. The value of 255 is a maximum number—target processors may use a smaller value.

7. A debug control register field specifies that $\overline{\text{EVTI}}$ pin action is to generate program trace synchronization, and the Event-In ($\overline{\text{EVTI}}$) pin has been asserted.

8. Upon overflow of the sequential instruction unit counter. Since a limited counter size must be implemented in the embedded processor, there will likely be sequential instruction sequences (with no taken branches) which will cause the counter to overflow.

9. Upon the occurrence of a watchpoint hit (optional). This trace message immediately follows the watchpoint hit message for target processors capable of immediately generating a Program Trace Synchronization Message. The program counter (PC) value included is the value of the PC at the time of the watchpoint hit.

Table 6-11 Program Trace Correction Message

Program Trace Correction Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	ADJUST	Variable	A number correcting the number of instruction units executed since the last taken branch. This number (unsigned) should be subtracted by the tool from the last I-CNT packet transmitted.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 10

Message Occurrence: This message is output by the target processor when it determines after a Program Trace Message has been sent, that the value in the *number of instruction units executed* packet is incorrect.

Note that if ADJUST = 1, this indicates that the last taken branch was actually cancelled. Consequently, in the next I-CNT packet transmitted, the taken branch that was cancelled will be counted as part of the sequential instruction units.

6.4.5 Data Trace

There are four different Public Messages associated with data tracing:

- Data Trace, Data Write
- Data Trace, Data Write with Sync
- Data Trace, Data Read
- Data Trace, Data Read with Sync

Table 6-12 Data Trace, Data Write Message

Data Trace, Data Write Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	The data value written.
1	U-ADDR	Variable	The unique portion of the data write address, which is relative to the previous Data Trace Message (read or write).
0	MAP	Device-specific	A number to indicate the memory map currently in use by the target processor. For targets with only a single memory map, this packet can be omitted.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 5

Message Occurrence: This message is output by the target processor when it detects a memory write that matches the debug logic's data trace attributes.

Table 6-13 Data Trace, Data Write with Sync Message

Data Trace, Data Write with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	The data value written.
1	F-ADDR	Variable	The full address of the memory location written. Most significant bits that have a value of 0 may be truncated.
1	CANCEL	Variable	Number of previous Data Trace Messages that should be ignored by the tool. This packet is generated only by processors performing speculative execution where a trace message may be output before it is known whether the data write actually occurred.
0	MAP	Device-specific	A number to indicate the memory map currently in use by the target processor. For targets with only a single memory map, this packet can be omitted.

Table 6-13 Data Trace, Data Write with Sync Message (Continued)

Data Trace, Data Write with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 13

Message Occurrence: This message is an alternative to the Data Trace, Data Write Message. It is output instead of a Data Trace, Data Write Message whenever a memory write occurs that matches the debug logic's data trace attributes, and when one of the following conditions has occurred:

1. The processor has exited from reset. This synchronization message is required to allow the *unique portion of the data write address* of following Data Trace, Data Write Messages to be correctly interpreted by the tool.

2. When data trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This synchronization message is required to allow the *unique portion of the data write address* of following Data Trace, Data Write Messages to be correctly interpreted by the tool.

4. The Event-In pin has been asserted and a debug control register field specifies that $\overline{\text{EVTI}}$ pin action is to generate data trace synchronization.

5. An overrun condition had previously occurred in which one or more data trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an ECODE value of 00010 or 00111 immediately prior to the Data Trace, Data Write with Sync Message.

6. The periodic Data Trace Message counter has expired indicating that 255 *without-sync* versions of Data Trace Messages have been sent since the last *with-sync* version. The value of 255 is a maximum number—target processors may use a smaller value.

7. A data write is detected following the processor exiting from debug mode.

Table 6-14 Data Trace, Data Read Message

Data Trace, Data Read Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	The data value read.
1	U-ADDR	Variable	The unique portion of the data read address, which is relative to the previous Data Trace Message (read or write).
0	MAP	Device-specific	A number to indicate the memory map currently in use by the target processor. For targets with only a single memory map, this packet can be omitted.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 6

Message Occurrence: This message is output by the target processor when it detects a memory read that matches the debug logic's data trace attributes.

Table 6-15 Data Trace, Data Read with Sync Message

Data Trace, Data Read with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	The data value read.
1	F-ADDR	Variable	The full address of the memory location read. Most significant bits that have a value of 0 may be truncated.
1	CANCEL	Variable	Number of previous Data Trace Messages that should be ignored by the tool. This packet is generated only by processors performing speculative execution where a trace message may be output before it is known whether the data read actually occurred.
0	MAP	Device-specific	A number to indicate the memory map currently in use by the target processor. For targets with only a single memory map, this packet can be omitted.

Table 6-15 Data Trace, Data Read with Sync Message (Continued)

Data Trace, Data Read with Sync Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 6

Message Occurrence: This message is an alternative to the Data Trace, Data Read Message. It is output instead of a Data Trace, Data Read Message whenever a memory read occurs that matches the debug logic's data trace attributes, and when one of the following conditions has occurred:

1. The processor has exited from reset. This synchronization message is required to allow the *unique portion of the data write address* of following Data Trace, Data Read Messages to be correctly interpreted by the tool.

2. When data trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This synchronization message is required to allow the *unique portion of the data write address* of following Data Trace, Data Read Messages to be correctly interpreted by the tool.

4. The Event-In pin has been asserted and a debug control register field specifies that $\overline{\text{EVTI}}$ pin action is to generate data trace synchronization.

5. An overrun condition had previously occurred in which one or more data trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an ECODE value of 00010 or 00111 immediately prior to the Data Trace, Data Read with Sync Message.

6. The periodic Data Trace Message counter has expired indicating that 255 *without-sync* versions of Data Trace Messages have been sent since the last *with-sync* version. The value of 255 is a maximum number—target processors may use a smaller value.

7. A data read is detected following the processor exiting from debug mode.

6.4.6 Data Acquisition

Table 6-16 Data Acquisition Message

Data Acquisition Message			Direction: from target
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DQDATA	Variable	One or more packets of data values.
1	IDTAG	Device-specific	Data ID tag. This specifies which group of data is included in the Data Acquisition Message.
6	TCODE	Fixed	Value = 7

Message Occurrence: This message is sent by a target when the target processor writes the value of 0x0 to the Data Acquisition Control Register.

6.4.7 Error

Table 6-17 Error Message

Error Message			Direction: from target
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
5	ECODE	Fixed	Error code. Refer to **Table 6-15**.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 8

Table 6-18 Error Codes

Error Code	Description
00000	Ownership trace overrun.
00001	Program trace overrun.
00010	Data trace overrun.
00011	Read/write access error (read or write error to user memory map). This error code applies only to targets that support the Read/Write Access Messages for NRRs (APPENDIX B).[1]
00100	Invalid message (message type not implemented). The Error Message is sent by the target as soon as the invalid message is detected.

Table 6-18 Error Codes (Continued)

Error Code	Description
00101	Invalid access opcode (NRR not implemented). This error code applies only to targets that support the Read/Write Access Messages for NRRs (APPENDIX B). The Error Message is sent by the target as soon as the invalid opcode is detected.
00110	Watchpoint overrun.
00111	Program and/or data and/or ownership trace overrun.
01000	Program trace and/or data trace and/or ownership trace and/or watchpoint overrun.
01001–10111	Reserved.
11000–11111	Vendor defined.

1. For targets that implement device-specific debug control and status registers and use Public Messages 22–26 to provide read/write access, an error condition is indicated by the ST field in the Target/Tool Response Message.

Overrun Errors: For any of the trace overrun categories, an Error Message, containing an overrun error code, is to inform the tool that the target has discarded trace occurrences because of insufficient space in its trace output queue. The Error Message is sent immediately prior to a synchronization message (OTM, BTM + Sync, DTM + Sync) as soon as space is available in the trace output queue.

6.4.8 Watchpoint Hit

Table 6-19 Watchpoint Message

Watchpoint Message			Direction: from target
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
2	WPHIT	Device-specific	This is an 8-bit field in which each bit position corresponds to a different watchpoint number. Bit positions 0 through 7 correspond to watchpoints 0 through 7. A “1” in a bit position indicates a watchpoint hit occurred.
0	SRC	Device-specific	Client that is source of message. For targets with only a single client, this packet can be omitted.
6	TCODE	Fixed	Value = 15

Message Occurrence: This message is sent by the target whenever a watchpoint hit occurs. Multiple watchpoint hits can be indicated in the same message. The debug logic in the target must ensure that Watchpoint Messages can never be

cancelled once they have been generated. If watchpoint hit occurrences are discarded because of insufficient space in the trace output queue, the target must send the tool an Error Message prior to the next Watchpoint Message actually sent so that the tool knows that one or more watchpoint hit occurrences were discarded.

6.4.9 Port Replacement

Table 6-20 Port Replacement—Output Message

Port Replacement—Output Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
16	OUT	Fixed	Each bit corresponds to one of the 16 low-speed I/O pins involved with port replacement. When the direction of the pin is an output, the pin state corresponds to the value of the bit in this packet. Pins defined as inputs are unaffected by corresponding bits in this packet.
16	DIR	Fixed	Each bit specifies the direction of one of the 16 low-speed I/O pins involved with port replacement. 0 = input 1 = output
6	TCODE	Fixed	Value = 20

Message Occurrence: This message is sent by the target to set up external port replacement logic on the target system. For low-speed I/O port bits defined as outputs, this message is also used to set the state of the pins.

Table 6-21 Port Replacement—Input Message

Port Replacement—Input Message			**Direction: from tool**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
16	IN	Fixed	Each bit corresponds to one of the 16 low-speed I/O pins involved with port replacement. When the direction of the pin is an input, this message is used to read the pin state.
6	TCODE	Fixed	Value = 21

Message Occurrence: This message is sent by the tool upon the occurrence of a change in the state of one or more input pins.

6.4.10 Read/Write Access of Nexus Recommended Development Registers

For target processors that implement the Nexus recommended development registers described in APPENDIX B, the following four messages are used by a tool to access debug control and status registers via the AUX. These four messages will not normally be used by target processors that implement device-specific registers.

Table 6-22 Target Ready Message

Target Ready Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
6	TCODE	Fixed	Value = 16

Message Occurrence: The target sends this message when it has completed the actions that were specified in a previous Read/Write Access Message from the tool and is able to accept another command. The tool sends this message when it is able to accept another message from the target as part of a block read sequence.

Table 6-23 Read Register Message

Read Register Message			**Direction: from tool**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	OPCODE	Fixed	Refer to APPENDIX B for a list of all opcodes that the tool can use to request the target to return specific status or data.
6	TCODE	Fixed	Value = 17

Message Occurrence: The tool sends this message when it wants to read control or status information from the target. The target responds to this message with a Read/Write Response Message containing the requested information.

Table 6-24 Write Register Message

Write Register Message			**Direction: from tool**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	REGVAL	Variable	Depending on the opcode selected, the message may include one or more fixed-length packets of information downloaded to the target.
8	OPCODE	Fixed	Refer to APPENDIX B for a list of all opcodes that the tool can use to send debug control commands and data to the target.
6	TCODE	Fixed	Value = 18

Message Occurrence: The tool uses this message to control debug resources within the target and to send data to the target. When the target has processed the message and is able to accept another Read/Write Access Message from the tool, it responds with a Target Ready Message.

Table 6-25 Read/Write Response Message

Read/Write Response Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	REGVAL	Variable	Depending on the opcode selected, the message includes one or more fixed-length packets of information.
6	TCODE	Fixed	Value = 19

Message Occurrence: This message is sent in the following circumstances:

- By a target in response to a Read Register Message issued by the tool. The information packets included in the message depend on the opcode in the original Read Register Message.
- By a target in response to a Write Register (with RWA opcode, RW field = Read) Message from the tool. The UDI Message contains a single data word of the size specified in the Write Register Message.

- By a target in response to each Target Ready Message from the tool which follows a block read command issued by the tool. The message contains a single data word of the size specified in the Write Register Message. The target continues to send messages until all data originally specified by the Access Count field has been transferred. The tool can prematurely terminate the data transfer sequence by responding to a message with a Write Register (with RWA opcode, RW field = Read, SC = 0) Message instead of with a Target Ready Message.

- By a tool in response to each Target Ready Message issued by the target, which follows a block write command issued by the tool. The message contains a single data word of the size specified in the Write Register Message. The tool continues to send messages until all data originally specified by the Access Count field has been transferred. The tool can prematurely terminate the data transfer sequence by sending a Write Register (with RWA opcode, RW field = Write, SC = 0) Message instead of with another UDI Message.

6.4.10.1 Read/Write Access of Nexus Recommended Registers (NRRs)—Protocol Examples

The examples in **Figure 6-2**, **Figure 6-3**, **Figure 6-4** and **Figure 6-5** use the NRRs concatenated as defined in B.10 Nexus Recommended Registers (NRRs) Concatenated for Better Transfer Efficiency on Page 144.

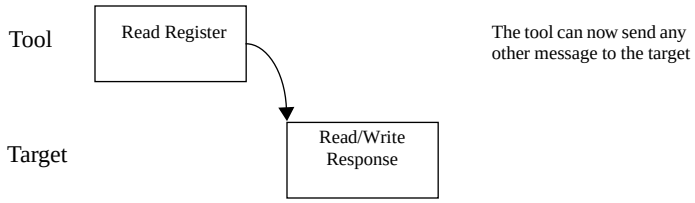


Figure 6-2 Tool Requesting Target Debug Status

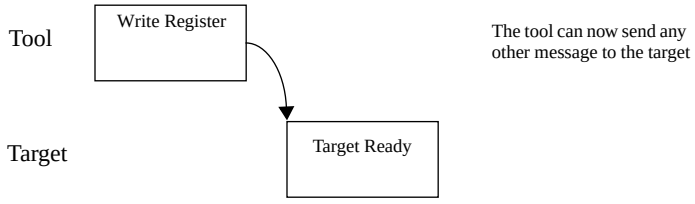


Figure 6-3 Tool Sending a Debug Command to the Target

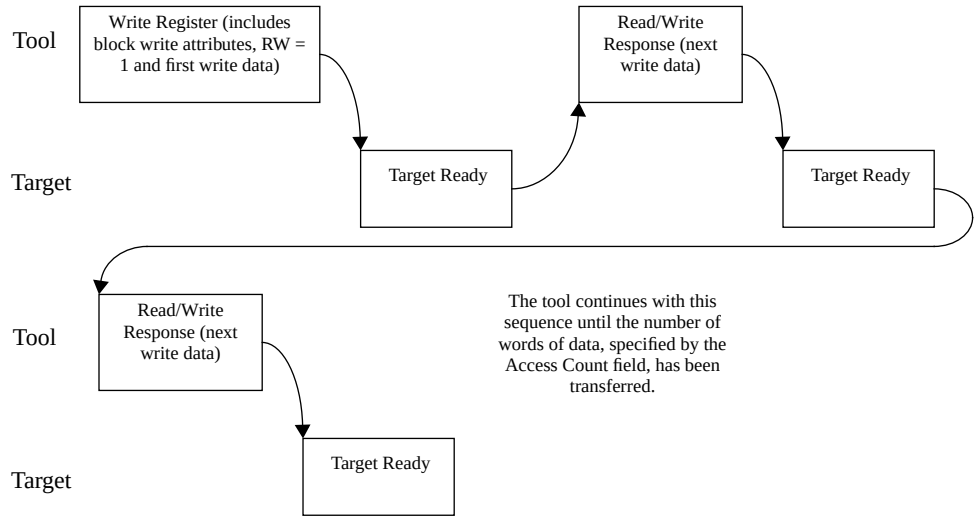
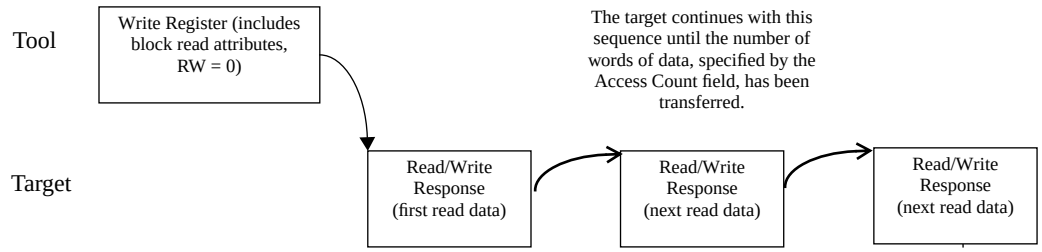


Figure 6-4 Block Write Command Issued by Tool



Note that the tool should be able to keep up with the transfer rate of the target for block reads. If the tool does not contain this capability, then block reads should not be requested.

Figure 6-5 Block Read Command Issued by Tool

6.4.11 Read/Write Access of Memory-Mapped Locations and Memory Substitution

The following five messages define the protocol which allows a tool to read or write memory-mapped locations in the target processor's internal address space and a target to read or write memory locations in the tool. This type of read/write access is used in the following circumstances:

- By a tool that implements device-specific development control and status registers (instead of the NRRs as defined in APPENDIX B).
- By a target that implements memory substitution in which code and/or data that is normally fetched from target memory is instead fetched from tool memory. This function is started by a target processor watchpoint hit and is ended by the tool.

- By a target that allocates a fixed portion of its internal memory map to provide AUX access. Processor reads or writes to this address range result in Read/Write Messages being issued to access memory that exists within the tool. As an example, a target's debug exception handler program may exist only within the tool and be fetched from the tool each time a debug exception occurs.

Table 6-26 Read Target/Tool Message

Read Target/Tool Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
1	ADDRESS	Variable	Device-specific field specifying the location to be read. The size of the address must match the address space supported by the target or tool.
3	DSZ	Fixed	Data Size 000 = 8-bit 001 = 16-bit 010 = 32-bit 011 = 64-bit 100–101 = Vendor defined 110–111 = Reserved for future data sizes Note: A target/tool does not need to support all of the above data sizes.
0	MAP	Device-specific	A number to indicate the memory map to be used in the target/tool. For targets or tools with only a single memory map, this packet can be omitted.
6	TCODE	Fixed	Value = 22

Message Occurrence: This message is output by a tool when it wants to read a location in the target's memory-mapped address space and by a target when it wants to read a location in the tool's memory space.

Table 6-27 Write Target/Tool Message

Write Target/Tool Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	Write data value of size DSZ.
1	ADDRESS	Variable	Device-specific field specifying the location to be written. The size of the address must match the address space supported by the target or tool.

Table 6-27 Write Target/Tool Message (Continued)

Write Target/Tool Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
3	DSZ	Fixed	Data Size 000 = 8-bit 001 = 16-bit 010 = 32-bit 011 = 64-bit 100–101 = Vendor defined 110–111 = Reserved for future data sizes Note: A target/tool does not need to support all of the above data sizes.
0	MAP	Device-specific	A number to indicate the memory map to be used in the target/tool. For targets or tools with only a single memory map, this packet can be omitted.
6	TCODE	Fixed	Value = 23

Message Occurrence: This message is output by a tool when it wants to write to a location in the target's memory-mapped address space, and by a target when it wants to write to a location in the tool's memory space.

Table 6-28 Read Next Target/Tool Data Message

Read Next Target/Tool Data Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
6	TCODE	Fixed	Value = 24

Message Occurrence: Read Next Target/Tool Data provides the most efficient method of reading consecutively-addressed data. Both tool and target are required to increment their internal address pointers according to the size of the data being transferred.

The tool sends this message when it has processed a prior Target Response Message and the tool's receive buffer can accommodate more read data. The target sends this message when it has processed a prior Tool Response Message and the target's receive buffer can accommodate more read data.

There is no limit to the amount of data that can be transferred using consecutive Read Next Target/Tool Data commands.

Table 6-29 Write Next Target/Tool Data Message

Write Next Target/Tool Data Message			**Direction: from tool, from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	Write data value of size determined by the DSZ packet in the most recent Write Target or Write Tool Message.
6	TCODE	Fixed	Value = 25

Message Occurrence: This message is initiated by the tool when it wants to write to the next consecutively-addressed location in the target, and by the target when it wants to write to the next consecutively-addressed location in the tool. Both tool and target are required to increment their internal address pointers according to the size of the data being transferred.

The tool sends this message when it has processed a prior Target Response Message and has more data available to send. The target sends this message when it has processed a prior Tool Response Message and has more data available to send.

There is no limit to the amount of data that can be transferred using consecutive Write Next Target/Tool Data commands.

Table 6-30 Target Response Message

Target Response Message			**Direction: from target**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	Read data, the size of which is determined by the DSZ packet in the most recent Read Target Message. This field does not exist if the previous message issued by the tool was a Write Target Message or if the target is unable to complete the previously requested read operation.
2	ST	Fixed	Status 00 = The previously requested read or write operation is able to be processed normally. 01 = The previously requested read or write operation cannot be completed. 1x = Reserved for future use.
6	TCODE	Fixed	Value = 26

Message Occurrence: This message is output by the target but the contents differ depending on whether the most recent read/write command issued by the tool was contained in a Read Target Message or a Write Target Message.

For read commands, the target sends a Target Response Message (containing data) as soon as it has retrieved the data requested by a previous Read Target Message or Read Next Target Data Message from the tool.

For write commands, the target sends a Target Response Message (with no data) as soon as the target's receive buffer is able to accept more data from the tool.

If the target is unable to process the function requested by the previous Read Target Message, Write Target Message, Read Next Target Data Message or Write Next Target Data Message, it sends a Target Response Message with the ST field = 01.

Table 6-31 Tool Response Message

Tool Response Message			**Direction: from tool**
Minimum Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	DATA	Variable	Read data, the size of which is determined by the DSZ packet in the most recent Read Tool Message. This field does not exist if the previous message issued by the target was a Write Tool Message or if the tool is unable to complete the previously requested read operation.
2	ST	Fixed	Status 00 = The previously requested read or write operation is able to be processed normally. 01 = The previously requested read or write operation cannot be completed. 10 = Memory substitution transfer complete. 11 = Reserved for future use.
6	TCODE	Fixed	Value = 26

Message Occurrence: This message is output by the tool but the contents differ depending on whether the most recent read/write command issued by the target was contained in a Read Tool Message or a Write Tool Message.

For read commands, the tool sends a Target Response Message (containing data) as soon as it has retrieved the data requested by a previous Read Tool Message or Read Next Tool Data Message from the target.

For write commands, the tool sends a Tool Response Message (with no data) as soon as the tool's receive buffer is able to accept more data from the target.

If the tool is unable to process the function requested by the previous Read Tool Message, Write Tool Message, Read Next Tool Data Message or Write Next Tool Data Message, it sends a Tool Response Message with the ST field = 01.

To support memory substitution, in which code that is normally fetched from target memory is instead fetched from tool memory, the tool must be able to inform the target when to stop requesting data, i.e. the tool determines when the substitution process should end. Memory substitution will typically be initiated by a target watchpoint hit and will be terminated by the tool. The tool informs the target not to request more data by setting the ST field = 10 in the response message which contains the final read data.

Read/Write Access of Memory-Mapped Registers—Protocol Examples

The following read/write protocol examples show how the AUX is used by a tool to access target memory space, and by a target to access tool memory space.

Tool Accessing Target: Figure 6-6 through **Figure 6-11** show the sequences of messages sent between a tool and a target when the tool wants to read or write memory-mapped address space in the target. To achieve maximum transfer performance, Read Next Target Data or Write Next Target Data Messages should be used wherever possible since these messages have the shortest length.

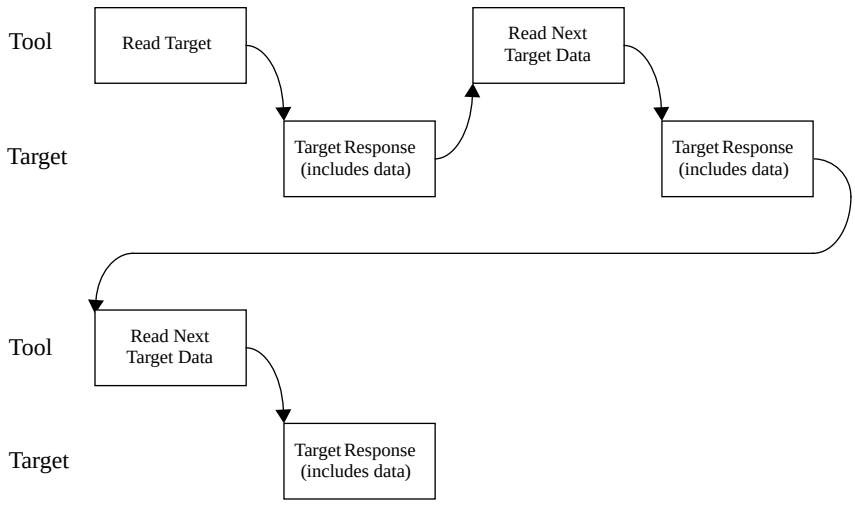


Figure 6-6 Reading Consecutively-Addressed Target Locations

The protocol permits only one outstanding request from the tool. In the above example, a Read Next Target Data command can be issued only after a response is received to the previous tool-initiated command.

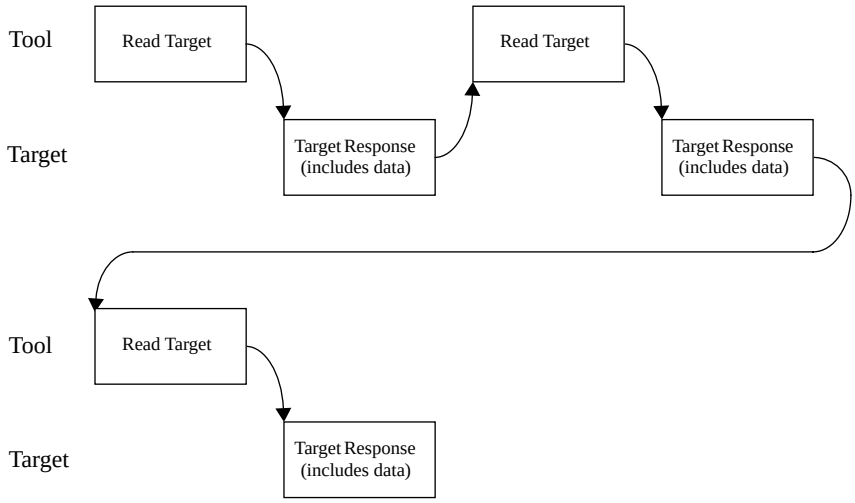


Figure 6-7 Reading Randomly-Addressed Target Locations

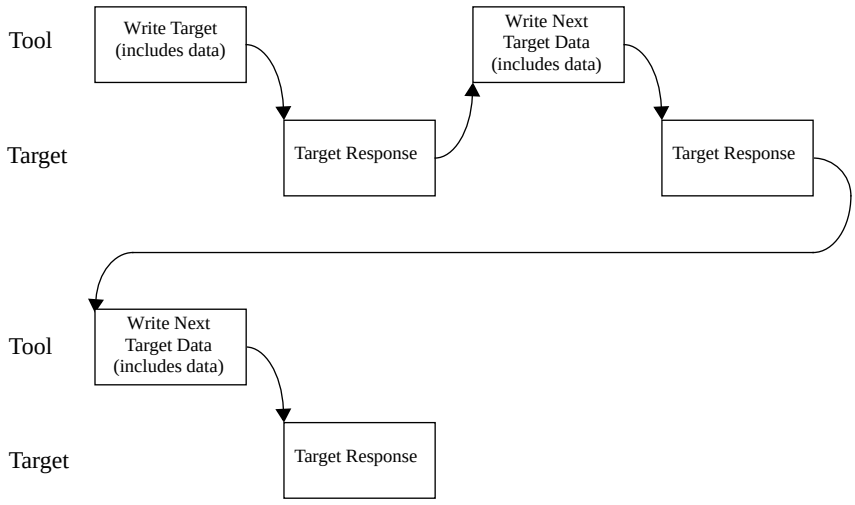


Figure 6-8 Writing Consecutively-Addressed Target Locations

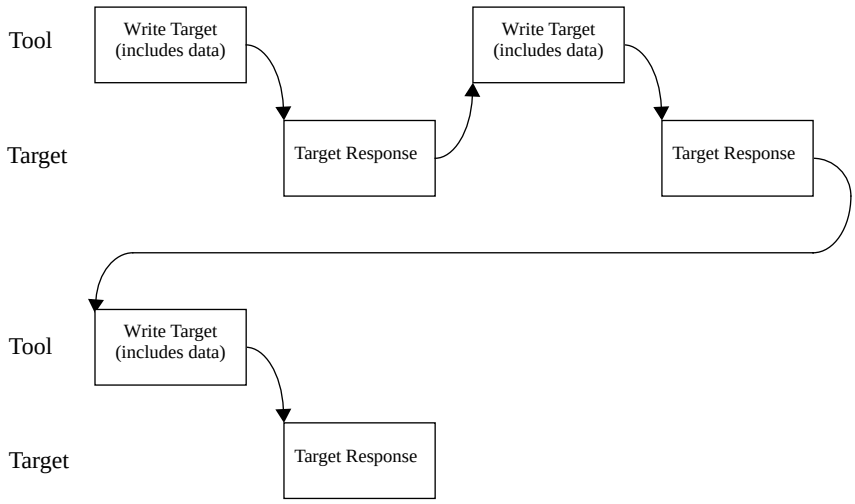


Figure 6-9 Writing Randomly-Addressed Target Locations

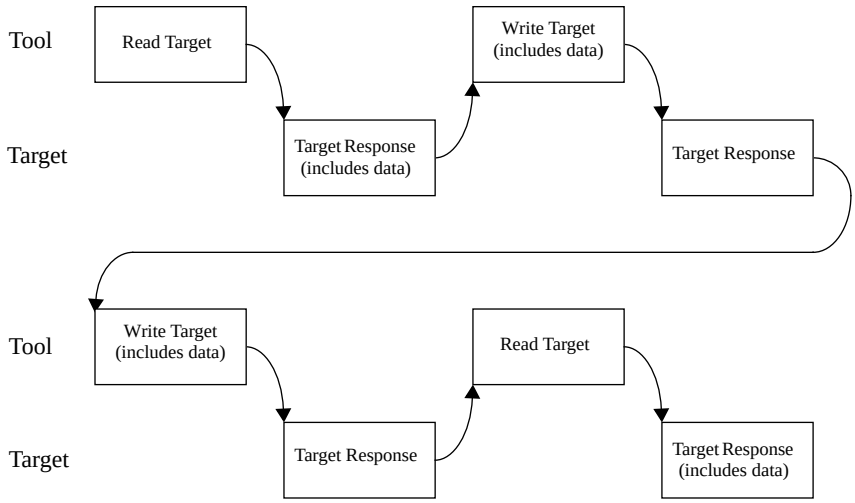


Figure 6-10 Intermixed Reading/Writing Randomly-Addressed Target Locations

In **Figure 6-10**, the Write Target command can be issued only after a response is received to the previous Read Target command.

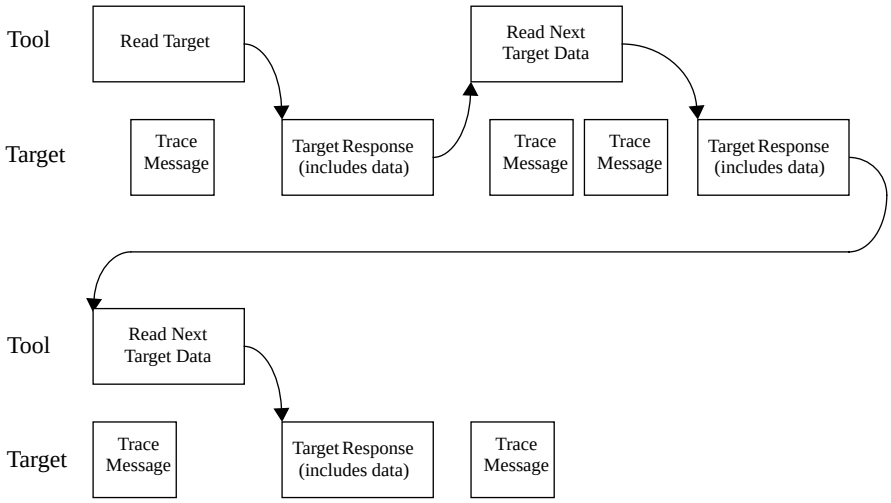


Figure 6-11 Trace Messages Intermixed with Read Target Locations

None of the earlier protocol examples showed trace messages being sent by the target. Trace messages are not acknowledged by the tool and these messages can be output any time the Auxiliary Output Port is not transmitting other messages. This example shows how the read/write protocol and trace messages can co-exist. This example also demonstrates that the AUX is full-duplex, that is, messages can occur in both directions simultaneously.

One point to note about the co-existence of read/write protocol and trace message output is that protocol responses from the target should not pass through the same output queue as trace messages. As soon as a protocol response has been prepared by the target, it must be transmitted at the first opportunity, that is, immediately following any trace message currently being transmitted, regardless of the number of other trace messages queued for output.

Termination: When large blocks of data are being read, the tool requests the next data word (when its buffer is able to accept more data) by issuing a Read Next Target Data Message.

A similar process occurs for block writes. The tool send the next data word after it receives acknowledgment from the target that the target is ready to accept more data.

The tool is always in control of the transfer process. The target has no prior knowledge of the amount of data to be transferred; the tool just stops the process by not sending any more Read Next Target Data or Write Next Target Data Messages. The target simply increments an address counter each time it receives a Read Next Target Data or Write Next Target Data Message. This address counter is automatically changed to a new value whenever another Read Target or Write Target Message is received.

During the transfer of a large block of data using Read Next Target Data or Write Next Target Data Message, the target may determine that it is unable to continue supplying read data or accepting write data. This could happen if the incrementing address points to non-existent memory or to a protected memory area. Upon detecting such a condition, the target issues a Target Response Message with the status (ST) field = 01 (meaning that the previously requested read or write operation cannot be completed). The tool stops sending any more Read Next Target Data or Write Next Target Data Messages and may then perform some recovery or error notification tasks.

Target Accessing Tool: A target uses the same protocol described in Section 6.4.11 Read/Write Access of Memory-Mapped Locations and Memory Substitution on Page 75 to access memory space within a tool, except that all messages occur in the reverse direction. To understand the protocol, simply swap the Tool and Target names on all the protocol diagrams in **Figure 6-6** through **Figure 6-10**.

Read Tool and Write Tool Messages include an optional Map packet for use in situations where multiple memory maps are supported by the tool. These alternative memory maps may be used to select different address spaces within the tool such as:

- Target boot image
- Debug exception handler
- Read/write data space

To support memory substitution, in which code that is normally fetched from target memory is instead fetched from tool memory, the tool must be able to inform the target when to stop requesting data, i.e. the tool determines when the substitution process should end. Memory substitution will typically be initiated by a target watchpoint hit and will be terminated by the tool. The tool informs the target not to request more data by setting the ST field = 10 in the response message that contains the final read data.

Read/write access can occur in both directions simultaneously. For example, a tool may initiate read/write access to the target without affecting any target-to-tool transfer of data currently in progress. This means that both the tool and the target must have sufficient receive buffer space to support two messages, a request from the other device and the response to a request.

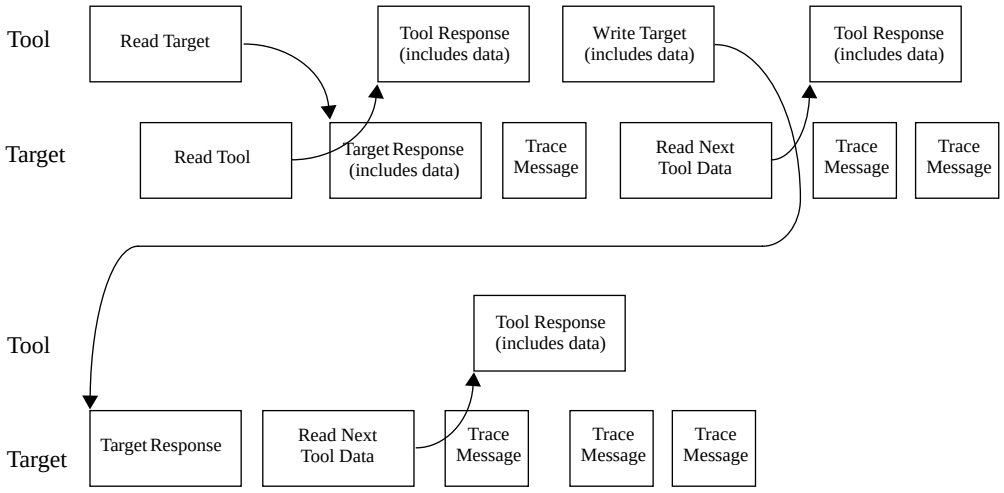


Figure 6-12 Simultaneous R/W Accesses by Tool and Target

6.4.12 Memory Substitution

Read/Write Access Messages are used to emulate a bus where instructions and data may be accessed via the AUX.

Table 6-32 Read/Write Access Messages

Public Message	Examples Events That Cause Message Generation
Read Tool	The target sends this message to read any memory location in the tool.
Write Tool	The target sends this message to write to any memory location in the tool.
Read Next Tool Data	The target sends this message to the tool as part of a block read function.
Write Next Tool Data	The target sends this message to the tool as part of a block write function.
Tool Response	1. The tool send this message, containing data, in response to a Read Tool command or a Read Next Tool Data command. The message indicates to the target that the tool can accept another Memory Substitution Message. The tool can terminate the memory substitution block read access by including an ST = 10. 2. The tool sends this message in response to a Write Tool command or a Write Next Tool Data command. The message indicates to the target that the tool can accept more data.

SECTION 7
Auxiliary Port Signals

Embedded processors complying to class 2, 3 or 4 shall provide the appropriate pin functions as shown in **Table 7-1**. Required and optional pin functions are designated by “R” and “O” respectively. Pins not allowed for an interface are shaded.

Table 7-1 Auxiliary Pin Functions Required per Interface Type

Pin Type	Direction	Full-duplex (Auxiliary In/Out)	Full-duplex w/IEEE 1149.1	Half-duplex w/IEEE 1149.1
MCKI	In	R		
MDI	In	R		
$\overline{\text{MSEI}}$	In	R		
MCKO	Out	R	R	
MDO	Out	R	R	
$\overline{\text{MSEO}}$	Out	R	R	
$\overline{\text{EVTI}}$	In	R	R	O
$\overline{\text{RSTI}}$	In	R		
$\overline{\text{EVTO}}$	Out	O	O	O

Note that the MCKO functions may be provided via a system clockout pin on the embedded processor.

7.1 Pin Functions

The auxiliary pin functions are described in **Table 7-2**.

Table 7-2 Auxiliary Pins

Auxiliary Pins	Description of Auxiliary Pins
MCKO	Message Clockout (MCKO) is a free-running output clock to development tools for timing of MDO and $\overline{\text{MSEO}}$ pin functions. MCKO can be independent of embedded processor system clock (CLOCKOUT). An embedded processor CLOCKOUT pin may be used as a functional equivalent for MCKO.
MDO[M:0]	Message Data Out (MDO[M:0]) are output pin(s) used for OTM, BTM, DTM, reads, memory substitution accesses, etc. External latching of MDO shall occur on rising edge of MCKO (or system clock). Depending upon bandwidth requirements, 1, 2, 4, 8 or more pins may be implemented.
$\overline{\text{MSEO}}$[1:0]	Message Start/End Out ($\overline{\text{MSEO}}$ [1:0]) are output pins that indicate when a message on the MDO pins has started, when a variable-length packet has ended and when the message has ended. Only 1 $\overline{\text{MSEO}}$ pin is required but up to 2 pins may be implemented for more efficient transfers. External latching of $\overline{\text{MSEO}}$ shall occur on the rising edge of MCKO (or system clock).
MCKI	Message Clockin (MCKI) is a free-running input clock from development tools for timing of MDI and $\overline{\text{MSEI}}$ pin functions. MCKI can be independent of the embedded processor system clock.
MDI[N:0]	Message Data In (MDI[N:0]) are input pin(s) used for downloading configuration information, writes to user resources, etc. Internal latching of MDI shall occur on the rising edge of MCKI. Depending upon bandwidth requirements, 1, 2, 4, 8 or more pins may be implemented.
$\overline{\text{MSEI}}$[1:0]	Message Start/End In ($\overline{\text{MSEI}}$ [1:0]) are input pins that indicate when a message on the MDI pins has started, when a variable-length packet has ended and when the message has ended. Only 1 $\overline{\text{MSEI}}$ pin is required but up to 2 pins may be implemented for more efficient transfers. Internal latching of $\overline{\text{MSEI}}$ shall occur on the rising edge of MCKI.
$\overline{\text{EVTI}}$	Event In ($\overline{\text{EVTI}}$) is an input where, when a high-to-low transition occurs, a processor is halted (breakpoint) or Program and Data Synchronization Messages are transmitted from the embedded processor.
$\overline{\text{RSTI}}$	Reset In ($\overline{\text{RSTI}}$) is for resetting the Nexus port resources.
$\overline{\text{EVTO}}$	Event Out ($\overline{\text{EVTO}}$) is an optional output pin to development tools comprising exact timing for a single breakpoint status indication. Upon a breakpoint occurrence of the programmed breakpoint source, $\overline{\text{EVTO}}$ is asserted for a minimum of 1 clock period of MCKO.

Example Ports: For a full-duplex AUX with IEEE 1149.1 pins, a minimum of 3 auxiliary pins are required for compliance, i.e. MDO, MSEO and $\overline{\text{EVTI}}$, assuming a system clockout pin can be used for MCKO. The performance classification, however, would also be minimal, and may only meet the transfer bandwidth requirements for low-end applications or for lower compliance classifications. The Nexus standard allows for additional transfer bandwidth with a scalable pin interface or transfer rate, as illustrated by the examples in **Table 7-3**.

Table 7-3 Example of Auxiliary Output Ports

Number of Pins for each Example					Comments
MDO	**$\overline{\text{MSEO}}$**	**MCKO**	**$\overline{\text{EVTI}}$**	**Total Pins**	
1	1	0	1	3	Base implementation.
2	1			4	2X faster than base implementation.
4	1			6	4X faster than base implementation.
4	2			7	1 clock faster per transfer .
8	2			11	> 8X faster than base implementation.
1	1	1		4	Independent clock allows for faster or slower transfer rate than with system clock reference.
2	1			5	
4	1			7	
4	2			8	
8	2			12	

Multi-Processor: The Nexus standard allows for embedded processor implementations that comprise multiple clients to utilize a single AUX, depending upon the transfer bandwidth requirement for the application. The AUX may be designated for a single client or shared by multiple clients on the embedded device during runtime. Messages transmitted via the AUX shall contain information defined by the Nexus standard indicating which client generated the message.

Single Master for Tool Connection: The Nexus standard does not support multiple tools connected directly to the Nexus input port. That is, arbitration for multiple external tools is not supported by the port. To connect multiple tools, the tools should either manage the arbitration, or a single low-level tool should be connected with multiple high-level tools interconnected and arbitrated by the single low-level tool.

Reset Configuration: Embedded processors complying to class 2, 3 or 4 shall receive reset configuration information via $\overline{\text{EVTI}}$, according to the Nexus standard, to completely enable/disable message transmission on the auxiliary output port. If message transmission is enabled, output messages shall be transmitted normally. If message transmission is disabled, auxiliary output pins shall be 3-stated and no messages shall be transmitted. One exception allowed is the AUX clockout pin. If the system clock is used as the MCKO function, then it is not required to 3-state the system clock via $\overline{\text{EVTI}}$ reset configuration.

Embedded processors complying to any compliance class and implementing LSIO port replacement shall receive reset configuration information via $\overline{\text{EVTI}}$, according to the Nexus standard, to enable/disable message transmission on the auxiliary output port. If message transmission is enabled, output messages will be transmitted normally with support for Port Replacement Messages according to the Nexus standard. If message transmission is disabled, auxiliary output pins shall provide device-specific LSIO capability.

Reset configuration information must be valid on $\overline{\text{EVTI}}$ for at least 4 system clocks of the embedded processor prior to negation of $\overline{\text{RSTI}}$ or $\overline{\text{TRST}}$ (IEEE 1149.1 reset pin) or during the IEEE 1149.1 Test Logic Reset state.

The $\overline{\text{EVTI}}$ pin shall comprise a pull-up resistor with the following reset configuration states.

Reset State	Description
0	Message transmission enabled.
1	Message transmission disabled (default).

Security: Device-specific enable/disable mechanisms internal to the embedded processor may be optionally provided for secure visibility of user resources on the embedded processor.

Port Replacement Support for MCUs: Port replacement support for MCUs containing LSIO is an option provided in the Nexus standard for LSIO that can tolerate the inherent latency of the port replacement mechanism. Up to16 bits of LSIO port replacement are allowed with the standard Port Replacement Messages transmitted via the AUX, as shown in **Figure 7-1**. The standard messages transmitted between the development tool and embedded processor provide the necessary information for the development tool to replace the LSIO port (with additional delay).

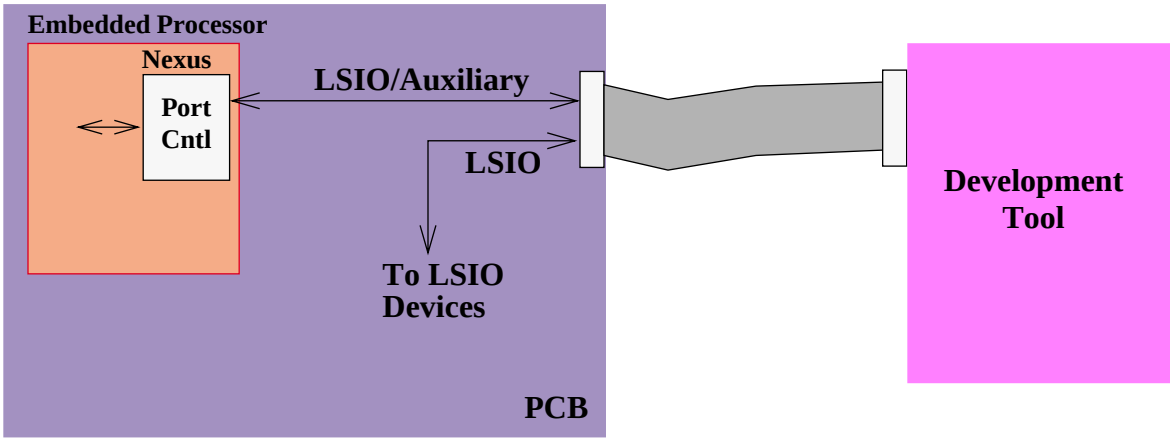


Figure 7-1 Port Replacement for MCUs containing LSIO pins

Most messages transmitted in a typical application will comprise development information. Upon occurrence of an LSIO state change, however, a Port Replacement Message will be transmitted. Port replacement messages will be transmitted either by the embedded processor to the tool (messages containing information for low-speed output pins) or by the tool to the embedded processor (messages containing information for low-speed input pins). Port Replacement Messages from the embedded processor will contain two essential packets: one packet indicating the direction of each LSIO and another indicating the state of all LSIO. Port Replacement Messages from the tool will contain one essential packet indicating the state of all LSIO. This information will be used by the embedded processor and tool to maintain the correct state and direction for all LSIO.

Note that if the development tool is not connected to perform the port replacement function, a special connector should be connected so that the LSIO signals are connected to the LSIO devices. For production boards that do not require the port replacement function, no connector is required if these signals are connected directly by board traces.

The development tool shall implement the following rules to assure proper port replacement:

- Prior to receiving the first Port Replacement Message after the embedded processor port has been reset, all replacement pins on the tool should default to input.
- Prior to receiving the first Port Replacement Message after the embedded processor port has been reset, the tool should not generate Port Replacement Messages to the embedded processor.
- When the processor writes to the LSIO port registers, a Port Replacement Message will be transmitted to the tool. The tool then drives the pins configured as outputs to their programmed states.
- Whenever any pin configured as an input changes, the tool transmits a Port Replacement Message to the embedded processor for update of the state internally (enabled interrupt may be generated).

Port Replacement Support for MPUs: Port replacement support for MPUs is an option provided in the Nexus standard. Since MPUs do not contain LSIO pins, a similar and yet slightly different technique is provided for simultaneously using both a primary pin function, such as a HSIO external bus port, and a secondary pin function, such as Nexus development pins. Due to the high-speed nature of the HSIO external bus port, only the data out portion of the Nexus port can be simultaneously shared with the primary function. Since the Nexus data out signals comprise the most stringent bandwidth requirements, however, this solution still provides a tremendous advantage in reducing the total number of actual development support pins. Refer to **Figure 7-2** for an illustration.

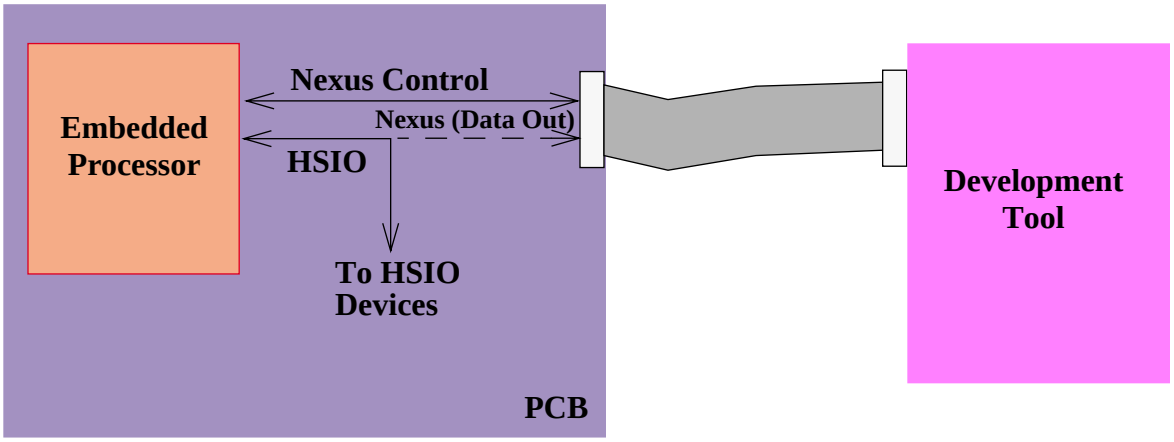


Figure 7-2 Port Replacement for MPUs

Most messages transmitted in a typical application will comprise information related to the primary function of the HSIO external port. During HSIO information transfer (e.g. external bus cycle), the Nexus control signals are negated and the development tool ignores the HSIO information. Upon the occurrence of a condition that generates Nexus output information (e.g. Data Read Message and Direct Branch Message), a corresponding Nexus message is transmitted out the port and captured by the tool. When a Nexus message is transmitted, the HSIO control signals (e.g. bus control pins) should remain negated.

SECTION 8
Auxiliary Port Message Protocol

The protocol for the embedded processor receiving and transmitting messages via the auxiliary pins shall be accomplished with the $\overline{\text{MSEI}}$ and $\overline{\text{MSEO}}$ pin functions respectively. A minimum of 1 and a maximum of 2 $\overline{\text{MSEI}}$ pins shall provide the protocol for the embedded processor receiving messages, and a minimum of 1 and a maximum of 2 $\overline{\text{MSEO}}$ pins shall provide the protocol for the embedded processor transmitting messages.

The $\overline{\text{MSEI/MSEO}}$ protocol comprises the following:

- Two “1”s followed by one “0” indicates start of message
- “0” followed by two or more “1”s indicates end of message
- “0” followed by “1” followed by a “0” indicates end of variable length packet
- “0”s at all other clocks during transmission of a message
- “1”s at all clocks during no message transmission (idle)

The same sequence is followed when using 1 or 2 $\overline{\text{MSEI/MSEO}}$ pins, but when using 2 $\overline{\text{MSEI/MSEO}}$ pins, it is possible for two sequences to occur on the same clock.

$\overline{\text{MSEI/MSEO}}$ is used to signal the end of variable-length packets, and not device-specific or fixed-length packets. $\overline{\text{MSEI/MSEO}}$ are sampled on the rising edge of MCKI/MCKO.

Figure 8-1 illustrates the state diagram for 1-pin $\overline{\text{MSEI/MSEO}}$ transfers. When using only 1 $\overline{\text{MSEI/MSEO}}$ pin, the “End Message” state does not contain valid data on the MDI/MDO pins. Also, it is not possible to have two consecutive “End Packet” Messages. This implies that the minimum packet size for a variable-length packet is two times the number of MDI/MDO pins. This ensures that a false end of message state is not entered by transmitting two consecutive 1’s on the $\overline{\text{MSEI}}$/$\overline{\text{MSEO}}$ pin before the actual end of message.

Figure 8-2 illustrates the use of 2-pin $\overline{\text{MSEO}}$ transfers. The 2-pin $\overline{\text{MSEI/MSEO}}$ option is more robust than the 1-pin option. Termination of the current message may immediately be followed by the start of the next message on the consecutive clocks. An extra clock to end the message is not necessary as with the 1-pin $\overline{\text{MSEI/MSEO}}$ option. The 2-pin option also allows for consecutive “End Packet” states. This can be an advantage when small, variable-sized packets are transferred.

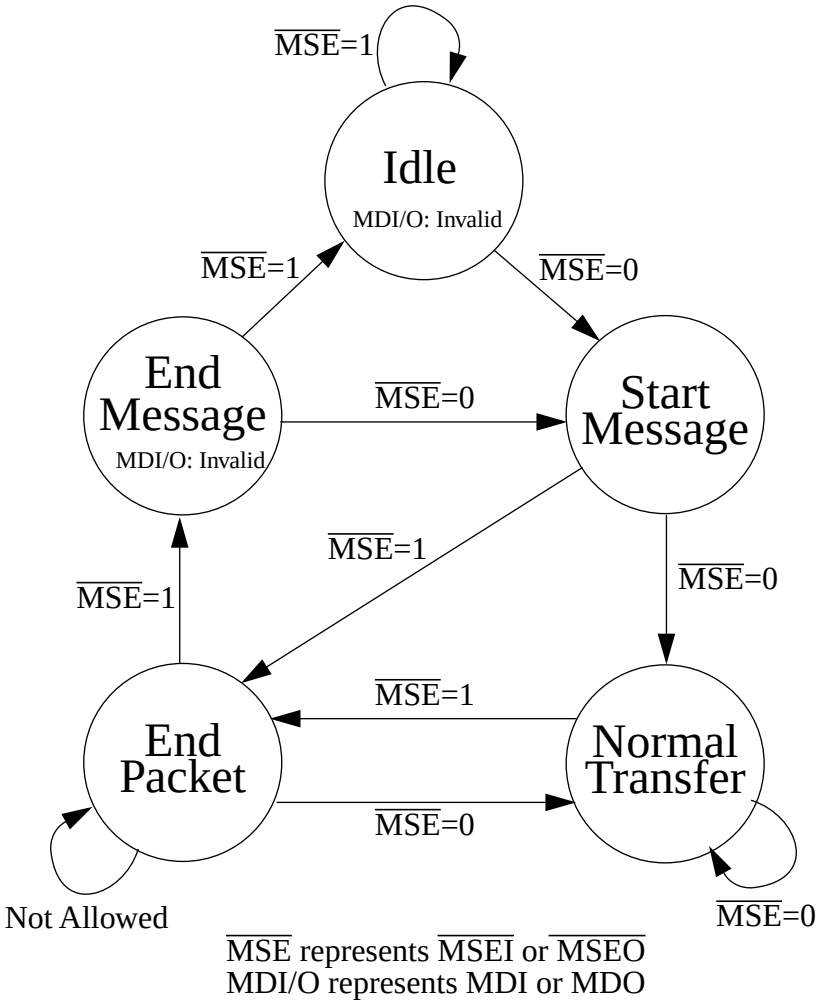
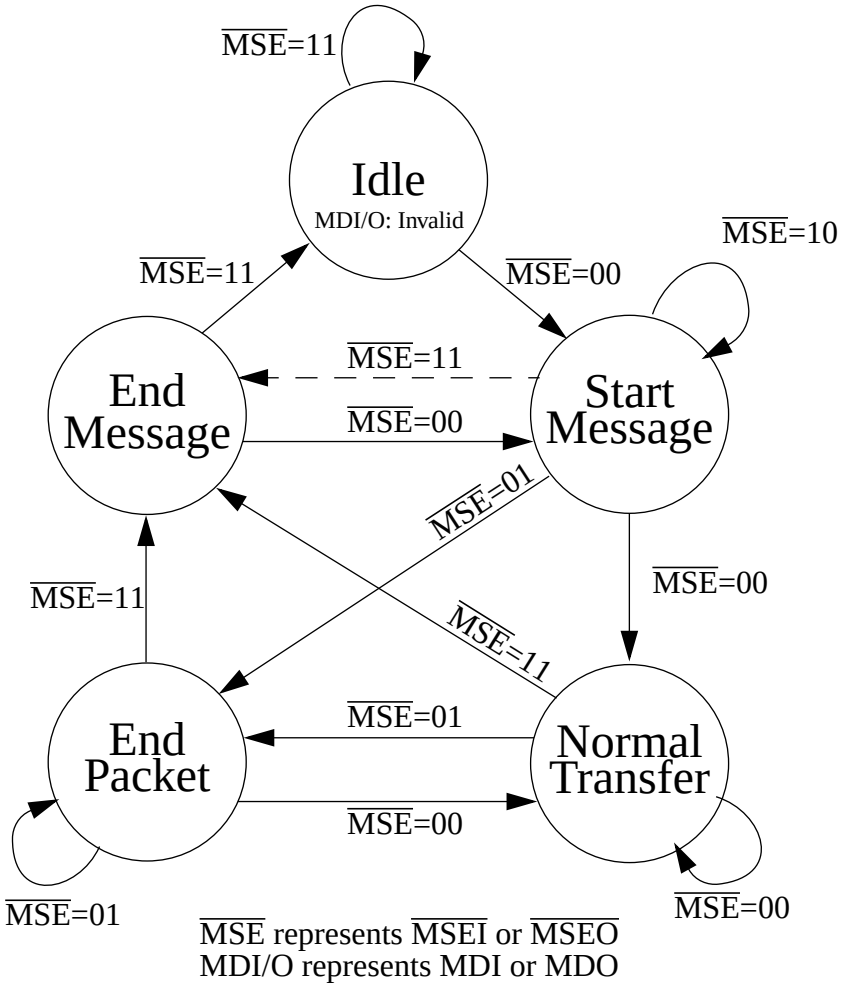


Figure 8-1 1-pin $\overline{\text{MSEI/MSEO}}$ Transfers



Notes:

1—The variable port size for MDO and $\overline{\text{MSE}}$ allows for increased transfer rates per clock.

2—The 1-pin $\overline{\text{MSE}}$ option should be selected when pin count is the most critical factor in the system and performance is not a priority.

3—The 2-pin $\overline{\text{MSE}}$ option should be chosen when performance is the top priority and pin count is secondary.

Figure 8-2 2-Pin $\overline{\text{MSEI}}$/$\overline{\text{MSEO}}$ Pin Transfers

Note that the “End Message” state may also indicate the end of a variable-length packet as well as the end of the message when using the 2-pin option.

Figure 8-3 illustrates the transfer protocol for the Indirect Branch Message. For purposes of illustration only 1 MDO pin and 1 $\overline{\text{MSEO}}$ pin are shown. MDO and $\overline{\text{MSEO}}$ are sampled on the rising edge of MCKO.

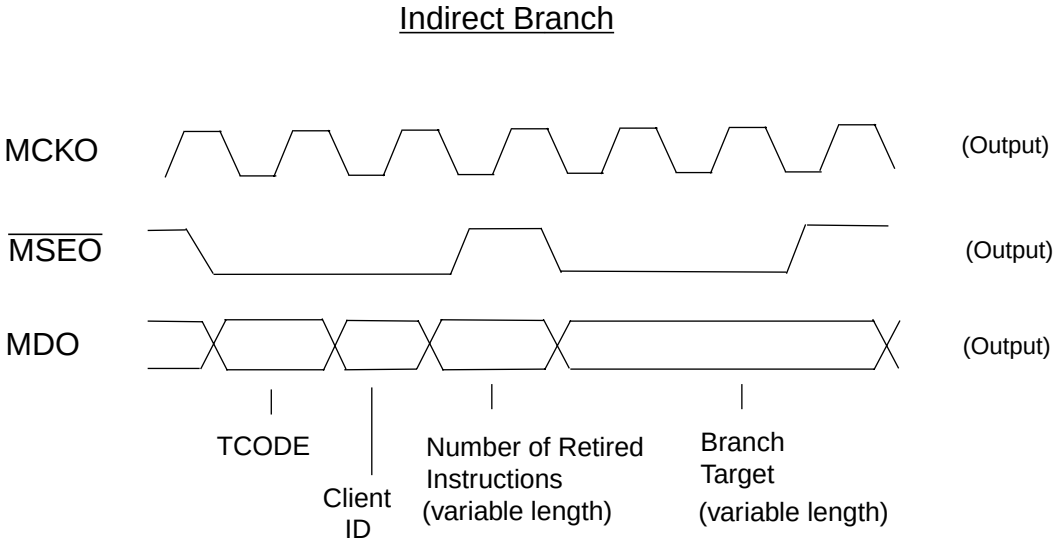


Figure 8-3 Timing Diagram for Indirect Branch

Table 8-1 and **Table 8-2** illustrate examples of 1-pin and 2-pin $\overline{\text{MSEO}}$ options for the same Indirect Branch Message.

Note that T0 and S0 are the least significant bits where:
Tx = TCODE number
Sx = Client which is source of message
Ix = Number of instruction units
Ax = Unique portion of the Address

Table 8-1 Indirect Branch Using the 1-Pin $\overline{\text{MSEO}}$ Option

Clock	MDO[3:0]				$\overline{\text{MSEO}}$[0]	
	3	2	1	0	0	Idle
0	X	X	X	X	1	Idle (or end of last message)
1	T3	T2	T1	T0	0	Start Message
2	S1	S0	T5	T4	0	Normal Transfer
3	I3	I2	I1	I0	0	Normal Transfer
4	I7	I6	I5	I4	1	End Packet
5	A3	A2	A1	A0	0	Normal Transfer
6	A7	A6	A5	A4	1	End Packet
7	X	X	X	X	1	End Message
8	T3	T2	T1	T0	0	Start Message

Table 8-2 Indirect Branch Using the 2-Pin $\overline{\text{MSEO}}$ Option

Clock	MDO[3:0]				$\overline{\text{MSEO}}$[1:0]		
	3	2	1	0	1	0	
0	X	X	X	X	1	1	Idle (or end of last message)
1	T3	T2	T1	T0	0	0	Start Message
2	S1	S0	T5	T4	0	0	Normal Transfer
3	I3	I2	I1	I0	0	0	Normal Transfer
4	I7	I6	I5	I4	0	1	End Packet
5	A3	A2	A1	A0	0	0	Normal Transfer
6	A7	A6	A5	A4	1	1	End Packet/ Message
7	T3	T2	T1	T0	0	0	Start Message

8.1 Rules for Messages

Embedded processors complying to classes 2, 3 and 4 shall provide messages via the AUX in a consistent manner as described below:

- A variable-sized packet within a message must end on a port boundary.
- A variable-sized packet may start within a port boundary only when following a fixed-length packet. (If two variable-sized packets end and start on the same clock, it is impossible to know which bit is from the last packet and which bit is from the next packet.)
- Whenever a variable-length packet is sized such that it does not end on a port boundary, it is necessary to extend and zero fill the remaining bits after the highest-order bit so that it can end on a port boundary.

  For example, if the MDO port is 4 bits wide, and the unique portion of an indirect address TCODE is 5 bits, then the remaining 3 bits of MDO must be packed with 0s.

Clock	MDO[3:0]				$\overline{\text{MSEO}}$[1:0]		
	3	2	1	0	1	0	
0	A3	A2	A1	A0	0	0	Normal Transfer
1	0	0	0	A4	0	1	End Packet

- Processors that do not have A0 and/or A1 address bits must be consistent in their representation of address values within all messages. That is, bits A0/A1 must always be included or excluded from all Public Messages.
- A data packet within a data message must be 8, 16, 32, or 64 bits in length.
- To improve message compression, multiple device-specific or fixed-length packets may start and end on a single clock.
- Each type of device-specific or fixed-length packet must be the same within all messages. For example, if a vendor implements 3 bits to identify the source processor, then all Public Messages with a source processor packet must be 3 bits in length.
- When a device-specific or fixed-length packet follows a variable sized packet, the device-specific or fixed-length packet must start on the port boundary.
- $\overline{\text{MSEI}}$/$\overline{\text{MSEO}}$ protocol must be followed for both input and output messages.

SECTION 9
IEEE 1149.1 Message Protocol

Embedded processors complying to class 1, 2, 3 or 4 may optionally implement messages via the IEEE 1149.1 interface according to this standard.

Two basic categories of messages may be implemented: solicited and unsolicited. Solicited messages are initiated and transmitted from an external controller to the embedded processor, e.g. to read an NRR. Unsolicited messages are generated by the embedded processor and are normally transmitted at random times. Unsolicited messages are most commonly transmitted via the AUX, however, a mechanism is described in 9.1 that allows for the retrieval of unsolicited messages via an IEEE 1149.1 interface.

9.1 IEEE 1149.1 Compatibility

An IEEE 1149.1 port used for this standard shall implement all the mandatory features of a standard IEEE 1149.1 port, including the “BYPASS” and “IDCODE” instructions. A 16-state IEEE 1149.1 TAP state machine will be used per the IEEE-1149.1 standard as illustrated in **Figure 9-1**.

The 5 required IEEE 1149.1 pins will be as follows:

- Test Data Input (TDI) provides for serial movement of data into the IEEE 1149.1 port.
- Test Data Output (TDO) provides for serial movement of data out of the IEEE 1149.1 port. All target accesses initiated via the IEEE 1149.1 port should be transmitted by the target via TDO (not via auxiliary output port).
- Test Clock Input (TCK) provides the clock for the IEEE 1149.1 port.
- Test Mode Select Input (TMS) provides access to the IEEE 1149.1 TAP state machine.
- Optionally, the Test Reset Input ($\overline{\text{TRST}}$) provides for asynchronous initialization of the IEEE 1149.1 controller.
- Optionally, the Ready Output ($\overline{\text{RDY}}$) is used to accelerate data accesses through the IEEE 1149.1 port.

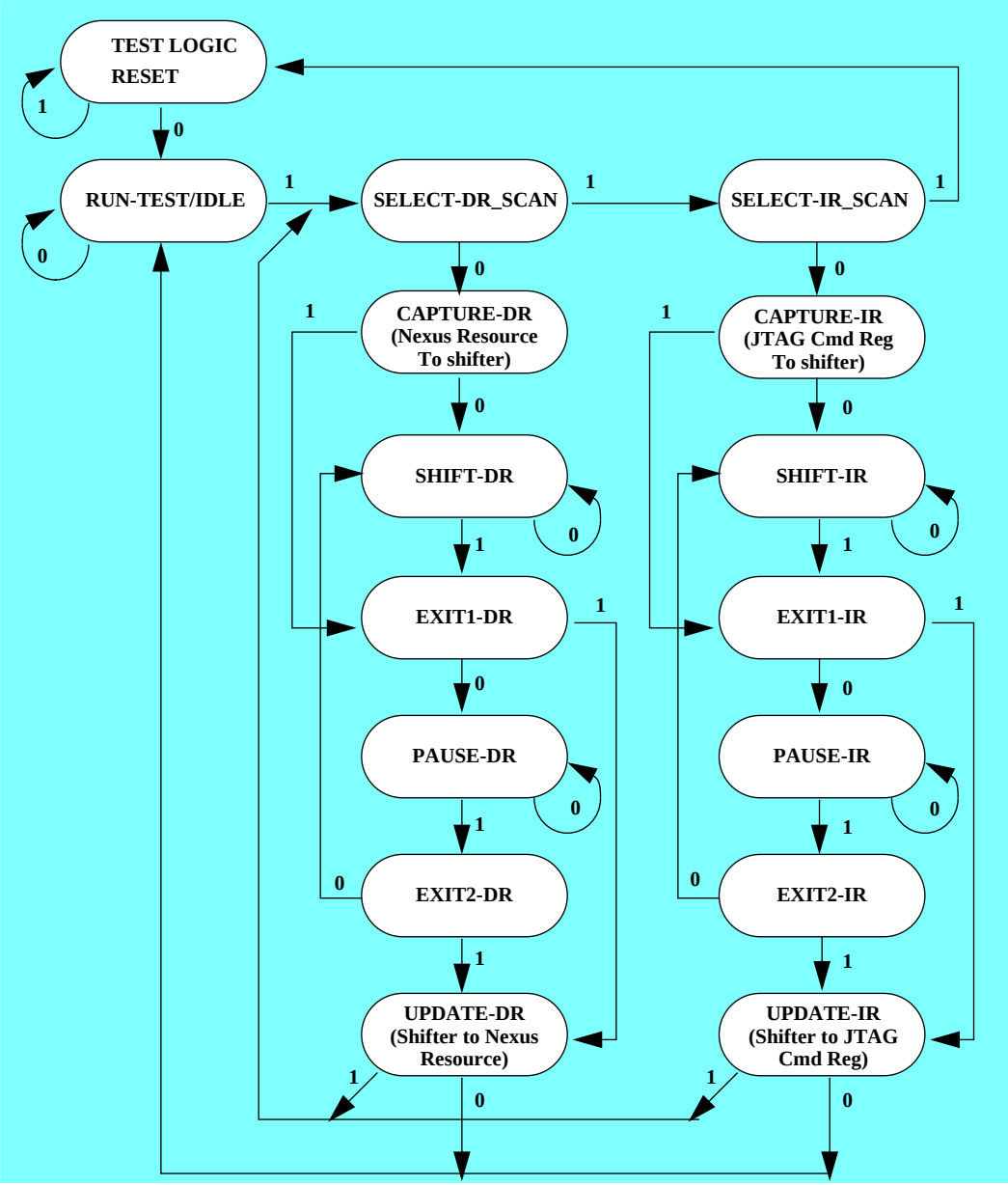


Figure 9-1 16-State IEEE 1149.1 Finite State Machine

Assertion of a power-on-reset signal on the embedded processor or the $\overline{\text{TRST}}$ pin causes the IEEE 1149.1 controller to default to being loaded with the IDCODE instruction upon exit of TEST-LOGIC-RESET controller state. This allows immediate entry to the SELECT-DR_SCAN path to retrieve the contents of the device ID. The LSB of the IDCODE must be a logic 1 so that examination of the first bit of data shifted out of a component during a data scan sequence immediately following exit from the TEST-LOGIC-RESET controller state will show whether a Device Identification (DID) Register is included in the design. The Nexus API may then retrieve the characteristics of the device to configure the software interface.

The system logic shall continue its normal operation undisturbed when the IEEE 1149.1 controller is decoding the IDCODE instruction. All NRRs must be accessible through the IEEE 1149.1 port independent of the state of the target processor.

Refer to the IEEE 1149.1 specification for further details on electrical and pin protocol compliance requirements. Additional information on the IEEE 1149.1 pin interface to connectors may be found in **APPENDIX A.** The IEEE 1149.1 NRRs may be found in **APPENDIX B.**

9.1.1 Optional Ready ($\overline{\text{RDY}}$) Output Pin

To increase the transfer rate of the IEEE 1149.1 port an additional pin may be implemented to signal when data is ready to be transferred to and from NRRs. This may eliminate the need to poll NRRs for status information for synchronization purposes. This capability becomes especially important when performing read/write access transfers to different speed target memories.

Without the use of a $\overline{\text{RDY}}$ pin, each time a read/write access transfer is made to a target memory location, it will be necessary to check if the memory transfer has completed. The function of the $\overline{\text{RDY}}$ pin will be to assert (asynchronously) to a logic low for a period of 4 target system clocks then de-assert whenever the read/write access transfer has completed.

The function of the $\overline{\text{RDY}}$ pin may also be used for unsolicited Public Messaging as described in 9.6 Reading Unsolicited Messages on Page 105.

Table 9-1 illustrates the IEEE 1149.1 sequence required to read the Device IDCODE immediately after assertion of the $\overline{\text{TRST}}$ pin, or after 5 TCK clocks with the TMS pin at a logic 1.

Table 9-1 IEEE 1149.1 Sequence to Read Device IDCODE After $\overline{\text{TRST}}$ Assertion

Step	TMS	IEEE 1149.1 State	Nexus State	Description
1	1	TEST-LOGIC-RESET	IDLE	IEEE 1149.1 controller in reset state.
2	0	RUN-TEST-IDLE	IDLE	IDCODE loaded into IEEE 1149.1 IR.
3	1	SELECT-DR_SCAN	IDLE	
4	0	CAPTURE-DR	IDLE	Load Device ID into TDI/TDO shifter.
5	0	SHIFT-DR	IDLE	TDO active and IEEE 1149.1 shifter presents a 1 in LSB.
N–1 TCKs			IDLE	
6	1	EXIT1-DR	IDLE	Last bit of Device ID shifted out to TDO.
7	1	UPDATE-DR	IDLE	
8	0	RUN-TEST-IDLE	IDLE	IEEE 1149.1 controller ready for Instruction.

9.2 Selecting the IEEE 1149.1 Port

Access to NRRs is enabled when the IEEE 1149.1 controller is decoding a device-specific “NEXUS-ENABLE” instruction entered via the SELECT-IR_SCAN path. When the IEEE 1149.1 controller passes through the UPDATE-IR state and decodes the NEXUS-ENABLE instruction, the Nexus controller will be reset to the NRR select state. The Nexus controller will have three states: idle, register select state and register data access state. **Table 9-2** illustrates the IEEE 1149.1 sequence to select the Nexus controller.

Table 9-2 IEEE 1149.1 Sequence to Enable Nexus Block for Communication

Step	TMS	IEEE 1149.1 State	Nexus State	Description
1	0	RUN-TEST-IDLE	IDLE	IEEE 1149.1 controller in reset state.
2	1	SELECT-DR_SCAN	IDLE	
3	1	SELECT-IR-SCAN	IDLE	
4	0	CAPTURE-IR	IDLE	Load last register select command into TDI/TDO shifter.
5	0	SHIFT-IR	IDLE	TDO becomes active and the IEEE 1149.1 shifter is ready. Shift *N*-1 bits of size of vendor-defined Nexus-Enable Instruction.
N–1 TCKs				
6	1	EXIT1-IR	IDLE	Last bit of Device ID shifted out to TDO.
7	1	UPDATE-IR	REG_SELECT	IEEE 1149.1 controller decoder. Nexus controller is forced to register select state.
8	0	RUN-TEST-IDLE	REG_SELECT	Nexus controller enabled and ready to receive commands.

9.3 Selecting an IEEE 1149.1 Register

When the IEEE 1149.1 “NEXUS-ENABLE” instruction is being decoded by the IEEE 1149.1 controller, the IEEE 1149.1 port allows tool/target communications via up to 128 IEEE 1149.1 NRRs. Each NRR is referenced by a unique register address index in the range 0 through 127.

All communication with the Nexus controller is performed via the SELECT-DR_SCAN path. The Nexus controller will default to a register select state when enabled. Accessing an NRR requires two passes through the SELECT-DR_SCAN path, one pass to select the NRR and the second pass to read/write the NRR data.

The first pass through the SELECT-DR_SCAN path is used to enter an 8-bit Nexus command consisting of a read/write control bit in the LSB followed by a 7-bit NRR address, as illustrated in **Figure 9-2**.

Figure 9-2 IEEE 1149.1 Controller Command Input

The second pass through the SELECT-DR_SCAN path is used to read or write the NRR data by shifting in the data LSB first during the SHIFT-DR state. When reading an NRR, the register value will be loaded into the IEEE 1149.1 shifter during the CAPTURE-DR state. When writing to an NRR, the value will be loaded by the IEEE 1149.1 shifter to the NRR during the UPDATE-DR state. When reading data from an NRR, there is no requirement to shift out the entire NRR contents, and shifting may be terminated once the required number of bits have been acquired. **Figure 9-3** illustrates the relationship between an IEEE 1149.1 TAP state machine and a Nexus controller state machine.

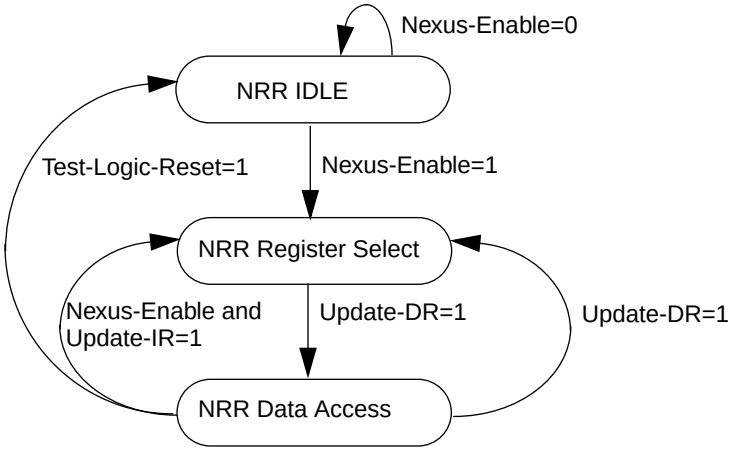


Figure 9-3 IEEE 1149.1 TAP State Machine relationship to Nexus Controller State Machine

Table 9-3 illustrates an IEEE 1149.1 sequence that will write a 32-bit value to an NRR.

Table 9-3 IEEE 1149.1 Sequence to Write to an NRR

Step	TMS	IEEE 1149.1 State	Nexus State	Description
1	0	RUN-TEST-IDLE	NRR REG_SELECT	IEEE 1149.1 controller in idle state.
2	1	SELECT-DR_SCAN	NRR REG_SELECT	
3	0	CAPTURE-DR	NRR REG_SELECT	IEEE 1149.1 shifter may be loaded with last value of register being decoded by Nexus controller or Nexus status information.
4	0	SHIFT-DR	NRR REG_SELECT	TDO becomes active and NRR address and write bit is shifted in.
7 TCKs			NRR REG_SELECT	
5	1	EXIT1-DR	NRR REG_SELECT	Last bit of NRR shifted into TDI.
6	1	UPDATE-DR	NRR REG_SELECT	Nexus controller decodes and selects register.
7	1	SELECT-DR_SCAN	NRR DATA ACCESS	Second pass through SELECT-DR_SCAN.
8	0	CAPTURE-DR	NRR DATA ACCESS	IEEE 1149.1 shifter may be loaded with last value of register being decoded by Nexus controller or Nexus status information.
9	0	SHIFT-DR	NRR DATA ACCESS	TDO becomes active and outputs current value of register while new value is shifted in through TDI.
N–1 TCKs			NRR DATA ACCESS	
10	1	EXIT1-DR	NRR DATA ACCESS	Last bit of NRR shifted out to TDO.
11	1	UPDATE-DR	NRR DATA ACCESS	Nexus controller writes value to register.
12	0	RUN-TEST/IDLE	NRR REG_SELECT	IEEE 1149.1 controller returns to idle state, or may return to SELECT-DR_SCAN state for new NRR register select. Total number of TCK clocks = 49 in this example.

9.4 Read/Write Access via the IEEE 1149.1 Port

The read/write access registers, as described in **APPENDIX B,** provide a means for transferring single or multiple data values through the auxiliary or IEEE 1149.1 port. When using the IEEE 1149.1 port, the RWCS and RWA Registers are initialized for the data transfer. Once initialization is complete, synchronization with the target must be handled by an external target controller. There will be two methods available for synchronization of data transfers.

The first method uses an optional pin called Ready for Transmission ($\overline{\text{RDY}}$). The $\overline{\text{RDY}}$ signal asserts (asynchronously) to indicate that the Nexus module is read for read access, or the write access is complete. An external development tool may then clock the IEEE 1149.1 port and perform the next read/write access. Use of a $\overline{\text{RDY}}$ pin permits data transfers in [16 + (data width)] TCK clocks, assuming the IEEE 1149.1 controller starts from and ends in the SELECT-DR_SCAN state.

Table 9-4 IEEE 1149.1 Sequence for Read/Write Access with $\overline{\text{RDY}}$ Pin

Step	TMS	IEEE 1149.1 State	Nexus State	Description
1	1	SELECT-DR_SCAN	NRR REG_SELECT	Starting point of this example.
2	0	CAPTURE-DR	NRR REG_SELECT	IEEE 1149.1 shifter may be loaded with last value of register being decoded by Nexus controller or Nexus status information.
3	0	SHIFT-DR	NRR REG_SELECT	TDO becomes active and Nexus R/W Data Register is selected for write. Data is then shifted from TDI.
7 TCKs			NRR REG_SELECT	
4	1	EXIT1-DR	NRR REG_SELECT	Last bit of Nexus R/W Data Register shifted from TDI.
5	1	UPDATE-DR	NRR REG_SELECT	Nexus controller decodes and selects register.
6	1	SELECT-DR_SCAN	NRR DATA ACCESS	Second pass through SELECT-DR_SCAN.
If Read Access, wait for $\overline{\text{RDY}}$ pin assertion.				
7	0	CAPTURE-DR	NRR DATA ACCESS	IEEE 1149.1 shifter may be loaded with last value of register being decoded by Nexus controller or Nexus status information.
8	0	SHIFT-DR	NRR DATA ACCESS	TDO becomes active and outputs current value of register while new value is shifted in through TDI.
N–1 TCKs			NRR DATA ACCESS	
9	1	EXIT1-DR	NRR DATA ACCESS	Last bit of NRR shifted out to TDO.
10	1	UPDATE-DR	NRR DATA ACCESS	Nexus controller writes value to register.
11	1	SELECT-DR_SCAN	NRR REG_SELECT	IEEE 1149.1 controller returns to SELECT-DR_SCAN state for new NRR select. Total number of TCK clocks = 48 in this example.
If the Write Access and Transfer count is greater than 1, wait for $\overline{\text{RDY}}$ pin assertion then go to Step 2, else go to idle state.				

If a $\overline{\text{RDY}}$ pin is not made available, polling the RWCS Register for an SC bit value of 0 is required. The polling method requires 65 TCK clocks for transfer of a 32-bit value.

9.5 Reading and Writing Public Messages

Public Messages may be read from or written to the target via the IEEE 1149.1 port. Public Message Writes are solicited messages that are generated by an external IEEE 1149.1 controller and are input into an Input Public Message Register (IPMR). The IPMR receives its TCODES and packets via multiple passes through the SELECT-DR_SCAN.

The IEEE 1149.1 protocol does not permit Public Messages to be generated from an embedded target microcontroller. Therefore, an Output Public Message Register (OPMR) must be made available for transmission of Public Messages from the embedded target microcontroller to an external IEEE 1149.1 controller. The IPMR and OPMR Registers may reside in the vendor-defined address range, as illustrated in APPENDIX B.

The external IEEE 1149.1 controller dictates the length of each packet for an Input Public Message and may terminate shifting in the SHIFT-DR state at any time. An end of message terminator is not required by an external IEEE 1149.1 controller to the Nexus controller when performing Input Public Message Writes.

Output Public Message Reads are unsolicited messages which are generated by the target processor and are read from the OPMR. These unsolicited messages may contain variable length packets of data. Two methods may be used for determining when an Output Public Message is available, when to terminate retrieving a variable length packet and when an Output Public Message is ended.

9.6 Reading Unsolicited Messages

It is also possible to detect when an unsolicited message is available in the OPMR. This method will require the selection of the OPMR followed by a read of the LSB of the OPMR to receive an Output Message Pending (OMP) status bit. If the OMP is a logic 0 the external IEEE 1149.1 controller may terminate OPMR shifting. If the OMP is logic 0 the Nexus controller will not advance to the register data access state, but instead will stay in the register select state.

An unsolicited output Public Message is ready for retrieval when the OMP is a logic 1 and the Nexus controller will advance to the register data access state. Resolution of variable length packets is determined by the bit width of the Output Message Register. The width of the Output Message Register will be vendor defined, where the vendor may optimize the register size depending upon the size of packets transmitted. **Figure 9-4** illustrates the IEEE 1149.1 TAP state machine for accessing the Public Message Registers as well as other NRRs.

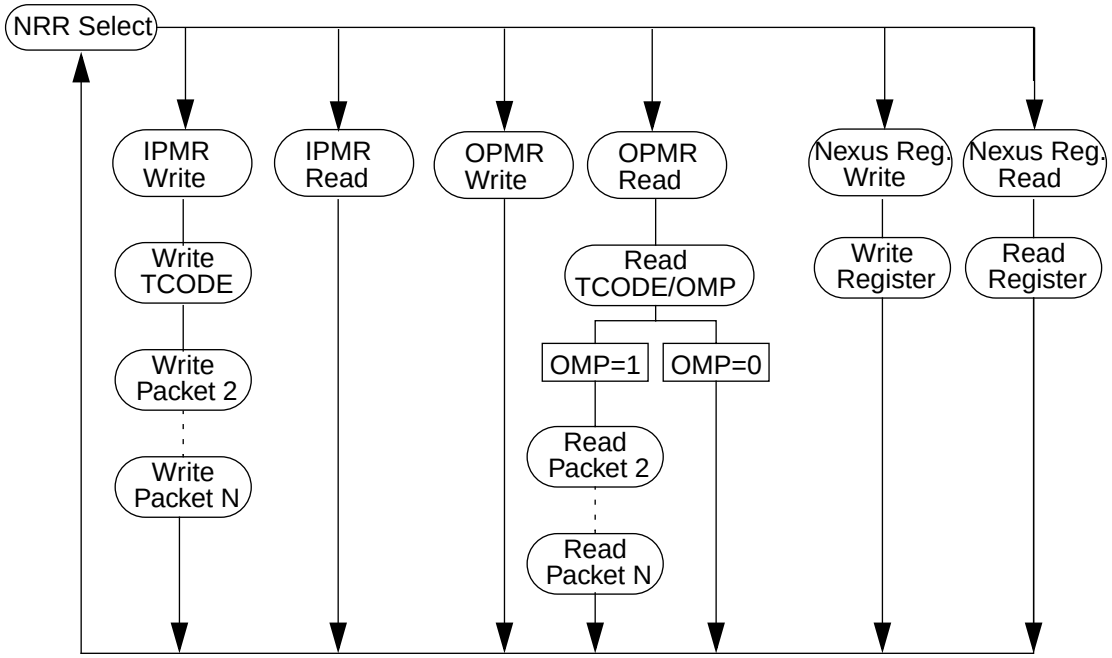


Figure 9-4 IEEE 1149.1 Controller State Diagram for Public Messaging

Note that each bubble, except for NRR Select, represents a complete pass through SELECT-DR_SCAN.

SECTION 10
Miscellaneous Topics

10.1 Multiple Address Threads

On embedded processors that implement data and program trace, there will be an address thread for each type of trace: the data address thread for both data read and Data Write Messages, and the instruction address thread for all Program Trace Messages. Messages containing a data address packet will be encoded and compressed using the data address most recently transmitted, thus creating a data address thread. Likewise, messages containing an instruction address packet will be encoded and compressed using the instruction address most recently transmitted, thus creating an instruction address thread.

There may be low cost implementations that only require 1 thread for both program and data trace. While this is not recommended and highly discouraged, it is reluctantly allowed. This is discouraged because of the increased bandwidth required and the severe feature set limitations it places on development tools. The bandwidth required is increased because locality of reference is lost when switching back and forth between instruction address space and data address space. For tools, storage enabling of message types is inhibited and modular design of packet encoding and decoding is prevented, thus limiting execution speed, etc. High-end cores should not even consider a 1-thread approach.

With 1 address thread, the next address (data or instruction) will be generated from the last message (Data Trace or Program Trace respectively).

10.2 Repeat Instructions and Hardware Loops

Some architectures may have explicit or implied flow control instructions. For example, many DSPs have hardware loops and repeat instructions. While it is possible to provide Vendor Defined Messages to handle such flow control instructions, they may also be handled as elegantly by using Public Messages.

10.2.1 Visibility for Repeat Instructions

In calculating the number of sequential instruction units executed for displaying direct or indirect branch messages, a repeat instruction (where the repeated instruction is non-interruptible) and the instruction to be repeated may be counted as one instruction each time it is executed. The sequential instruction count provided then would effectively “unroll” the repeat instructions.

If repeat instructions are interrupted, however, then a vendor-defined “repeat” message may be preferred. A “repeat” message may contain:

- The number of sequential instruction units executed since last taken branch and
- The number of the times instruction(s) are repeated.

The development tool must then compute the program trace, and correctly display when there is an exception and a return to finish the repeat count. If the device supports degenerate cases of repeat (such as repeating repeat instructions), more special considerations may be required to accurately deal with these cases.

10.2.2 Visibility for Hardware Loops

Visibility of hardware loops can be provided by transferring Indirect Branch Messages at or near the bottom of each hardware loop (when instruction fetching resumes at the top of the loop). Although Indirect Branch Messages are one of the longer message lengths, hardware loop messages will likely be compact since the target address does not change between each repetition of the loop. Thus the size of the message may be similar to a direct branch message.

While it is possible to generate Vendor Defined Messages that mark the entry into a hardware loop and the exit from a hardware loop, this method may add to both the complexity for generating these messages on the embedded device, as well as to the development tool in attempting to reconstruct the trace.

Some architectures provide implicit ways of exiting hardware loops, as well as other factors that may add to the complexity of providing visibility for hardware loops. These potential issues should be considered by the vendor during implementation.

10.3 Simultaneous Development of Multiple Embedded Processors

To facilitate development of multiple embedded processors interconnected by an existing serial communication bus standard, control and status information defined in this standard may be required to be accessible in the programmer’s model. If this is required, precautions should be taken to ensure that:

- Development resources are used only for development, and not for application purposes.
- Security should be provided for proprietary applications to restrict access to the application program.

APPENDIX A
Connector and Electrical Specifications

A.1 Connection Options

Table A-1 lists the connector options and the signals in each option.

Table A-1 Signal Summary

Pin Name	Connector A	Connector B Option 1 - IEEE 1149.1	Connector B Option 2 - Auxiliary Port	Connector B Option 3 - Combined	Connector C	Comments
MCKI	—	—	1	—	1	Auxiliary Port
MDI	—	—	2	—	4	
$\overline{\text{MSEI}}$	—	—	1	—	1	
MCKO	—	—	1	1	1	
MDO	—	—	4	2	8	
$\overline{\text{MSEO}}$	—	—	1	1	2	
$\overline{\text{EVTO}}$	1	1	1	1	1	
$\overline{\text{EVTI}}$	1	1	1	1	1	
$\overline{\text{RSTI}}$	—	—	1	—	1	
PORT	—	—	—	—	16	Port Replacement
IEEE 1149.1 Pins	5	5	—	5	—	IEEE 1149.1
$\overline{\text{RDY}}$	1	1	—	1	—	—
VREF	1	1	1	1	1	System Signals
$\overline{\text{RESET}}$	1	1	1	1	1	
CLOCKOUT	1	1	—	—	—	
Vendor Defined	1	1	1	1	2	—
UBATT	—	—	—	—	2	—
GROUND	8	13	13	13	38	—
TOTAL SIGNALS	12	12	16	15	42	—
TOTAL PINS	20	30	30	30	80	—

A.1.1 Signal Descriptions

Signal description used throughout this appendix is as follows:

OUT = output from the target to the development tool
IN = input to the target from the development tool

Refer to 7.1 Pin Functions on Page 87 for a description of the following signals:

MDO, $\overline{\text{MSEO}}$, MCKO, MDI, $\overline{\text{MSEI}}$, MCKI, $\overline{\text{RSTI}}$, $\overline{\text{EVTI}}$, $\overline{\text{EVTO}}$

Refer to the IEEE 1149.1 standard for a description of the following signals:

TDO, TDI, TCK, TMS, TRST

CLOCKOUT

This is the system clock from the target processor. CLOCKOUT helps development tools to determine the proper rate for TCK. CLOCKOUT can also be used to indicate target activity and used for MCKO where it matches the needs of the interface.

$\overline{\text{RESET}}$

This signal will cause the target to enter its reset state. The tool and target should use open-drain output drivers for this pin.

Vendor Defined

This signal may be used as needed by the target developer. Tool vendors should design their tools such that this signal can be configured as an input or output. This signal should be at a low enough slew rate as to not cause cross talk on adjacent pins.

VREF

This signal is used to establish the signaling levels of the debug interface of the target system. Any current drawn from this pin should be limited to that needed for voltage translation and/or signal interpolation and is not intended to supply logic functions or power. VREF is not necessarily at the target processor VDD level.

PORT[15:0]

Port replacement is a concept in which up to 16 low-speed I/O pins of the target processor can also be used to carry AUX signals. The development tool connects to the original I/O devices on the target system via the PORT pins. The development tool performs the input/output functions on behalf of the target when the tool receives Port Replacement Messages from the target processor.

The PORT[15:0] pins replace the signals directly across from them as shown in the connector C pin-out in **Table A-10**.

UBATT

Vender defined power supply pins. Not to be used as logic signals.

A.2 Connector A (IEEE 1149.1 Interface)

This connector is used for IEEE 1149.1 protocol only.

Connector A consists of IEEE 1149.1 signals which are $\overline{\text{TMS}}$, $\overline{\text{TRST}}$, TDI, TDO, and TCK. CLOCKOUT, $\overline{\text{RESET}}$, $\overline{\text{RDY}}$, $\overline{\text{EVTO}}$ and $\overline{\text{EVTI}}$ are added for enhanced performance and flexibility.

A.2.1 Signal Layout

Table A-2 lists connector A signals.

Table A-2 Connector A

Signal Name	I/O	Pin	Pin	I/O	Signal Name
$\overline{\text{RESET}}$	IN	1	2	O	VREF
$\overline{\text{EVTI}}$*	IN	3	4	—	GND
$\overline{\text{TRST}}$*	IN	5	6	—	GND
TMS	IN	7	8	—	GND
TDI	IN	9	10	—	GND
TCK	IN	11	12	—	GND
TDO	OUT	13	14	—	GND
CLOCKOUT*	OUT	15	16	—	GND
$\overline{\text{EVTO}}$*	OUT	17	18	—	GND
$\overline{\text{RDY}}$*	OUT	19	20	I/O	Vendor defined*

* Optional signals

A.2.2 Implementation Considerations

Because of its location on pin 2, VREF is used as a virtual ground for $\overline{\text{RESET}}$ on pin 1. Therefore, VREF should have decoupling capacitors at both ends of the cable connected to ground.

It is recommended that the signals in **Table A-3** be connected to pull-ups on the target to prevent floating signals when the tool is not connected or powered-on.

Table A-3 Signals for Pull-Ups on the Target

Signals	Pull-up
TMS, TCK, TDI, $\overline{\text{TRST}}$, $\overline{\text{RESET}}$, $\overline{\text{EVTI}}$	10K Ω

It is recommended that the following signals in **Table A-4** be connected to pull-ups on the tool to prevent floating signals when the target is not connected or powered-on.

Table A-4 Signals for Pull-Ups on the Tool

Signals	Pull-up
CLOCKOUT, $\overline{\text{EVTO}}$, $\overline{\text{RDY}}$	10K Ω

The target may need a jumper in the CLOCKOUT path near its source to prevent excessive radiated noise on the signal. This target design consideration eliminates the CLOCKOUT path between the CPU and the debug connector when debug operations are not being performed.

WARNING

Any optional signals not used by the target must be left unconnected at the target debug connector.

A.2.3 Mechanical Specifications

On the target system, the AMP, System 50, 20-pin, board mounted connector is recommended. The part number (P/N) is 104549-2. This a vertical, surface mount type header with a shroud and two mounting holes (see Detail A in **Figure A-1**[g]).

Other styles of System 50 board mounted connectors are available, such as thru-hole and horizontal mounting. Only those header connectors that mate with the AMP Ribbon Cable Connector System (P/N 111196-4) should be used.

g. Permission to reprint Figure A-1 and Figure A-2 was granted by AMP Incorporated, Harrisburg, PA.

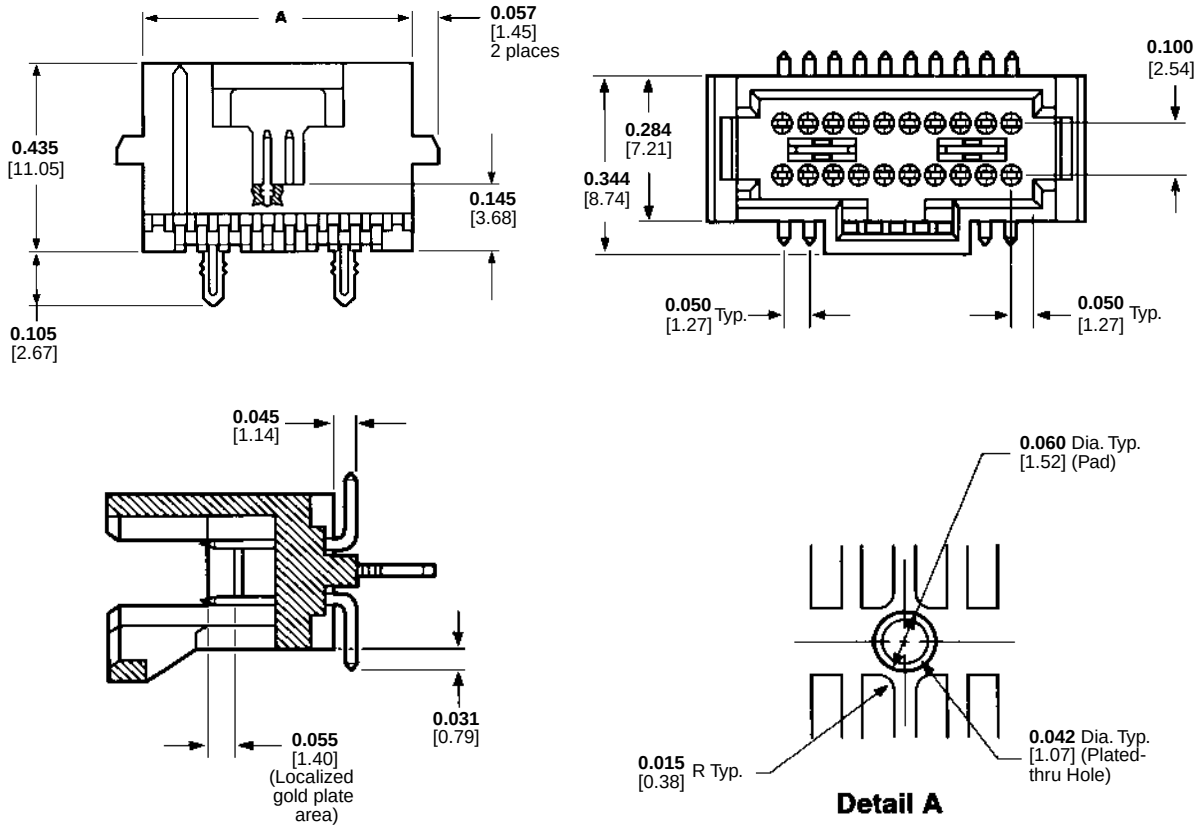


Figure A-1 AMP, System 50 Header (Surface Mount)

A.2.3.1 The PC Board (PCB) Connector Layout

AMP specifies the surface mount PCB layout shown in **Figure A-2**.

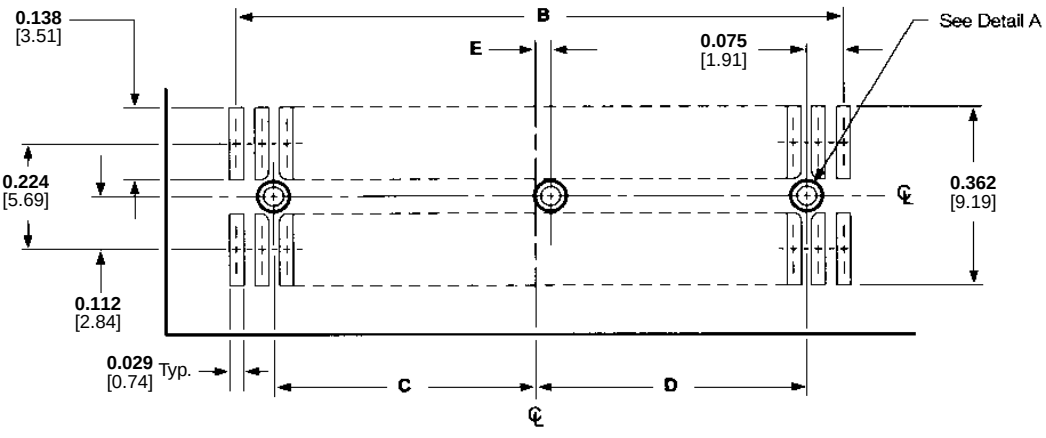


Figure A-2 Surface Mount Layout

Table A-5 shows the board layout for the PCB connector (numbers in square brackets are in millimeters).

Table A-5 Connector A Dimensions

	Dimensions
Dim. A	0.630 in. [16.00 mm]
Dim. B	0.450 in. [11.43 mm]
Dim. C	0.150 in. [3.80 mm]
Dim. D	0.150 in. [3.80 mm]

Development tools may use the AMP Ribbon Cable Connector System (AMP-LATCH) part number 111196-4. Tool vendors are responsible for assembling their own cables.

If a different 20-pin connector is desired, the chosen connector must conform to the same 0.050 in. pitch pin spacing and 0.100 in. row spacing as in the AMP connector (see **Table A-1**).

A.3 Connector B

This connector can be used in one of three configurations:

1. IEEE 1149.1 mode—IEEE 1149.1 only
2. AUX configuration—auxiliary port with 4 data-out pins and 2 data-in pins
3. Combined configuration—IEEE 1149.1 and auxiliary output with 2 data-out pins

The combined configuration may be utilized in those applications where the additional bandwidth of the AUX is needed but JTAG is still required. It is also suitable for those applications where there are constraints on the available board space or pin-out limitations on the target. The most common use of this configuration is for static debugging using JTAG with program trace on the AUX during runtime.

The AUX control registers and AUX input functions are accessed through the JTAG port. Direct memory read accesses may use JTAG or the auxiliary output port.

A.3.1 Signal Layout

Table A-6 identifies the pin-out for Connector B, listing the three configurations.

Table A-6 Connector B

Signal Name	Signal Name	Signal Name	I/O	Pin	Pin	I/O	Signal Name
1) IEEE 1149.1	*2) Auxiliary*	*3) Combined*					
$\overline{\text{RESET}}$	$\overline{\text{RESET}}$	$\overline{\text{RESET}}$	IN	1	2	O	VREF
$\overline{\text{EVTI}}$*	$\overline{\text{EVTI}}$	$\overline{\text{EVTI}}$	IN	3	4	—	GND
$\overline{\text{TRST}}$*	$\overline{\text{RSTI}}$	$\overline{\text{TRST}}$*	IN	5	6	—	GND
TMS	RESERVED	TMS	IN	7	8	—	GND
RESERVED	MDI1*	RESERVED	IN	9	10	—	GND
TDI	MDI0	TDI	IN	11	12	—	GND
TCK	MCKI	TCK	IN	13	14	—	GND
RESERVED	$\overline{\text{MSEI}}$	RESERVED	IN	15	16	—	GND
TDO	MDO3*	TDO	OUT	17	18	—	GND
$\overline{\text{RDY}}$*	MDO2*	$\overline{\text{RDY}}$*	OUT	19	20	—	GND
RESERVED	MDO1*	MDO1*	OUT	21	22	—	GND
RESERVED	MDO0	MDO0	OUT	23	24	—	GND
CLOCKOUT*	MCKO	MCKO	OUT	25	26	—	GND
RESERVED	$\overline{\text{MSEO}}$	$\overline{\text{MSEO}}$	OUT	27	28	—	GND
$\overline{\text{EVTO}}$*	$\overline{\text{EVTO}}$*	$\overline{\text{EVTO}}$*	OUT	29	30	I/O	Vendor defined*

* Optional signals

A.3.2 Implementation Considerations

Because of its location on pin 2, VREF is used as a virtual ground for $\overline{\text{RESET}}$ on pin 1. Therefore, VREF should have decoupling capacitors at both ends of the cable connected to ground.

It is recommended that the signals in **Table A-7** be connected to pull-ups on the target to prevent floating signals when the tool is not connected or powered-on.

Table A-7 Signals Connected To Pull-Ups On the Target

Signals	Pull-up
$\overline{\text{RESET}}$, $\overline{\text{EVTI}}$, $\overline{\text{RSTI}}$, $\overline{\text{MSEI}}$, $\overline{\text{TRST}}$, TMS, TDI MDI0, MDI1, MCKI	10K Ω

It is recommended that the signals in **Table A-8** be connected to pull-ups on the tool to prevent floating signals when the target is not connected or powered-on.

Table A-8 Signals To Be Connected To Pull-Ups On the Tool

Signals	Pull-up
MDO[3:0], MCKO, $\overline{\text{MSEO}}$, $\overline{\text{EVTO}}$, $\overline{\text{RDY}}$, TDO	10K Ω

WARNING

Any optional signals not used by the target must be left unconnected at the target debug connector.

A.3.3 Mechanical Specifications

On the target system, the AMP System 50, 30-pin, board mounted connector should be used (P/N 104549-5). This is a vertical, surface mount type header with a shroud and two mounting holes.

Other styles of System 50 board mounted connectors are available, such as thru-hole and horizontal mounting. Only those header connectors that mate with AMP Ribbon Cable Connector System (P/N 111196-7) should be used.

A.3.3.1 The PCB Connector Layout

Refer to **Figure A-2** for dimensions identified in **Table A-9**.

Table A-9 Connector B Dimensions

	Dimensions
Dim. A	0.880 in. [22.35 mm]
Dim. B	0.700 in. [17.78 mm]
Dim. C	0.275 in. [6.99 mm]
Dim. D	0.275 in. [6.99 mm]

Development tools may use the AMP Ribbon Cable Connector System (AMP-LATCH) part number 111196-7. Tool vendors are responsible for assembling their own cables.

If a different 30-pin connector is desired, the chosen connector must conform to the same 0.050 in. pitch pin spacing and 0.100 in. row spacing as the AMP connector.

A.4 Connector C (Auxiliary Port and Port Replacement)

Connector C is intended for target processors that support a wide AUX and/or for applications that can use port replacement.

Since the environmental and electrical requirements for this application vary, a custom target-to-tool cable assembly is required for each application while the board-to-board target connector remains the same.

A.4.1 Signal Layout

Table A-10 represents the pin-out for the active signals on Connector C.

Table A-10 Connector C

Signal Name (1)	I/O (2)	Pin (3)	Pin (4)	Pin (5)	Pin (6)	I/O (7)	Signal Name (8)
$\overline{\text{RESET}}$	IN	1	2-UBATT*	3	4	OUT	VREF
$\overline{\text{EVTI}}$*	IN	5	6	7	8	OUT	$\overline{\text{EVTO}}$*
$\overline{\text{RSTI}}$	IN	9	10	11	12	I/O	Vendor defined*
MDI3*	IN	13	14	15	16	I/O	Vendor defined*
MDI2*	IN	17	18	19	20	I/O	PORT15*
MDI1*	IN	21	22	23	24	I/O	PORT14*
MDI0	IN	25	26	27	28	I/O	PORT13*
MCKI	IN	29	30	31	32	I/O	PORT12*
$\overline{\text{MSEI}}$	IN	33	34	35	36	I/O	PORT11*
MDO7*	OUT	37	38	39	40	I/O	PORT10*
MDO6*	OUT	41	42	43	44	I/O	PORT9*
MDO5*	OUT	45	46	47	48	I/O	PORT8*
MDO4*	OUT	49	50	51	52	I/O	PORT7*
MDO3*	OUT	53	54	55	56	I/O	PORT6*
MDO2*	OUT	57	58	59	60	I/O	PORT5*
MDO1*	OUT	61	62	63	64	I/O	PORT4*
MDO0	OUT	65	66	67	68	I/O	PORT3*
MCKO	OUT	69	70	71	72	I/O	PORT2*
$\overline{\text{MSEO1}}$*	OUT	73	74	75	76	I/O	PORT1*
$\overline{\text{MSEO0}}$	OUT	77	78	79-UBATT*	80	I/O	PORT0*

* Optional signals

The pins in columns 4 and 5 (excluding pins 2 and 79) are connected to ground. This essentially provides a ground return per signal.

A.4.2 Implementation Considerations

It is recommended that the signals in **Table A-11** be connected to pull-ups on the target to prevent floating signals when the tool is not connected or powered-on.

Table A-11 Signals To Be Connected To Pull-Ups On the Target

Signals	Pull-up
$\overline{\text{RESET}}$, $\overline{\text{EVTI}}$, $\overline{\text{RSTI}}$, $\overline{\text{MSEI}}$, MDI[3:0], MCKI	10K Ω

It is recommended that the signals in **Table A-12** be connected to pull-ups on the tool to prevent floating signals when the target is not connected or powered-on.

Table A-12 Signals To Be Connected To Pull-Ups On the Tool

Signals	Pull-up
MDO[7:0], MCKO, $\overline{\text{MSEO}}$[1:0], $\overline{\text{EVTO}}$, PORT[15:0]	10K Ω

WARNING

Any optional signals not used by the target must be left unconnected at the target debug connector.

A.4.3 Mechanical Specifications

On the target system, a Samtec MOLC series (0.050x0.050) header should be used. This connector offers surface mount (see **Figure A-3**[h]), through hole (see **Figure A-4**) or mixed technology PCB mounting. This header mates with the FOLC series.

WARNING

The MOLC/FOLC series connectors are not keyed, so it is imperative that the target have pin 1 properly marked. The cable assembly and tools must also have pin 1 clearly identified.

Since these are board-to-board connectors, the tool vendor is responsible for assembling a cable based on customer requirements.

h. Permission to reprint Figure A-3, Figure A-4, Figure A-5 and Figure A-6 was granted by Samtec Inc., New Albany, IN.

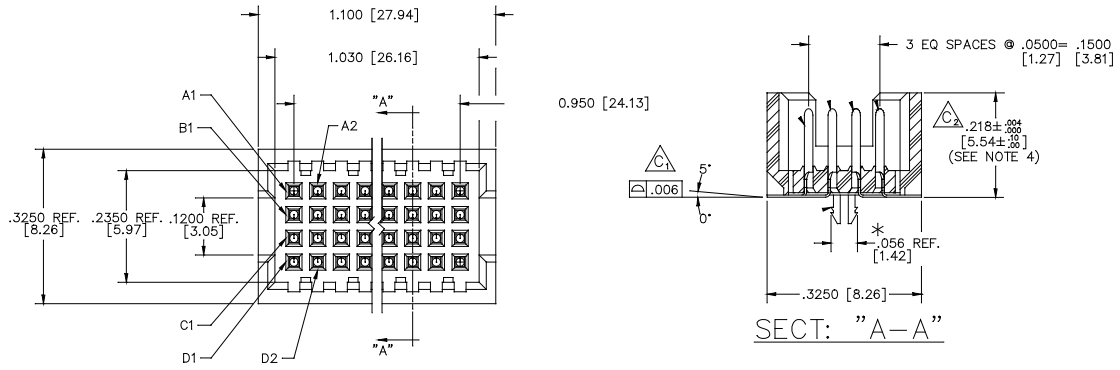


Figure A-3 Surface Mount Header

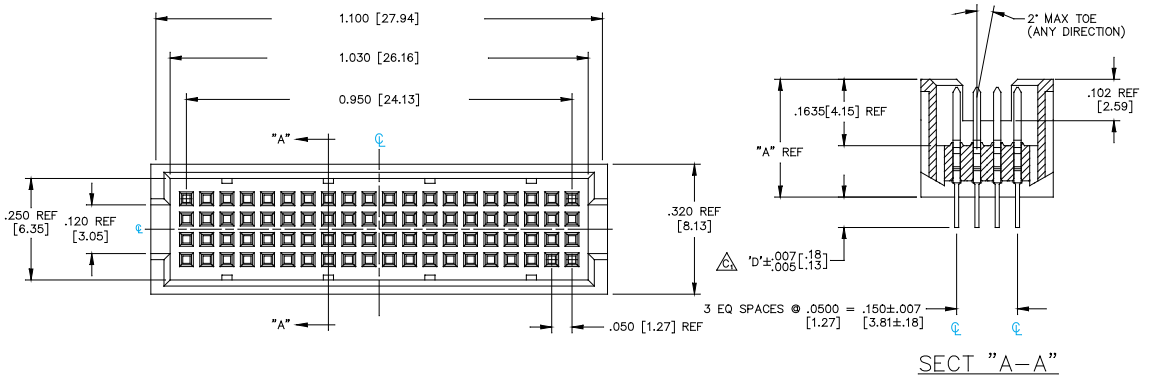


Figure A-4 Thru-Hole Header

A.4.3.1 The PCB Connector Layout

Figure A-5 illustrates the layout for a surface mount connector. **Figure A-6** illustrates the layout for a thru-hole connector.

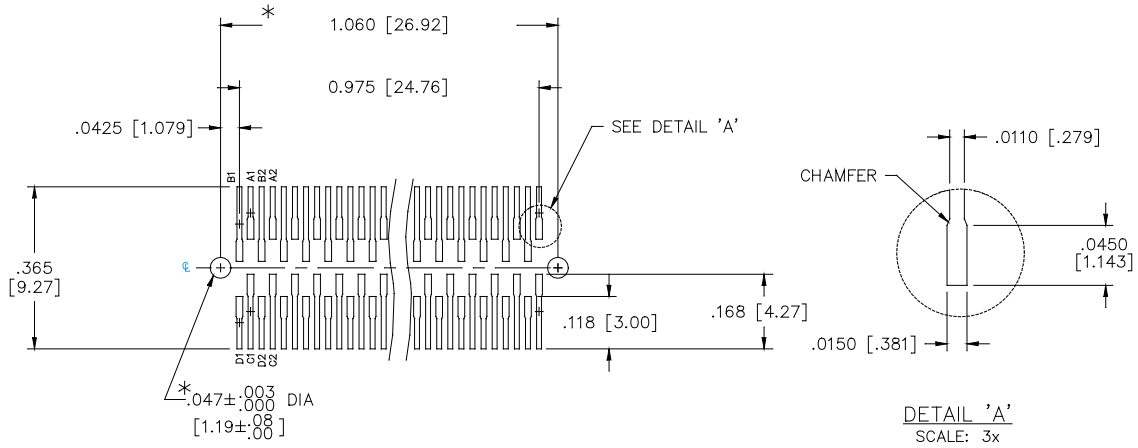


Figure A-5 Surface Mount PCB Layout

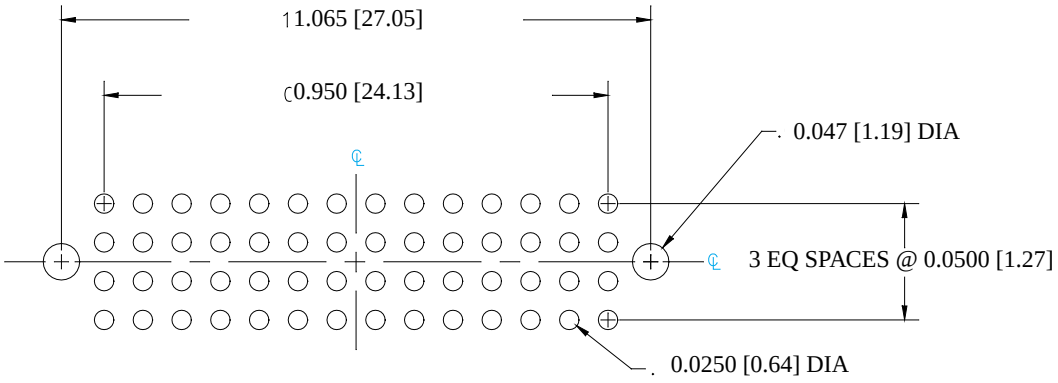


Figure A-6 Thru-hole PCB Layout

If a different 80-pin connector is desired, the chosen connector must conform to the same 0.050 in. pitch pin spacing, 0.050 in. row spacing, and 4x20 format as the Samtec connector.

A.5 DC Electrical Characteristics

Table A-13 lists the electrical characteristics for the signals used in the Nexus interface.

Table A-13 DC Electrical Characteristics

Characteristic	VREF Voltage	Min	Max	Unit
Input Low Voltage	VREF 2.8 V to 5 V	–0.3	0.8	V
Input High Voltage		2.0	1.2 (VREF)	V
Input Low Voltage	VREF below 2.8 V	–0.3	0.3 (VREF)	V
Input High Voltage		0.7 (VREF)	1.2 (VREF)	V
VREF Output Current	—	—	1	mA

All DC characteristics apply to the IEEE 1149.1 and AUX interfaces.

Output voltage levels need to be sufficient to satisfy the associated input requirements with a suitable margin.

The tool must sense the voltage on the VREF pin before attempting to drive outputs.

Absolute maximum tool output voltage is VREF + 20%.

The tool must not draw more than 1 mA of current from the VREF. It is a good idea to put a current-limiting resistor in series with the VREF, but the value should be minimal so as not to degrade the value of the VREF at the tool.

A.6 AC Electrical Characteristics—General

Input rise and fall times are measured at 20–80% values.

All setup and hold times are measured from the 50% point of the respective clock edge and the 50% point of the logic signal.

All measurements are made assuming a minimum capacitive loading of 25 pF.

A.7 AC Electrical Characteristics—IEEE 1149.1 Interface

Table A-14 lists the timing constraints for the IEEE 1149.1 interface.

Figure A-7gives a pictorial representation of critical timing in **Table A-14**.

Table A-14 AC Electrical Characteristics—IEEE 1149.1

Number	Characteristic	Min	Max	Unit
1	TCK Cycle Time (**Tc**)	30	—-	ns
2	TCK Duty Cycle	40	60	%
3	Rise and Fall Times (20–80%)	0	3	ns
4	$\overline{\text{TRST}}$ Setup Time to TCK Falling Edge	(0.30)Tc	—	ns
5	$\overline{\text{TRST}}$ Assert Time	(0.30)Tc	—	ns
6	TMS, TDI Data Setup Time	(0.20)Tc	—	ns
7	TMS, TDI Data Hold Time	(0.10)Tc	—	ns
8	TCK Low to TDO Data Valid	(–0.10)Tc	(0.20)Tc	ns
9	$\overline{\text{EVTO}}$ Pulse Width	(1.0) System Clock	—	ns
10	$\overline{\text{EVTI}}$ Pulse Width	(4.0)Tc	—	ns

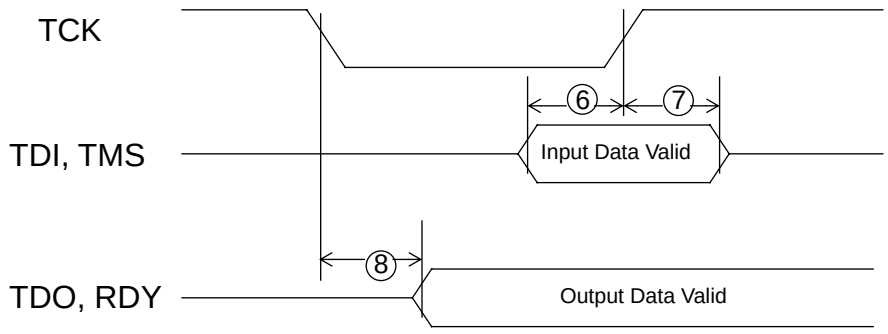


Figure A-7 IEEE 1149.1 Timing Diagram

A.8 AC Electrical Characteristics—Auxiliary Port

Table A-15 lists the timing constraints for the AUX interface. **Figure A-8** illustrates the critical timing for the clock to data on the input. **Figure A-9** illustrates the critical timing for the clock to data on the output.

Table A-15 AC Electrical Characteristics—Auxiliary Port

Number	Characteristic	Min	Max	Unit
1	MCKO Cycle Time (**Tco**)	5	—	ns
2	MCKO Duty Cycle	40	60	%
3	Output Rise and Fall Times	0	3	ns
4	MCKO low to MDO Data Valid	(–0.10)Tco	(0.20) Tco	ns
5	MCKI Cycle Time (**Tci**)	5	—	ns
6	MCKI Duty Cycle	40	60	%
7	Input Rise and Fall Times	0	3	ns
8	MDI Setup Time	(0.20)Tci	—	ns
9	MDI Hold TIme	(0.10)Tci	—	ns
10	$\overline{\text{RSTI}}$ Pulse Width	(4.0) Tco	—	ns
11	MCKO low to $\overline{\text{EVTO}}$ Valid	(–0.10)Tco	(0.20) Tco	ns
12	$\overline{\text{EVTI}}$ Pulse Width	(4.0) Tco	-—	ns
13	$\overline{\text{EVTI}}$ to $\overline{\text{RSTI}}$ Setup (at reset only)	(4.0) System Clock	—	ns
14	$\overline{\text{EVTI}}$ to $\overline{\text{RSTI}}$ Hold (at reset only)	(4.0) System Clock	—	ns

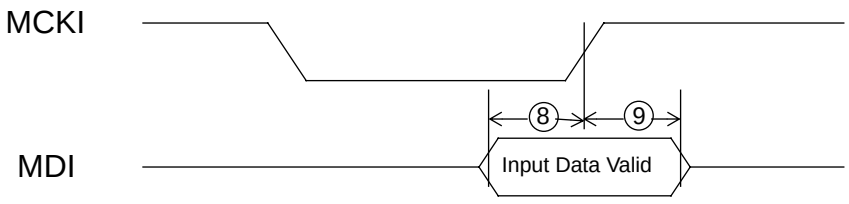


Figure A-8 Auxiliary Port Data Input Timing Diagram

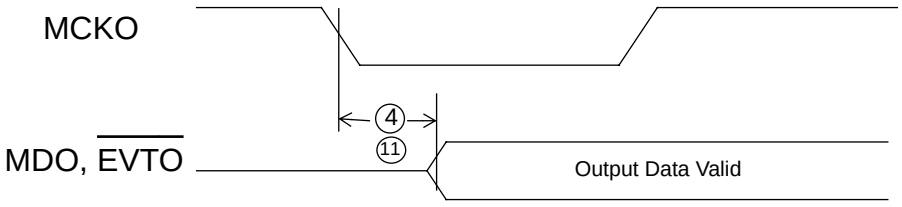


Figure A-9 Auxiliary Port Data Output Timing Diagram

MDO and $\overline{\text{EVTO}}$ data is held valid until the next MCKO low transition.

When $\overline{\text{RSTI}}$ is asserted, $\overline{\text{EVTI}}$ is used to enable or disable the AUX (see **Figure A-10** and **Figure A-11**). Because MCKO probably is not active at this point, the timing must be based on the system clock. Since the system clock is not realized on the connector, its value must be known by the tool.

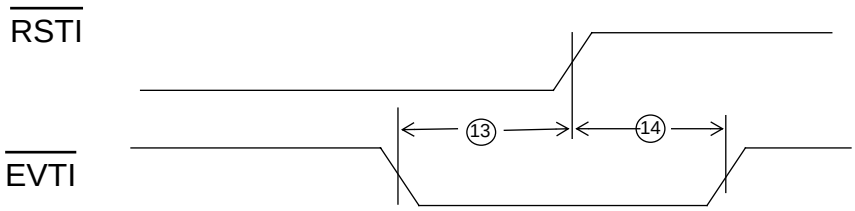


Figure A-10 Enable Auxiliary From $\overline{\text{RSTI}}$

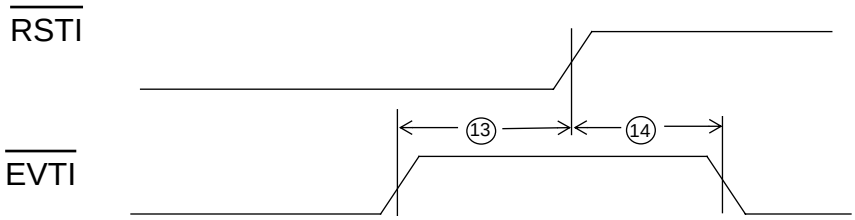


Figure A-11 Disable Auxiliary From $\overline{\text{RSTI}}$

A.9 Terminations

Because of the high-speed natures of the IEEE 1149.1 and auxiliary ports, it is recommended that the target and tool both employ a point-to-point series termination scheme (see **Figure A-12** and **Figure A-13**).

Z_{out} = Output impedance of the driver
Z_{target} = Impedance of traces on the target printed circuit board
Z_{cable} = Characteristic impedance of cable
Z_{tool} = Impedance of traces on the tool printed circuit board
R_t = Source terminators

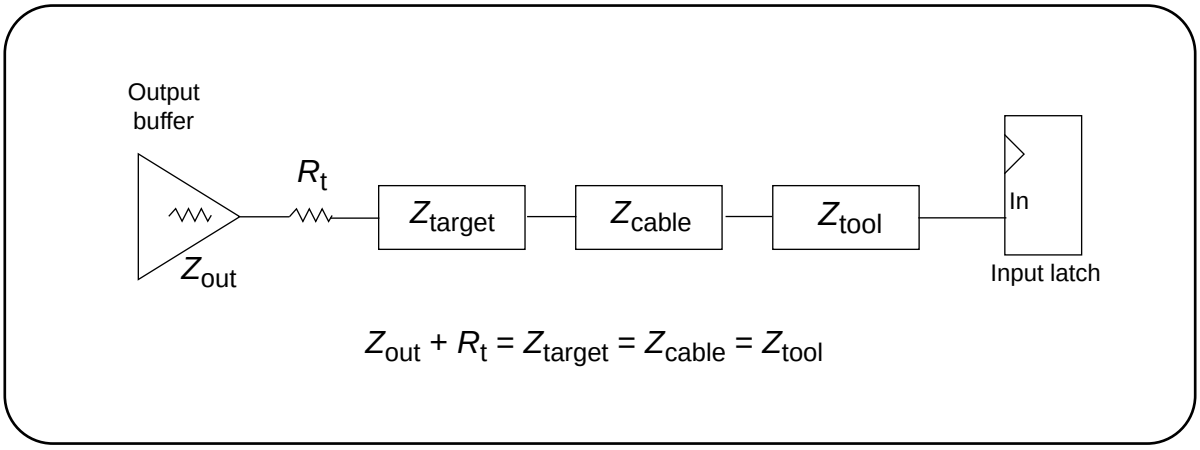


Figure A-12 Target Output Source Termination

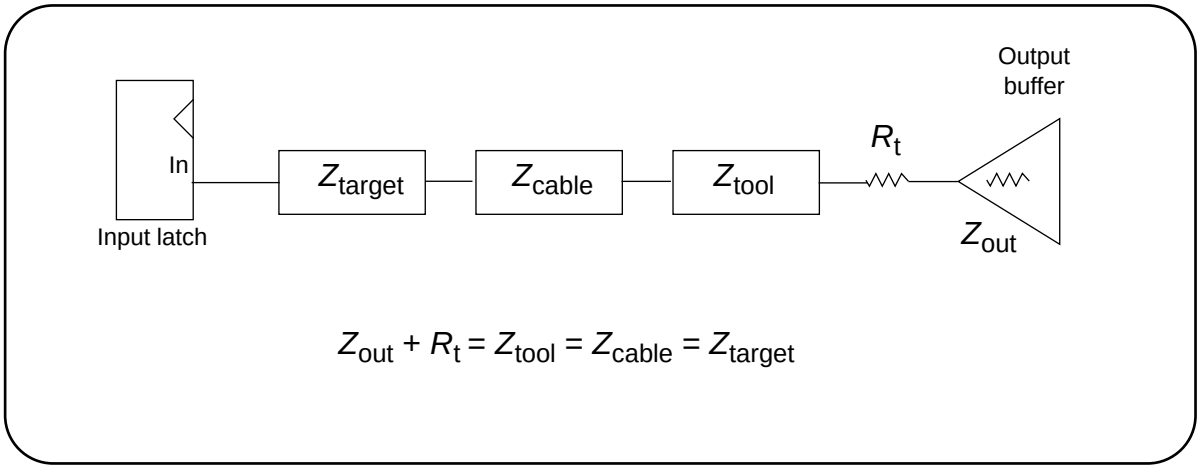


Figure A-13 Tool Output Source Termination

APPENDIX B
Recommendations for Access to Control and Status Registers

Silicon vendors must implement the API requirements for tool software compatibility as described in SECTION 5. In addition, embedded processors complying to class 2, 3 or 4 are required to implement the AUX message protocol and the required Public Messages as described in SECTION 5 and SECTION 8. There are no requirements in the Nexus standard, however, regarding conformity of development registers which are accessed by tools for control and status.

This appendix contains only recommendations (not requirements) for silicon vendors in implementing development registers that are accessed by tools for control and status.

NOTE

Development tool vendors should not design tools based upon the contents of this appendix. Obtain the silicon vendor's API and product specification for development tool design.

The Nexus standard supports development for up to 32 clients[i] on an embedded processor. Each client on embedded processors complying to class 1 may provide development tool access to control and status according to these recommendations via the IEEE 1149.1 interface. Each client on embedded processors complying to class 2, 3 or 4 may provide development tool access to control and status according to these recommendations via either the AUX or the IEEE 1149.1 interface.

Embedded processors may provide development control and status registers according to **Table B-1**, **Table B-2** and **Table B-3**. **Table B-1** illustrates the "NEXUS-ENABLE" instruction. Writing an appropriate value to the "NEXUS-ENABLE" instruction, as defined by the silicon vendor, will enable access to NRRs illustrated in **Table B-3**. Additionally, the Device ID information identifies key attributes to the development tool concerning the embedded processor.

In **Table B-2** the Client Select control selects one of the clients on the embedded processor for access. Once the client is selected, control and status accesses are directed to the selected client. An alternate client can be selected at any time during operation.

i. Refer to 1.1 Terms and Definitions on Page 2.

Table B-1 IEEE 1149.1 Register Map

Control/ Status	Compliance Class	Access Opcode	Read/ Write
IEEE 1149.1 Public Opcodes	—	Device-specific	—
Device ID Register[1]	All	Device-specific	R
NEXUS-ENABLE[2] (SECTION 9)	All	Device-specific	R/W

1. The ID Register is defined by the IEEE 1149.1 standard.
2. Only needed for IEEE 1149.1 port (and not AUX).

Table B-2 Nexus Clients

Control/ Status	Compliance Class	Access Opcode	Read/ Write
Device ID	All	0	R
Client Select	4[1]	1	R/W
Shared by all Nexus Clients	—	2–63	—
Reserved	—	64–127	—
Device-specific	—	128–255	—

1. If embedded processor contains multiple clients, then Client Select is required.

The NRR indices as shown in **Table B-3** shall be identical for accesses via the IEEE 1149.1 interface and the AUX. The fields associated with each opcode accessed via the IEEE 1149.1 interface shall be identical in size and function to the packets accessed for each opcode via the AUX. The Public Messages in SECTION 6 prescribe the method for accessing recommended control and status registers.

Table B-3 also defines control and status access as indicated per clients of class 2, 3 or 4 complying embedded processors. Device-specific register space is also provided so that vendor-defined development functions may be implemented. For embedded processors complying to class 2, 3 or 4, the device-specific registers may comprise the transfer registers for interfacing with a processor, e.g. Program Counter and Processor Status.

Table B-3 Recommended Registers for Nexus Clients

Nexus Recommended Register (NRR)	Compliance Class	Register Index	Read/ Write
Device ID (DID) (auxiliary only)	All	0	R
Client Select Control (CSC)	2, 3, 4[1]	1	R/W
Development Control	2, 3, 4	2	R/W
Reserved for Development Control	—	3	—
Development Status (DS)	4	4	R
Reserved for Development Status	—	5	—
User Base Address (UBA)	2, 3, 4	6	R[2]
Read/Write Access Control/Status (RWCS)	3, 4	7	R/W
Reserved for Read/Write Access Control/Status	—	8	—
Read/Write Access Address (RWA)	3, 4	9	R/W
Read/Write Access Data (RWD)	3, 4	10	R/W
Watchpoint Trigger (WT)	4	11	R/W
Reserved for Watchpoint Trigger	—	12	—
Data Trace Control (DTC)	3, 4	13	R/W
Data Trace Start Address (DTSA) (2)	3, 4	14–15	—
Data Trace Start Address (Reserved - 2)	—	16–17	—
Data Trace End Address (DTEA) (2)	3, 4	18–19	—
Data Trace End Address (Reserved - 2)	—	20–21	—
Breakpoint/Watchpoint Control (BWC) (2)	4	22–23	R/W
Breakpoint/Watchpoint Control (Reserved - 6)	—	24–29	—
Breakpoint/Watchpoint Address (BWA) (2)	4	30–31	R/W
Breakpoint/Watchpoint Address (Reserved - 6)	—	32–37	—
Breakpoint/Watchpoint Data (BWD) (2)	4	38–39	R/W
Breakpoint/Watchpoint Data (Reserved - 6)	—	40–45	—
Reserved for future Nexus functionality	—	46–> 54	—
Re-mapped NRRs (see B.10 Nexus Recommended Registers (NRRs) Concatenated for Better Transfer Efficiency on Page 144)	—	55–> 63	—
Vendor defined	—	64–> 127	—
Reserved for future Nexus functionality[3]	—	128–> 255	—

1. Needed if there are multiple clients on an embedded processor.
2. May also be read/write access for development tool configuration of UBA.
3. IEEE 1149.1 is not capable of Access Future Reserved.

Reset: All control and status information shall be reset by one of the following:

- IEEE 1149.1 Test Logic Reset state or assertion of $\overline{\text{TRST}}$
- Assertion of $\overline{\text{RSTI}}$

No control or status information shall be reset for system reset on the embedded processor.

Access with the IEEE 1149.1 Interface: The IEEE 1149.1 state machine is shown in **Figure 9-1**. The value shown adjacent to each arc represents the value of the TMS signal sampled on the rising edge of the TCK signal.

Access to NRRs is enabled by loading a single instruction (“NEXUS-ENABLE”) into the IEEE 1149.1 IR. Once the IEEE 1149.1 “NEXUS-ENABLE” instruction has been loaded, the IEEE 1149.1 port allows tool/target communications with all NRRs according to the register index in **Table B-3**.

Reading/writing of an NRR then requires two (2) passes through the Data-Scan path of the IEEE 1149.1 state machine.

1. The first pass through the DR selects the NRR to be accessed by providing an index (see **Table B-3**), and the direction (read/write). This is achieved by loading an 8-bit value into the IEEE 1149.1 DR. This register has the following format:

MSB	**LSB**
7 bit NRR Address	R/W

Read/Write (R/W):

0 = Read
1 = Write

Nexus Recommended Register (NRR) Address:
Selected from values in **Table B-3**

2. The second pass through the DR then shifts the data in or out of the IEEE 1149.1 port, LSB first.

    a. During a read access, data is latched from the selected NRR when the IEEE 1149.1 state machine passes through the “Capture-DR” state.

    b. During a write access, data is latched into the selected NRR when the IEEE 1149.1 state machine passes through the “Update-DR” state.

Access with the Auxiliary Port: The control and status registers are accessed via the four Public Messages—Target Ready, Read Register (tool requests information), Write Register (tool provides information) and Read/Write Response (from tool or target).

To write control or status the following sequence would be required:

- If a prior Target Ready Message was transmitted by the target, then the tool transmits a Write Register Message, which contains write attributes and a register value to be written.
- The tool waits for the Target Ready Message before initiating the next access.

To read control or status the following sequence would be required:

- If a prior Target Ready Message was transmitted by the target, then the tool transmits a Read Register Message, which contains read attributes.
- When the target reads data, it transmits a Read/Write Response Message containing read data. The target is now ready for the next access.

Control and Status Information: This section describes the fields comprising each control and status register. The control registers in this section are organized such that the most used bits are located in the most significant bits of the registers. This allows for short write sequences from the tool to write only a few bit fields.

For many of the control and status opcodes defined in this section there are bits reserved as vendor defined. Device-specific development features and operations may be included in these designated bits. For tools not implementing these vendor-defined development features, the fields should not be written or set to a value of 0. The setting of 0 is designated as the default state.

B.1 Device ID (DID) Register

Accessing DID provides key attributes to the development tool concerning the embedded processor. This information assists the development tool in determining configuration and features of the device. For classes 2, 3 and 4 embedded processors, this information is also transmitted via the auxiliary output port upon exit of AUX reset.

For embedded processors with a full AUX, the DID shown here should be implemented in compliance to the register organization and bit field definitions as specified in the IEEE 1149.1. For embedded processors with an IEEE 1149.1 interface used for this standard, the DID Register defined by the 1149.1 standard must be implemented. In this case, the DID Register defined in this sub-section is not necessary.

The fields include embedded processor information containing the manufacturer ID, product number and revision number. In general, the revision number must be changed (i.e. incremented) whenever the embedded processor has a mask revision that will disrupt the tools in any manner (see **Table B-4**).

Table B-4 DID Register

Bit Number	Field Name	Description
31–28	RN	Revision Number
27–12	PN	Product Number
11–1	MID	Manufacturer ID
0	—	Reserved

B.2 Client Select Control (CSC) Register

CSC contains a single 5-bit field which, when written to, selects the client to be accessed via the IEEE 1149.1 interface or the AUX. The encodings of CSC are device-specific. The setting of CS selects which client is accessed for access opcodes 1–127.

This register is recommended if there are multiple clients on the embedded processor (see **Table B-5**).

Table B-5 CSC Register

Bit Number	Field Name	Description
7–5	—	Reserved
4–0	CS	Client select

B.3 Development Control Register

The Development Control Register is used for basic development control of a client. DBE enables debug mode and DBR allows for a software mechanism to enter debug mode. If debug mode is enabled then asserting DBR, power-on reset or an exception may cause the processor to halt and enter debug mode. Enabling debug mode is necessary to use features such as single stepping and breakpoints.

The TM field enables BTM, DTM and OTM. One or all types of trace may be enabled by TM, or via a watchpoint occurrence (refer to B.7 Watchpoint Trigger (WT) Register).

If EIC = 00 and program and/or data trace are enabled, a high-to-low transition on $\overline{\text{EVTI}}$ will cause program and/or data trace synchronization respectively. If EIC = 01, a high-to-low transition on $\overline{\text{EVTI}}$ will cause a breakpoint to occur. If EIC = 10, no operation will occur regardless of the state on $\overline{\text{EVTI}}$.

The MS and SS bit fields determine how the processor will operate when DBR is negated. If MS = SS = 0 then normal operation will commence when DBR is negated. If MS = 0 and SS = 1, then a single step will occur when DBR is negated with internal memory access. If MS = 1 and SS = 0, then operation will commence when DBR is negated with instruction/data access via the AUX. If MS = SS = 1, then a single step will occur when DBR is negated with instruction/data access via the AUX.

When MS = 1, the state of the SO bits determines which combination of instruction and data accesses are substituted so that memory accesses are made via the AUX or IEEE 1149.1 interface. If MS = 0, memory substitution is not enabled and memory accesses are made to the target memory system.

OVC is used to determine control for overrun of BTM and DTM. Overruns can be handled by displaying an Overrun Message to development tools, delaying the processor to avoid BTM overruns, delaying the processor to avoid DTM overruns or delaying the processor to avoid both BTM and DTM overruns.

CBI is an optional control bit that, when enabled, gates a global, wired-OR breakpoint signal to the client. When the global breakpoint signal is asserted and the CBI is asserted, it causes a breakpoint to occur on the client. Each client should also wire-OR its breakpoint status output to this global breakpoint signal. When CBI is negated, the client will only break for breakpoint conditions internal to the client.

For embedded processors complying to class 2 or 3, the only development control field required is TM. For this case all other fields except the vendor-defined field shall be reserved and contain the same number of bits (see **Table B-6**).

Table B-6 Development Control Register

Bit Number	Field Name	Description
31–24	—	Vendor defined
23–15	—	Reserved
14	CBI	CBI - Client Breakpoint Input (optional) 0 = Break for internal breakpoints only 1 = Break for other clients' breakpoints also
13	DBE	DBE - Debug Enable (class 4) 0 = Debug mode disabled 1 = Debug mode enabled
12	DBR	DBR - Debug Request (class 4) 0 = Exit Debug mode 1 = Request debug mode
11	MS	MS - Memory Substitution (class 4) 0 = Use instructions and data in target memory 1 = Access instruction/data through AUX
10–9	SO	SO - Substitution Operands (class 4) 00 = Instructions and Data 01 = Instructions only 10 = Data only 11 = Reserved
8	SS	SS - Step Enable (class 4) 0 = Single step disabled 1 = Single step enabled
7–5	OVC	OVC - Overrun Control (class 4) 000 = Generate overrun messages 001 = Delay processor for BTM overruns 010 = Delay processor for DTM and OTM overruns 011 = Delay processor for BTM, DTM and OTM overruns 100–111 = Reserved
4–3	EIC	EIC - $\overline{\text{EVTI}}$ Control (class 2, 3, 4) 00 = $\overline{\text{EVTI}}$ for program and data trace synchronization 01 = $\overline{\text{EVTI}}$ for breakpoint generation 10 = No operation 11 = Reserved
2–0	TM	TM - Trace Mode (class 2, 3, 4) 000 = No Trace 1XX = BTM Enabled X1X = DTM Enabled XX1 = OTM Enabled

B.4 Development Status (DS) Register

When debug mode is entered the condition is detected by reading the DBS bit in DS, or by observing the Debug Status Message on the auxiliary pins. The SSS will also be set if debug mode is entered after a single step. The HWB and SWB also indicate if a hardware breakpoint (e.g. address comparator) or a software breakpoint (e.g. breakpoint instruction) caused the processor to halt and enter debug mode. The BPn bits indicate which breakpoint occurred.

Other conditions that may impact development support are detecting when the processor is in a Low Power mode or a non-recoverable hardware error has occurred. STP and HWE may be implemented to indicate these conditions.

The DS Register is read-only. All status bits are dynamic and do not require clearing.

This register is recommended for embedded processors complying to class 4. The contents of the DS Register are transmitted out the auxiliary pins upon a change in state of any bit (see **Table B-7**).

Table B-7 DS Register

Bit Number	Field Name	Description
31–24	—	Vendor defined
23–17	—	Reserved
15–8	BP7-0	BPn - Breakpoint Status 0 = No breakpoint 1 = Breakpoint occurred
7–6	—	Reserved
5	DBS	DBS - Debug Status 0 = Processor not halted 1 = Processor halted in Debug mode
4	STP	STP - Stop Status 0 = Processor not stopped 1 = Processor stopped in Low Power mode
3	HWE	HWE - HW Error 0 = No HW error 1 = Non-recoverable HW error occurred
2	HWB	HWB - HW Bkpt Status 0 = No HW breakpoint 1 = HW breakpoint
1	SWB	SWB - SW Bkpt Status 0 = No SW breakpoint 1 = SW breakpoint
0	SSS	SSS - Single Step Status 0 = Processor not halted 1 = Processor halted in Debug mode after single step

B.5 User Base Address (UBA) Register

UBA provides visibility for the development tool to determine what the setting is for the device-specific user base address. UBA is the memory map base address for user access to specific resources of the Nexus development port. If needed, UBA may be writable by the development tool to configure the memory map base address for user access.

User access to the Nexus development port is required for OTM and DQM, and reserved for other uses. The size of UBA is vendor defined (see **Table B-8**).

Table B-8 UBA Register

Bit Number	Packet Name	Description
Vendor defined	UBA	Device-specific user base address.

The memory map for user access of development features is shown in **Table B-9**, where offset is the base word size of the embedded processor.

Table B-9 Memory Map for User Accesses

Memory Map Location	Description
(UBA) + 2 x Offset	Reserved for future use.
(UBA) + 1 x Offset	Reserved for future use.
(UBA)	Ownership Trace Register.
(UBA) – 1 x Offset	Data Acquisition Control .
(UBA) – N x Offset	Location for DQMs where N is data ID.

The UBA register is recommended for embedded processors complying to class 2, 3 or 4.

Ownership Trace Register (OTR): OTR shall be provided only for general-purpose processor clients of embedded processors complying to classes 2, 3 and 4. OTR provides a register to which an operating system can write an ID for the current task/process. The size of OTR is vendor defined.

DQM: DQMs are achieved by user writes to appropriate locations in the memory map shown in **Table B-9**. The write information is queued up for messaging via the auxiliary pins. The location in the DQM portion of the UBA Register map that is written to determine the data ID tag for the message, with the exception of the Data Acquisition Control, which is used for DQM queue control.

DQM Data written to a location in the UBA Register map are queued up until a value of 0x0 is written to Data Acquisition Control, at which point the ID tag and data values are transferred on the auxiliary pins (refer to Data Messaging CODE in 4.2.10 Data Acquisition on Page 26). A DQM transfer is also started if the message queue fills up, or if another location in the DQM Register map is written prior to when 0x0 is written to Data Acquisition Control. In the event of the queue filling up prior to 0x0 being written to the address pointed to by UBA, subsequent writes to locations in the DQM portion of the register map will be stalled until queue space becomes available.

For simplicity of HW implementation, DQM IDs will be 2 or greater.

B.6 Read/Write Access Registers

The Read/Write Access feature provides DMA-like access to internal memory-mapped resources when the client is halted or during runtime. Three registers are used for the Read/Write Access feature:

- Read/Write Access Control/Status (RWCS)
- Read/Write Access Address (RWA)
- Read/Write Access Data (RWD)

The tool will write to a user memory map location by first updating the RWA and RWD Registers with the user address and data to be written, and then by updating the RWCS Register with the write access attributes. The tool will read from a user memory map location by first updating the RWA Register with the user address to be read, and then by updating the RWCS Register with the read access attributes.

These registers are recommended for embedded processors complying to class 3 or 4.

More detailed information on using the Read/Write Access feature is included in the following paragraph.

Access with the IEEE 1149.1 Interface: The IEEE 1149.1 state machine is shown in **Figure 9-1**. The value shown adjacent to each arc represents the value of the TMS signal sampled on the rising edge of the TCK signal.

1. For a block read the following sequence would be required:

    a. Initialize the Read/Write Access Address Register (RWA) through the IEEE 1149.1 access method outlined previously using the NRR index of 9 (see **Table B-3**).

b. Initialize the Read/Write Access Control/Status Register (RWCS) through the IEEE 1149.1 access method outlined previously using the NRR index of 7 (see **Table B-3**).

c. The read data will then be transferred to the RWD Register. When completed (without error), the Nexus block decrements the number in the CNT field and sets the DV bit. This indicates that the device is ready for the next access.

d. The data can then be read from the Read/Write Access Data Register (RWD) through the IEEE 1149.1 access method outlined previously using the NRR index of 10 (see **Table B-3**).

e. Once the RWD value has been read, the RWA will then be incremented to the next word of size SZ and Step 1c will be repeated. When the CNT field reaches a value of 0, the AC bit is cleared indicating the end of the read access.

2. For a block write the following sequence would be required:

a. Initialize the Read/Write Access Address (RWA) Register through the IEEE 1149.1 access method outlined previously using the NRR index of 9 (see **Table B-3**).

b. Initialize the Read/Write Access Data (RWD) Register through the IEEE 1149.1 access method outlined previously using the NRR index of 10 (see **Table B-3**).

c. Initialize the Read/Write Access Control/Status (RWCS) Register through the IEEE 1149.1 access method outlined previously using the NRR index of 7 (see **Table B-3**).

d. The Nexus block will then transfer the data value from the RWD Register to the memory-mapped address in the Read/Write Access Address (RWA) Register. When completed (without error), the Nexus block decrements the number in the CNT field and clears the DV bit. This indicates that the device is ready for the next access.

e. Repeat Step 2b until the CNT field has a value of 0. When this occurs, the AC bit will be cleared indicating the end of the write access.

Access with the Auxiliary Port: Refer to 6.4.10.1 Read/Write Access of Nexus Recommended Registers (NRRs)—Protocol Examples on Page 74.

B.6.1 Read/Write Access Control/Status (RWCS) Register

The SZ, RW, PR, MAP and CNT fields are written to by the tool to set up access attributes. The AC field is asserted by the tool to initiate an access or is negated by the tool to cancel an access in progress. The AC field is negated by the embedded processor upon completion of the access requested by the tool.

SZ and RW determine the access size and whether it is a read or write. The PR bits are intended to allow for implementations that perform a variety of access priorities, from a lowest-intrusive access (0b00) to a highest-intrusive access (0b11). The exact meaning of the encodings are vendor defined.

The MAP bits are intended to allow for multiple memory maps to be accessed. The primary processor memory map should be designated as the default (MAP = 000). Secondary memory maps, such as special-purpose processor memory maps, which are implemented in some processor architectures may also require access.

To request a block move, CNT is set by the tool to a value greater than 0. The address range for a block move is from RWA to RWA + CNT. The CNT field should not be decremented by the embedded processor during an in-progress block move. Upon completion of a block move the embedded processor should negate the AC field and set the CNT field to a value of 0.

If the RWCS Register is written to while any single or block access is in progress, the target will terminate the access, including any remaining block accesses, within one access cycle of the target. In this case, the access in progress when the RWCS Register is written is not guaranteed to complete (see **Table B-10**).

Table B-10 Read/Write Access Status Bit Encoding

DV	ERR	Read Action	Write Action
0	0	Read Access has not completed	Write Access completed without error
0	1	Read Access error has occurred	Write Access error has occurred
1	0	Read Access completed without error	Write Access has not completed
1	1	Not Allowed	Not allowed

If an error is generated during a block access, the block access will be terminated (see **Table B-11**).

Table B-11 Read/Write Block Access

Bit Number	Field Name	Description
31	AC	AC - Access Control 0 = End Access 1 = Start access
30	RW	RW - Read/Write 0 = Read access 1 = Write access
29–27	SZ	SZ - Word Size 000 = 8-bit 001 = 16-bit 010 = 32-bit 011 = 64-bit 1xx = Reserved
26–24	MAP	MAP - Map Select 000 = Primary memory map 001–111 = Other memory maps
23–22	PR	PR - Priority bb = Access priority
21–16	—	Reserved
15–2	CNT	CNT - Access Count hhhh = Number of accesses of word size SZ
1	ERR	Last access generated an error
0	DV	Data Valid in RWD

B.6.2 Read/Write Access Address (RWA) Register

The RWA Register is used by the tool to program the address of user memory-mapped resource to be accessed, or the lowest address (i.e. lowest unsigned value) for a block move (CNT > 0). The address range for a block move is from RWA to RWA + CNT.

The size of RWA is vendor defined (see **Table B-12**).

Table B-12 RWA Register

Bit Number	Packet Name	Description
Vendor defined	RWA	User memory-mapped address to be accessed.

B.6.3 Read/Write Access Data (RWD) Register

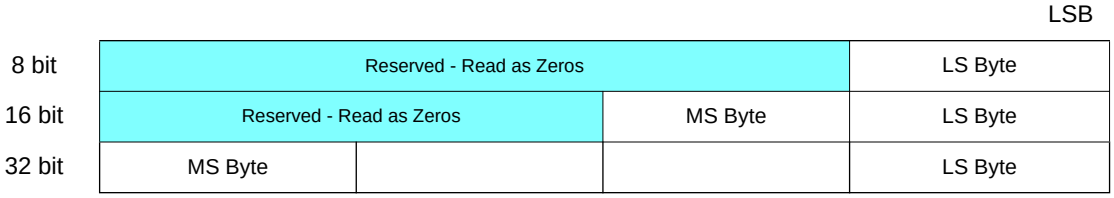
The RWD Register is used to contain the data to be written for the next block write access, and the read data for completed read accesses.

The size of RWD is vendor defined (see **Table B-13**).

Table B-13 RWD Register

Bit Number	Packet Name	Description
Vendor defined	RWD	Data read from a user memory-mapped location or to be written to a user memory-mapped location.

For read and write accesses the register may contain different sizes of data. The following is the organization for three different sizes of data.



B.7 Watchpoint Trigger (WT) Register

The WT Register allows the watchpoints defined in the breakpoint/watchpoint registers (refer to B.9 Breakpoint/Watchpoint Registers on Page 142) to be assigned to trigger actions. PTS and PTE select watchpoints to enable and disable program trace, effectively producing an address and/or data related “window” for triggering program trace. DTS and DTE select watchpoints to enable and disable data trace, effectively producing an address and/or data related “window” for triggering data trace. Program and/or data trace is triggered via the WT setting if the TM field (refer to B.3 Development Control Register on Page 130) has not already enabled program and/or data trace.

MSS selects a watchpoint to trigger memory substitution. (See **Table B-14.**) Refer to B.3 Development Control Register on Page 130 for additional fields related to memory substitution.

The WT register is recommended for embedded processors complying to class 4.

Table B-14 WT Register

Bit Number	Field Name	Description
31–29	PTS	PTS - Program Trace Start 000 = Trigger disabled 001 = 111 Use watchpoint 1–7
28–26	PTE	PTE - Program Trace End 000 = Trigger disabled 001 = 111 Use watchpoint 1–7
25–23	DTS	DTS - Data Trace Start 000 = Trigger disabled 001 = 111 Use watchpoint 1–7
22–20	DTE	DTE - Data Trace End 000 = Trigger disabled 001 = 111 Use watchpoint 1–7
19–17	MSS	MSS - Memory Substitution Start 000 = Trigger disabled 001 = 111 Use watchpoint 1–7
16–8	—	Reserved
7–0	—	Vendor defined

B.8 Data Trace Registers

The data trace registers allow DTM to be restricted to reads, writes or both for a programmable user address range. Three registers are used for selecting the data trace attributes:

- Data Trace Control (DTC)
- Data Trace Start Address (DTSA)
- Data Trace End Address (DTEA)

These registers are recommended for embedded processors complying to class 3 or 4.

B.8.1 Data Trace Control (DTC) Register

RWTn selects for each data trace channel (up to 6 data trace channels) if no trace messages are generated, or if reads, writes or both generate Data Trace Messages. If RWTn selects data trace for reads and/or writes, all selected accesses within the address range specified by DTSA to DTEA, end points inclusive, will generate Data Trace Messages (see **Table B-15**).

Table B-15 DTC Register

Bit Number	Field Name	Description
31–30	RWT0	RWT0 - Read/Write Trace 0 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
29–28	RWT1	RWT1 - Read/Write Trace 1 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
27–26	RWT2	RWT2 - Read/Write Trace 2 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
25–24	RWT3	RWT3 - Read/Write Trace 3 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
23–22	RWT4	RWT4 - Read/Write Trace 4 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
21–20	RWT5	RWT5 - Read/Write Trace 5 00 = No Data Trace Messages generated x1 = Enable data read trace 1x = Enable data write trace
19–8	—	Reserved
7–0	—	Vendor defined

B.8.2 Data Trace Start and End Address Registers (DTSA and DTEA)

The DTSA and DTEA Registers are used by the tool to program the start and end addresses for a data trace channel. If RWTn selects data trace for reads and/or writes, all selected accesses within the address range specified by DTSA to DTEA, end points inclusive, will generate Data Trace Messages.

The size of the DTSA and DTEA Registers are device specific (see **Table B-16** and **Table B-17**).

Table B-16 DTSA Register

Bit Number	Packet Name	Description
Vendor defined	DTSA	Start address for data trace visibility.

Table B-17 DTEA Register

Bit Number	Packet Name	Description
Vendor defined	DTEA	End address for data trace visibility.

B.9 Breakpoint/Watchpoint Registers

The breakpoint/watchpoint registers provide control for breakpoint and watchpoint logic. Three registers are used for controlling the breakpoints/watchpoints:

- Breakpoint/Watchpoint Control (BWC)
- Breakpoint/Watchpoint Address (BWA)
- Breakpoint/Watchpoint Data (BWD)

These registers are recommended for embedded processors complying to class 4.

B.9.1 Breakpoint/Watchpoint Control Register (BWC)

For all breakpoints to be enabled, DBE must be set to enable debug mode (refer to B.3 Development Control Register on Page 130). When debug mode is enabled, individual breakpoints can be enabled with BWE. Watchpoints are enabled with BWE regardless of the state of DBE.

BRW selects if a read, write or any access will cause a breakpoint. BME selects the data mask enable to be on a particular byte, half-word (2-byte) or word (4-byte) lane. Since the breakpoint data size unit is device specific, BSU is read-only to indicate to the tool if the data size unit is 1 byte, 2 bytes or 4 bytes. For example with 32-bit machines, the 4 most significant bits of BME may be reserved and the least significant bits may be used to enable masking of byte lanes (assuming BSU = 00). BWO selects the breakpoint operand as instruction or data, if the BWA and/or the BWD Registers are used for the breakpoint condition.

EOC selects if the breakpoint status indication is output on the EVTO pin (see **Table B-18**).

Watchpoints can be assigned actions listed in **Table B-14.**

If logical conditions of breakpoint or watchpoint detections are needed, or if counting *N* watchpoints is needed for development, the vendor-defined field can be defined to provide these or other features.

Table B-18 BWC Register

Bit Number	Field Name	Description
31–30	BWE	BWE - Breakpoint/Watchpoint Enable 00 = Disabled 01 = Breakpoint enabled 10 = Reserved 11 = Watchpoint enabled
29–28	BRW	BRW - Breakpoint/Watchpoint Read/Write Select 00 = Break on read access 01 = Break on write access 10 = Break on any access 11 = Reserved
27–20	BME	BME - Breakpoint/Watchpoint Data Mask Enable 1XXXXXXXX = Mask MS data size unit ⋮ XXXXXXXX1 = Mask LS data size unit
19–18	BSU	BSU - Breakpoint/Watchpoint Data Size Unit (read only) 00 = Data size unit is 1 byte 01 = Data size unit is 2 bytes 10 = Data size unit is 4 bytes 11 = Reserved
17–15	BWO	BWO - Breakpoint/Watchpoint Operand 1XX = Compare with BWA value X1X = Compare with BWD value XX0 = Compare for instruction types XX1 = Compare for data types
14	EOC	EOC - $\overline{\text{EVTO}}$ Control (optional) 0 = Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ 1 = Breakpoint/watchpoint status indication is output on $\overline{\text{EVTO}}$
13–8	—	Reserved
7–0	—	Vendor defined

B.9.2 Breakpoint/Watchpoint Address (BWA)

BWA is used to compare against address operands (address of instruction or data). The size of the BWA is vendor defined (see **Table B-19**).

Table B-19 Breakpoint/Watchpoint Address

Bit Number	Packet Name	Description
Vendor defined	BWA	Address of instruction or data for breakpoint or watchpoint generation.

B.9.3 Breakpoint/Watchpoint Data (BWD)

BWD is used to compare against data operands (instruction opcode or data value). The size of the BWD is vendor defined (see **Table B-20**).

Table B-20 Breakpoint/Watchpoint Data

Bit Number	Packet Name	Description
Vendor defined	BWD	Instruction opcode or data value for breakpoint or watchpoint generation.

B.10 Nexus Recommended Registers (NRRs) Concatenated for Better Transfer Efficiency

The NRRs may be concatenated as shown in **Table B-21** for better efficiency of transfers between the target and tool. For example, performing writes to configure the RWCS, RWA and RWD Registers to write a value to a user memory-mapped location requires only one Write Register Message instead of three (one for each register).

Table B-21 Nexus Recommended Registers (NRRs) Concatenated

Nexus Concatenated Registers	Register Index	Read/ Write
RWCS || RWA || RWD	55	R/W
BWC0 || BWA0 || BWD0	56	R/W
BWC1 || BWA1 || BWD1	57	R/W
BWC2 || BWA2 || BWD2	58	R/W
BWC3 || BWA3 || BWD3	59	R/W
BWC4 || BWA4 || BWD4	60	R/W
BWC5 || BWA5 || BWD5	61	R/W
BWC6 || BWA6 || BWD6	62	R/W
BWC7 || BWA7 || BWD7	63	R/W

For the Write Register Message or Read/Write Response Message, the REGVAL packet will contain the right-most register (LSB first), followed by the center register (LSB first), followed by the left-most register of **Table B-21**.

APPENDIX C
Data Acquisition in Tuning for Applications

For applications such as automotive powertrain, disk drive control and wireless, visibility of selected program variables (called calibration variables) must be provided to enable accurate tuning of selected program constants (called calibration constants). When calibration variables are stored in internal RAM, the data must be acquired from the embedded processor during runtime. Additionally, when calibration constants are stored in internal ROM, these constants must be tuned during runtime to determine the optimal values.

C.1 Data Acquisition or Measurement of Calibration Variables

Two options are explained in the following paragraphs to meet these data acquisition needs. The first utilizes DTM and the second utilizes the read/write access feature. Lastly, support in the Nexus standard for program tuning is explained.

DTM Option: A technique to accomplish data acquisition would be to set up a data trace window for all internal embedded processor memory-mapped locations which require acquisition. Depending upon the application, this window may include non-calibration data. Coherency (demarcating old data from new data) would be provided with a specific embedded processor data write sequence or a watchpoint occurrence and message. Care should be taken to assure that the data trace bandwidth requirements do not exceed the performance capability of the AUX.

Alternately, the embedded processor could queue up calibration variables for acquisition by the development tool by writing them to contiguous locations in a data trace window, e.g. contiguous locations in system RAM. Dedicated locations in the data trace window would be used to distinguish each group of calibration variables. Coherency would be provided with a specific embedded processor data write sequence or a watchpoint occurrence and message. Again, care should be taken to assure that the data trace bandwidth requirements do not exceed the performance capability of the AUX.

Read/Write Access Option: A technique to accomplish data acquisition would be to designate contiguous locations in a system RAM for all calibration variables. Calibration variables would be copied by the embedded processor from the source to these RAM locations prior to acquisition by the tool. A specific embedded processor data write sequence or a watchpoint occurrence and message would be used to signal the tool to acquire the calibration variables. The tool would acquire the calibration variables using the read/write access feature.

C.2 Tuning of Calibration Constants

The Nexus standard provides features to support program execution tuning, also referred to as calibration constant tuning. This is required when tuning electro-mechanical systems for a variety of loads, such as for automotive powertrain and disk drive applications.

The Nexus standard provides download capability for calibration constants to be tuned during runtime using a vendor-defined tuning block internal to the embedded processor. The read/write access feature provides access to vendor-defined blocks, either via the IEEE 1149.1 interface or the auxiliary pin interface, when the processor is halted or running. The auxiliary pin interface may be preferred for better performance capability, e.g. if simultaneous tuning and rapid prototyping are required.

Prior art solutions used as the vendor-defined tuning block include a bondout version of the embedded processor that allows an external RAM to overlay calibration constants in the internal ROM. The overlay RAM is accessible by the development tool. To provide coherency of modifications from the development tool, the overlay may comprise two identical RAMs, which are alternately enabled for overlay. The disabled RAM would be available to the tool for the latest tuning information, and would then be swapped in. For this prior art, all accesses could be managed by the development tool via the AUX.

APPENDIX D
Topics for Discussion

This appendix contains some ideas that have been discussed within the consortium. Your feedback would also be appreciated. The consortium web site at http://www.ieee-isto.org/Nexus5001/ can be used to provide feedback.

D.1 Minor Classification Changes/Ownership Trace Alternative Use

Successful adoption of this standard will be in part tied to the additional cost of implementing a chosen classification of Nexus on a vendor's embedded processor. Of all the Nexus features, the Data Trace feature may require the most additional silicon cost. Due to this, the Data Trace feature may not be acceptable for all applications. To help cover the need for basic data visibility if the Data Trace feature is not implemented, it has been suggested that the OTM feature be allowed to also support data visibility.

As defined in this specification, an OTM is generated by a client writing to a device-specific address the current process or task ID. It is suggested that writes of other types of operands also be allowed, such as the value for data parameters. To identify which data parameter is displayed by an OTM, the PROCESS packet (see SECTION 6) should include both an internal (or reduced) address for the data parameter and the value of the data parameter. To indicate if the OTM contains a process/task ID or a data parameter address and value, the upper portion of the PROCESS packet (as decoded by the embedded processor's API) should be zero or the lower portion should be the process/task ID.

Since the packet length for PROCESS is device specific, any packet length may be implemented.

Finally, if data trace is truly considered for only higher-scale embedded processors, it should be moved from class 3 to class 4.

D.2 Reduced Pin Count Option

Some applications require a special, extremely low pin count for the Nexus interface. This may be due to the small target board size or to environmental reasons. The ideal case would be two high-speed signals—one for input and one for output.

A suggestion has been made to evaluate if a reduced 4-pin Nexus interface may be supported. This interface would support the basic Nexus protocol but with fewer pins.

D.3 Multiple Nexus Compliant Embedded Processors on a Single Target Board

Some more discussion is required to ensure that this can readily be accomplished with the current definition of the Nexus AUX.

The designated approaches are as follows:

- Provide a separate board connector and tool for each embedded processor. This is not ideal since multiple tools are required.
- Provide a shared board connector and tool for all embedded processors, but with a separate $\overline{\text{EVTI}}$ connector signal for each embedded processor. The $\overline{\text{EVTI}}$ signals can be used to enable the Nexus interface on a single embedded processor, while the Nexus interfaces on the remaining embedded processors are disabled. A single tool can be used, however, only a single embedded processor can be debugged at one time (while all others are also running). There are currently no connector recommendations for this application.

Feedback is solicited to determine if these are the required use models for development with multiple embedded processors, and if there are any problems that may not have been considered.

Support for multiple clients on a single embedded processor is addressed by this standard. A single client may be selected for receiving messages to the embedded processor. The source client generating output messages may also be identified via a packet in applicable Public Messages.

D.4 High-Frequency Auxiliary Port

For AUX implementations which exceed 200 MHz, a reduced DC voltage specification will likely be required. Electrical specifications have been investigated and connector pin(s), in addition to those shown in **Table A-1**, may be needed for reference voltage.

D.5 Additional Functionality for Event-In Pin

Add the ability to configure $\overline{\text{EVTI}}$ to optionally force the client into Debug Mode, Start BTM, start DTM, etc. This allows for a tool to use an external system event, such as an analog signal value change, to force the configured clients into Debug Mode or Start Trace Messages, etc.

D.6 Add an Exception Status Packet to Indirect Branch Messages

With the current definition, there is no difference between an Indirect Branch Message transmitted because of a branch instruction and a change of flow caused by an exception event. It is proposed that a 1-bit packet be added to the Indirect Branch Messages to indicate whether or not the message was the result of an exception.

Currently when reconstructing opcode flow, the logic analyzer tool checks whether or not a branch opcode exists at the location where an Indirect Branch Message was transmitted. If not, an out-of-sync message is tagged on that location. This is done because of historical problems encountered when program files are revised and reloaded into the target system and the tool is not informed of the new program. Information in the program files displayed can thus be subtly different from what was actually executed by the processor, often causing customers to waste precious time chasing non-existent problems. With this 1-bit packet, the tool could still reliably inform the customer of a program file synchronization problem.

D.7 Additional use of $\overline{\text{RDY}}$ pin

There have been suggestions to allow for the $\overline{\text{RDY}}$ pin to function similarly to the $\overline{\text{MSEO}}$ pin. Using this suggestion, the IEEE 1149.1 interface may be used as a low-bandwidth substitute for the AUX.

When the Output Public Message Register has been selected and the $\overline{\text{RDY}}$ pin is asserted, the development tool provides the IEEE 1149.1 clock, and then the $\overline{\text{RDY}}$ pin behaves as per the $\overline{\text{MSEO}}$ protocol. The development tool shifts out the message via the IEEE 1149.1 TDO pin, as determined by the message shifted out and the $\overline{\text{MSEO}}$ protocol.

Once the development tool starts reading the Output Public Message Register, the $\overline{\text{RDY}}$ pin will provide packet end indication of variable length packets and message end indications. Thus the external tool will be able to determine the message length to shift out and the size of the variable length fields.

After the Output Public Message Register has been shifted out, the IEEE 1149.1 state machine need not be transitioned to IDLE, but can be stationed at SELECT-DR_SCAN until the next message ready indication is signaled on the $\overline{\text{RDY}}$ pin. This way, IEEE 1149.1 related overhead may be minimized to a few cycles.

APPENDIX E
References

IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture (Includes IEEE Std 1149.1a-1993).

The Evolution of Powertrain Microcontrollers and its Impact on Development Processes and Tools, Motorola/Hewlett Packard white paper.