



Arteris™
THE NETWORK-ON-CHIP COMPANY

Arteris od16681v1 – ©2016 Qualcomm Technologies, Inc. – Confidential and Proprietary Qualcomm Technologies, Inc. – Arteris:
subject to NDA – FlexNoC 3.1.1 – Observability – Designer's Manual – 18 February 2016

FLEXNoC 3.1.1

Observability

Designer's Manual

Qualcomm Interconnect Technology Center

www.arteris.com

Confidential and Proprietary – Qualcomm Technologies, Inc.

©2016 Qualcomm Technologies, Inc.

Not to be used, copied, reproduced, or modified in whole or in part, nor its contents revealed in any manner to others without the express written permission of the copyright holder.

Arteris, FlexNoC, FlexWay, FlexLLI, FlexArtist, FlexExplorer, and FlexVerifier, are trademarks or registered trademarks of Arteris, Inc. The Arteris logo and Arteris cover art is used with permission.

Qualcomm is a trademark of QUALCOMM Incorporated, registered in the United States and other countries. All QUALCOMM Incorporated trademarks are used with permission. Other product and brand names may be trademarks or registered trademarks of their respective owners.

od16440v1
Preliminary

About the FlexNoC Observability Designer's Manual

The *FlexNoC Observability Designer's Manual* is for system architects, hardware designers, and software programmers who need to understand and implement FlexNoC interconnect observation technology.

FlexNoC documentation is available directly through the FlexNoC FlexArtist software help system, which you can access from the software by pressing the F1 key, or by clicking the corresponding command on the [Help](#) menu.

Most major parts in the help system are also available as printable PDF files located in the doc/PDF/ subdirectory of the FlexNoC software installation directory. The PDF version of the release note is located in the doc/ReleaseNote/ subdirectory. The FlexNoC software installation includes Adobe Reader, which you can use either to open the pre-compiled index file *twpIndexReleasePDF.pdx* included with the PDFs, or to compile your own index by selecting option [All PDF documents in](#), and specifying subdirectory doc/PDF/ in the reader search dialog box.

PDF searchable index of PDF files (../PDF/twpIndexReleasePDF.pdx)

PDF version *About Arteris Publications*

(../PDF/twpAbout_Arteris_Publications.pdf)

PDF version *FlexNoC Release Note*

(../ReleaseNote/rnpFlexNoC_ReleaseNote.pdf)

PDF version *FlexNoC Getting Started Guide*

(../PDF/ugpFlexNoC_Getting_Started.pdf)

PDF version *FlexNoC Concepts* (../PDF/dmpFlexNoC_Concepts.pdf)

Designer's Manuals

PDF version *FlexNoC Addressing and Security*

(../PDF/dmpFlexNoC_AddressingAndSecurity.pdf)

PDF version *FlexNoC Clocking and Reset*

(../PDF/dmpFlexNoC_ClockingAndReset.pdf)

PDF version *FlexNoC Memory Units* (../PDF/dmpFlexNoC_MemoryUnits.pdf)

PDF version *FlexNoC NIU Transaction Handling*

(../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf)

PDF version *FlexNoC Observability* (../PDF/dmpFlexNoC_Observability.pdf)

PDF version *FlexNoC Physical Implementation*

(../PDF/dmpFlexNoC_Physical_Implementation.pdf)

PDF version *FlexNoC Power Management*

(../PDF/dmpFlexNoC_Power_Management.pdf)

PDF version *FlexNoC Quality-of-Service* (../PDF/dmpFlexNoC_QoS.pdf)

PDF version *FlexNoC Run-time Configuration*
(../PDF/dmpFlexNoC_Run-time_Configuration.pdf)

PDF version *FlexNoC Resilience Features*
(../PDF/dmpFlexNoC_ResilienceFeatures.pdf)

Configuration Guides

PDF version *FlexNoC FlexLLI Controller Configuration*
(../PDF/cgpFlexNoC_FlexLLI_Controller_Configuration.pdf)

PDF version *FlexNoC FlexLLI PSI Controller Configuration*
(../PDF/cgpFlexNoC_FlexLLI_PSI_Controller_Configuration.pdf)

Software User's Guides

PDF version *FlexNoC FlexArtist Software*
(../PDF/ugpFlexNoC_FlexArtist_Software.pdf)

PDF version *FlexNoC FlexArtist Exploration*
(../PDF/ugpFlexNoC_FlexArtist_Exploration.pdf)

PDF version *FlexNoC FlexVerifier Software*
(../PDF/ugpFlexNoC_FlexVerifier_Software.pdf)

od10123v1
Preliminary

Contents

About the FlexNoC Observability Designer's Manual

iii

Concepts	1
Error detection, logging, and reporting	1
Error sources and associated error processing	2
Unified and differentiated error modes	3
Error reporting and logging in observers	4
Packet and transaction probes	4
Universal probes	7
Probing transactions external to the NoC	7
Observer output	9
Time stamping the observation subsystem	9
Observation network	10
Observation-related security issues	10
Observation power management	10
Configuration	11
Creating an observation subsystem	11
Adding an observer	12
Choosing appropriate probe points	12
Adding and connecting probes	13
Probing on response paths	15
Configuring the observer	16
Configuring error logging	17
Configuring the observer output interface	18
Configuring probes	19
Packet tracing	19
Collecting statistics	20
Configuring probes for transaction profiling	21
Configuring probes for security filtering	22
Tuning the observation network	23
Serialization considerations	23
Clock domain crossing considerations	24
Buffering considerations	24
Power domain management considerations	24
General tuning recommendations	25
Operation	27
Logging errors	29
Reconstructing transaction addresses	29
Tracing packets	32
Basic packet tracing	32
Filtering packets	32
Tracing error packets	36
Combining filters	36
Configuring and managing trace alarm events	37
Collecting statistics	37
Counting packets over a fixed period	38
Configuring and managing the statistics alarm	38
Counting filtered packets over a fixed period	39
Measuring bandwidth	39
Transaction profiling operation	40

Measuring latency	40
Technical Reference	43
FlexNoC packet formats.....	43
Variable width packet fields and optional fields	44
Packet header fields.....	45
Packet payload fields	49
Routeld packet field structure and Addr values.....	50
Preambles	52
Observation probes.....	53
Error probe.....	53
Packet probe.....	53
Transaction filter	53
Transaction profiler	54
Universal probe functional description	54
Probe signals	58
Probe parameters.....	66
Probe registers	86
Universal Probe registers.....	115
Observers.....	115
Error logging features and codes.....	115
ATB formatting	116
STPv2 stream generation	121
Power disconnection of the observer output.....	125
Observer signals.....	125
Observer parameters.....	128
Observer registers.....	132
Glossary	141
Index	147

CHAPTER ONE

Concepts

IN THIS CHAPTER

Error detection, logging, and reporting	1
Packet and transaction probes	4
Universal probes	7
Observer output	9
Observation network	10
Observation-related security issues	10
Observation power management	10



This section describes Arteris FlexNoC observation debug support for the Arteris interconnect.

The technology, which allows activity within the NoC to be observed and reported, provides the following services:

- § Error detection, logging, and reporting.
- § Tracing packets.
- § Computing traffic or link statistics.
- § Profiling transaction latencies.
- § Reporting events or statistics to the system via industry standard third-party interfaces, such as ARM® ATB Coresight™, or MIPI® STPv2.

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

od10155v1

Error detection, logging, and reporting

Certain interconnect hardware components can generate errors in particular circumstances:

- § Generic initiator NIUs, when decoding addresses or applying security criteria.
- § Generic or specific target NIUs that are incapable of processing a particular initiator transaction.

§ The target IP itself, which can also return error responses to transactions.

If an initiator NIU detects an invalid transaction, it processes the transaction and sends it to the target normally, possibly to a target specifically chosen as an entry point for error logging of transactions with invalid addresses. Because the initiator NIU tags the requests as errored, the target NIU simply returns the transactions to their response ports without processing.

To catch all possible errors, whether generated by interconnect hardware units or target IPs, it is advisable therefore to probe the network downstream of targets. If several target response paths are multiplexed, a single probe point suffices to catch all errors for that group of targets.

For configurations with many targets but few initiators, such as in peripheral subsystems, the optimal solution may be to set probe points at the output of response links, just upstream of initiators (response inputs).

od2212v2

Error sources and associated error processing

NoC units can generate errors in particular circumstances:

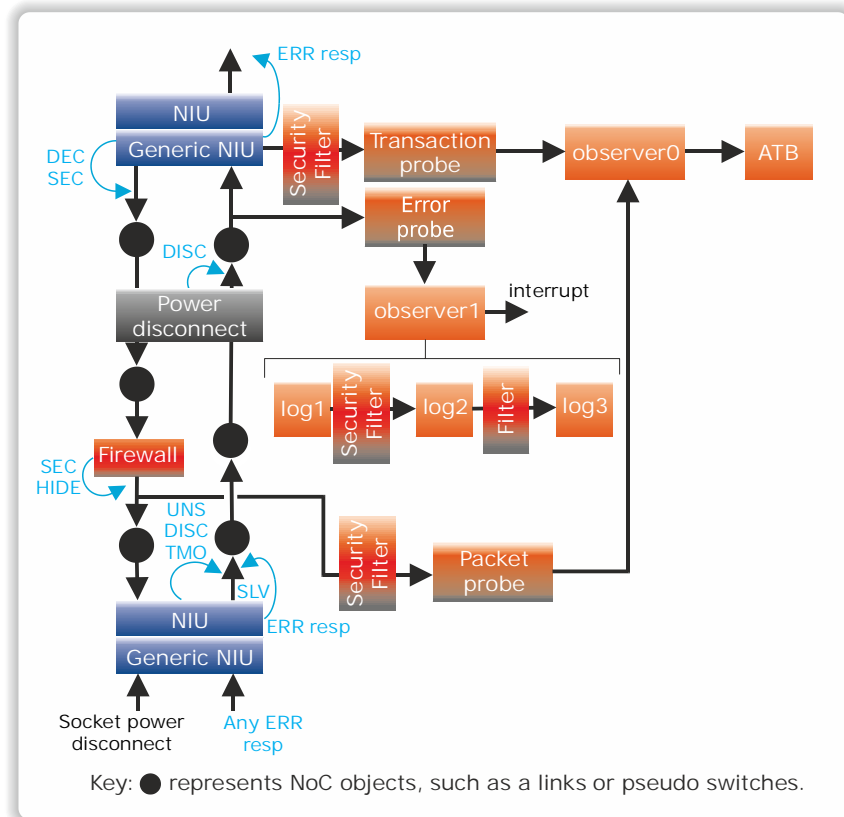
- § Generic initiator NIUs, when decoding addresses or applying security criteria.
- § Security firewalls, when detecting illegal packets.
- § Generic or specific target NIUs that are incapable of processing a particular initiator transaction.
- § Target IP itself, which can return error responses to transactions.

When an initiator NIU or a firewall detects an invalid transaction, it is tagged with an error code and sent to a target NIU, which forwards it directly to its response output, bypassing the target. Target NIUs also forward error responses from the target IP cores on their response output.

Probing the response output of an NIU catches all possible errors from any initiator to that target. If several target response paths are multiplexed, a single probe point suffices to catch all errors for that group of targets.

NOTE For NoCs with many targets but few initiators, probing errors at initiator response input is more efficient than at target response outputs, since fewer probe points are necessary.

Error sources within the NoC and associated error processing are described in the following figure.



ob8411v2

NOTE The instantiated error probes, packet probes, power disconnect, and so on, represent a small sample of the full range of FlexNoC technology, which includes, for example, request vs. response probing, ability to instantiate FW before power disconnect, and so forth.

In the figure, blue arrows indicate units that can detect errors, associated generated error codes (if in differentiated error mode), and packet output.

EXAMPLE 1 FW is a user-defined firewall. It checks its packet input against security criteria and assigns SEC or HIDE error codes to faulty packets.

EXAMPLE 2 When in disconnected state, power disconnect unit generates response packets with DISC error code.

The right-hand side of the diagram describes the security aspects of error logging and forwarding to the ATB interface.

od10157v1
Preliminary

Unified and differentiated error modes

FlexNoC error detection, reporting, and logging can be set in one of two modes:

- § Unified error type (default).
- § Differentiated error types.

NOTE The differentiated mode is obtained by setting parameter `useErrorCodes` to `True` (Specification: Global parameters).

In unified error type mode, a single Boolean status bit ERR is used to mark packets in error.

In differentiated error codes, the packets also carry an error code qualifier that identifies the error type, as shown in the following table.

Code	Value	Source	Type
SLV	0	Target	Target error detected by slave.
DEC	1	Initiator NIU	Address decode error.
UNS	2	Target NIU	Unsupported request.
DISC	3	Power Disconnect	Disconnected target or domain.
SEC	4	Initiator NIU or Firewall	Security violation.
HIDE	5	Firewall	Hidden security violation, reported as OK to initiator.
TMO	6	Target NIU	Time-out.
RSV	7	None	Reserved.

Among other purposes, error codes can be used by observation network components to steer error processing—for example, to log secure transaction errors in a separate, secure logger.

Error codes can also be used during the SoC software debug stage to more easily identify error causes.

The special HIDE code enables firewalls to hide errors from initiators. An initiator NIU that receives a HIDE error code will silently drop the error and respond to the initiator as if the error never happened.

NOTE In-band error reporting to NoC initiators through the socket protocol is not differentiated: a single code is returned to the initiator.

od10158v1

Error reporting and logging in observers

FlexNoC error probes detect packets in error and forward them to the associated observer via the observation network. The error packets are recorded by error loggers in the observer. An error logger can be programmed to assert an output signal when an error packet is received. The signal can be used as a system interrupt. Observers can contain one or more loggers and associated interrupt signals, each dedicated to specific error codes or error security levels.

By default, a new error can only be logged when the previous one is cleared by proper register access. If an error probe reports new errors and the associated logger is already full, then subsequent errors are dropped. Alternatively, the error logger can be programmed to queue up error packets and only accept them once the currently logged error has been cleared.

Several observers can be configured according to a user-defined strategy. For example, an error logging subsystem and associated observer can be defined for each NoC power domain.

od18528v1

Packet and transaction probes

FlexNoC packet probes provide the following features:

- § Run-time programmable selection of packet probe points.
- § Run-time programmable filtering for packet tracing.
- § Recording of traffic and link statistics.
- § Event counting of signals external to the NoC.
- § Generation of trace or statistic packets, or both, for the observation network.
- § Optional time stamping of packets.

FlexNoC transaction probes provide the following features:

- § Run-time programmable selection of transaction probe points.
- § Transaction profiling in terms of latency or number of pending transactions.
- § Event counting of signals external to the NoC.
- § Generation of statistic packets for the observation network.
- § Optional time stamping of packets.

od10160v1
Preliminary

Universal probes

od14785v1
Preliminary

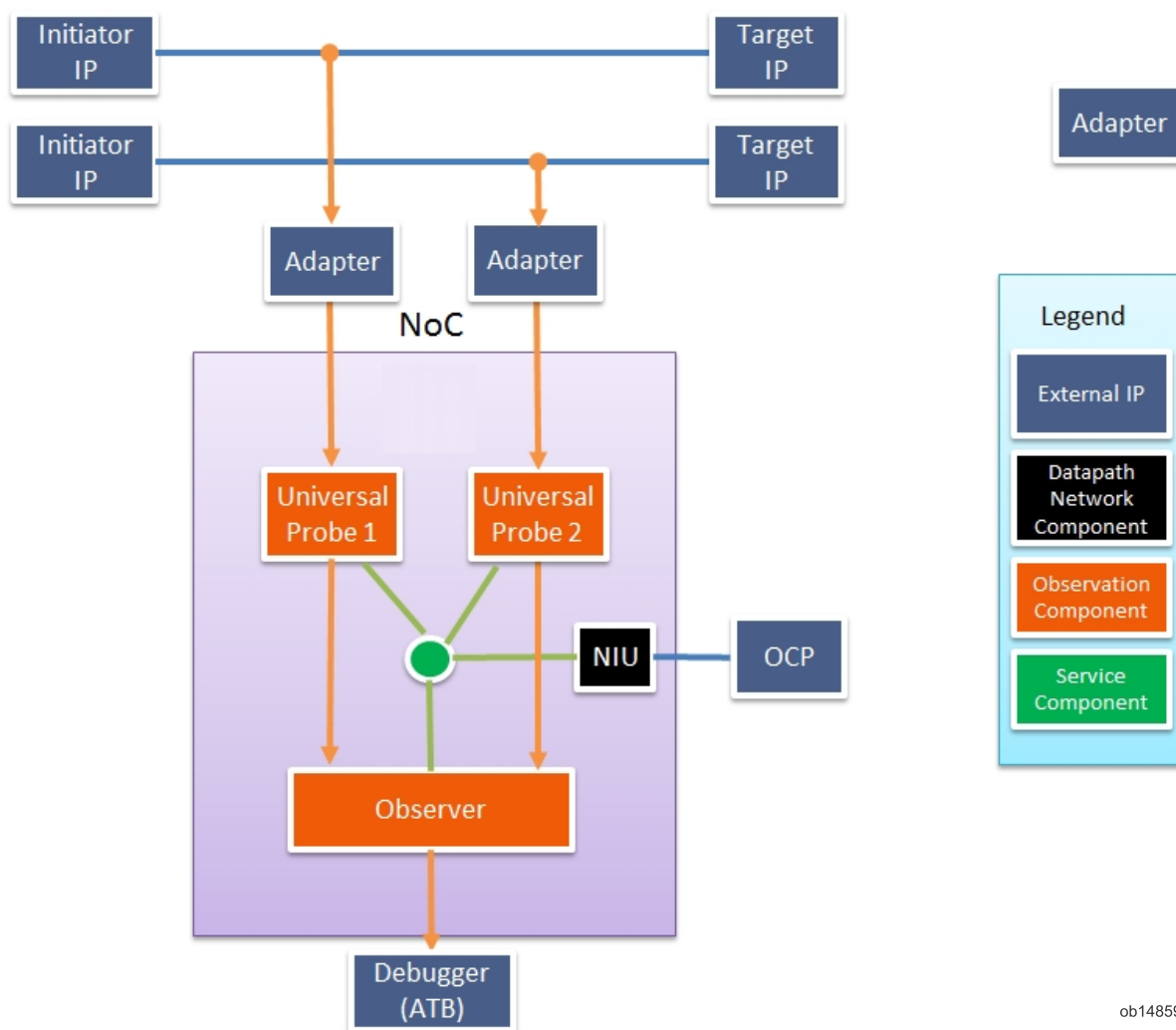
Probing transactions external to the NoC

Standalone probe system

The universal probe can also be used to monitor transactions between IP modules even when no FlexNoC interconnect is used for the system interconnect.

In this case, a dedicated NoC can be created to contain the universal probes. Such a NoC must only include *internal* target NIUs, whether they be service target or other target NIUs. The flows to these target NIUs are used only as programming interfaces to internal interconnect components, such as firewalls.

The following figure shows an example diagram of such a "standalone" FlexNoC probe system.



In the diagram:

- § Two universal probes are instantiated, which observe separate transaction paths between the external IP modules.
- § Both universal probes are connected to the same observer.
- § An OCP target port provides register access.

A standalone probe system can implement the following features, and their ancillary signals, to augment available observability operations in an interconnect:

- § CoreSight-compliant triggers
- § Alarms
- § External events
- § ATB encapsulation
- § Statistic suspend input control

More information about these features is provided in related reference documentation.

The following subsections cover issues pertinent to standalone probe systems as well as observation systems used in FlexNoC interconnects.

RELATED REFERENCES

CoreSight cross-triggering signals ...	59
Configuring and managing the statistics alarm	38
Configuring and managing trace alarm events	37
Encapsulation of ATB packets	123
Collecting statistics	20

od14790v1
Preliminary

Probe identification

Each universal probe instantiated in a standalone probe system should have a unique probe identifier, specified at design time. The ATB ID of the associated observer, which can be set at run time, should also be unique within the system. This allows unique identification of the probe source in the ATB traffic out of the standalone probe system.

Clock considerations

od14864v1
Preliminary

Clock domains boundaries, whether synchronous or asynchronous, can be freely traversed within the observation network, as well as within the datapath and service networks. Hence, the ATB endpoint can run on its own clock separately from the universal probe clock.

Probe request ports must use the same clock as the probe to which they are connected, whereas transaction ports can use a clock different from the probe clock.

All clock domains, including optional separate clocks for the transaction ports, are managed through the FlexNoC clock manager units whose content is defined by the integrator according to the desired clock/reset methodology.

Each unit instantiated in the standalone probe system has its own separate clock gating. Hence, the clocks to the observation units are not activated until a request or transaction event occurs on a universal probe port or a service access is made to configure a component.

Observer output

An observer can be connected to the host system via an ATB™ interface, with optional MIPI STPv2 encapsulation and time stamping. The probed data can be forwarded either to an off-chip interface or to an SoC storage buffer for analysis by an on-chip debugger.

NOTE When an observer is only connected to error logging probes, no debug output is required.

od10161v1

Time stamping the observation subsystem

To assist debug, time stamps can be added to the observation subsystem. Observed events are time stamped in packet probes. The observer sends the time stamp, relative to a previous time stamp, with the probed data in STPv2 packets. The observer also sends a time stamp reference, which allows the SoC debug module to determine the time at which each probe event occurred.

Time stamping probed events

The system time stamp is supplied by the SoC via a probe input signal. The probe adds the time stamp to the observation packet as it exits the probe onto the observation network. Because the number of pipeline stages in the probe is limited and fixed, there are only a few cycles of uncertainty in the time stamp value inserted in the packet, relative to the actual timing of the probed event.

The time stamp is added to the observation packet, regardless of whether the probe is configured for tracing or statistic collection.

The time stamp remains attached to the observation packet until the packet reaches the observer. If the observer debug output is configured in STPv2 mode, the time stamp value is extracted. The difference between the current time stamp and the previous time stamp value, which was sent in the STPv2 stream, is calculated. This delta value is appended to the first STPv2 packet embedding the ATB packet.

Providing a time stamp reference

The observer can include a time stamp in the STPv2 synchronization sequence, thus allowing the SoC debug module to resynchronize to this reference. This time stamp is supplied to the observer on a dedicated time stamp input signal. This signal should be the same as the one used to supply the time stamp to the observer probes.

RELATED REFERENCES

Synchronization sequence generation
..... 121

od10179v1

Observation network

Observation data is routed from probes to observers over a dedicated observation network. The network can cross clock and power domains, be pipelined and comprise FIFOs, rate adapters, and serialization adapters, all of which are configured like an ordinary transport datapath.

The actual bandwidth required by the observation network depends upon its usage. For example, if only error logging is performed, virtually no bandwidth is required. Even if it is used to convey statistics information, as well as errors, the required bandwidth is still minimal, depending on the number of probes, the number of counters in each probe and the frequency of the statistic reports. However, packet tracing demands higher bandwidth, particularly if payload data is traced, as well as header information.

od10162v1
Preliminary

Observation-related security issues

To avoid a security breach of the SoC through the use of the observation features, it may be necessary to block selected events from reaching the observation network or the observer, when in non-secure debug mode.

This can be achieved through security functionality at the inputs of probes or observers. These security filters can prevent packets with certain characteristics from entering the corresponding probes or to be logged or forwarded to the ATB output. Several standard criteria can be used for such filtering; user-defined logic can also be used if complex criteria are necessary.

od10163v1
Preliminary

Observation power management

An observation subsystem may be:

- § inside the same power domain as its probe points: for example an observation subsystem that logs error when and only when the probed domain is also active,
- § inside a different power domain : for example, an observation subsystem that traces packets and compute statistics only when the SoC is under control of a debugger, saving power in application-oriented operation modes.
- § Partitioning the observation subsystems into power domains is user-defined.
- § In addition, all observation units benefit from the same clock gating power optimization techniques than the other FlexNoC units.

od10164v1
Preliminary

CHAPTER TWO

Configuration

IN THIS CHAPTER

Creating an observation subsystem	11
Configuring the observer	16
Configuring probes.....	19
Tuning the observation network	23



Design-time configuration involves creating an observation subsystem, configuring observers, probes, and timestamps, and tuning the observation network.

In this section, numerous configuration examples refer to parameters that are described in more detail in related reference documentation.

od10165v1

RELATED REFERENCES

Observer parameters.....	128
Probe parameters	66

Creating an observation subsystem

FlexNoC observation subsystems comprise FlexNoC probes, observation network components, and observer units. The subsystem processes information from *probe points* and forwards it to the observer or reports it to the system using registers and interrupt signals.

A FlexNoC interconnect design can contain several independent observation subsystems, each focusing on different tasks such as error logging, or different locations such as power domains.

Once configured with FlexNoC FlexArtist software, subsystem units can be modified, and probe points and probes can be added or removed as required by changes to the specification and architecture in the probed datapath.

Although observation subsystem units can be added in any order with FlexNoC FlexArtist software, the preferred method is to follow the sequence of topics in this section, which makes correct probe configuration more straightforward.

od10166v1

Adding an observer

The first step in creating FlexNoC observation subsystems is to add an observer unit in order to log errors, or encapsulate traced packets of statistics packets for forwarding to supported third-party, industry-standard interfaces.

An observer unit is not required if you only intend to configure probes for computing statistics whose results are to be consulted exclusively through software register access, instead of being forwarded to an off-chip interface.

Because they affect the boundary of the NoC, observer units are configured with FlexNoC FlexArtist software during the specification phase of the FlexNoC workflow.

⇒ *To create a new observer unit*

- 1 Open the **Observation** page of the FlexArtist design editor **Specification** view **Interface** section.
- 2 Right-click anywhere over the page, and on the shortcut menu that opens, point to **New**, then click **Observer**.
- 3 In the *New object* dialog box that opens, type the name of the new object, or click **OK** to accept the default.

FlexNoC FlexArtist adds the new object to the list in the first column of the page.

- 4 For the new observer unit, select a clock from the list for parameter *clock*.

NOTE The observer's clock can be in a clock regime that is different from the regime of the clocks of the associated probes.

- 5 If the observer is intended to log errors in addition to other possible functions, click the corresponding cell in parameter column *errorLoggers*, and add at least one list element.

or

If the observer is intended to log errors only, set parameter *debugOutput* to **None**, **ATB**, or **STPv2**, as required.

od10167v1
od10167v1

Choosing appropriate probe points

The following kinds of FlexNoC probe points can be used to provide information to probes:

Error probe point Always on the output of a link or NIU, that is, on the request side for initiator NIUs, and on the response side for target NIUs. Each probe point is connected to a single error probe created automatically by FlexNoC FlexArtist software.

Datapath probe point Always on the output of a link. Connected to packet probes. A single packet probe can be shared by several datapath probe points.

WARNING Probing more than one link on a given path with the same packet probe may create timing loops in synthesis. Probing both the request and response paths with the same packet probe may create a deadlock condition when the probe is configured to be intrusive (see "Adding and connecting probes" on page 13).

Memory scheduler probe point Allows probing downstream of the scheduler arbiter. Connected to a packet probe created automatically by FlexNoC FlexArtist software. Cannot be shared by any other probe point.

Transaction probe point Only available on initiator NIUs, and probes inside the NIU itself. Used for transaction profiling. Connected to a transaction probe.

IMPORTANT FlexNoC PDDs must contain at least one probe of an appropriate type in order to configure probe points with FlexNoC FlexArtist software.

Probe point selection matrix

The following table shows how to set FlexNoC FlexArtist architecture object parameters in order to implement a particular probe point.

Probe point	Link param. observation	Init. NIU param. reqObservation	Targ. NIU param. rspObservation
(None)	None	None	None
Error	Errors	Errors	Errors
Packet	Packets		Packets ^a
Transaction		Transactions	


Table notes

^a Only legal for memory schedulers.

è To add a probe point

- 1 Open the relevant page in the **Unassigned Variable** design editor (Architecture: Datapath Transport: Topology).
- 2 In the diagram, select a link, NIU, or memory scheduler on which you want to place a probe point.
- 3 Right-click the selected object: on the shortcut menu that opens, point to **Open with**, then click **Customizer**.
- 4 In the *Customizer* dialog box, add the appropriate probe point to the selected link or NIU by setting the relevant observation group parameters and sub-parameters.

è To display probe points

- 1 Open the relevant page in the **Unassigned Variable** design editor (Architecture: Datapath Transport: Topology).
- 2 On the commands menu , click **Configure view**.
- 3 In the dialog box that opens, locate the *Object content* area, then in sub-area *Probe*, click **Display**.

od19560v1

od19560v1
od19560v1

Adding and connecting probes

An error probe is automatically generated when a probe point of type **Errors** is created. If the observation subsystem is only intended to log errors, skip to the next step.

A probe definition is required for probe points of type [Packets](#) and [Transactions](#). Several probe points of the same type can share the same probe hardware, reducing the overall probing subsystem cost.

NOTE A probe defined for a memory scheduler probe point cannot be used by other probe points

Probes are created and configured in the design editor [Architecture](#) view (Architecture: Observability: Probe) by right-clicking and selecting [New](#), then [Probe](#).

The [Probe](#) page also lists the probe points by type. If some probe points of type [Transactions](#) and [Packets](#) have not yet been connected to a probe, assign their parameter *probe* to one of the defined probes in this page.

NOTE Detailed probe configuration should not be attempted until all probe points have been assigned a probe, because probe parameters depend on the type of their associated probe points.

The basic configuration of a probe includes the following steps:

- 1 Assign a clock source by setting parameter *clock*. The clock net of a Probe and all its [Packets](#) points, if any, must be the same, or at least they must be the same logical clock, the probe clock may be a copy of the clock of the probe points in order to be able to independently shut off the probe subsystem.
- 2 Assign a destination observer using the *observer* parameter. The value may be set to **None** only for statistics-only probes with results only looked up through S/W registers and interrupt signals.
- 3 In the case of a packet probe, set probe parameter *mode* to [Spy](#) or [Intrusive](#). When set to [Intrusive](#), the probe can inject flow control in the datapath if bandwidth is insufficient to output all packet tracing information through the observer. The probe never injects flow control in the datapath when in [Spy](#) mode.

NOTE Intrusive mode can offer a more detailed debug mechanism but since packet transport can potentially be blocked, a serious loss of functionality in the SoC may occur, leading to a possible security issue.

NOTE Even though the debug mode may be configured as Intrusive, the actual mode used at runtime can be programmed to use either Spy or Intrusive mode.

NOTE Spy mode makes the datapath fully independent from the probe and allows the probe to be in a different power domain than to its probe point.

- 4 Assign a *probed* to the probe. The *probed* is a probe identifier that will be encapsulated in all probe output packets, enabling the observation software to identify the source of the packet. All probes connected to the same observer must be assigned a different *probed* value. Different observers can be identified with their ATB ID, providing the observation software with unique source identification capability.

Viewing observation network connectivity

At this stage, the various units constituting the observation network and their connectivity have been defined. At any time, you can check the sources of observation packets (probes manually created in the [Probe](#) page and error probes automatically created by [Errors](#) probe points) and their observer destinations in the observation topology graphical display (Architecture: Observability: Topology).

The topology of this network, that is, the multiplexing scheme, serialization, clock domain crossings, and so on, should be tuned only after all units are correctly configured. See section [Configuring probes for security filtering](#) (on page 22) later in this document.

od10169v1
Preliminary

Probing on response paths

To save gates for context storage, target NIUs neither store the address and user flag fields from request packets, nor forward them to response packets unless there is an [Errors](#) or [Packets](#) probe point in a *response* path of the target NIU.

If there is a [Packets](#) probe point in a response path, the address field and all user flags are copied from request packets by the target NIU into response packets, enabling detailed filtering of packet probes.

If there is an [Errors](#) probe point in a response path and global parameter [keepAddressAndFlagsInResponsePacketsForErrorProbes](#) (Architecture: Global parameters) is set to `True`, the address field and all user flags will be copied by the target NIU into the response packets, for all types of errors.

Note that in the case of a DEC error, to guarantee that the full transaction address can be reconstituted from the logged address, a complete memory map must be defined. This can be achieved either using targets with [access](#) set to `None`, or assigning a default error target.

Alternatively, if [keepAddressAndFlagsInResponsePacketsForErrorProbes](#) is set to `False`, the copying of the address field and the user flags from request packets by the target NIU into response packets is determined by the type of error recorded and the source of that error. This behavior is summarized in the following table.

Error code	Source of error	Address copied to response packets?	User flags copied to response packets
SLV	Slave: AXI, ACELite AHB (ERROR) NSP (SLV)	No ¹	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²
UNS	Slave (NSP)	No ¹	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²

Error code	Source of error	Address copied to response packets?	User flags copied to response packets
	NoC	Yes	All user flags
DISC	Slave (NSP)	No ¹	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²
	NoC	Yes	All user flags
SEC	Slave (NSP)	No ¹	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²
	NoC	Yes	All user flags
DEC	Slave: AXI, ACELite NSP (DEC)	No ¹	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²
	NoC	Yes ³	All user flags
TMO	Slave (NSP) Timeout Unit ³	No	Selected individually in parameter <code>userFlagsKeptInResponsePacketsForErrorProbes</code> ²

Table notes

- 1 The request address information is no longer available in the target NIU once an error of type SLV is generated by the target response.
- 2 Typically only user flags that are used in the security filter of the error probe are selected, along with others that carry pertinent information to the logged errors. Parameter `userFlagsKeptInResponsePacketsForErrorProbes` is located in the design editor **Architecture** view (Architecture: Global parameters).
- 3 To ensure that the transaction address can be fully reconstituted from the logged address (see "Reconstructing transaction addresses" on page 29), it is strongly recommended to define a complete memory map. This can be achieved either by using targets with parameter `access` set to `None`, or by assigning a default error target.
- 4 The timeout unit is in the generic domain; there is no transport timeout. In all cases, probing request links is less expensive than probing response links because the address and user flags do not need to be copied into response packets by target NIUs. However, probing request links does not allow probing of errors that are generated by target NIUs or by targets.

od14243v1
Preliminary

Configuring the observer

FlexNoC observer configuration details are completed in the FlexArtist design editor **Specification** view **Interface** section **Observation** page.

od10171v1

Configuring error logging

If the observer is connected to Errors probe points, it will log error packets (request or responses or both, depending on the probe point location) caught by the error probes.

Without security filtering

If there is no security concern about logged errors, then the configuration is straightforward: create a single logger in the *erroLoggers* list, set with *Standard* filtering, and default values for sub-parameters, that is:

- § empty *ignoredErrorCodes*,
- § *flagFiltering* values, if any user flags have been defined in the specification, left at x,
- § *forward* set to *False*.

These settings imply that all errors are logged in the logger, regardless of the packet error code, or any packet user flag attribute value, and once logged they are not forwarded to the next stage of the observer — which would be the ATB output if any, since there is no other logger.

With security filtering

Security concerns for error logging may have several causes, such as:

- § Separating errors logged for "secure transactions" from "non-secure transactions".
- § Separating errors logged for "non-secure transactions" trying to illegally access secure memory regions, from transactions addressing unmapped regions.

And more generally, separately hiding or reporting and logging errors based on the packet error code, and transaction user attributes.

NOTE User flags can be made available in response packets for security criteria (see "Probing on response paths" on page 15).

To that end the observer can be configured with independent, chained error loggers. Each logger has its logging registers, which can be made accessible with different security levels, its output interrupt signals, which can be sent to different interrupt controllers, and its catches and logs error packets based matching defined filtering criteria. Packets matching the defined criteria are optionally forwarded to the next stage, while packets not matching the criteria are always forwarded to the next stage.

If an error packet reaches the last error logger (of a chain of 1 or more error loggers) before the ATB endpoint, the error packet may be routed to the ATB endpoint (if either there is no match with the security filter criteria of the logger or parameter *Forward* is set to *True*). In this case, the error packet is carried in the ATB packet using the same format as that for a Trace Packet (header only).

Example 1: Separating loggers for secure transactions and non-secure transactions

Assuming the "secure" information is a user flag named *secure*:

§ Logger 0 is set with **Standard filtering**, no *ignoredErrorCodes*, *flagFiltering* for userFlag *secure* set to 1, *forward* = False.

§ Logger 1 uses default values.

The first logger will catch error packets marked as secure, log them, and only pass the non-secure ones to the next stage, which can thus only log non-secure error packets.

Example 2: Separate loggers for security errors

A security error is defined as a firewall or basic security scheme setting a security (SEC) error code in the packets:

§ Logger 0 is set with **Standard filtering**, *ignoredErrorCodes* list includes all codes except SEC, default *flagFiltering*, *forward* = False.

§ Logger 2 is set with **Standard filtering**, no *ignoredErrorCodes*, default *flagFiltering*, *forward* = False.

The first logger will catch error packets with error code SEC, log them, and only pass the non-SEC ones to the next stage, which will log them.

Complex schemes can be achieved using combinations of error code filtering and flag filtering capabilities in **Standard Filtering** mode. The two filtering stages, however, are independent: first ignore some error codes, and then apply only mask/match flag filtering capability to the errors that were not already ignored.

Using the **User-defined filtering** mode overcomes these limitations, by allowing the user to provide an arbitrary combinatorial equation taking into account flags and error codes, as well as external sideband signals.

NOTE Security filtering in error loggers only applies to error packets, not to observation packets from packet probes. Packets from packet probes are always forwarded to the next stage by error loggers, and have their own security mechanism.

od10173v1
Preliminary

Configuring the observer output interface

The following procedure is used to configure the output of an observer for connection to a SoC debug system.

⇒ *To configure observer output for SoC debug systems*

- 1 In the **Observation** page of the FlexArtist design editor **Specification** view **Interface** section, set parameter *debugOutput* to ATB or STPv2.

NOTE Alternatively, you can set parameter *debugOutput* to None if the observer is only connected to Errors probe points: in this case, skip the rest of this procedure.

- 2 Set sub-parameter *wData*, jointly with the observer clock rate, in order to ensure sufficient throughput of debug data.
- 3 If *debugOutput* is set to STPv2 and the STP stream needs to include time stamps, use parameter *wTimeStamp* to define the width of signal *TimeStamp* (Specification: Interface: Observation: debugOutput).
- 4 If power domains have been specified, set *power* sub-parameter *IPpowerDomain* (Specification: Interface: Observation: debugOutput: power) to the power domain of the SoC debug module that is connected to the observer.

Also set the disconnect protocol with *power* sub-parameter *disconnect* (Specification: Interface: Observation: debugOutput: power).

od10172v1
Preliminary

Configuring probes

Probes can handle a variety of tasks depending on their probe points:

- § Packet filtering and tracing, for probes with a *packets probe* point.
- § Packet filtering, statistics accumulation, and output, for probes with a *Packets probe* point.
- § Transaction profiling, statistics accumulation, and output, for probes that have at least one *transactions probe* point.

Packet filtering resources are shared between the various uses of the same probe, as well as statistics accumulation and output. Consequently, a single probe may handle both tracing and statistics collection tasks.

It is common to separate packet probes from transaction profiling probes and merge their output in the observation network prior to the observer. Although this duplicates counters for statistics accumulation and formatting hardware, it simplifies probe resource allocation during programming while maintaining a single stream of data at the observer output.

Separate configurations require creating probes that respectively handle only transactions probe or packets probe points.

od10174v1

Packet tracing

A FlexNoC packet probe has at least one packet probe point that can be configured for packet tracing or statistics collection, or both.

è *To configure a packet probe for tracing*

- 1 Open the relevant page in the **Unassigned Variable** design editor (Architecture: Observability: Probe).

Set parameter *nFilter* to the number of filters required.

NOTE Filters are used to select packets based on run-time criteria. If *nFilter* is set to 0, all packets are accepted; packet tracing depends entirely on the value of field *TraceEn* in register *MainCtl*.

- 2 If filtering on user flags is required, set parameter *allowFilterOnUser* to True.

WARNING Filtering on many user flags can be expensive in gates.

- 3 If packet payloads need to be included in the trace as well as headers, set parameter *allowPayloadTracing* to True.

NOTE Probing payloads on request links only gives access to write data payloads, while probing payloads on response links only gives access to response data payloads.

Once configured, payload tracing can be activated at run time through register *MainCtl* field *PayloadEn*, although this may require a lot more bandwidth than packet header tracing.

RELATED REGISTERS

MainCtl	87
PayloadEn	88

Collecting statistics

è *To configure a probe to collect statistics*

- 1 Open the relevant page in the **Unassigned Variable** design editor (Architecture: Observability: Probe).
- 2 Set parameter *nFilter* to the number of filters required. Filters are used to select packets based on run-time criteria.
- 3 Set parameter *statisticsCollection* to True.
- 4 Set parameter *nStatisticsCounter* to specify the number of statistics counters to instantiate in the probe.

If the probe is used only by Transactions probe points, packet filtering is not used and *nFilter* is forced to 0.

If the probe is used by transaction probe points, which requires parameter *transactionProfiling* to be set to True, the minimum permissible value of *nStatisticsCounter* is the greater of the probe's associated transaction profiler parameters *nObservable* and (*nComparators* + 1). This constraint ensures that, as a minimum, the following transaction profiling scenarios are supported:

§ Events in all transaction profiling bins associated with a selected NIU can be counted.

§ Events in a selected bin for each of the observed NIUs can be counted.

To count events in all transaction profiling bins for *nObservable* NIUs simultaneously, *nStatisticsCounters* must be set to (*nObservable* × (*nComparators* + 1)).

- 5 Set the width of the statistics counters in parameter *wStatisticsCounter*.
- 6 Set parameter *statisticsCounterAlarm* to True if an alarm must be generated (see "Configuring and managing the statistics alarm" on page 38) when a statistic counter reaches a particular threshold upon a statistics dump.
- 7 Set parameter *allowStatisticsSuspend* to True if the ability to suspend statistics collection temporarily is required.

When set, the parameter adds an output port *StatAlarm* to the probe. If required, this port can be connected to a top-level NoC port to drive an interrupt.

This creates an input port *StatSuspend* on the probe, which should be connected port to a top-level NoC port. Statistics collection will be suspended when this port is driven to 1.

NOTE This function is typically used if the device can be put into a step-by-step mode under the control of a debugger. In such a mode, statistic results would be different and could cause the statistics alarm to unduly fire.

- 8 Set parameter *nExternalEvent* to the required number of event inputs to be counted, up to a maximum of 32.

External events are pulses from external sources, for example coming from a target IP. As for other event types, the mapping of external events to statistic counters is programmable at run time.

RELATED PARAMETERS

transactionProfiling	73
statisticsCounterAlarm	72

od12761v1
Preliminary
od12761v1
Preliminary

Configuring probes for transaction profiling

Probes that are used by at least one transaction probe point can measure transaction characteristics such as transaction delay, or the number of pending transactions. The type of measurement can be changed at run time.

Measured values are compared against a set of programmable thresholds and recorded in statistics counters. At the end of the measurement period, counter values can be used to construct a histogram of the transaction characteristics of interest.

NOTE Transaction probes can be used even when not connected to a FlexNoC observation network: alarms can be used to freeze the counters and read stable values.

Configuration example: measuring transaction delay

The following procedure configures a transaction probe to measure the spread of transaction delay of two NIUs, *init0* and *init1*, with the delay categorized as follows.

Histogram bin	Range of delays (clock cycles)
0	0 to 1
1	2 to 4
2	5 to 9
3	10 to 99
4	100+

The design assumptions are:

- § *init0* and *init1* have values of 2 and 16, respectively, set for parameter *nPendingTrans*.
- § *init0* and *init1* are mapped to transaction probe ports 0 and 1.
- § No external events are "connected" to the transaction probe.

è To configure measurement of transaction delay

- 1 Open the relevant page in the **Unassigned Variable** design editor (Architecture: Observability: Probe).
- 2 Check that parameter *transactionProfiling* is set to **True** for the probe of interest.
- 3 Set the number of NIUs to be observed simultaneously, *nObservable*, to 2.
- 4 Set the number of transaction profiling counters, *nCounters*, to 24 (four counters are required for *init0*, and 20 for *init1*).

NOTE Consult related reference documentation for complete parameter descriptions.

This allows all potential transactions to be tracked.

- 5 Set parameter *wCounters* to 7 to accommodate the upper delay threshold of 100.
- 6 Set the number of delay thresholds in parameter *delayThresholds* to 4. Assign threshold values to parameter list elements as follows:
delayThresholds(0) = 2
delayThresholds(1) = 5
delayThresholds(2) = 10
delayThresholds(3) = 100
- 7 Set parameter *nComparators* to 4 to create five histogram bins.
- 8 Set parameter *statisticsCollection* to True.
- 9 Set parameter *nStatisticsCounter* to 10 so that events in all delay bins from both ports of the transaction probe can be counted.
- 10 Set parameter *wStatisticsCounter* to a value sufficiently large to allow statistics counters to handle all delay events.

In the worst case situation when all transactions have the same delay, counters must be large enough to count the total number of transactions during the expected measurement period.

RELATED REFERENCES

Measuring latency 40

od10177v1

Configuring probes for security filtering

FlexNoC security filtering in probes is configured via probe parameter *securityFiltering*. Both standard and user-defined filtering schemes are supported.

When configured for standard filtering, packets are passed into the probe based on the following conditions:

- § Matching packet user flags.

EXAMPLE Secure transactions, labeled with a user flag *secure* set to 1, may be excluded from the probe by setting sub-parameter *secure* of parameter *flagFiltering* to 0, with the other flags at set to x.

- § An optional SPIDEN input indicating the security state of the debug subsystem.

NOTE When parameter *useSPIDENInput* is set to True, the security filtering mechanism is disabled when the external input *SPIDEN* is set to 1, indicating that the debug subsystem is in a secure state.

When configured for User-defined filtering, more complex schemes are possible, with dependencies between flags, or between one or more input sideband signals, or both. A customer cell is instantiated allowing an arbitrary combinatorial equation between selected packet user flags and sideband signals.

od10178v1
Preliminary

RELATED PARAMETERS

nFilter.....	67
securityFiltering	77
flagFiltering	77
flagFiltering	129
useSPIDENinput.....	78

Tuning the observation network

The observation network topology can be viewed and edited in the design editor [Architecture](#) view (Architecture: Observability: Topology).

Its connectivity is entirely fixed by the associations of probes to observers, defined in the previous steps.

The observation network is a packet-based network offering the same features as the datapath network with the following exceptions:

- § It is a "push-only" network from probes to observers (even though it may carry traces of datapath response packets). This implies special precautions before powering-off the associated power domain.
- § There may be no data payload in the packets: there is no data payload out of error probes, nor out of trace-only probes without payload data tracing.

od10180v1

Serialization considerations

The width of the packet headers in the observation network can be found by selecting the structure in the FlexArtist software Project Tree pane and clicking the observation Transport NTTP Package in the Information pane.

The width of each field is listed together with the total size of the header. The size of an observation packet header is similar to that of a datapath packet header. There are differences in the *Urgency*, *Echo*, and *Debug* fields, and an additional field may be present to carry the time stamp.

Serializing packets can reduce routing congestion when the observation bandwidth is very small, especially if there are many probes that are physically distributed in the floor plan. Observation bandwidth is small, for instance, when there are only logging probes, or when statistics are gathered infrequently.

NOTE When there is no payload, for example with only error probes, the serialization of the observation network defaults to 8-bit serialization. This setting saves a lot of wires, and to preserve it end-to-end, any inserted links in the topology should be set with *nBytePerWord*=1, *headerPenalty*=AUTO.

When the observation bandwidth is higher, then it is usually limited by the available bandwidth of the off-chip interface downstream of the ATB output.

The serialization of the network should be chosen so that each probe can provide data to the observer at the maximum observer output rate (which may be lower than ATB data width multiplied by observer clock rate), but not more.

The optimum serialization for each path may depend on respective clock rates of the probe and the observer. In general, probes are expected to run at least as fast as the observer, so that choosing the same serialization as the observer width is a reasonable trade-off.

od10181v1
Preliminary

Clock domain crossing considerations

Asynchronous domain clock domain crossings require FIFOs that are 6-word deep for full throughput. To reduce the cost of clock domain crossings, serialize and multiplex sources with common clock before crossing to a new clock domain (assuming that the sources are not too distant in the floor plan).

od10182v1
Preliminary

Buffering considerations

The bandwidth considerations that lead to serialization are only true when considering average bandwidth. In some cases, high peak bandwidth may be useful at probe outputs:

- § When tracing packets: several consecutive packets may match the programmed filter and exit the probe, headed to the observer. The resulting peak bandwidth may exceed the observer bandwidth.
- § When computing statistics: the statistics gathering is frozen while the statistics packet is being output from the probe. If this process takes too long, many events that should be counted will be missed.
- § When several probes are active together: packets may collide at a switch, and stop at the switch input while it is processing another input.
- § Configuring a FIFO on the path from the probe to the observer allows packets to be stored temporarily and processed at a slower rate by the observer. Only when the FIFO runs out of space will new incoming traced packets be dropped, or statistics collection be frozen, as before.

Such a FIFO can be dedicated to a probe output, or common to several probes, that is, several probe outputs connected to the same switch.

od10183v1
Preliminary

Power domain management considerations

Because FlexNoC observation networks are of the "push-only" type:

- § a probe does not know if a packet has reached the observer, and
- § the power controller for the observation domain cannot know if a packet is still in transit in the observation network.

Consequently, at worst, the packet could be an error packet that is logged and triggers an interrupt. The power domain in which the observer is located should not be shut down until such packets have reached the observer.

To allow enough time for a packet from a probe to transit the network, regardless of the packet type, an observer waits for a number of clock cycles set by parameter *interruptPropagationDelay* before allowing the power controller to shut down the observer power domain (Structure: Parameters: Nodes).

NOTE To avoid race issues, it is preferable to disable tracing and statistic collection probes by software before shutting down the probe power domain.

If the connection between a transaction probe point and its probe crosses power domains, any filtered events sent from the transaction filter to the transaction probe are ignored if the probe clock is turned off.

If the observation network between a probe and the observer endpoint crosses power domains, then parameter *powerStall* for the transport disconnect unit inferred by FlexNoC FlexArtist software must be set to `CONST_ERR`. In this case, when the destination domain is turned off, observation packets are dropped at the power domain boundary because there is no response path in the observation network to convey the error back to the initiator.

More information is available in related concept and reference documentation ([../PDF/dmpFlexNoC_Power_Management.pdf](#)).

od10184v1
Preliminary

General tuning recommendations

- 1 Create separate observers for error logging and tracing.
- 2 A separate observer for error logging should be created for each power domain. This ensures that whenever an error probe point is powered, any error packets can be logged.
- 3 The observer for tracing should be put in a power domain of type `SUPPLY`. In normal operation, this allows the observer logic to be powered off.
- 4 Check parameter *nBytePerWord* throughout the observation network, especially at clock boundaries (Architecture: Observability: Serialization). The default values may be larger than required.
- 5 Check the effect of parameter *headerPenalty* in the observation network on the SoC debug system (Architecture: Observability: Serialization). For example, if *headerPenalty* is set to `TWO`, an error logger will assert signal *Fault* one cycle later.

od19652v1

CHAPTER THREE

Operation

IN THIS CHAPTER

Logging errors	29
Tracing packets	32
Collecting statistics	37
Transaction profiling operation	40



This section describes Arteris FlexNoC observability operation, in particular, the run-time programming sequences required to control and extract information from the observation system.

Information in this section assumes that FlexNoC observation units have already been configured at design time (see "Configuration" on page 11) with FlexNoC FlexArtist software.

Detailed descriptions of registers mentioned in this section are available in the related technical reference section.

Gathering design reference information

To help you with programming, you can use FlexArtist software to export your FlexNoC design structure as a plain text file. This file provides datapath network information such as route IDs, packet probe points, and transaction ports.

Another helpful export is the design register map, which lists all implemented registers, including details of reset values, bit widths, and access types.

When you export design structures or register maps, FlexNoC FlexArtist software prompts you to create a file named by default according to the syntax:

StructureName.{**info**|**txt**}

where *StructureName* is the name of a valid FlexArtist structure object selected for the export operation, followed by the file name extension **info**, for design structure files, or **txt**, for design register maps. You can change the default name, if desired.

è *To generate the design structure file*

§ On the [Implementation](#) menu, point to [RTL output](#), then click [Export Structure information](#) to export the structure information file (.info).

è *To export the design register map*

§ On the [Creation](#) menu, point to [Export](#), then click [Register Map](#).

RELATED REFERENCES

Probe registers	86
Observer registers	132

Logging errors

An essential aspect of FlexNoC error logging is the reconstruction of initiator and target transaction addresses. This section describes the calculation procedure with an example. The particular case of decode errors is also explained.

od10187v1

Reconstructing transaction addresses

Because the logged address of the error packet, *Addr*, is only an offset within the address region for the initiator–target mapping concerned, the corresponding transaction address for the initiator or target must be reconstructed.

The mapping associated with the logged packet must first be identified by inspecting packet *Routeld* subfields *InitFlow*, *TargFlow*, and *TargSubRange*. From these subfield values, collectively known as the "aperture", the local address can be found.

The corresponding initiator or target transaction address can then be reconstructed from the sum of *Addr* and the respective initiator or target local address.

Identifying an initiator–target mapping associated with a logged packet

The initiator–target mapping that corresponds to the logged packet is found by extracting *Routeld* from register *ErrLog1*, and additionally, *ErrLog2*, in the case where *Routeld* exceeds 32 bits.

è *To export a structure information file*

§ On the [Implementation](#) menu, point to [RTL output](#), then click [Export Structure information](#) to export the structure information file (.info).

è *To identify an initiator–target mapping*

1 In the structure information file exported with FlexArtist software, locate the section "Datapath Routeld decomposition".

This section contains the bit ranges for the *Routeld* subfields *InitFlow*, *TargFlow*, and *TargSubRange*. Using these bit ranges, map the contents of *ErrLog1*, and *ErrLog2* if necessary, to these constituent subfields.

2 In subsection "InitFlow field" of the file, identify the initiator flow corresponding to the value of *InitFlow*.

3 In subsection "TargFlow field", identify the target flow corresponding to the value of *TargFlow*.

4 In subsection "Aperture values", identify the initiator–target mapping using the initiator and target flows identified in the previous steps, as well as the value of *TargSubRange*. Note the values of *Init localAddress* and *Targ localAddress*.

Calculating initiator–target transaction addresses

Once the initiator–target mapping associated with the logged packet has been identified, the complete transaction address, either from the perspective of the initiator or the target, can be calculated.

Initiator transaction address

From the perspective of the initiator, *InitAddr*, is calculated by:

$$\text{InitAddr} = \text{Init localAddress} \mid \text{Addr}$$

where:

Addr is the logged packet transport address *Addr* given in register *ErrLog3* and, additionally, *ErrLog4*, in the case where the address exceeds 32 bits.

Init localAddress is the value from the identified initiator-target mapping in subsection “Aperture values” of the structure information file.

IMPORTANT Parameters *localAddress* and *globalAddress* (Specification: Mappings: Memory) cannot be used to calculate the transaction address.

Target transaction address

The transaction address from the perspective of the target, *TargAddr*, is given by:

$$\text{TargAddr} = \text{Targ localAddress} \mid \text{Addr}$$

where the definition of the terms are the same as those for calculating the initiator transaction address.

RELATED PARAMETERS

keepAddressAndFlagsInResponsePacketsForErrorProbes 69

od14469v2
Preliminary

Address reconstruction example

In this example, an initiator *I* has a single flow, connected to target *T* also with a single flow, but two mappings of 64 KB each. The initiator mapping starts at a local address of 0x0 and is mapped to global address 0x0. The size of the address range is 4 GB.

One of the 64 KB mappings to target *T* starts at a local address of 0x0 and is mapped to global address 0x0. The second 64 KB mapping to target *T* starts at a local address of 0x0 and is mapped to global address 0x100000.

These mappings are specified in the FlexNoC FlexArtist design editor [Specification](#) view (Specification: Mappings: Memory). Upon export the values of *Init localAddress* and *Targ localAddress* are generated by FlexArtist and stored in the exported structure information file, as shown in the following figure.

Column		access	localAddress	globalAddress	mask	modes	readPermissions	writePermissions	powerAutoResponse	securityLabel	comment
Initiator											
I/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False			
Target											
T/T/0/0	ReadWrite	0x0	0x0	0xFFFF (64K)				False			
T/T/0/1	ReadWrite	0x0	0x10_0000	0xFFFF (64K)				False			



Excerpt from exported structure information file

Aperture values:

Init flow Targ flow Targ subrange : Init mapping **Init localAddress**, Targ mapping **Targ localAddress**

I/I/0	T/T/0	0x0	:	0	,	0x0	,	0	,	0x0
I/I/0	T/T/0	0x1	:	0	,	0x100000	,	1	,	0x0

ob14507v1

Consider the case where the error logger registers contain the following values for a logged packet:

ErrLog1 = *Routeld* = 1

ErrLog3 = *Addr* = 0x100

Decomposing *Routeld*, *Targ SubRange* is 1.

Therefore, the initiator transaction address is:

```
InitAddr = Init localAddress | Addr
          = 0x100000 | 0x100
          = 0x100100
```

Logged addresses for decode errors

When the type of logged error is an address decode error, that is, when register *ErrLog0* field *ErrCode* is DEC, then the transaction address is given directly by the packet header field *Addr* in *ErrLog3*, and optionally *ErrLog4*, provided that the following two conditions are met:

- § The address decode error was generated by the generic-to-transport side of an initiator NIU.
- § A default target is defined for the initiator associated with the error. Note that it is not necessary to specify the same default target for each of the initiators.

IMPORTANT It is strongly recommended to add a default target if error logging is configured.

od14471v1

In section "Aperture values" of the exported structure information file, the values in columns *Init mapping* and *Targ mapping* are set to "NA" (not applicable) for any mapping which references a target that has been designated as the default target for a given initiator. This indicates that the mapping is not defined by the user.

If no default target is defined, then the number of transport address bits that are logged is given by:

$$\log_2(\text{smallest target address range accessible by initiator})$$

od14474v1

Tracing packets

The examples in this section assume that the necessary parameters have been set during probe configuration (see "Packet tracing" on page 19).

od10189v1

Basic packet tracing

TIP Trace port numbering for a given packet probe is displayed in the Information pane when a packet probe is selected. This information can also be obtained from the structure information file.

To trace packets, the packet probe must be programmed as follows:

- 1 Set field *TraceEn* of register *MainCtl* to 1 to enable forwarding of traced packets to the connected observer.
Optionally set field *PayloadEn* of register *MainCtl* to 1 if the packet payload should be included in the trace.
- 2 Set register *TracePortSel* to the value corresponding to the probe point of interest.
- 3 Set field *GlobalEn* of register *CfgCtl* to 1.
- 4 Set register *FilterLut* to -1 (all bits set).

NOTE A lookup table acts as a simple gate when there are no filters.

This programming sequence is sufficient only when the probe is configured without filters, that is parameter *nFilter* is set to 0 (Architecture: Observability: Probe). For details on how to program probes with filters, see the following section.

od10190v1

Filtering packets

The total number of filters is set by parameter *nFilter* (Architecture: Observability: Probe). In the packet probe, define the selection criteria using the filter registers.

The following list comprises the complete set of filter registers, where the value for filter *n* ranges from 0 to (*nFilter* – 1). However, not all of these registers may be available, depending on the configuration of the packet probe.

<i>Filters_N_RouteIdBase</i>	<i>Filters_N_Status</i>
<i>Filters_N_RouteIdMask</i>	<i>Filters_N_Length</i>
<i>Filters_N_AddrBase_Low</i>	<i>Filters_N_Urgency</i>
<i>Filters_N_AddrBase_High</i>	<i>Filters_N_UserBase</i>

Filters_N_WindowSize *Filters_N_UserMask*
Filters_N_SecurityBase *Filters_N_UserBaseHigh*
Filters_N_SecurityMask *Filters_N_UserMaskHigh*
Filters_N_Opcode

Filtering on an address range

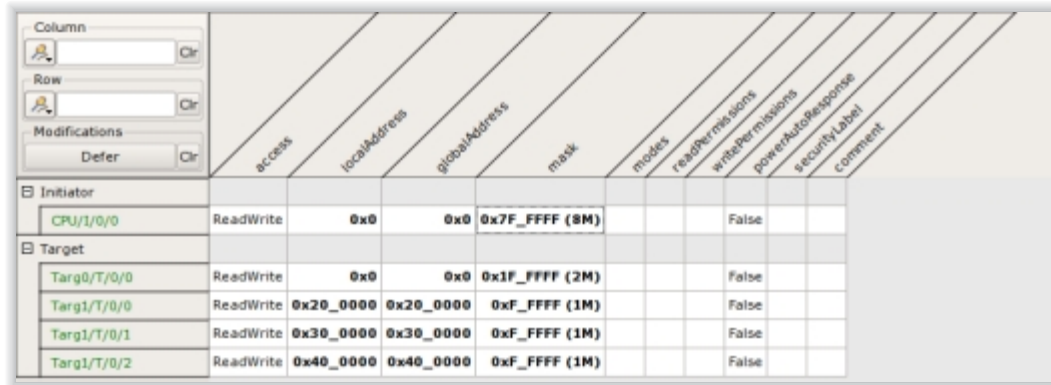
Packets which target a given address range of a specific mapping may be filtered by programming the set of filter registers as described in the table below:

Register	Value
Filters_N_RouteIdBase	RouteId subfields <i>InitFlow</i> , <i>TargFlow</i> , and <i>TargSubRange</i>
Filters_N_RouteIdMask	Set bits to "1" if <i>RouteId</i> fields are to be used by filter
Filters_N_AddrBase_Low/High	Address offset from initiator or target local address ¹
Filters_N_WindowSize	Log ₂ of address range

Table notes

- 1 The address offset is calculated with respect to either the initiator local address *Init localAddress* or the target local address *Targ localAddress*, depending on whether packets are to be selected from the perspective of the initiator or the target.
The values of *RouteId* subfields *InitFlow*, *TargFlow*, and *TargSubRange*, *Init localAddress* and *Targ localAddress* are provided in the structure information file (.info).
- è To export a structure information file
- § On the [Implementation](#) menu, point to [RTL output](#), then click [Export Structure information](#) to export the structure information file (.info).
- è To identify the route ID and local address information for the mapping
- 1 In the structure information file, locate the section "Datapath RouteId decomposition".
- 2 In subsection "InitFlow field", locate the value of *InitFlow* which corresponds to the initiator of interest.
- 3 In subsection "TargFlow field", locate the value of *TargFlow* which corresponds to the target of interest.
- 4 In subsection "Aperture values", locate the values of *TargSubRange* and *Init localAddress* or *Targ localAddress*, relevant to the address region of interest.
- 5 In subsection "wRouteId", note the bit ranges used by *InitFlow*, *TargFlow*, and *TargSubRange* in *RouteId*.

In the example NoC design below, in which one initiator *CPU* is connected to two targets *Targ0* and *Targ1*, it is required to select packets which target addresses ranging from 0x310000 to 0x318000. This address range falls within the second 1 MB mapping of *Targ1*, labeled *Targ1/T/0/1*, as shown in the memory map screen capture in the following figure (Specification: Mappings: Memory).



	access	localAddress	globalAddress	mask	modes	readPermissions	writePermissions	powerAutoResponse	securityLabel	comment
Initiator										
CPU/I/0/0	ReadWrite	0x0	0x0	0x7F_FFFF (8M)			False			
Target										
Targ0/T/0/0	ReadWrite	0x0	0x0	0x1F_FFFF (2M)			False			
Targ1/T/0/0	ReadWrite	0x20_0000	0x20_0000	0xF_FFFF (1M)			False			
Targ1/T/0/1	ReadWrite	0x30_0000	0x30_0000	0xF_FFFF (1M)			False			
Targ1/T/0/2	ReadWrite	0x40_0000	0x40_0000	0xF_FFFF (1M)			False			

ob14916v1

The route ID and aperture information in the exported structure information file is shown in the following figure. Note that in this example, *RouteId* does not include the sub-field *InitFlow* since there is only one initiator.

```

=====
Datapath RouteId decomposition
=====

wRouteId: 8 bits
-----
Field Type   : Bit(s)
-----
TargFlow      : 7
Targ SubRange : 6 .. 5
SeqId         : 4 .. 0

InitFlow field:
-----
value (0x) : flow
-----
0           : Flow:/Specification/CPU/I/0

TargFlow field:
-----
value (0x) : flow
-----
0           : Flow:/Specification/Targ0/T/0
1           : Flow:/Specification/Targ1/T/0

Aperture values:
-----
Init flow Targ flow Targ subrange : Init mapping , Init localAddress , Targ mapping , Targ localAddress
-----
CPU/I/0 Targ0/T/0 0x0             : 0           , 0x0             , 0           , 0x0
CPU/I/0 Targ1/T/0 0x0             : 0           , 0x200000    , 0           , 0x200000
CPU/I/0 Targ1/T/0 0x1             : 0           , 0x300000    , 1           , 0x300000
CPU/I/0 Targ1/T/0 0x2             : 0           , 0x400000    , 2           , 0x400000

```

ob14919v1

Program the registers of filter 0 as follows:

- Set the fields of *Filters_0_RouteIdBase* to:
Filters_0_RouteIdBase(7) = *TargFlow* = 1
Filters_0_RouteIdBase(6:5) = *TargSubRange* = 01
Filters_0_RouteIdBase(4:0) = *SeqId* = 00000
giving a value of 8'b10100000.

- 2 Set the bits of *Filters_0_RouteldMask* to 1 for those components of *Routeld* which should be used by the filter:
 $\text{Filters_0_RouteldMask} = 8'b11100000$
- 3 Program *Filters_0_AddrBase_Low* with the start address offset, with respect to *Targ localAddress*:
 $\text{Filters_0_AddrBase_Low} = 0x310000 - \text{Targ localAddress} = 0x310000 - 0x300000 = 0x10000$
- 4 Set the size of the address range in *Filters_0_WindowSize*:
 $\text{Filters_0_WindowSize} = \log_2(0x318000 - 0x310000) = \log_2(0x8000) = 15.$

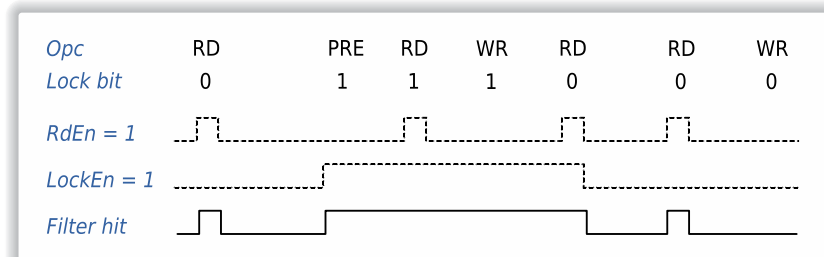
Selecting packets based on packet opcode

Packets can be selected based on the opcode field (*Opc*) in the packet header using register *Filters_N_Opcode*. In the following example, the filter is required to select all packets with *Opc* equal to RD and all packets belonging to a locked sequence.

Program the register fields of *Filters_N_Opcode* as follows:

Field name	Setting
RdEn	1
LockEn	1

The other register fields are set to 0. The following diagram shows a sequence of packets with the resulting filter hits.



ob14563v1

Excluding packet header fields from filter criteria

To ignore a specific packet header field in the filter, set its associated register with the value in the following table:

Register name	Setting
Filters_N_RouteldMask	0
Filters_N_WindowSize	-1 ¹
Filters_N_SecurityMask	0
Filters_N_Opcode	0xF
Filters_N_Status	0x3
Filters_N_Length	0xF
Filters_N_Urgency	0
Filters_N_UserMask (if present)	0
Filters_N_UserMaskHigh (if present)	0

Table notes

- 1 Setting register *Filters_N_WindowSize* to **-1** programs the filter to accept packets which target anywhere in the full address range supported by the transport network. Hence the address is effectively excluded from the filter criteria.

Tracing all packets

If it is required to trace all packets with a packet probe that is configured with filters, that is, when parameter *nFilter* is set to a value greater than **0**, then the probe's filtering function must be programmed as follows:

- 1 Program one of the filters to accept all packet header field values. Set the filter registers to the values given in the table in the preceding section "Excluding packet header fields from filter criteria".
- 2 Program register *FilterLut* to select the output of filter programmed in the previous step.

od10191v1
Preliminary

RELATED REFERENCES

Packet probe filter registers.....	95
Measuring latency.....	40

Tracing error packets

Error packets, that is packets with header field *Status* set to **ERR**, are not traced by default. To include error packets in the trace output, set register field *ErrEn* of register *MainCtl* to **1**, as well as field *ErrEn*.

od17348v1

Combining filters

A look-up table (LUT) is instantiated in the probe, which allows a hit to be generated on any logical combination of packet filters. The logic function is programmed via register *FilterLut*. The value programmed in the register corresponds to the truth table applied on the filter outputs.

For example, with two filters *F0* and *F1*, sixteen possible functions can be defined, a subset of which is shown in the following table.

Filter hit		Required filter combination					
F1	F0	F0	F1	F1 & F0	F1 F0	F1 ^ F0	~ F0
0	0	0	0	0	0	0	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	1	1
1	1	1	1	1	1	0	0
		Value to program in FilterLut					
		0xA	0xC	0x8	0xE	0x6	0x5

NOTE In this example, setting register *FilterLut* to **0xF** will generate a LUT hit for any combination of filter hits, effectively bypassing the filters.

od10192v1

RELATED REGISTERS

FilterLut.....	91
----------------	----

Configuring and managing trace alarm events

Trace alarm events can be separately enabled, recorded, and cleared using registers *TraceAlarmEn*, *TraceAlarmStatus*, and *TraceAlarmClr* respectively. In each register, a dedicated bit is associated with each filter hit; the MSB is associated with the LUT hit.

The trace alarm logic can be configured to drive output signal *TraceAlarm* if one or more trace alarm events are recorded in register *TraceAlarmStatus*. Signal *TraceAlarm* can be used directly as an interrupt source, or can be routed through a sideband manager.

⇒ *To use the trace alarm feature*

- 1 Enable hits from selected filters 0 to (*nFilter* – 1) to generate a trace alarm by setting register *TraceAlarmEn* bits 0 to (*nFilter* – 1) to 1.
- § Optionally, enable LUT hits to generate a trace alarm by setting the MSB of *TraceAlarmEn*.
- 1 Set field *AlarmEn* to 1 in register *MainCtl* in order to activate signal *TraceAlarm* when a trace alarm occurs.

If *TraceAlarm* is asserted, the interrupt service routine should:

- 1 Identify the source or sources of the trace alarm by reading register *TraceAlarmStatus*.
- 2 Clear the source or sources of the trace alarm by setting the corresponding bits to 1 in register *TraceAlarmClr*. Signal *TraceAlarm* will be de-asserted when all bits in *TraceAlarmStatus* have been cleared.

RELATED REGISTERS

TraceAlarmStatus.....	92
TraceAlarmClr.....	92
TraceAlarmEn.....	92

od10193v1
Preliminary

Collecting statistics

The following programming sequence examples assume that the necessary parameters have been set (see "Collecting statistics" on page 20).

RELATED REFERENCES

Collecting statistics.....	20
----------------------------	----

od10194v1

Counting packets over a fixed period

TIP Trace port numbering for a given packet probe is displayed in the Information pane when a packet probe is selected, as well as in the structure information file.

The following programming sequence counts packets at a given probe point using statistic counter 0.

- 1 Set field *StatEn* to 1 in register *MainCtl*.
- 2 Set register *Counters_0_PortSel* to the value corresponding to the probe point of interest.
- 3 Set register *Counters_0_Src* to 0x6 (PKT) to count packets.
- 4 Specify the period during which the packets should be counted by setting register *StatPeriod* to:
 $\log_2(\text{interval expressed in number of probe clock cycles})$
- 5 Set field *GlobalEn* of register *CfgCtl* to 1 to enable packet counting.

Once time $2^{\text{StatPeriod}}$ has elapsed, the number of packets counted is dumped to the observer.

od10195v1

Configuring and managing the statistics alarm

The following programming sequence asserts the statistics alarm signal *StatAlarm* if the number of packets exceeds a specified threshold when the statistics dump occurs.

- 1 Set register *StatAlarmMax* to the number of packets that should trigger the statistics alarm.
- 2 Set the counter alarm mode to *MAX* by setting register *Counters_0_AlarmMode* to 2. This triggers the alarm if the count value is greater than *StatAlarmMax*.
- 3 Set field *StatCondDump* to 0 in register *MainCtl* to generate a statistics alarm event and freeze the statistic counter value if the statistic counter exceeds the value of *StatAlarmMax* when a statistics dump occurs.
- 4 Set field *AlarmEn* to 1 in register *MainCtl* and ensure that bit 0 of register *StatAlarmEn* is set to its default value of 1 to activate signal *StatAlarm* in the event of a statistics alarm.

In the event that *StatAlarm* is asserted when the statistics dump period expires, the interrupt service routine should:

- 1 Read the number of packets contained in register *Counters_0_Val*.
- 2 Clear the statistics alarm by setting bit 0 to 1 in register *StatAlarmClr*. Signal *StatAlarm* will be de-asserted.

NOTE Statistics counters remain frozen and the dump timer is suspended until bit 0 is set to 1 in register *StatAlarmClr*.

Alternatively, the statistics alarm function can be configured to *not* drive signal *StatAlarm* when a statistics alarm event occurs. This signal can be disabled by clearing bit 0 of register *StatAlarmEn* to 0. Furthermore, because signal *TraceAlarm* is enabled by setting field *AlarmEn* in register *MainCtl*, *StatAlarm* can be disabled without affecting the behavior of *TraceAlarm*.

od22458v1

Counting filtered packets over a fixed period

The following programming sequence counts hits from packet filter 2 using statistic counter 0.

NOTE The important distinction between this sequence and the one in a related programming example (see "Counting packets over a fixed period" on page 38) is that the probe port is selected using register *TracePortSel* rather than *Counters_0_PortSel*

- 1 Set field *StatEn* to 1 in register *MainCtl*.
- 2 Set register *TracePortSel* to the value corresponding to the probe port of interest.
- 3 Set register *Counters_0_Src* to 0xE (FILT2) to count packets from filter 2.
- 4 Specify the period during which the packets should be counted by setting register *StatPeriod* to:
 $\log_2(\text{interval expressed in number of probe clock cycles})$
- 5 Set field *GlobalEn* of register *CfgCtl* to 1 to enable packet counting.

Once time $2^{\text{StatPeriod}}$ has elapsed, the number of filtered packets counted is dumped to the observer.

od11600v1

Measuring bandwidth

The following programming sequence example shows how a packet probe can be used to measure the average bandwidth, over a given interval, at a probe point.

Some important points to note about this example are:

- § Statistics counters are chained together to support the maximum theoretical bandwidth. Counter 0 is configured to count bytes; counter 1 increments when counter 0 rolls over.
- § The counter values are dumped to an observer after time $2^{\text{StatPeriod}}$.
- § A maximum bandwidth threshold is defined. A statistics alarm signal is asserted if this threshold is exceeded when statistics data is dumped. The comparison is made against the concatenated value of the two counters.

The programming sequence is as follows:

- 1 Set field *StatEn* to 1 in register *MainCtl*.
- 2 Set register *Counters_0_PortSel* to the value corresponding to the probe point of interest.
- 3 Set register *Counters_0_Src* to 0x8 (BYTES) to count bytes.
- 4 Set register *Counters_1_Src* to 0x10 (CHAIN) to increment when counter 0 wraps.
- 5 Specify the period during which the bytes should be counted by setting register *StatPeriod* to:
 $\log_2(\text{interval expressed in number of probe clock cycles})$
- 6 Set register *Counters_0_AlarmMode* to 0x2 (MAX) and *Counters_1_AlarmMode* to 0x0 (OFF).

A statistics alarm will be raised if the combined count of counter 0 and counter 1 is greater than the value specified in register *StatAlarmMax*.

- 7 Set register *StatAlarmMax* to a value corresponding to the upper bandwidth limit required to trigger the statistics alarm.
- 8 Set field *AlarmEn* to 1 in register *MainCtl*. Ensure bit 0 of register *StatAlarmEn* is set to 1 (default value) if the alarm output signal *StatAlarm* is required.
- 9 Set field *GlobalEn* of register *CfgCtl* to 1 to enable the counting of bytes.

In the event that *StatAlarm* is asserted, the interrupt service routine should:

- 1 Optionally read the number of packets contained in registers *Counters_0_Val* and *Counters_1_Val*.
- 2 Clear the statistics alarm by setting bit 0 to 1 in *StatAlarmClr*. The signal *StatAlarm* will be de-asserted.

NOTE The statistics counters are frozen and the dump timer is suspended until bit 0 is set to 1 in register *StatAlarmClr*.

od10197v1
Preliminary

Transaction profiling operation

The programming sequence examples in this section constitute, for a given category of use cases, the *obligatory* base from which to develop sequences to handle specific needs within the category.

od10198v1
Preliminary

Measuring latency

The following example of a run-time programming sequence shows how a transaction probe can be used to measure transaction latency. The example assumes appropriate probe configuration at design time (see "Configuring probes for transaction profiling" on page 21).

The measurement specification is:

- § For *init0*, monitor delays for read transactions only.
- § For *init1*, monitor delays for read and write transactions.

⇒ *To program the transaction profiler unit*

- 1 Ensure that register *En* is 0.
- 2 Set register *ObservedSel_0* to 0 to monitor transactions from *init0* on observed transaction port 0.
- 3 Set register *ObservedSel_1* to 1 to monitor transactions from *init1* on observed transaction port 1.
- 4 Set register *Mode* to 2'b00 to set the event mode to DELAY for both transaction ports.
- 5 Set register *nTenureLines_0* to 1 to allocate one "tenure line", that is, four transaction profiling counters, to *init0*. The remaining five tenure lines are automatically allocated to *init1*.
- 6 Set the threshold registers for transaction port 0. These select the actual delay thresholds, from the set of thresholds defined in parameter *delayThresholds*, which is used to categorize the delay in transactions issued by *init0*.

Thresholds_0_0 = 0 ⇒ *delay threshold*(0) = 2

$Thresholds_0_1 = 1 \Rightarrow delay_threshold(1) = 5$
 $Thresholds_0_2 = 2 \Rightarrow delay_threshold(2) = 10$
 $Thresholds_0_3 = 3 \Rightarrow delay_threshold(3) = 100$

- 7 Set the threshold registers for transaction port 1. These select the actual delay thresholds, from the set of thresholds defined in parameter *delayThresholds*, which is used to categorize the delay in transactions issued by *init1*.

$Thresholds_1_0 = 0 \Rightarrow delay_threshold(0) = 2$
 $Thresholds_1_1 = 1 \Rightarrow delay_threshold(1) = 5$
 $Thresholds_1_2 = 2 \Rightarrow delay_threshold(2) = 10$
 $Thresholds_1_3 = 3 \Rightarrow delay_threshold(3) = 100$

⑤ To program the packet probe statistics subsystem

The statistics counters in the packet probe unit of the transaction probe must be programmed to count the delay events recorded by the transaction profiler unit.

- 1 Map the five delay event bins, associated with transactions issued by *init0*, from the transaction profiler unit to statistics counters in the packet probe unit by programming the following registers:Ⓜ

Counters_0_Src = 0x20 => Counts the number of transactions from *init0* with a delay between 0 and 1.

Counters_1_Src = 0x21 => Counts the number of transactions from *init0* with a delay between 2 and 4.

Counters_2_Src = 0x22 => Counts the number of transactions from *init0* with a delay between 5 and 9.

Counters_3_Src = 0x23 => Counts the number of transactions from *init0* with a delay between 10 and 99.

Counters_4_Src = 0x24 => Counts the number of transactions from *init0* with a delay greater than 100.

- 2 Map the five delay event bins, associated with transactions issued by *init1*, from the transaction profiler unit to statistics counters in the packet probe unit by programming the following registers:

Counters_5_Src = 0x25 => Counts the number of transactions from *init1* with a delay between 0 and 1.

Counters_6_Src = 0x26 => Counts the number of transactions from *init1* with a delay between 2 and 4.

Counters_7_Src = 0x27 => Counts the number of transactions from *init1* with a delay between 5 and 9.

Counters_8_Src = 0x28 => Counts the number of transactions from *init1* with a delay between 10 and 99.

Counters_9_Src = 0x29 => Counts the number of transactions from *init1* with a delay greater than 100.

- 3 Set register *StatPeriod* to $\log_2(\text{period})$, that is, the period during which delay events should be counted before being sent to the observer.
- 4 Set field *StatEn* of register *MainCtl* to 1 to enable the statistics counters to count delay events.

⑤ To enable the transaction probe

- 1 Set field *GlobalEn* of register *CfgCtl* to 1 to enable the statistics counters.
- 2 Set register *En* to 1 to enable the transaction profiling counters.

è *To program the init0 NIU transaction filter*

- 1 Set register *Mode* to 1 to set the delay mode to LATENCY.
- 2 Set field *RdEn* of register *Opcode* to 1 to monitor read transactions only.

è *To program the init1 NIU transaction filter*

- 1 Set register *Mode* to 1 to set the delay mode to LATENCY.
- 2 Set fields *RdEn* and *WrEn* of register *Opcode* both to 1 to monitor read and write transactions.

od10199v1
Revised

RELATED REGISTERS

Mode 108

RELATED REFERENCES

Configuring probes for transaction profiling..... 21

CHAPTER FOUR

Technical Reference

IN THIS CHAPTER

FlexNoC packet formats	43
Observation probes.....	53
Observers	115



Arteris FlexNoC observability technology comprises two principle components: observation probes and observer units.

The section provides detailed technical reference information for these components, along with the FlexNoC packet format description needed to analyze error logs and packet trace information.

od10200v1

FlexNoC packet formats

FlexNoC technology supports two packet formats:

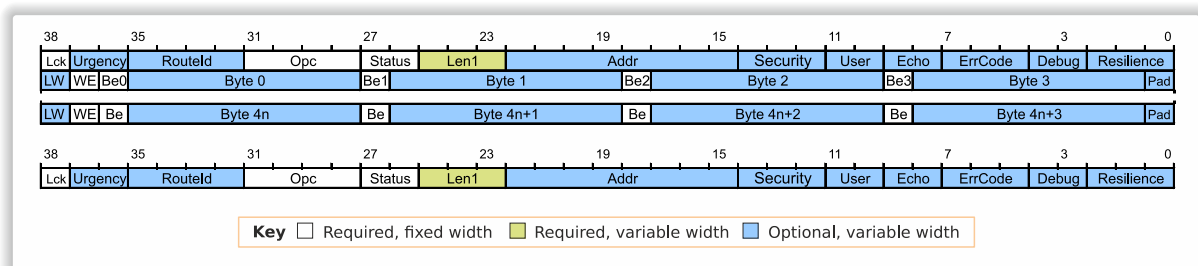
- § Packets with payload, such as write requests, read responses, and some preamble requests.
- § Packets without payload, that is, all other types.

The header of a packet has the same format independent of the request or response status. For example, a read response header has the same format as a write response header or a read request header. Most of the fields are invariant between the request packet header and the associated response header of a transaction.

Packet format is tuned for a particular FlexNoC instance depending on the NoC configuration, but is fixed for all packets, whether they are on the datapath network, or the observation network, in a given FlexNoC architecture. This implies a uniform programming model for all packet probes or error logging within the same network.

Serialization of packets, however, may vary from link to link. Performance statistics, such as the number of cycles taken to transfer packets, may therefore differ depending on the probe point. Conversely, error logging or packet tracing are independent of the original packet serialization.

The following figure shows a simple format defined by a 4-byte payload width and a 1-cycle header penalty. Colored fields are either optional or variable in width. Packet transmission starts from top to bottom and left to right.



ob14081v1

od11979v1

Variable width packet fields and optional fields

Packet fields, which are either of variable width or optional, or both, are defined by network parameters. The values of these network parameters are calculated by FlexNoCFlexArtist software during NoC synthesis.

A separate set of network parameters is created for each network type. The set for the datapath network, for example, is shown in the following table.

Network parameter name	Associated packet field	Field type
wUrgency	Urgency	Header
wSecurity	Security	Header
wAddr	Addr	Header
wLen1	Len1	Header
wRouteld	Routeld	Header
wEcho	Echo	Header
wUser	User	Header
wDebug	Debug	Header
useLastWord	LastWord	Payload
useErrCode	ErrCode	Header
wByte	Byte	Payload
wResilience	Resilience	Header

Network parameter values can be displayed with FlexNoC FlexArtist software.

od14273v1

è To display network parameter values

- 1 Select a structure object in the FlexArtist Project Tree pane.
- 2 In the Information pane, locate the *Transport NTTP Packages* area.
- 3 Click the [datapath](#), [observation](#), or [service](#) links to display the corresponding network report in the Contextual Help page.

An example of the display is shown in the following figure.

**Datapath network
NTTP package**
Header Size: 51 bits

Parameters:

wUrgency	0	wSecurity	
wRouteId	5	wUser	0
wLen1	7	wEcho	0
wAddr	32	useErrCode	False
wPriority	0	wDebug	0
		wResilience	0
		useLastWord	True
		wByte	8

od14266v2

Network parameter details are also included in the `.info` file generated by the FlexArtist software command [Export Structure Information](#), available on the [Implementation](#) menu, submenu [RTL output](#).

od14273v1

Packet header fields

FlexNoC packets contain the following header fields, presented in MSB-to-LSB order.

od11980v1

Lock (Lck)

When the *Lock (Lck)* bit is set, arbitration will not take place at the end of the packet. The following packet must take the same route and no other packet will be inserted in between. Lock capability is used either in conjunction with Exclusive Read or for Linked sequences.

This bit is always present and is in the first word of the header.

od13984v1
Preliminary

Urgency

The *Urgency* field contains the urgency level of the packet. The number of urgency levels is specified by parameter *nUrgencyLevel*. Because bar-graph encoding is used for timing reasons, the number of bits in this field is equal to number of urgency levels minus one. The lowest urgency is 0...00, with 0...01, 0...11 up to 1...11 increasing in urgency.

Urgency has no impact on the ordering rules and is generated by the initiator NIUs, either from socket information or from internal computation like a bandwidth regulator.

This field is in the first word of the header. It is not invariant, and its width is set by datapath or observation network parameter *wUrgency*.

od13985v1
Preliminary

RouteId

The *RouteId* field uniquely identifies an *initiator-mapping-target-mapping* pair. This information is used by routing tables to steer the packet inside the network. The set of possible route IDs is determined by both the connectivity table and the memory map.

The following table shows the MSB-to-LSB breakdown of *Routeld* into its constituent sub-fields.

Sub-field name	Description
InitFlow	System-unique number identifying the transaction initiator flow.
TargFlow	System-unique number identifying the transaction target flow.
TargSubRange	Unique number for a given target flow, identifying address regions from the same initiator flow to the same target flow.
SeqId	Sequence number assigned by the initiator NIU.

TIP The width of these sub-fields is displayed by clicking the datapath transport NTTP package displayed in the FlexArtist Information pane when a structure object is selected in the Project Tree. This width is also given in the exported structure information file *StructureName.info*.

The width of this invariant field is set by datapath or observation network parameter *wRouteld*.

od13986v1
Preliminary

RELATED REGISTERS

ErrLog1	135
Filters_N_RouteldMask	96
Filters_N_RouteldBase	96

od13986v1
Preliminary

Opc

The 4-bit *Opc* field indicates the transaction type. This field is invariant, except for RDL and WRC values, which are updated by exclusive access monitors, and are always present.

The field has the following encoding:

- 0: RD** Read. Data is read by an initiator from a target with incrementing addressing.
- 1: RDW** Wrapped Read. Data is read by an initiator from a target with wrapping addressing.
- 2: RDL** Linked Read. This read allocates a monitor in the Target with incrementing addressing.
- 3: RDX** Exclusive Read with incrementing addressing. It must be immediately followed by a write with the same values of *Addr*, *Len*, and *Status*, within a locked sequence (Lock = 1 on RDX and 0 on WR). This sequence guarantees atomic execution of RDX and its associated WR.
- 4: WR** Write. Data is written by an initiator to a target with incrementing addressing.
- 5: WRW** Wrapped Write. Data is written by an initiator to a target with wrapping addressing.
- 6: WRC** Conditional Write. This write is only executed if there is a matching monitor in the target. Otherwise the *Status* value FAIL is sent back to the initiator. Incrementing addressing is used.
- 7: Reserved.**
- 8: PRE** Preamble Packet (see "Preambles" on page 52) of a Linked Sequence. This is used to signal special operations such as entering a locked sequence or transactions, or non-native bursts such as Fixed or 2D bursts.

NOTE Preamble packets only propagate ahead of associated request packets, and are processed and dropped by target NIUs. The packets never appear when probing response links. The status of a PRE packet must be *REQ* because this type of packet does not have a response.

9: URG Urgency Packet. Used for pressure and urgency management. The status of an URG packet is always *REQ*. Only *Urgency* and *Path* fields are significant in these packets. The other fields are all 0.

10-15: Reserved.

od14685v1
Preliminary

Status

The 2-bit field *Status* indicates the evolution of a transaction through the NoC. The field, which is not invariant and is always present, has the following encoding:

0: REQ The packet contains an unprocessed request.

1: ERR The packet contains a response with an error condition. A packet with status *ERR* never contains a payload.

2: RSP The packet contains a response and the associated request has been successfully processed.

3: Status code 3 can have two meanings:

FAIL A Conditional Write request has not been processed because the associated monitor was cleared by some other write access.

or

CONT The packet is a response fragment to a read which is not the first fragment. This status occurs when the target interleaves read responses. It is only allowed when *OpC* is set to **RD**.

Any agent is allowed to generate status *ERR*. Only target NIUs are allowed to generate status *RSP*, *FAIL* or *CONT*. Target NIUs execute requests only when the packet status is *REQ*.

od13989v1

Len1

The *Len1* field indicates the number of payload bytes, minus 1, handled by the transaction.

This field is invariant and is always present. Its width is determined by the datapath or observation network parameter *wLen1*.

od13990v1

Addr

The *Addr* field indicates the start address of the transaction, expressed in bytes, in the target address space. This address is produced by the address translation tables of the initiator NIUs.

The width of this field is set by the datapath or observation network parameter *wAddr*.

A target NIU only copies this field from request packets to response packets in one of the following cases:

- § if there is a **Packets** probe point on the response path
- § if there is an **Errors** probe point on the response path, global parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to **False** and the error is of type UNS, DISC, SEC or DEC and it has been generated within the NoC transport.

- § if there is an **Errors** probe point on the response path and global parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to **True**.

If none of these conditions are satisfied, field *Addr* is still present in the response packet but its value is invalid.

RELATED PARAMETERS

keepAddressAndFlagsInResponsePacketsForErrorProbes 69

od13991v1
Preliminary

Security

The *Security* field contains the security information of the transaction specified by the security flags in the specification. Security flags are stored in the security field in alphabetical ASCII order (lowest ASCII codes left most).

This field is invariant and its width is set by datapath or observation network parameter *wSecurity*.

od13992v1
Preliminary

User

User information transported as-is from initiator to target, as specified by user flags. User flags are stored in the *User* field in alphabetical ASCII order, that is, the lowest numeric ASCII encoding starts from the LSB.

The width of this field is set by the datapath or observation network parameter *wUser*.

A target NIU only copies packet header field *User* from request packets to response packets if one of the following conditions is satisfied:

- § There is a **Packets** probe point on the response path, in which case all user flags are copied to the field.
- § There is an **Errors** probe point on the response path, global parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to **False**, the error is of type UNS, DISC, SEC or DEC and it has been generated within the NoC transport. In this case, all user flags are copied to the field.
- § There is an **Errors** probe point on the response path, global parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to **False**, and it is a slave error (SLV). In this case, only those flags which are set to **True** in global parameter *userFlagsKeptInResponsePacketsForErrorProbes* are copied into the response packets.
- § There is an **Errors** probe point on the response path and global parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to **True**. In this case, all user flags are copied to the field.

If none of these conditions are satisfied, field *User* is still present in the response packet but its contents is invalid.

RELATED PARAMETERS

keepAddressAndFlagsInResponsePacketsForErrorProbes 69
userFlagsKeptInResponsePacketsForErrorProbes 70

od13993v1
Preliminary

Echo

The *Echo* field contains private information necessary for proper operation of the various FlexNoC units, but unrelated to any particular functional attribute of the transaction. It can be ignored for all observation purposes.

This field is invariant and its width is set by the datapath or observation network parameter *wEcho*.

od13994v1
Preliminary

ErrCode

The optional 3-bit *ErrCode* field indicates the type of error, and is relevant when *Status* = ERR, otherwise *ErrCode* defaults to SLV. This field is only created when the *differentiated error* mode has been activated by setting parameter *useErrorCodes* to True (Specification: Global parameters).

Encoding for this field is shown in the following table.

Code	Value	Source	Type
SLV	0	Target	Target error detected by slave.
DEC	1	Initiator NIU	Address decode error.
UNS	2	Target NIU	Unsupported request.
DISC	3	Power Disconnect	Disconnected target or domain.
SEC	4	Initiator NIU or Firewall	Security violation.
HIDE	5	Firewall	Hidden security violation, reported as OK to initiator.
TMO	6	Target NIU	Time-out.
RSV	7	None	Reserved.

od13995v1
Preliminary

Debug

The *Debug* field is used internally by probes. The width of this field is set by the datapath or observation network parameter *wDebug*. In the case of the datapath network, *wDebug* is always 0, and consequently, can be ignored for observation purposes.

od13996v1
Preliminary

Resilience

The *Resilience* field contains parity or ECC information..

The width of this field is set by the datapath or observation network parameter *wResilience*.

od13997v1
Preliminary

Packet payload fields

FlexNoC packets contain the following payload fields, presented in MSB-to-LSB order.

NOTE In general, FlexNoC probes are configured to observe only headers, not payloads.

od14109v1
Preliminary

LastWord (LW)

The packet payload field *LastWord* (LW) bit is asserted to 1 to indicate the last payload fragment of an interleaved response. It is not asserted for a write payload. This bit is mandatory when response interleaving is supported.

This bit is present when datapath or observation network parameter *useLastWord* is set to **True**.

WIDTH 1 bit

od13998v1
Preliminary

WordErr

The packet payload field *WordErr* (WE) bit is asserted to 1 when the associated read payload contains at least one byte in error, such as a parity error, for example.

It is de-asserted when associated with a write payload.

WIDTH 1 bit

od13999v1

Be

A *Be* (byte enable) bit is associated with each byte in the payload.

For a write request, a payload byte must be written when its *Be* field is equal to 1, or discarded otherwise.

For a read response, a payload byte contains valid read data when its associated *Be* is equal to 1, or no data otherwise. This situation occurs when a read response has been chopped by the target NIU because of target interleaving capability.

Be is 0 when it does not correspond to an addressed byte, regardless of whether it is a read or a write.

od14000v1

Byte

The *Byte* field contains the packet payload and optionally a parity bit in an ECC field when the datapath or observation network parameter *wByte* is set to a value greater than 8. The payload always occupies bits (7:0) of the *Byte* field.

NOTE Because byte addresses increase from left to right, packet format is big-endian (on page 44).

od14001v1

Routeld packet field structure and Addr values

For the purpose of analyzing a packet trace, or filtering packets by source, destination, or both, by using the *routelDMask* and *routelDMatch* probe registers, it is useful to know the sub-component structure of packet field *Routeld*.

Filtering packets by address ranges, or understanding target addresses, also requires understanding the relationships between the *Addr* field transported in the packet, the *Routeld*, and the original initiator address.

Field *Routeld* comprises the sub-fields shown in the following table.

Sub-field name	Description
----------------	-------------

Sub-field name	Description
InitFlow	System-unique number identifying the transaction initiator flow.
TargFlow	System-unique number identifying the transaction target flow.
TargSubRange	Unique number for a given target flow, identifying address regions from the same initiator flow to the same target flow.
SeqId	Sequence number assigned by the initiator NIU.

NOTE The width and position of these sub-fields, and their associated initiator and target flows or subrange encodings for a given FlexNoC instance, are displayed in the FlexArtist Information pane when a structure object is selected. This information is also included in the *StructureName.info.xml* file generated when exporting designs with FlexArtist software.

od11983v1
Preliminary

Once the *Initflow*, *TargFlow*, and *TargSubRange* information, also referred to collectively as the aperture, has been extracted from a packet, field *Addr* indicates the offset of the addressed bytes within that particular address region mapped from the initiator to the target. The corresponding initiator and target addresses of the transaction can then be reconstructed from the initiator and target.

From the target perspective, the target request address corresponding to the packet is the sum of the *Addr* packet field, and the local address of the target mapping.

From an initiator perspective, the definition of the address and connectivity map allows a single initiator mapping for the *Initflow* to match the target mapping in the global address space. The initiator request address corresponding to the packet is the sum of the *Addr* packet field, and the local address of the initiator mapping.

NOTE The only way to find the associated initiator mapping is to analyze step-by-step the definition of the address map configured for the NoC instance. The list of such initiator–target mapping associations can be found in exported *StructureName.info.xml* files. The aperture list contains information equivalent to the FlexNoC FlexArtist view of the memory map (Specification: Mappings: Memory Map), including the mapping local addresses required to reconstruct the initiator and target address.

Example

Initiator *I* has a single flow, connected to target *T* also with a single flow, but two mappings of 1 KB each. Initiator mapping starts at local address 0x0, global address 0x0, size 4 GB.

One of the 1 KB mappings to target *T* starts a local address of 0x0, global address 0x0. The second 1 KB mapping to target *T* starts a local address of 0x0, global address 0x100000,.

After identifying packets containing the identifier flow for *I* in *InitFlow*, the identifier flow for *T* in *TargFlow*, and either a *TargSubRange* 0 or 1, the initiator and target address for the associated transactions can be determined. Walking through the list of the two apertures in this address map, first the initiator mapping and the target mapping, by selecting the mappings that have the same sub-range number as the *Routeld* sub-field *TargSubRange*. Then, for example:

§ An observed packet with *Addr* 0x100 for the initiator mapping and first target mapping example corresponds to: initiator address = 0x0 + 0x100 = 0x100, target address = 0x0 + 0x100 = 0x100.

- § An observed packet *Addr* of 0x100 for the initiator mapping and second target mapping example corresponds to: initiator address = 0x0 + 0x100 = 0x100, target address = 0x100000 + 0x100 = 0x100100.

od11983v1
Preliminary

Preambles

Preambles are packets with *Opc*=PRE, which are the first of a Locked sequence of request packets (i.e., a series of successive request packets kept together during arbitration), optionally sending additional information to the target for special purposes related to the processing of transactions part of the locked sequence. They are used for example to carry the necessary information to reproduce Fixed or 2D bursts at target NIUs.

They may have an associated payload of 4 or 8 bytes to carry the extra information.

NOTE For complex preambles, that is, those with a data payload indicated by *Len1* greater than 0, such as Fixed or 2D bursts, the preamble type is only available in the payload itself. If payload tracing is not configured in a probe, only simple preambles can have their type identified, using bits *Addr* (3:0) of the preamble.

When the preamble type can be identified by *Addr* (3:0), the values are:

- 0x0 Abort preamble** Terminates the locked sequence, as also indicated by a value of 0 in bit *Lck* of this preamble.
- 0x1 Lock preamble** Only acts as the beginning of a Locked sequence, signaling arbiters to keep requests together until (and including) one that is marked with bit *Lck* at 0. The behavior of the transaction part of the Locked sequence is not affected by this preamble.
- 0x2 Null read preamble** Only possible in NoCs where at least one initiator is capable of sending a read null request. Followed by a single read request of one byte with *Lck* equal to 0, changes the meaning of that read transaction to act as a read of zero bytes at the destination ([../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf](#)).
- 0x3 RdCondWr preamble** Only possible in NoCs where at least one PIF protocol initiator supports RdCondWr operation ([../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf](#)). Followed by three additional requests in the locked sequence, that is, two writes followed by a read with *Lck* equal to 0, all other relevant request fields (Address, Length, burst type, source, destination) being identical.

If the RDCondWR operation is supported at the destination, the effect of the locked sequence is that the data returned by the read is compared to the first write request data payload, and only if they are equal, the second write is executed, otherwise it is discarded. If not supported at destination, an error is returned for the three requests part of the Locked sequence.

Preambles for Fixed and 2D bursts are complex preambles (with *Len1* greater than 0) requiring examination of their associated payload to evaluate their precise effect. They are followed by a single request, whose *Addr* and *Len1* fields are closely related to the address and size of the request.

NOTE The *Addr*field in a read or write request following a complex preamble is rounded to a multiple of 128 bytes. The exact transaction address is reconstructed at the target NIU by using additional information present in the preamble payload. For more information about how to interpret fixed or 2D bursts preamble using the associated preamble payload, contact Arteris support services (mailto:support@arteris.com).

od11984v1
Preliminary

Observation probes

This section provides detailed technical reference information for the FlexNoC Observability technology collectively called *observation probe* units.

od10201v1
Preliminary

Error probe

Packet headers are stored in the probe and are sent to the probe output if the packet header is in error, or if the subsequent error status of the data payload belonging to the packet indicates a late target error.

No flow control is injected at the probed link; there is no timing impact on the datapath. If another error is detected while there is an unprocessed error packet still pending in the probe, the second error will be ignored.

od10202v1
Preliminary

Packet probe

The FlexNoC *packet probe* unit is implemented whenever trace and statistics collections features are enabled in FlexNoC FlexArtist software. The unit can be used in conjunction with the FlexNoC *transaction profiler* unit to create a *transaction probe*.

Main features

The unit has the following main features:

- § Support for any transport packet format and serialization.
- § Trace mode with programmable packet filters.
 - § Optional packet payload tracing.
 - § Programmable flow-control modes: *Overflow Management* and *Intrusive*
- § Statistics can be collected and sent to an observer either periodically or after manual triggering.
- § Alarms on both trace filters and statistics counters.
- § Optional time stamping of probed events.

od10203v1

Transaction filter

The FlexNoC *transaction filter* unit uses programmable criteria to select request and response events.

The unit is integrated into the NIU and is connected to the NIU generic-to-transport unit. More than one transaction filter can be connected to a FlexNoC transaction profiler unit.

Transaction profiler

The FlexNoC *transaction profiler* unit receives events from one or more transaction filters and organizes them into histogram bins, defined by programmable thresholds.

The unit is connected to a standard Packet Probe via the unit external event input ports.

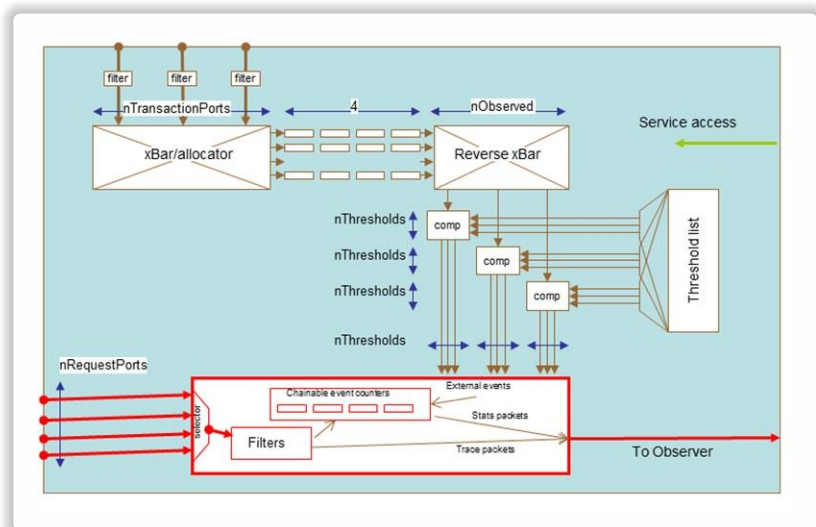
od10207v1
Preliminary

Universal probe functional description

The FlexNoC *universal probe* unit is constructed from the same FlexNoC hardware IP library modules used for transaction probe and packet probe ONUs, described in related reference documentation.

The probe has its own specific probe ports for observing request-only and transaction interfaces. Because the probe is not intrusive, the signals of the probe ports are input-only. FlexNoC interconnect-specific features are not included, however.

The following figure shows a functional diagram of a universal probe with a complete set of available port types and sub-units.



ob8863v1

In the diagram:

- § Request ports are red, and are connected to the packet probe ONU, outlined in red. The output of the packet probe sub-unit connects to the observer. Standard packet probe features filtering, tracing, and statistics collection are available, but payload tracing is *not* supported.
- § Transaction processing functions are outlined in brown. Transaction filters are instantiated on each transaction port. The filtered outputs are connected to the transaction profiler unit, the outputs of which are connected to the external input ports of the packet probe sub-unit. The same features are thus available as when probing transactions internally with a NoC: transaction filtering and profiling, and statistics gathering.

The actual type of probe ports and sub-units instantiated in a given universal probe depends on how the probe is configured. For example, if no transaction ports are specified, the elements outlined in brown in the diagram are not present. If statistics collection is not required, then a trace-only version of the packet probe sub-unit is instantiated.

The same interface can be connected to a request-only port and a transaction port if both functions are desired and all signals required by the ports are available. However, if width of signals common to the transaction port and request-only port differ, they should be padded outside the probe to the wider set.

RELATED REFERENCES

Packet probe	53
Transaction filter	53
Transaction profiler	54

od14860v1
Preliminary

Request-only ports

Universal probe request-only probe ports support the following functionality:

- § Filtered tracing (request forwarded to the observer).
- § Statistics collection of transaction state or filtered request events.

Request-only ports are typically used to gather statistics on the state (idle time versus busy time) of the probed interface or counting request events in order, for example, to calculate the bandwidth achieved over the interface.

Packet probe features and limitations apply to the request-only probe ports:

- § All request-only ports of the same probe are synchronous to the probe clock, and only a single port can be selected for probing at run time.
- § Request-only ports do not support the packet probe optional data-tracing feature. All request-only ports for a given probe use the same signal configuration.
- § Request-only ports cannot probe reads and writes during the same cycle. Therefore, when probing an AXI interface, separate universal probes must be connected to the independent read and write channels.

od14827v1
Preliminary

Transaction ports

Transaction ports are used when the probed interface needs to be characterized in terms of request acceptance delay (handshake), request-to-response latency, or number of pending transactions.

The features and limitations of transaction ports are the same as those found when probing transactions within the NoC:

- § Port clock can be asynchronous to probe clock.
- § Dedicated transaction filter per port.
- § Simultaneous monitoring of multiple ports on the same clock cycle.

Therefore, unlike request-only ports, it is possible to observe read and write transactions of an AXI interface with a single universal probe with two transaction ports connected to the write and read channels.

od14828v1

Adapting the probed interface to universal probe ports

The external interface to be probed must first be adapted to conform to the interfaces of the request or the transaction ports, or both, of the FlexNoC universal probe. This adaptation is the responsibility of the integrator and is instantiated externally to the NoC.

Depending on the specifics of the interface to be probed, and the type of observation measurements to be made, the adapter is required to perform some or all of the following functions:

- § Signal mapping
- § Opcode generation
- § Transaction length generation
- § Transaction ID generation

Generating the opcode

The universal probe expects the transaction type to be encoded according to the opcode encoding scheme of the FlexNoC transport, as described in FlexNoC packet formats documentation (see "FlexNoC packet formats" on page 43).

The following table contains a subset of the opcodes that are relevant for universal probes.

Transaction type	Opcode	
	Name	Value
Read	RD	0
Wrapped read	RDW	1
Linked read	RDL	2
Exclusive read	RDX	3
Write	WR	4
Wrapped write	WRW	5
Conditional write	WRC	6

The transaction type of the probed interface must therefore conform to this encoding. As an example, the table below shows a *suggested* re-encoding when interfacing to an AXI write channel:

AwBurst	AwLock		Opcode
	AXI v3	AXI v4	
00 or 01	00	0	WR
	01	1	WRC
	10	not applicable	WR ¹
10	X	X	WR ²

Table notes

- 1 The FlexNoC packet format does not define an opcode value for locked accesses. It is suggested that a user signal of the probe port is assigned to indicate a locked access.

- 2 If the interface is probed by a transaction port, wrapped bursts must be mapped to opcode WR or RD because they are not supported by the port transaction filter. It is also recommended to map wrapping requests to RD and WR for request ports. If filtering is required on wrapped bursts, a user bit may be used to indicate the presence of such bursts.

Computing the payload length

The universal probe expects information on burst length applied to its probe ports to be expressed in bytes. Other base units must be first converted. For example, AXI burst lengths are expressed in terms of a number of transactions. The following formula regenerates the burst length in bytes, based on the AXI signals *AxLen*, *AxSize*, and *AxAddr*:

$$\text{Burst length in bytes} = (\text{AxLen} + 1) \times 2^{\text{AxSize}} - \text{AxAddr} \% (2^{\text{AxSize}})$$

Generating the transaction ID when measuring latency for interface protocols which support out-of-order transactions

The integrator must supply a unique identifier for each transaction request from a given initiator, within its current pool of pending transactions, to the probe transaction port. Also, for each corresponding response, an identifier with the same value must be supplied to the probe port. The universal probe requires available, unique identifiers, in order to compute latency. The adapter for the universal probe must allocate such identifiers if they are not already part of the probed interface.

When probing and adapting partially-ordered interfaces, which may return out-of-order (OOO) responses to requests with different IDs, but in-order responses for requests with identical IDs, the logic to generate such unique transaction identifiers can become complex.

Note that the probe can start probing a particular interface at run time while transactions are already pending on the interface. In this case, the responses to pending transactions are ignored because their transaction identifiers cannot be associated with transaction identifiers of known requests. This transaction pipeline realignment mechanism at the start of probing is the main reason why the probe needs unique request and response identifiers for each pending transaction.

Transaction constraints for Universal Probes

od14832v1
Preliminary

The underlying observability hardware units used to construct the universal probe impose certain constraints on transactions. When these units are instantiated individually within the NoC, these constraints are handled implicitly. It is important, therefore, to ensure that transactions, which are observed by a universal probe, respect these constraints, which are listed in the following table by transaction type.

Opcode name	Packet header field constraints
RD, WR	$(\text{Addr} + \text{Len}1) < 2^{\text{wAddr}}$ *Note
RDW, WRW	$((\text{Len}1 - 1) = 2^{\text{x}})$ and $(\text{Len}1 > 0)$
RDL, WRC	$((\text{Len}1 - 1) = 2^{\text{x}})$ and $(\text{Addr} \% (\text{Len}1 - 1) = 0)$ and $(\text{Len}1 < 128)$

Table notes

* *wAddr* is the width of the packet header field *Addr*.

od14820v1
Preliminary

Probe signals

od10208v1
Preliminary

TraceAlarm

Output signal *TraceAlarm* is asserted to 1 when all of the following conditions are met:

- § A packet filter hit or LUT hit occurs.
- § The corresponding bit in register *TraceAlarmEn* has been set to 1.
- § Field *AlarmEn* has been set to 1 in register *MainCtl*.

NOTE When parameter *nFilter* is set to 0 and the LSB of register *FilterLut* is set to 1, a LUT hit always occurs. This continuously asserts *TraceAlarm*. If this is not required, clear the LSB of *TraceAlarmEn* to 0.

TraceAlarm can either be connected directly to the NoC periphery, or it can be routed through a sideband manager.

od10210v1

StatAlarm

Output signal *StatAlarm* is driven by bit 0 of register *StatAlarmStatus*, when bit 0 of register *StatAlarmEn* is 1 and field *AlarmEn* in register *MainCtl* is set to 1.

This signal is only implemented when parameter *statisticsCounterAlarm* is set to True.

od22452v1
Preliminary

StatSuspend

When *StatSuspend* is driven to 1, the statistics counters are frozen and the dump timer is suspended *after a statistics dump*.

When driven to 0, *StatSuspend* clears the values in the statistic counters, enables event counting, and activates the dump timer, provided:

- § both field *StatEn* of register *MainCtl* is 1 and field *GlobalEn* of register *CfgCtl* is 1, or
- § alternatively, channel 0 of signal *CtiTrigIn* is 1.

This signal is only implemented when parameter *allowStatisticsSuspend* is set to True.

NOTE This signal does not affect the trace output.

od8254v1

ExtEvent

Input signal *ExtEvent.n* is the means by which events external to the NoC can be counted by a statistics counter in the referenced probe. The number of external inputs, *n*, is defined by parameter *nExternalEvent*.

ExtEvent.n is selected as the source of a statistics counter using the FlexNoC packet probe counter register *Counters_M_Src*. While the signal is asserted to 1, the counter will increment on the probe clock.

od13958v1

TimeStamp

Input signal *TimeStamp* carries the time stamp value used by the probe to stamp the time in the observation packet.

The signal value must be absolute and encoded in natural unsigned format.

Probe parameter *wTimeStamp* sets the signal width (Architecture: Observability: Probe). If set to 0, the signal is not created, and the observation packet is stamped with a time of 0.

NOTE A common time stamp signal can be used for all probes that belong to the same clock regime.

CoreSight cross-triggering signalsod13959v1
Preliminary

FlexNoC cross-triggering signals are used to enable the probe, forward alarms, and initiate statistics dumps by an external module via the ARM® CoreSight™ cross-triggering protocol.

CtiTrigInod10212v1
Preliminary

Signal *CtiTrigIn* controls the probe via two CoreSight channels. Therefore, although it is an 8-bit signal, only the two LSBs are implemented.

NOTE The signals pass through re-synchronizer cells whether or not the clock domain of the signal source is asynchronous to the probe clock.

TRIGIN Channel 0 The probe is enabled when *CtiTrigIn(0)* is driven to 1. It is equivalent to setting field *GlobalEn* to 1 in register *CfgCtl*. Clearing the signal to 0 disables the probe.

TRIGIN Channel 1 A statistics frame is dumped, all event counters are cleared, and statistics gathering resumes when *CtiTrigIn(1)* is driven to 1. It is equivalent to a pulse on register *StatGo*. Clearing the signal to 0 has no effect.

WIDTH (7:0)

DIRECTION Input

od10213v1

CtiTrigInAck

Signal *CtiTrigInAck* acknowledges events received on *CtiTrigIn*. Because only two CoreSight channels are implemented, only the two LSBs of the 8-bit signal are used.

WIDTH (7:0)

DIRECTION Output

od10214v1

CtiTrigOut

Output signal *CtiTrigOut* reports alarm events from the probe via two Coresight channels. Therefore, although it is an 8-bit signal, only the two LSBs are implemented.

TRIGOUT Channel 0 *CtiTrigOut(0)* is asserted to 1 when a packet has been selected by the LUT to match the programmed criteria in the filters and the corresponding bits in register *TraceAlarmEn* are set to 1. It is equivalent to the assertion of signal *TraceAlarm*. The signal is asserted by setting field *AlarmEn* to 1 in register *MainCtl*.

CtiTrigOut(0) is de-asserted to 0 when input signal *CtiTrigOutAck(0)* is asserted to 1 and register *TraceAlarmClr* is set to 1 to clear the alarm.

TRIGOUT Channel 1 *CtiTrigOut(1)* is asserted to 1 when a statistics alarm event occurs. It is equivalent to the assertion of signal *StatAlarm*. The signal is enabled when the following registers are configured, and the "without dump" option for statistics gathering with alarm mode is activated:

Register *MainCtl*:

Field *StatEn* = 1

Field *AlarmEn* = 1

Field *StatCondDump* = 0

The corresponding bit in register *StatAlarmEn* is set to its default, 1.

CtiTrigOut(1) is de-asserted to 0 when input signal *CtiTrigOutAck(1)* is asserted to 1 and the corresponding bit in register *StatAlarmClr* is set to 1 to clear the alarm.

WIDTH (7:0)

DIRECTION Output

CtiTrigOutAck

Input signal *CtiTrigOutAck* acknowledges events received on *CtiTrigOut*. If signal *CtiTrigOut* is not connected, the input *CtiTrigOutAck* must be forced to 1.

Because only two CoreSight channels are implemented, only the two LSBs of the 8-bit signal are used.

NOTE The signals pass through re-synchronizer cells whether or not the clock domain of the signal source is asynchronous to the probe clock.

WIDTH (7:0)

DIRECTION Input

od14237v1
Preliminary

User-defined filtering signals

When probe parameter *securityFiltering* is set to User-defined filtering (Architecture: Observability: Probe), custom filtering logic can be implemented in a dedicated customer cell. By default, parameter *coreName*, which determines the name of the corresponding module(Architecture: Observability: Probe: securityFiltering), is set to *ProbeName_Core* for probe *ProbeName*.

Probe filtering modules are included in files *rtl.customerCells.v* or *rtl.CustomerCells.vhd* when NoCs are exported to Verilog or VHDL respectively. File names can be modified by setting parameter *descriptionPath* (Project Manager: Exports: Export Options: exportOption).

User flags can optionally be used as additional filtering criteria inputs. The list of flags is set by *interfaceFlags* (Architecture: Observability: Probe: securityFiltering).

od10216v1

Verilog module example

The following RTL code example shows a module in a Verilog export:

```
module Probe_Core (Flag_x, Flag_y, Fwd, Vld);
  input Flag_x;
  input Flag_y;
  output Fwd;
  input Vld;
  assign Fwd = "user-defined expression";
endmodule
```

In the module:

- § Optional user flags *Flag_x* and *Flag_y* are used as filtering criteria inputs.
- § Output signal *Fwd*, whose value is computed by the custom filtering logic, is driven to 1 or 0 when packets can be processed or discarded respectively by the corresponding probe.
- § Input signal *Vld*, when asserted **high**, indicates that module input information is valid.

od18384v1

RELATED PARAMETERS

securityFiltering	77
interfaceFlags.....	78
coreName	78

Universal probe signals

The FlexNoC universal probe has two interfaces respectively for transaction and request ports.

od8816v1

Universal Probe Request port

The universal probe request port interface has the following signals.

od14792v1

Valid

Signal *Valid* should be driven to 1 from the beginning of the request until the end of the request. In conjunction with request port signal *Ready*, this signal defines the state of the interface, as described in the following table.

<i>Valid</i>	<i>Ready</i>	<i>Event type</i>
0	X	IDLE
1	0	BUSY
1	1	Request

Events **IDLE** and **BUSY** indicate the flow-control state of the request interface. The probe can count such events but cannot filter them. In contrast, request events are filtered before they are counted.

More information on event filtering and counting in the universal probe is available in related reference documentation.

od8834v1
Preliminary

RELATED REFERENCES

Packet probe filter registers.....	95
Packet Probe counter registers.....	101

Ready

Signal *Ready* should be driven to 1 when the transaction request has been accepted by the target. This signal is used by the universal probe in conjunction with *Valid* in order to determine the state or event type of the transaction request.

od14851v1
Preliminary

Id

Signal *Id* should be driven by the request identifier that indicates the source of the transaction. Its value can be mapped to user flags within the NoC, which can then be used to filter the transaction request. The width of the signal is determined by parameter *wId* in the parameter group *nRequestPorts*.

The value on this signal must not change when signal *Valid* is set to 1 until signal *Ready* is asserted to 1.

od14852v1
Preliminary

Address

Signal *Address* should be driven by the transaction address. Its value can be used to filter the transaction request.

Signal width is determined by parameter *wAddress* in parameter group *nRequestPorts*.

Signal value must not change, when signal *Valid* is set to 1, until signal *Ready* is asserted to 1.

od8838v1
Preliminary

Opcode

Signal *Opcode* should be driven by the type of transaction. The value must comply with the encoding used for the FlexNoC packet header field *Opc*, described in related reference documentation. The opcode information can be used to filter transactions.

When signal *Valid* is set to 1, the value on *Opcode* must not change until signal *Ready* is asserted to 1.

IMPORTANT Opcodes PRE, RDX and all reserved values are not supported by the universal probe, and can cause unexpected behavior.

od14854v1

RELATED REFERENCES

Opc.....	46
----------	----

Length

Signal *Length* must be supplied with the value of the transaction length, expressed in bytes.

Signal width is set by parameter *wAddress* in the parameter group *nRequestPorts*.

The value on this signal must not change when signal *Valid* is set to 1 until signal *Ready* is asserted to 1.

Prio

The priority level of the transaction, encoded in binary, should be applied to signal *Prio*. The universal probe maps this to the FlexNoC packet header field *Urgency*, allowing transactions to be filtered using packet probe register *Filters_N_Urgency*.

The width of the signal is set by parameter *wPrio* in the parameter group *nRequestPorts*.

The value on this signal must not change while signal *Valid* is 1 until signal *Ready* is asserted to 1.

od14836v1
Preliminary

User

Signal *User* should be driven by any in-band qualifiers for the transaction. Its value can be mapped to user flags within the NoC, which can then be used to filter the transaction.

The width of the signal is determined by parameter *wUser* (Specification: Interface: Observation: nRequestPorts).

NOTE Although the maximum value of parameter *wUser* is 80, only the lower 64 user bits can be used for filtering at universal probe request ports.

od14855v1
Preliminary

Security

Signal *Security* should be driven by the security access type of the transaction. Its value can be mapped to security flags within the NoC, which can then be used to filter the transaction. The width of the signal is determined by parameter *wSecurity* in parameter group *nRequestPorts*.

od14856v1
Preliminary

Universal Probe Transaction port

The universal probe transaction port interface has the following request and response signals.

od14857v1
Preliminary

Req_Valid

Signal *Req_Valid* should be driven to 1 for the duration of the transaction request. The universal probe uses the signal in conjunction with *Req_Ready* when measuring request latency (handshake delay). *Req_Valid* is also used with *Req_Tld*, *Rsp_Ready*, and *Rsp_Tld* when the universal probe is programmed to measure request-to-response latency.

od14791v1

Req_Ready

Signal *Req_Ready* should be driven to 1 when the transaction request has been accepted by the target. This signal is used by the universal probe in conjunction with *Req_Valid* when measuring request latency.

od14841v1
Preliminary

Req_Id

Signal *Req_Id* should be driven by the request identifier that indicates the source of the transaction. Its value can be mapped to user flags within the NoC, which can then be used to filter the transaction. The width of the signal is determined by parameter *wId* in parameter group *nTransactionPorts*.

Req_Address

Signal *Req_Address* is driven by the transaction address. Its value can be used to filter the transaction. The width of the signal is determined by parameter *wAddress* in parameter group *nTransactionPorts*.

Req_Opcode

Signal *Req_Opcode* should indicate the type of transaction. The value must comply with the encoding used for the FlexNoC packet header field *Opc*, described in related reference documentation. The opcode information can be used to filter transactions.

IMPORTANT Opcodes PRE, RDX and all reserved values are not supported by the universal probe and can cause unexpected behavior.

RELATED REFERENCES

Opc.....46

Req_TId

Signal *Req_TId* should be driven by the request identifier of the transaction, and must be supplied to the universal probe in order to measure the request-to-response latency on an interface whose protocol supports out-of-order transactions.

The universal probe uses this signal in conjunction with signal *Rsp_TId* to calculate the transaction latency.

The integrator must ensure that the value applied to *Req_TId* is unique for all pending transactions of a given master.

Req_User

Signal *Req_User* should be driven by any in-band qualifiers for the transaction. Its value can be mapped to user flags within the NoC, which can then be used to filter the transaction. The width of the signal is determined by parameter *wUser* in the parameter group *nTransactionPorts*.

NOTE Although the maximum value of *wUser* is 80, only the lower 32 user bits can be used for filtering in the transaction port of the universal probe.

Req_Security

Signal *Req_Security* should be driven by the security access type of the transaction. Its value can be mapped to security flags within the NoC, which can then be used to filter the transaction. The width of the signal is determined by parameter *wSecurity* in the parameter group *nTransactionPorts*.

od14839v1
Preliminary

Rsp_Valid

Signal *Rsp_Valid* should be driven to 1 from the beginning of the transaction response to the end of the response. However, it is only taken into account by the universal probe for the last response when signal *Rsp_Last* is asserted to 1.

od14846v1
Preliminary

Rsp_Ready

Signal *Rsp_Ready* should be driven to 1 when the transaction response has been accepted by the initiator. This signal is used by the universal probe in conjunction with *Req_Valid*, *Req_Tld* and *Rsp_Tld* to measure request-to-response latency.

od14847v1

Rsp_Last

Signal *Rsp_Last* should be driven to 1 for the final response word in the transaction response. This signal avoids the need to store the request length in the universal probe and count responses when interfacing to an SRMD protocol, such as AXI or OCP, or when using response interleaving.

The signal must not change while *Req_Valid* is set to 1 until *Rsp_Ready* is asserted to 1.

od14848v1
Preliminary

Rsp_Tld

Signal *Rsp_Tld* should be connected to the response identifier of the transaction that is generated by the target.

Rsp_Tld must be supplied to the universal probe when measuring request-to-response latency on an interface whose protocol supports out-of-order transactions. The probe uses this signal in conjunction with *Req_Tld* to calculate transaction latency.

This signal must not change while *Req_Valid* is set to 1 until *Rsp_Ready* is asserted to 1.

The integrator is responsible for ensuring that the value of the response identifier for a given transaction is the same as the request identifier, in order to allow the universal probe to re-associate the response with its request.

Responses to requests that did not match the request filter criteria, or responses to requests that were pending before the probe was activated, are ignored and do not affect statistics.

A single read response with *Rsp_Last* set to 1 is expected to have a response identifier that matches a pending request identifier. This constraint also applies to writes. If the protocol of the probed interface supports write requests without expecting a response, universal probe behavior will be unpredictable. On the other hand, early write responses are supported as long as the condition of having a single transaction pending with a given transaction identifier is preserved.

od14849v1
Preliminary

Probe parameters

Probe parameters are displayed in the [Probe](#) page of the FlexNoC FlexArtist design editor [Architecture](#) view [Observability](#) section.

od19586v1

observation

Parameter *observation* defines the type of probe to be connected to a link. When set to [Errors](#), an error probe is created; when set to [Packets](#), a packet probe is created. When set to [None](#), no observation service is available.

LOCATION Architecture: Observability: Probe

VALUES None (default), Errors, Packets

od19587v1

transactionStatPipe

When parameter *transactionStatPipe* is set to [True](#), a pipeline stage is added to the referenced transaction probe upstream of the transaction profiler and downstream of the clock adapter, if present.

The parameter can be set to [True](#) to resolve the timing issues that arise when NIUs that share a single probe are physically distant from each other, for instance when certain NIUs are located at the die edge, and others are located at the die center.

LOCATION Architecture: Observability: Probe

VALUES True, False (default)

od17714v1

module

Parameter *module* specifies the name of the RTL module in which the probe is instantiated.

The parameter is available for packet probes assigned to links, and transaction probes.

NOTE Modules are specified in the FlexArtist design editor [Architecture](#) view (Architecture: Modules).

LOCATION Architecture: Observability: Probe

VALUES module name, inherited (default)

od19589v1

observer

Parameter *observer* specifies the observer unit to which the observation packets, generated by the probe, are sent.

Observer units are created in the [Observation](#) page of the design editor [Specification](#) view [Interface](#) section.

When set to [NONE](#), no connection is made to an observer. This value is used when packet data is only to be read from the probe registers, and there is no requirement to send it as ATB packets to the host debug system.

LOCATION Architecture: Observability: Probe

VALUES None, List of available observers

nFilter

Parameter *nFilter* sets the number of filters to be instantiated in the FlexNoC packet probe unit tracing subsystem, which is used to select packets for tracing and statistics collection.

The parameter sets the maximum number of filters that can be combined and selected via the filter LUT in order to filter packets to be traced in the subsystem. If *nFilter* is set to 0, all packets are accepted and packet tracing depends entirely on the value of field *TraceEn* in register *MainCtl*. Alternatively, the output of an individual filter, either filtered packets or bytes, can be selected as the event type for statistic counters in the probe statistic collection subsystem.

NOTE Parameter *nFilter* is forced to 0, and packet filtering is not used, when the probe is connected to transaction probe points, that is, when parameter *transactionProfiling* is set to True.

Parameter *nFilter* can be set to a value greater than 0 even when parameter *tracing* is set to False. However, to prohibit unauthorized access, *control setting* parameters *freeze* and *rstVal* for field *TraceEn* of register *MainCtl* are hard-coded to True and 0 respectively.

LOCATION Architecture: Observability: Probe

VALUES 0–4

RELATED PARAMETERS

Security filtering parameters77

allowFilterOnUser

When set to True, parameter *allowFilterOnUser* enables filtering with user bits as criteria.

LOCATION Architecture: Observability: Probe

VALUES True, False (default)

RELATED REGISTERS

UserBase..... 110
UserMask 110

allowFilterOnEnabledBytes

When configuring packet probes, parameter *allowFilterOnEnabledBytes* should be set to True if probe statistics counters are to be programmed to count enabled bytes either from:

- § packets selected by the packet probe filters, that is, when register *Counters_M_Src* is set to *FILT_BYTE_EN*, or
- § from the packet probe LUT, when register *Counters_M_Src* is set to *LUT_BYTE_EN*.

For multi-port packet probes, the parameter can only be set to True if the serialization is the same across all ports.

For transaction probes, which comprise a transaction profiler and a packet probe, the parameter does not apply, and FlexNoC FlexArtist software forces *allowFilterOnEnabledBytes* to *False*.

LOCATION Architecture: Observability: Probe

VALUES True (default), False

od13159v1

wTimeStam p

Probe parameter *wTimeStam p* sets the width of *TimeStam p*, a probe input signal.

This parameter is only available when parameter *debugOutput* of the observer, to which the probe is connected, is set to STPv2 (Specification: Interface: Observation: debugOutput). In this case, the value of *wTimeStam p* is forced to the same value as parameter *wTimeStam p* of the observer (Specification: Interface: Observation).

LOCATION Architecture: Observability: Probe

VALUES 0–48 (default = N/A)

od15375v1
Preliminary

tracing

When set to *True*, parameter group *tracing* enables the probe trace function, adds alarm output signal *TraceAlarm* to the probe, and makes sub-parameters available for configuration.

LOCATION Architecture: Observability: Probe

VALUES True, False

od10225v1

allowPayloadTracing

When parameter *allowPayloadTracing* is set to *True*, the probe trace includes the payload of incoming packets when fields *TraceEn* and *PayloadEn* of register *MainCtl* are set.

If the probe has multiple ports, the register is preceded by a multiplexer that can switch the selected input port at a packet or locked-sequence boundary.

In addition, the following constraints apply to multiple port probes:

- § When *allowPayloadTracing* is set to *True*, each port must have the same value of parameter *headerPenalty*. The ports must also have the same value of parameter *nBytePerWord*, which must be greater than zero.
- § When *allowPayloadTracing* is set to *False*, the following constraints apply according to the value of *headerPenalty*:

<i>headerPenalt y</i>	<i>Constraint</i>
NONE	No constraint. Each port may be set to NONE or ONE independently.
ONE	
TWO	All ports must have the same values of headerPenalty and nBytePerWord.
AUTO	

LOCATION Architecture: Observability: Probe: tracing

VALUES True, False

od7680v1

crossTrigger

Parameter *crossTrigger* defines the protocol used for cross triggering. The only protocol supported is CoreSight.

LOCATION Architecture: Observability: Probe

VALUES None, CoreSight

od10227v1
Preliminary

Error probing parameters for response paths

Error probing parameters set the global policy on the copying of address and user flags from request packets to response packets by target NIUs that have at least one response path configured with an *Errors* probe point.

The parameters only appear in FlexArtist software when at least one target NIU has an *Errors* probe point on a response path.

od12095v1

keepAddressAndFlagsInResponsePacketsForErrorProbes

If an *Errors* probe point is specified on a response path and parameter *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to *True*, the address and all user flags in request packets are copied into response packets by the target NIU, for all types of errors.

This makes it possible to record address and user flag values in the error probe even when errors come from slaves (error code SLV), at the cost, however, of more storage overhead in the target NIU. More information is available in related reference documentation.

If *keepAddressAndFlagsInResponsePacketsForErrorProbes* is set to *False*, then the address and all user flags will only be copied to response packets if either of the following conditions are met:

- § A *Packets* probe point is on a response path of the target NIU.
- § An *Errors* probe point is on a response path of the target NIU and the error is of type UNS, DISC, SEC, or DEC, and it has been generated within the NoC transport. In the case of a slave error (SLV), only the 7 address LSBs are copied because other bits are no longer available in the target NIU when the error is raised. Furthermore, only user flags which are set to *True* in parameter *userFlagsKeptInResponsePacketsForErrorProbes* are copied into the response packet.

If neither of the preceding conditions are satisfied, field *Addr* is still present in the response packet, but the field value is invalid.

LOCATION Architecture: Global parameters

VALUES True, False (default)

od14239v1

RELATED REFERENCES

Reconstructing transaction addresses
..... 29

`userFlagsKeptInResponsePacketsForErrorProbes`

When parameter `userFlagsKeptInResponsePacketsForErrorProbes` is set to `True`, and an `Errors` probe point is specified on a response path, it is possible to individually select user flags that are to be copied by the target NIU from request packets to response packets in the event of a slave error (SLV).

Certain user flag values must be preserved because they are used in the security filter of the error probe, making it possible to record flag values in the error logger, at the cost, however, of more storage overhead in the target NIU.

NOTE If parameter `keepAddressAndFlagsInResponsePacketsForErrorProbes` is set to `True`, all user flags are copied into response packets, including for errors of type SLV. Therefore, all entries in `userFlagsKeptInResponsePacketsForErrorProbes` must be set to `True`.

LOCATION Architecture: Global parameters

VALUES `True`, `False` (default)

od14238v1

Packet probing parameters

The following packet probing parameters are available.

od10229v1
Preliminary

`probe`

Parameter `probe` specifies the name of the packet probe that is connected to the probe point.

LOCATION Architecture: Observability: Probe

VALUES List of available packet probes

od10230v1
Preliminary

`clock`

Parameter `clock` sets the clock for the referenced FlexArtist design element.

Clocks are created and configured in the [Clock](#) page of the design editor [Specification](#) view [Domain](#) section.

LOCATION Various places in FlexArtist editors

VALUES List of available clocks

od4827v1

`mode`

Parameter `mode` sets the referenced packet probe to one of two operating modes: `Spy` (default) or `Intrusive` (legacy FlexNoC 2.6. mode).

When set to `Spy`, the probe is non-intrusive: if the trace port is not ready, the trace is discarded, but transport flow is not blocked, and therefore no back pressure is applied. The probe power domain can be different from the domain of the associated link. The probe clock must not require any adaptation: it must have the same frequency and phase as the probed link, although it can have a different source.

When set to `Intrusive`, if the trace port is not ready, packet flow is blocked, and back pressure is applied. In this mode, probe clocks and probed link clocks must be the same.

A probe connected to a link can be set to either mode. A probe connected exclusively to transaction probe points can be set to **Spy** mode only, and its clock frequency must be greater than or equal to probe point clock frequency.

NOTE For backward compatibility, FlexNoC FlexArtist software automatically upgrades designs developed with preceding FlexNoC versions by setting all probes in the design to Intrusive mode operation.

LOCATION Architecture: Observability: Probe

VALUES Spy (default), Intrusive

Statistics collection parameters

od10233v1
Preliminary

The following statistics collection parameters are available.

od10234v1

statisticsCollection

When set to **True**, parameter group *statisticsCollection* enables statistics collection for the referenced probe, and group sub-parameters are displayed.

NOTE When a probe is configured for transaction profiling, that is, when parameter *transactionProfiling* is set to **True**, *statisticsCollection* is forced to **True**.

LOCATION Architecture: Observability: Probe

VALUES True, False

RELATED REGISTERS

Counters_M_Src.....102

od6417v2

nStatisticsCounter

When set, parameter *nStatisticsCounter* specifies the number of statistics counters in the probe. The counters accumulate occurrences of events or values coming from the packet filtering logic, transaction profiling logic, or external events.

If the probe is used by transaction probe points, which requires parameter *transactionProfiling* to be set to **True**, the minimum permissible value of *nStatisticsCounter* is the greater of the probe's associated transaction profiler parameters *nObservable* and (*nComparators* + 1). This constraint ensures that, as a minimum, the following transaction profiling scenarios are supported:

- § Events in all transaction profiling bins associated with a selected NIU can be counted.
- § Events in a selected bin for each of the observed NIUs can be counted.

To count events in all transaction profiling bins for *nObservable* NIUs simultaneously, *nStatisticsCounters* must be set to (*nObservable* × (*nComparators* + 1)).

The maximum value of *nStatisticsCounter* is restricted to:

$$2 \times (nObservable \times (nComparators + 1) + nExternalEvent)$$

up to a maximum value of 32. This represents the minimum logic resources required to support the worst case configuration where chained statistic counters are mapped to all bins for all observed ports and all external event inputs.

LOCATION Architecture: Observability: Probe: statisticsCollection

VALUES 1–32

od18502v1

wStatisticsCounter

Parameter *wStatisticsCounter* sets the width, in bits, of the statistics counters in the probe.

LOCATION Architecture: Observability: Probe: statisticsCollection

VALUES 8–16

od6419v1
Preliminary

statisticsCounterAlarm

When parameter *statisticsCounterAlarm* is set to **True**, an alarm signal, *StatAlarm*, is added to the probe.

When set to **False**, the following registers are reserved:

StatAlarmMin

StatAlarmMax

StatAlarmStatus

StatAlarmClr

StatAlarmEn

LOCATION Architecture: Observability: Probe: statisticsCollection

VALUES True, False.

od10238v1

RELATED REGISTERS

StatAlarmMin	93
StatCondDump	89

allowStatisticsSuspend

When parameter *allowStatisticsSuspend* is set to **True**, statistics collection can be temporarily suspended under the control of an external tactical input. The input should be connected to port *StatSuspend* of the referenced probe.

LOCATION Architecture: Observability: Probe: statisticsCollection

VALUES True, False (default)

od9900v1

nExternalEvent

Parameter *nExternalEvent* sets the number of event signals collected by the referenced probe. Sources external to the NoC can drive the signals.

The signals can be connected to probe statistics counters by setting parameter *transactionProfiling* to **True** (Architecture: Observability: Probe), which reserves a number of probe inputs for transaction-profiling histogram bins. The number of reserved inputs is equal to:

$$r = nObservable \times (nComparators + 1)$$

Consequently, the maximum *available* value for *nExternalEvent* is reduced to $\max(nExternalEvent) - r$.

VALUES 0–256

LOCATION Architecture: Observability: Probe: statisticsCollection

RELATED PARAMETERS

transactionProfiling..... 73

od7679v1

Transaction profiling parameters

The following transaction profiling parameters are available.

transactionProfiling

When parameter group *transactionProfiling* is set to *True*, transaction profiling is enabled for the referenced probe, and sub-parameters become available.

FlexNoC FlexArtist software sets this parameter to *True* if one or more Transactions probe points have been assigned to the probe.

LOCATION Architecture: Observability: Probe

VALUES True, False

od10241v1
Preliminary

nCounters

Parameter *nCounters* defines the number of transaction profiling counters in the referenced transaction probe. The value of *nCounters* must be a multiple of 4 up to a maximum of 32. The *nCounters* transaction profiling counters are assigned to the observed NIUs at run time, in groups of four, by programming the registers *nTenureLines_i*.

NOTE These counters are distinct from the statistics counters that accumulate the histogram data.

The number of transaction profiling counters determines the total number of transaction contexts, associated with the probed NIUs, which can be tracked simultaneously by the transaction probe. The number of contexts for a given observed port depends on the port transaction profiling mode, set via transaction profiler register *Mode*.

When the mode is set to *DELAY*, the probed NIU creates a context for each transaction it issues. Each context is mapped to a separate transaction profiling counter from the set of counters that have been allocated to the observed NIU.

However, the actual number of transaction profiling counters required is greater than the number of transactions to be tracked. This overhead depends on the configuration of the NIU. Furthermore, the number of counters required must be rounded up to the next multiple of 4 because the counters can only be allocated to NIUs in groups of four.

Therefore, the number of transaction profiling counters required to track the delay of *N* transactions of a single probed NIU is:

od10242v1
Preliminary

$$\text{transaction profiling counters per probed NIU} = 4\{\text{ceil}[(N + 1) + A + B + C + D] / 4\}$$

where:

A = 1 if the *postDecode* forward pipe is enabled in the NIU, otherwise A = 0.

B = 1 if the *postSplit* forward pipe is enabled in the NIU, otherwise B = 0.

C = 1 if the *postSplit* backward pipe is enabled in the NIU, otherwise C = 0.

D = 1 if the NIU is configured for fixed bursts by setting parameter *useFixed* (Specification: Interface: NIU: protocol) to **True** and the data width of the target is less than the data width of the initiator, otherwise D = 0.

The preceding calculation should be repeated for all observed NIUs for which the associated transaction probe port is programmed for delay mode. The required value of *nCounters*, up to the limit of 32, then becomes:

$$nCounters = \text{sum}(\text{transaction profiling counters per probed NIU})$$

If more than 32 counters are required, then it will not be possible to track all potential transactions and these will be excluded from the transaction profile analysis.

NOTE Specifying a value for *nCounters* that is greater than the value obtained with the preceding expression is not permitted because the extra counters cannot be exploited, and would incur a waste of logic resources.

When the transaction profiling mode is set to **PENDING**, only one transaction profiling counter per observed NIU, from the set of counters that have been allocated to the NIU, is used.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 4–32

od10243v1

wCounters

Parameter *wCounters* sets the width for all the transaction profiling counters in the referenced transaction probe.

The width determines the maximum threshold value that can be assigned to the elements of parameters *delayThresholds* or *pendingThresholds*, where:

$$\text{maximum value of threshold element} = (2^{wCounters}) - 1$$

NOTE The counters do not wrap but saturate at a value of -1 .

The lower limit on *wCounters* is determined by the largest value of parameter *nPendingTrans* (Architecture: Datapath NIU: NIU) for all the NIUs observed by the probe.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 3–11

od10244v1

nObservable

Parameter *nObservable* sets the number of transaction ports that can be observed simultaneously.

Setting lower values reduces the amount of logic required for multiplexers and comparators and shortens timing paths. The setting should be high enough however to handle the maximum number of transactions to be counted simultaneously.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 1–4

od10245v1

delayThresholds

Parameter *delayThresholds* specifies a list of *possible* threshold levels that can be used to define histogram bins when measuring transaction latency.

Each bin is incremented if the measured delay or latency falls within the bounds of that bin. The lower and upper limits of all the possible bins are defined as:

```
For k = 0,
    bin(k) < delayThresholds(k)
For k = 1 to (delayThresholds - 2)
    delayThresholds(k - 1) <= bin(k) < delayThresholds(k)
For k = (delayThresholds - 1)
    bin(k) >= delayThresholds(k)
```

This set of thresholds is used for transaction profiling when the bit corresponding to the probe port number in the transaction profiler register *Mode* is set to DELAY. The particular thresholds used by the comparators are selected via registers *Thresholds_i_j*.

The maximum value that list elements can take is determined by the width of the transaction profiling counters, which is set by parameter *wCounters*:

Max value of delay threshold = $2^{wCounters} - 1$

NOTE If the number of elements in both parameter arrays *delayThresholds* and *pendingThresholds* is zero (Architecture: Observability: Probe: transactionProfiling), threshold values are directly programmed into the registers *Thresholds_i_j*.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 1–2047

od17415v1

RELATED PARAMETERS

pendingThresholds.....75

RELATED REGISTERS

Thresholds_i_j.....113
Mode112

pendingThresholds

Parameter *pendingThresholds* specifies a list of *possible* threshold levels that can be used to define histogram bins when measuring the number of pending transactions.

Each bin is incremented if the number of pending transactions falls within the bounds of that bin. The lower and upper limits of all the possible bins are defined as:

```

For k = 0,
    bin(k) < pendingThresholds(k)
For k = 1 to (pendingThresholds - 2)
    pendingThresholds(k - 1) <= bin(k) < pendingThresholds(k)
For k = (pendingThresholds - 1)
    bin(k) >= pendingThresholds(k)

```

This set of thresholds is used for transaction profiling when the bit corresponding to the probe port number in the transaction profiler register *Mode* is set to **PENDING**. The particular thresholds used by the comparators are selected via registers *Thresholds_i_j*.

The maximum value that list elements can take is determined by the width of the transaction profiling counters, which is set by parameter *wCounters*:

Max value of pending threshold = $2^{wCounters} - 1$

NOTE If the number of elements in both parameter arrays *delayThresholds* and *pendingThresholds* is zero (Architecture: Observability: Probe: transactionProfiling), threshold values are directly programmed into the registers *Thresholds_i_j*.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 1–2047

RELATED PARAMETERS

delayThresholds 75

RELATED REGISTERS

Thresholds_i_j 113

Mode 112

od17416v1

nComparators

Parameter *nComparators* sets the number of comparators used to sort measured latency, or the number of pending transactions, into histogram bins based on the active thresholds.

The number of bins *per observed port* is $(nComparators + 1)$.

The range of values which can be assigned to *nComparators* depends on the number of thresholds specified in parameter arrays *delayThresholds* and *pendingThresholds*.

If *delayThresholds* and *pendingThresholds* both contain non-zero values, where comparator values are selected by setting registers *Thresholds_i_j* to the index values of the threshold arrays ("fixed" threshold mode), then *nComparators* must satisfy:

$nComparators \leq \min(\text{number of elements in } delayThresholds, \text{number of elements in } pendingThresholds)$

This constraint makes it advisable to maintain the same number of list elements in the delay and pending thresholds lists. If necessary to obtain parity, add thresholds of one type.

If *delayThresholds* and *pendingThresholds* both contain zero values, where comparator values are programmed directly in registers *Thresholds_i_j* ("programmable" threshold mode), then *nComparators* can take any value between 1 and 4.

NOTE Logic increases by a factor of *nObservable* as *nComparators* increases.

LOCATION Architecture: Observability: Probe: transactionProfiling

VALUES 1–4

RELATED PARAMETERS

<i>delayThresholds</i>	75
<i>pendingThresholds</i>	75
<i>nObservable</i>	74

od17311v1

Security filtering parameters

The following security filtering parameters are available.

securityFiltering

Parameter group *securityFiltering* defines the type of security filtering used by the referenced probe.

When set to **Standard filtering**, sub-parameters are available to configure user flag masks, and optional SPIDEN input, to determine which packets can enter a probe.

When set to **User-defined filtering**, sub-parameters are available to configure complex schemes with dependencies between flags or one or more input sideband signals, or both.

LOCATION Architecture: Observability: Probe

VALUES None, Standard filtering, User-defined filtering

RELATED PARAMETERS

<i>nFilter</i>	67
----------------------	----

RELATED REFERENCES

Configuring probes for security filtering	22
---	----

od10249v1
Preliminary

od10250v1

Standard filtering parameters

The following standard filtering parameters are available.

flagFiltering

Parameter *flagFiltering* defines a match pattern for supported interface signals, as well as user and security flags. When the pattern matches, the packet is sent to the observer.

LOCATION Architecture: Observability: Probe: securityFiltering

VALUES x, 0, 1

od10252v1
Preliminary

od10253v1

useSPIDENinput

When parameter *useSPIDENinput* is set to **True**, input signal *SPIDEN* is used by the security filter. If no *SPIDEN* input is available, the security filter considers the input to be **0**.

LOCATION Architecture: Details: Probe: securityFiltering

VALUES True, False (default)

od8876v1
Preliminary

useSPNIDENinput

Parameter *useSPNIDENinput* enables *SPIDEN* input for the security filter of the referenced probe point.

When the *SPIDEN* signal is driven to **1**, packets are not filtered. When the signal is driven to **0**, or when the signal is not present, only packets whose user and security flags match the setting for parameter group *flagFiltering* (Architecture: Observability: Probe: securityFiltering) are allowed to proceed.

LOCATION Architecture: Observability: Probe: securityFiltering

VALUES True, False (default)

od19396v1
Preliminary

useNIDENinput

Parameter *useNIDENinput* enables *NIDEN* input for the security filter of the referenced probe point.

When the *NIDEN* signal is driven to **1**, or when the signal is not present, packets are not filtered. When the signal is driven to **0**, all packets are blocked.

LOCATION Architecture: Observability: Probe: securityFiltering

VALUES True, False (default)

od19397v1
Preliminary

User-defined filtering parameters

The following user-defined filtering parameters are available.

od10255v1
Preliminary

coreName

Parameter *coreName* specifies the name of the customer cell which will contain the filtering logic associated with the referenced probe. By default, *coreName* is set to the name of the probe, and suffixed with *_Core*.

LOCATION Architecture: Details: Probe: securityFiltering

VALUES text

od10256v1
Preliminary

interfaceFlags

Parameter *interfaceFlags* lists the interconnect protocol, user and security flags that will be made available at the port interface of the customer cell.

LOCATION Architecture: Details: Probe: securityFiltering

VALUES List of flags

od10257v1
Preliminary

sidebands

Parameter *sidebands* sets a list of sideband signals connected to the referenced design element.

The signals are characterized by sub-parameters *name*, *direction*, and *width*.

In certain cases, sub-parameter *async* can be used to specify that the signal is asynchronous with the design element core.

LOCATION Architecture: Datapath Transport: Firewall: version, other places in FlexArtist software

VALUES 0-128

od5529v2

Universal Probe parameters

Because the universal probe is constructed from underlying FlexNoC observability hardware, its configuration uses many of the parameters associated with observability packet probing and transaction profiling functions.

In addition, the probe has its own set of dedicated parameters, exclusively described in this section. More information concerning other parameters associated with constituent observability units is available in related reference documentation.

od14807v1

RELATED PARAMETERS

Probe parameters 66

clock

Parameter *clock* sets the clock for the referenced FlexArtist design element.

Clocks are created and configured in the [Clock](#) page of the design editor [Specification](#) view [Domain](#) section.

LOCATION Various places in FlexArtist editors

VALUES List of available clocks

od4827v1

IPowerDomain

Parameter *IPowerDomain* specifies the power domain of the interface which is probed by the universal probe. The value is common to all request-only ports and transaction probes for a given universal probe.

The parameter only effects the exported power intent file: isolation cells are specified on the probe port signals of the universal probe if all of the following conditions are satisfied:

- § The probed interface and universal probe are in different power domains.
- § The probed interface is of type [SUPPLY](#).
- § The power intent parameter *addIsolationCellOnInputPorts* is set to [True](#).

LOCATION Specification: Interface: Observation

VALUES List of available power domains

od14902v1

nRequestPorts

Parameter group *nRequestPorts* sets the number of configurable request ports. When set to a value other than 0, the parameter provides access to parameter groups *signalConfiguration* and *portNames*.

LOCATION Specification: Interface: Observation

VALUES 0–8

od14809v1
Preliminary

signalConfiguration

Parameter group *signalConfiguration* configures the widths of the request port signals of the referenced Universal Probe. The values of its sub-parameters are common to all the ports.

VALUES List

od14909v1
Preliminary

wId

Parameter *wId* sets the width of signal *Id* on universal probe request ports.

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 0–16

od14873v1
Preliminary

wAddress

Parameter *wAddress* sets the width of signal *Address* on universal probe request ports.

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 8–48

od14874v1

wLength

Parameter *wLength* sets the width of signal *Length* on universal probe request ports.

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 2–12

od14880v1
Preliminary

wPrio

Parameter *wPrio* sets the width of signal *Prio* on the universal probe request ports.

Prio is mapped by the universal probe to the FlexNoC packet header field *Urgency*. Because this field is bar-graph-encoded, it must be sized according to *wPrio*, by setting the value for parameter *nUrgencyLevel* as follows:

$$nUrgencyLevel = 2^{wPrio} - 1$$

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 0–3

wUserod14881v1
Preliminary

Parameter *wUser* sets the width, in bits, of signal *User* on universal probe request ports.

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 0–80

wSecurity

od14878v1

Parameter *wSecurity* sets the width of signal *Security* on the universal probe request ports.

LOCATION Specification: Interface: Observation: nRequestPorts:
signalConfiguration

VALUES 0–8

portNamesod14879v1
Preliminary

The names of universal probe request ports are specified by parameter *portNames*. The default name is the name of the universal probe suffixed with the port number.

The name of each universal probe request port signal in the exported RTL netlist is concatenated from the port and signal names.

LOCATION Specification: Interface: Observation: nRequestPorts

VALUES 0–*nRequestPorts*

nTransactionPortsod14882v1
Preliminary

Parameter group *nTransactionPorts* sets the number of configurable transaction ports. When set to a value other than 0, provides access to parameter groups *signalConfiguration*, *portClocks*, and *portNames*.

LOCATION Specification: Interface: Observation

VALUES 0–16

signalConfigurationod14810v1
Preliminary

Parameter group *signalConfiguration* configures the widths of the transaction port signals of the referenced universal probe. The values of its sub-parameters are common to all the ports.

VALUES List of port signal widths

wIdod14908v1
Preliminary

Parameter *wId* sets the width of signal *Req_Id* on universal probe transaction ports.

LOCATION Specification: Interface: Observation: nTransactionPorts:
signalConfiguration

VALUES 0–16

od14883v1
Preliminary

wTld

Parameter *wTld* sets the width of signals *Req_Tld* and *Rsp_Tld* on universal probe transaction ports.

LOCATION Specification: Interface: Observation: nTransactionPorts:
signalConfiguration

VALUES 1–7

od14884v1
Preliminary

wAddress

Parameter *wAddress* sets the width of signal *Req_Address* on universal probe transaction ports.

LOCATION Specification: Interface: Observation: nTransactionPorts:
signalConfiguration

VALUES 8–48

od14885v1

wUser

Parameter *wUser* sets the width, in bits, of signal *Req_User* on universal probe transaction ports.

LOCATION Specification: Interface: Observation: nTransactionPorts:
signalConfiguration

VALUES 0–80

od14886v1

wSecurity

Parameter *wSecurity* sets the width of signal *Req_Security* on universal probe transaction ports.

LOCATION Specification: Interface: Observation: nTransactionPorts:
signalConfiguration

VALUES 0–8

od14887v1
Preliminary

portClocks

The clocks used for universal probe transaction ports are specified by parameter *portClocks*. By default, the universal probe clock is used, as specified by parameter *clock* in the design editor [Specification](#) view [Interface](#) section [Observation](#) page.

LOCATION Specification: Interface: Observation: nTransactionPorts

VALUES 0–*nTransactionPorts*

od14889v1
Preliminary

portNames

The names of the universal probe transaction ports are specified by parameter *portNames*. The default name is the name of the universal probe suffixed with a port number.

The name of each universal probe transaction port signal in the exported RTL netlist is concatenated from the port and signal names.

LOCATION Specification: Interface: Observation: nTransactionPorts

VALUES 0–*nTransactionPorts*

od14888v1
Preliminary

Security filtering

Parameter group *securityFiltering* sets filtering for Universal Probe unit request ports and transaction ports.

requestPorts

Parameter group *requestPorts* sets the type of security filtering to be applied to the request-only ports of the universal probe.

When set to *Standard filtering*, sub-parameters *userBitMatch*, *securityBitMatch*, and *useSPIDENinput* become available.

When set to *User-defined filtering*, sub-parameters *coreName* and *sidebands* become available.

When set to *None*, no security filtering is applied.

VALUES None (default), Standard filtering, User-defined filtering

od8826v1
Legacy
Archive

userBitMatch

Parameter *userBitMatch* specifies the security filter match criteria for the bits of signal *User* of the universal probe request-only ports.

NOTE These criteria are ignored when parameter *useSPIDENinput* is set to *True* and signal *SPIDEN* to the universal probe is set to 1.

LOCATION Architecture: Observability: Probe: securityFiltering: requestPorts

VALUES x(default), 0, 1

od14891v1
Preliminary

securityBitMatch

Parameter *securityBitMatch* specifies the security filter match criteria for the bits of signal *Security* of universal probe request-only ports.

NOTE These criteria are ignored when parameter *useSPIDENinput* is set to *True* and signal *SPIDEN* to the universal probe is set to 1.

LOCATION Architecture: Observability: Probe: securityFiltering: requestPorts

VALUES x(default), 0, 1

od14892v1
Preliminary

useSPIDENinput

When parameter *useSPIDENinput* is set to *True* and input signal *SPIDEN* is 1, all transactions on request ports pass through security filtering, regardless of match criteria specified in security parameters *userBitMatch* and *securityBitMatch* for the request port.

LOCATION Architecture: Observability: Probe: securityFiltering: requestPorts

VALUES True, False (default)

od14895v1

od14813v1
Preliminary

coreName

Parameter *coreName* specifies the name of the customer cell which implements the user-defined security filtering logic for universal probe request ports. The default name is the concatenation of the probe name and the suffix *_Core*.

LOCATION Architecture: Observability: Probe: securityFiltering: requestPorts

VALUE customer cell name

od14811v1
Preliminary

sidebands

Parameter *sidebands* sets a list of sideband signals connected to the referenced design element.

The signals are characterized by sub-parameters *name*, *direction*, and *width*.

In certain cases, sub-parameter *async* can be used to specify that the signal is asynchronous with the design element core.

LOCATION Architecture: Datapath Transport: Firewall: version, other places in FlexArtist software

VALUES 0-128

od5529v2

name

Sideband sub-parameter *Name* sets the name of the referenced sideband signal.

VALUES Text string

od16423v1

direction

Parameter *direction* sets the input or output direction of the referenced port or signal.

FlexNoC FlexArtist software forces the parameter value for certain port types, such as *user-type* ports for example, which can only be input ports.

VALUES Input, output

od2311v1

width

Parameter *width* sets the width, in bits, of the referenced port or signal.

The range of available values varies according to the type of the signal or port.

VALUES Depend on signal or port

od2312v1

async

Sideband sub-parameter *async* must be set to *True* if the core sideband signal source is generated by a clock regime different from the one used by the customer cell.

VALUES True, False (default)

od16426v1

transactionPorts

Parameter group *transactionPorts* sets the type of security filtering to be applied to the transaction ports of the universal probe.

When set to **Standard** filtering, sub-parameters *userBitMatch*, *securityBitMatch*, and *useSPIDENinput* become available.

When set to **User-defined** filtering, sub-parameters *coreName* and *sidebands* become available.

When set to **None**, no filtering is applied.

VALUES None (default), Standard filtering, User-defined filtering

od14898v1
Preliminary

userBitMatch

Parameter *userBitMatch* specifies the security filter match criteria for the bits of signal *Req_User* of the universal probe transaction ports.

NOTE These criteria are ignored when parameter *useSPIDENinput* is set to **True** and signal *SPIDEN* to the universal probe is set to 1.

LOCATION Architecture: Observability: Probe: securityFiltering:
transactionPorts

VALUES x(default), 0, 1

od14899v1
Preliminary

securityBitMatch

Parameter *securityBitMatch* specifies the security filter match criteria for the bits of signal *Req_Security* of universal probe transaction ports.

NOTE These criteria are ignored when parameter *useSPIDENinput* is set to **True** and signal *SPIDEN* to the universal probe is set to 1.

LOCATION Architecture: Observability: Probe: securityFiltering:
transactionPorts

VALUES x(default), 0, 1

od14900v1

useSPIDENinput

When parameter *useSPIDENinput* is set to **True** and input signal *SPIDEN* is 1, all transactions on request ports pass through security filtering, regardless of match criteria specified in security parameters *userBitMatch* and *securityBitMatch* for the transaction port.

LOCATION Architecture: Observability: Probe: securityFiltering:
transactionPorts

VALUES True, False (default)

od14814v1
Preliminary

coreName

Parameter *coreName* specifies the name of the customer cell which implements the user-defined security filtering logic for universal probe transaction ports. The default name is the concatenation of the probe name and the suffix *_Core*.

LOCATION Architecture: Observability: Probe: securityFiltering:
transactionPorts

VALUES customer cell name

od14815v1
Preliminary

sidebands

Parameter *sidebands* sets a list of sideband signals connected to the referenced design element.

The signals are characterized by sub-parameters *name*, *direction*, and *width*.

In certain cases, sub-parameter *async* can be used to specify that the signal is asynchronous with the design element core.

LOCATION Architecture: Datapath Transport: Firewall: version, other places in FlexArtist software

VALUES 0-128

od5529v2

name

Sideband sub-parameter *Name* sets the name of the referenced sideband signal.

VALUES Text string

od16423v1

direction

Parameter *direction* sets the input or output direction of the referenced port or signal.

FlexNoC FlexArtist software forces the parameter value for certain port types, such as *user-type* ports for example, which can only be input ports.

VALUES Input, output

od2311v1

width

Parameter *width* sets the width, in bits, of the referenced port or signal.

The range of available values varies according to the type of the signal or port.

VALUES Depend on signal or port

od2312v1

async

Sideband sub-parameter *async* must be set to *True* if the core sideband signal source is generated by a clock regime different from the one used by the customer cell.

VALUES True, False (default)

od16426v1

Probe registers

Probe functions, such as tracing, filtering, statistics collection and transaction profiling, are run-time programmable via registers. For a given function, in addition to the specific registers for that function, the register set includes standard identification registers ([../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf](#)).

The base address for each function is specified in parameter *base* (Architecture: Service: Registers).

Reset values of read–write registers can be specified in parameter *rstVal* (Structure: Parameters: Control Setting).

NOTE Setting *rstVal* to Set by jumper is not supported for all probe registers.

Packet probe global registers

od10264v1

The FlexNoC *packet probe* unit has the following global configuration registers that can be programmed for packet tracing and statistics collection.

NOTE The last column in the table indicates if the operation is guaranteed after the register is written to when a probe is active, that is, when field *Active* of register *CfgCtl* is 1.

Address offset	Name	Type	Retention	Operation guaranteed when probe active
0x0008	MainCtl	Control	Yes	No
0x000C	CfgCtl	Control	Yes	Yes
0x0010	TracePortSel ^{1,2}	Control	Yes	Yes
0x0014	FilterLut ²	Control	Yes	No
0x0018	TraceAlarmEn ²	Control	Yes	No
0x001C	TraceAlarmStatus ²	Status	No	not applicable (read only)
0x0020	TraceAlarmClr ²	Control	Yes	Yes
0x0024	StatPeriod ³	Control	Yes	No
0x0028	StatGo	Control	Yes	Yes
0x002C	StatAlarmMin ³	Control	Yes	No
0x0030	StatAlarmMax ³	Control	Yes	No
0x0034	StatAlarmStatus ³	Status	No	not applicable (read only)
0x0038	StatAlarmClr ³	Control	Yes	Yes
0x003C	StatAlarmEn ³	Control	Yes	Yes

Table notes

- 1 The register is only present if the packet probe is connected to more than one packet probe point.
- 2 The register is not available if probe parameter *transactionProfiling* is set to True.
- 3 The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

RELATED PARAMETERS

statisticsCollection 71

od14240v1
Revised

MainCtl

Register *MainCtl* contains the following probe control bits.

Bit range	Field name	Applies to transaction probe?
0	ErrEn	No
1	TraceEn	No
2	PayloadEn	No
3	StatEn	Yes
4	AlarmEn	Yes
5	StatCondDump	Yes
6	IntrusiveMode	No
7	FiltByteAlwaysChainableEn	Yes

od12097v1

ErrEn

Register field *ErrEn* enables the probe to send any packet with Error status to an observer, independently of filtering mechanisms, thus constituting a simple supplementary global filter.

od10267v1
Preliminary

TraceEn

Register field *TraceEn* enables the probe to send traced packets to an observer.

The register field is not available if the probe parameter *transactionProfiling* is set to True.

od10268v1

PayloadEn

When register field *PayloadEn* is set to 1, the packet trace includes payload as well as headers. When set to 0, only headers are reported.

Parameter *allowPayloadTracing* must be set to True for *PayloadEn* to have an effect on the packet trace format.

The register field is not available if the probe parameter *transactionProfiling* is set to True.

This register field has no effect with Universal Probes.

od15205v1

StatEn

When set to 1, register field *StatEn* clears the values in the statistic counters, enables event counting, and activates the dump timer, provided:

- § field *GlobalEn* of register *CfgCtl* is set to 1, or
alternatively, channel 0 of signal *CtiTrigIn* is 1,
- and**

- § if implemented, signal *StatSuspend* is 0.

When *StatEn* is cleared to 0, the statistics counters are frozen and the dump timer is suspended *after a statistics dump*.

od10270v1

AlarmEn

When set to 1, register field *AlarmEn* enables the trace alarm output signal *TraceAlarm*. This signal will then be asserted to 1 if a trace alarm event occurs (one or more bits are set in register *TraceAlarmStatus*).

When set to 1, register field *AlarmEn* also enables the statistics alarm output signal *StatAlarm*. This signal will then be asserted to 1 if a statistics alarm event occurs (the corresponding bit is set in register *StatAlarmStatus*) and the corresponding bit in register *StatAlarmEn* is set to 1 (default value).

When *AlarmEn* is cleared to 0, both alarm output signals are driven to 0.

od14241v1
Preliminary

StatCondDump

If register field *StatCondDump* is 1, statistic counter values are only dumped into statistics frames when a statistics dump request occurs if *one or more statistics counters contains a value that is out-of-bounds*, as defined via registers *StatAlarmMin*, *StatAlarmMax*, and *Counters_M_AlarmMode*. The statistics counters are reset, statistics collection automatically resumes, but no alarm event occurs:

- § Output signal *StatAlarm* is not asserted, even if it is enabled.
- § Register *StatAlarmStatus* is inoperative.

If register field *StatCondDump* is 0, statistic counter values are dumped into statistics frames when a statistics dump request occurs. The statistics counters are reset and statistics collection automatically resumes. However, if one or more statistics counters contains a value that is out-of-bounds, the statistics counters are frozen, further dumping is suspended, and an alarm event is generated:

- § Output signal *StatAlarm* is asserted if it is enabled.
- § The corresponding bit in register *StatAlarmStatus* is set to 1.

NOTE To implement field *StatCondDump*, parameter *statisticsCounterAlarm* must be set to True, otherwise the field is reserved.

StatCondDump operates with both automatic (register *StatPeriod*) and manual (register *StatGo*) statistics dump request mechanisms.

The following table summarizes *StatCondDump* operation.

StatCondDump	Statistic counters out-of-bounds at dump ?	Statistics dump ?	Statistic counters frozen ?	Alarm event generated ?
0	No	Yes	No	No
	Yes	Yes	Yes	Yes
1	No	No	Not applicable	No
	Yes	Yes	No	No

RELATED PARAMETERS

statisticsCounterAlarm 72

od7692v4

IntrusiveMode

When set to 1, register field *IntrusiveMode* enables trace operation in *intrusive flow-control* mode. In this mode, all observed packets are sent to the output of the observer. A flow control mechanism is used at the input of the observer in order to guarantee that no packets are dropped.

When *IntrusiveMode* is set to 0, tracing uses *Overflow flow-control* mode. In case of a header or payload overflow, packets are dropped and the overflow is signaled to the observer.

In spy mode, if the trace port is not ready, the trace will be discarded but transport flow is not blocked, unlike in intrusive mode where a non-ready trace port would block the flow.

For more information see related references.

NOTE If probe parameter *mode* is set to *Spy* then *IntrusiveMode* is hard-coded to 0.

This register field has no effect with Universal Probes.

od15206v1

FiltByteAlwaysChainableEn

Register field *FiltByteAlwaysChainableEn* controls mapping of filter outputs to counters when counting either bytes or enabled bytes.

When the field is set to 0, filter outputs are mapped to all counters. As a result, only bytes or enabled bytes for filters that are mapped to even counters can be counted using a pair of chained counters.

When set to 1, filter outputs are mapped only to even counters, in which case bytes or enabled bytes from any filter can be counted using a pair of chained counters.

For an example of the effect of *FiltByteAlwaysChainableEn*, see event types *FILT_BYTE* or *FILT_BYTE_EN* in the description of related register *Counters_M_Src*.

od12092v1

RELATED REGISTERS

Counters_M_Src..... 102

CfgCtl

Register *CfgCtl* contains global enable and active bits. The register, which must be used by software before changing certain packet probe global registers, has two fields:

Bit range	Field name
0	GlobalEn
1	Active

od10274v1

GlobalEn

Setting register field *GlobalEn* to 1 enables the tracing and statistics collection subsystems of the packet probe.

When the field is cleared or reset, all fields of register *MainCtl* are also cleared or reset.

od7684v1

Active

Register field *Active* is used to inform software that the probe is active. Probe configuration is not allowed during the active state. This bit is raised when bit *GlobalEn* is set, and is cleared a few cycles after setting *GlobalEn* to zero (probe is Idle).

od10276v1
Preliminary**TracePortSel**

Register *TracePortSel* specifies the probe point to be used by the packet probe for tracing or collecting statistics, when one of the following filter events is selected for that counter (see register *Counters_M_Src*):

LUT	FILTO	FILT1	FILT2	FILT3
LUT_BYTE_EN	LUT_BYTE	FILT_BYTE_EN	FILT_BYTE	

When filtered packets or bytes are being traced or counted, or both, changes to this register are only taken into account by the trace filters at packet boundaries. On the other hand, changes are seen immediately by statistic counters when counting other event types.

The number of bits in register *TracePortSel* is:

\log_2 (number of probe points to which the probe is connected)

The mapping of probe port numbers to probe points for a given packet probe is visible in the:

- § FlexNoC FlexArtist [Information](#) pane when the probe is selected in the [Probe](#) page of the Architecture Editor [Details](#) view.
- § Mapping description section of the exported structure information file.

od7694v1

FilterLut

Register *FilterLut* controls the filter look-up table (LUT), which is used to select which filters are used and in what logical combination. The packet is forwarded to the observer if the packet header fields match the programmed combination of filter criteria.

NOTE Even when there is only one packet filter, it must be explicitly selected using *FilterLut* in order for the packets to be filtered.

The number of bits in register *FilterLut* depends on the number of filters, set by parameter *nFilter*, and is equal to $2^{nFilter}$.

When *nFilter* is set to 0, *FilterLut* acts as a simple gate that determines whether or not packets are routed to statistics counters: set *FilterLut* to 1 to forward packets to counters.

The register is not available if probe parameter *transactionProfiling* is set to True.

od10004v1

RELATED PARAMETERS

[nFilter](#) 67

RELATED REFERENCES

[Combining filters](#) 36

TraceAlarmEn

Register *TraceAlarmEn* enables hits from either trace filters or the look-up table (LUT) to be recorded in the trace alarm status register *TraceAlarmStatus*.

The number of bits in register *TraceAlarmEn* is equal to the setting for parameter *nFilter* + 1.

Setting any of the bits 0 to (*nFilter* – 1) to 1 will cause hits from filters 0 to (*nFilter* – 1) to assert the corresponding bit in register *TraceAlarmStatus*. Setting bit (*nFilter*) to 1 will cause LUT hits to set bit (*nFilter*) in register *TraceAlarmStatus*.

The register is not available if probe parameter *transactionProfiling* is set to True.

When parameter *nFilter* is set to 0, *TraceAlarmEn* is reserved.

od10279v1
Preliminary

RELATED REFERENCES

Configuring and managing trace alarm events37

TraceAlarmStatus

Register *TraceAlarmStatus* is a read-only register that indicates which look-up table or filter has been matched by a packet.

The number of bits in *TraceAlarmStatus* is equal to the setting for parameter *nFilter* + 1.

A value of 1 in bits 0 to (*nFilter* – 1) indicates a hit by filters 0 to (*nFilter* – 1) respectively. A value of 1 in bit (*nFilter*) indicates a LUT hit.

When parameter *nFilter* is set to 0, *TraceAlarmStatus* is reserved.

The register is not available if probe parameter *transactionProfiling* is set to True.

od10280v1

TraceAlarmClr

Setting a bit to 1 in register *TraceAlarmClr* clears the corresponding bit in register *TraceAlarmStatus*.

The number of bits in register *TraceAlarmClr* is equal to the setting for parameter *nFilter* + 1.

When parameter *nFilter* is set to 0, *TraceAlarmClr* is reserved.

NOTE The written value is not stored in the register. A read always returns 0.

The register is not available if probe parameter *transactionProfiling* is set to True.

od10281v1

StatPeriod

Register *StatPeriod* is a 5-bit register that sets a period ranging from 2 cycles to 2 gigacycles of the probe clock, during which statistics are collected before being dumped automatically.

Setting *StatPeriod* to a value greater than 0 implicitly enables automatic mode operation for statistics collection. When it is set to its default value of 0, automatic dump mode is disabled, and register *StatGo* may be used to trigger a statistics dump manually.

The following table shows the statistic dump period as a function of *StatPeriod*.

StatPeriod	Statistic dump period (probe clock cycles)
0	Automatic dump disabled
1–31	$2^{\text{StatPeriod}} - 1$

The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

Control field. Bits 4:0

RELATED PARAMETERS

statisticsCollection 71

od14534v1
Revised

StatGo

Writing to the 1-bit pulse register *StatGo* generates a statistics dump. The register is active when statistics collection operates in manual mode, that is, when register *StatPeriod* is set to 0.

NOTE The written value is not stored in the register. A read always returns 0.

od10283v1
Preliminary

StatAlarmMin

Register *StatAlarmMin* specifies the lower threshold used to trigger the statistics alarm. If a statistics counter has a value less than *StatAlarmMin* when a statistics dump occurs and one or more registers *Counters_M_AlarmMode* are set to either MIN or MIN_MAX, then the corresponding bit in register *StatAlarmStatus* is set.

The number of bits is equal to twice the value of parameter *wStatisticsCounter*, in order to support an alarm threshold for a chained counter.

The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

When parameter *statisticsCounterAlarm* is set to False (Architecture: Observability: Probe: statisticsCollection), register *StatAlarmMin* is reserved.

RELATED REGISTERS

Counters_M_Src 102

od10284v1

StatAlarmMax

Register *StatAlarmMax* specifies the upper threshold used to trigger the statistics alarm.

Bit 0 in register *StatAlarmStatus* is set if a statistics counter, whose register *Counters_M_AlarmMode* is set to either MAX or MIN_MAX, has a value greater than *StatAlarmMax* when a statistics dump occurs.

Bit 1 in register *StatAlarmStatus* is set if a statistics counter, whose register *Counters_M_AlarmMode* is set to either MAX or MIN_MAX, has a value greater than *StatAlarmMax* at any time.

The number of bits is equal to twice the value of parameter *wStatisticsCounter*, in order to support an alarm threshold for a chained counter.

The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

When parameter *statisticsCounterAlarm* is set to False (Architecture: Observability: Probe: statisticsCollection), register *StatAlarmMax* is reserved.

od22457v1
Review SME

StatAlarmStatus

Register *StatAlarmStatus* is a 2-bit read-only register.

Bit 0 is set when at least one statistics counter contains a value that is out-of-bounds *at the moment that the statistic counters are dumped*, whether initiated either using register *StatGo*, or after an elapsed period specified in register *StatPeriod*. The values in the statistic counters are frozen while bit 0 of *StatAlarmStatus* is 1.

Bit 1 is set when at least one statistics counter, for which a maximum threshold has been defined, contains a value that exceeds *at any time* the threshold. The values in the statistic counters are not frozen while bit 1 of *StatAlarmStatus* is 1.

The threshold limits are defined by registers *StatAlarmMin*, *StatAlarmMax* and *Counters_M_AlarmMode*.

The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

NOTE Parameter *statisticsCounterAlarm* must be set to True (Architecture: Observability: Probe: statisticsCollection), and register field *StatCondDump* must be set to 0, otherwise *StatAlarmStatus* is reserved.

od22455v1

StatAlarmClr

Register *StatAlarmClr* is a 2-bit register.

Set bit 0 to 1 to clear bit 0 of register *StatAlarmStatus*.

Set bit 1 to 1 to clear bit 1 of register *StatAlarmStatus*.

The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

When parameter *statisticsCounterAlarm* is set to False, *StatAlarmClr* is reserved.

NOTE The written value is not stored in the register. A read always returns 0.

od22456v1

StatAlarmEn

Register *StatAlarmEn* is a 2-bit register.

Set bit 0 to 1 to enable output signals *StatAlarm* and *CtiTrigOut(1)*.

Set bit 1 to 1 to enable output signal *StatAlarmMax*.

The register is available only if parameter *statisticsCollection* is set to **True** (Architecture: Observability: Probe).

When parameter *statisticsCounterAlarm* is set to **False** (Architecture: Observability: Probe: statisticsCollection), register *StatAlarmEn* is reserved.

RELATED SIGNALS

StatAlarm.....58

od22454v1

Packet probe filter registers

The possible set of registers in a FlexNoC *packet probe* filter is shown in the following table, where *N* is the filter number, determined by parameter *nFilter*, ranging from 0 to (*nFilter* – 1).

WARNING Filter operation can only be guaranteed if changes to register values are made when the probe is not active, that is, when field *Active* is 0 in register *CfgCtl*.

Address offset	Register name	Type	Retention
0x0044 + 0x003C × N	Filters_N_RouteIdBase	Control	Yes
0x0048 + 0x003C × N	Filters_N_RouteIdMask	Control	Yes
0x004C + 0x003C × N	Filters_N_AddrBase_Low	Control	Yes
0x0050 + 0x003C × N	Filters_N_AddrBase_High	Control	Yes
0x0054 + 0x003C × N	Filters_N_WindowSize	Control	Yes
0x0058 + 0x003C × N	Filters_N_SecurityBase	Control	Yes
0x005C + 0x003C × N	Filters_N_SecurityMask	Control	Yes
0x0060 + 0x003C × N	Filters_N_Opcode	Control	Yes
0x0064 + 0x003C × N	Filters_N_Status	Control	Yes
0x0068 + 0x003C × N	Filters_N_Length	Control	Yes
0x006C + 0x003C × N	Filters_N_Urgency	Control	Yes
0x0070 + 0x003C × N	Filters_N_UserBase	Control	Yes
0x0074 + 0x003C × N	Filters_N_UserMask	Control	Yes
0x0078 + 0x003C × N	Filters_N_UserBaseHigh	Control	Yes
0x007C + 0x003C × N	Filters_N_UserMaskHigh	Control	Yes

Constraints on register availability

The following registers are available only with certain FlexArtist software configurations, or with specific licenses:

Filters_N_AddrBase_High Is available only if transport address width, set by datapath network parameter *wAddr*, is greater than 32 bits.

Filters_N_UserBase and *Filters_N_UserMask* Are available only if user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe).

Filters_N_UserBase_High and *Filters_N_UserMask_High* Are available only if more than 32 user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe).

od10288v1

RELATED PARAMETERS

nFilter	67
allowFilterOnUser	67

RELATED REFERENCES

Filtering packets.....	32
------------------------	----

TIP To find the route ID for a given probe point, select the structure in the project tree pane. In the information pane, locate the corresponding structure summary. To display the datapath network Routeld and constituent components, click datapathRoutesId.

Filters_N_RouteldBase

Register *Filters_N_RouteldBase* specifies which value of *Routeld* should be used to filter packets. This register is used in conjunction with register *Filters_N_RouteldMask*.

Routeld and its subfields *InitFlow*, *TargFlow*, and *SeqId* are described in related reference documentation.

NOTE If *Routeld* is greater than 32 bits, filtering is performed on the 32 MSBs. In this case, it is not possible to filter on *SeqId* because this sub-field is mapped to the LSBs of *Routeld*. However, because sequence IDs are only used internally between NIUs, they are not necessary for debugging or statistics gathering.

RELATED REFERENCES

Routeld	45
---------------	----

od10289v1

Filters_N_RouteldMask

Register *Filters_N_RouteldMask* specifies the mask used on *Routeld* to filter packets.

A hit on filter *n* occurs if:

PacketRouteID & *Filters_N_RouteldMask* = *Filters_N_RouteldBase* & *Filters_N_RouteldMask*

Routeld and its subfields are described in related reference documentation.

NOTE If *Routeld* is greater than 32 bits, filtering is performed on the 32 MSBs. In this case, it is not possible to filter on *SeqId* because this sub-field is mapped to the LSBs of *Routeld*. However, because sequence IDs are only used internally between NIUs, they are not necessary for debugging or statistics gathering.

RELATED REFERENCES

Routeld	45
---------------	----

od10290v1

Filters_N_AddrBase_Low

Register *Filters_N_AddrBase_Low* specifies the values of the lower transport address bits, used in conjunction with register *Filters_N_WindowSize*, to filter packets.

The overall address base is the concatenation of *Filters_N_AddrBase_Low* and *Filters_N_AddrBase_High*:

$$\text{Address Base} = (\text{Filters_N_AddrBase_High}, \text{Filters_N_AddrBase_Low})$$

The number of bits in the register is equal to the width of the transport address, set by datapath network parameter $wAddr$, up to a maximum size of 32 bits.

Filters_N_AddrBase_High

Register *Filters_N_AddrBase_High* specifies the values of the upper address bits used in conjunction with register *Filters_N_WindowSize* to filter packets.

The register is available only if transport address width, set by datapath network parameter $wAddr$, is greater than 32 bits. The number of bits in the register is then equal to $(wAddr - 32)$.

od10291v1
Preliminary

Filters_N_WindowSize

Register *Filters_N_WindowSize* specifies the encoded address mask used to filter packets. The mask value is:

$$\text{Address Mask} = \sim(2^{\text{Filters_N_WindowSize}} - 1)$$

A hit on filter n occurs if:

$$\text{PacketAddress} \& \text{AddressMask} = \text{AddressBase} \& \text{AddressMask}$$

The number of bits in this register is equal to:

$$\text{ceil}(\log_2(wAddr + 1))$$

where $wAddr$ is the datapath network parameter.

od10292v1

Filters_N_SecurityBase

Register *Filters_N_SecurityBase* specifies, for filter n , the *values* to be matched against bits in the packet header field *Security* of transport packets. Bit *Filters_N_SecurityBase*(m) is mapped to *Security*(m), where m is a value ranging from 0 to $(wSecurity - 1)$ and $wSecurity$ is the datapath network parameter.

The register is used in conjunction with register *Filters_N_SecurityMask*.

The number of bits in the register is set by $wSecurity$.

od10293v1

Filters_N_SecurityMask

Register *Filters_N_SecurityMask* specifies, for filter n , which bits of packet header field *Security* are used to filter transport packets. Bit *Security*(m) is added to the filter when bit *Filters_N_SecurityMask*(m) is set to 1, where m is a value ranging from 0 to $(wSecurity - 1)$ and $wUser$ is the datapath network parameter.

A hit on filter n occurs if:

$$\text{PacketSecurity} \& \text{Filters_N_SecurityMask} = \text{Filters_N_SecurityBase} \& \text{Filters_N_SecurityMask}$$

The number of bits in the register is set by datapath network parameter $wSecurity$.

od10294v1

od10295v1

Filters_N_Opcode

Register *Filters_N_Opcode* contains four fields that control which packet opcodes are used to select packets by Filter *N*.

Bit range	Field name
0	RdEn
1	WrEn
2	LockEn
3	UrgEn

If multiple register fields are activated, then the set of packet opcodes that are used as the filter criteria is the logical OR of the opcodes of the individual register fields.

IMPORTANT No packets are selected by the filter when *Filters_N_Opcode* has a value of 0.

Opcode descriptions for packet header field *Opc* are provided in related reference documentation.

od10296v1

RELATED REFERENCES

Opc.....46

RdEn

When register field *RdEn* is set to 1, packets with packet header field *Opc* set to RD or RDW are selected.

Not all types of read packets are filtered: packets where *Opc* is RDL and RDX are not selected.

RD packets that are part of a locked sequence initiated by a preamble packet (*Opc* is PRE) are filtered.

TYPE Control

BIT RANGE 0

od10297v1

WrEn

When register field *WrEn* is set to 1, packets with packet header field *Opc* set to WR and WRW packets are selected.

Not all types of write packets are filtered: packets whose field *Opc* is set to WRC are not selected.

WR packets that are part of a locked sequence initiated by a preamble packet (*Opc* is PRE) are filtered.

TYPE Control

BIT RANGE 1

od10298v1
Preliminary

NOTE Despite its name, field *LockEn* does not *only* operate on the state of the lock bit.

LockEn

When register field *LockEn* is set to 1, packets are selected if they belong either to exclusive accesses, that is, when packet header field *Opc* is RDL or WRC, or locked sequences, that is, initiated with a packet that has *Opc* set to RDX or PRE.

In the case of an exclusive access, a WRC packet corresponding to a preceding RDL packet is selected automatically.

In the case of a locked sequence, packets are selected up to the final packet in which the packet header bit *Lock* is de-asserted to 0. This means that in case of a RDX-WR locked sequence, the WR packet is selected as well as the RDX packet. For a locked sequence initiated by a preamble (*Opc* is PRE), the packet indicating the end of the locked sequence (*Lock* = 0) is selected as well as the preceding packets comprising the locked sequence.

TYPE Control

BIT RANGE 2

od10299v1
Preliminary

UrgEn

When register field *UrgEn* is set to 1, urgency packets, that is packets whose packet header field *Opc* is set to URG, are selected.

TYPE Control

BIT RANGE 3

od10300v1

Filters_N_Status

Register *Filters_N_Status* is a 2-bit register that sets criteria for filter *n* based on packet status type.

Bit range	Field name
0	ReqEn
1	RspEn

od10301v1
Preliminary

ReqEn

Set register field *ReqEn* to 1 to select packets whose header field *Status* is REQ.

NOTE To select packets whose *Status* is ERR, set field *ErrEn* in register *MainCtl* to 1.

od10302v1

RspEn

Set register field *RspEn* to 1 to select packets whose header field *Status* is RSP, FAIL or CONT.

NOTE To select packets whose *Status* is ERR, set field *ErrEn* in register *MainCtl* to 1.

od10303v1

Filters_N_Length

Register *Filters_N_Length* is 4-bit register that selects candidate packets if the number of bytes is less than or equal to $2^{\text{Filters_N_Length}}$.

od10304v1
Preliminary

Filters_N_Urgency

Register *Filters_N_Urgency* specifies the minimum urgency level used to filter packets. A packet is selected if its packet header field *Urgency* is greater than or equal to the value of the register.

NOTE The value in *Filters_N_Urgency* is stored as a binary, unlike field *Urgency*, which uses bar-graph encoding.

od10305v1

Filters_N_UserBase

Register *Filters_N_UserBase* specifies, for filter n , the values to be matched against the lower bits in the packet header field *User* of transport packets. Bit *Filters_N_UserBase*(m) is mapped to *User*(m), where m is a value ranging from 0 to ($wUser - 1$) and $wUser$ is the datapath network parameter.

This register is used in conjunction with register *Filters_N_UserMask*.

The overall match string is the concatenation of *Filters_N_UserBase* and *Filters_N_UserBaseHigh*:

User Base = (Filters_N_UserBaseHigh, Filters_N_UserBase)

The register is available only if user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe). Register size is determined by $wUser$.

od10306v1

Filters_N_UserBaseHigh

Register *Filters_N_UserBaseHigh* specifies, for filter n , the values to be matched against the upper bits in the packet header field *User* of transport packets. Bit *Filters_N_UserBaseHigh*($m - 32$) is mapped to *User*($m - 32$), where m is a value ranging from 32 to ($wUser - 1$) and $wUser$ is the datapath network parameter.

The register is available only if more than 32 user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe). The number of bits in the register is then equal to ($wUser - 32$).

od10307v1

Filters_N_UserMask

Register *Filters_N_UserMask* specifies, for filter n , which lower bits of packet header field *User* are used to filter transport packets. Bit *User*(m) is added to the filter when bit *Filters_N_UserMask*(m) is set to 1, where m is a value ranging from 0 to ($wUser - 1$) and $wUser$ is the datapath network parameter.

The overall user mask is the concatenation of *Filters_N_UserMask* and *Filters_N_UserMaskHigh*:

User Mask = (Filters_N_UserMaskHigh, Filters_N_UserMask)

A hit on filter n occurs if:

PacketUserFlags & UserMask = UserBase & UserMask

The register is available only if user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe). Register size is determined by datapath network parameter $wUser$.

od10308v1

Filters_N_UserMaskHigh

Register *Filters_N_UserMaskHigh* specifies, for filter *n*, which upper bits of packet header field *User* are used to filter transport packets. Bit *User*(*m* – 32) is added to the filter when bit *Filters_N_UserMaskHigh*(*m* – 32) is set to 1, where *m* is a value ranging from 32 to (*wUser* – 1) and *wUser* is the datapath network parameter.

The register is available only if more than 32 user flags exist, and when probe parameter *allowFilterOnUser* is set to **True** (Architecture: Observability: Probe). The number of bits in the register is then equal to (*wUser* – 32).

od10309v1

Packet Probe counter registers

Each FlexNoC *packet probe* statistics counter has the following registers, where *m* is the counter number ranging from zero to (*nStatisticsCounter* – 1).

WARNING Filter operation can only be guaranteed if changes to register values are made when the probe is not active, that is, when field *Active* is 0 in register *CfgCtl*.

Address offset	Name	Type	Retention
0x0134 + 0x0014 × <i>m</i>	Counters_M_PortSel	Control	Yes
0x0138 + 0x0014 × <i>m</i>	Counters_M_Src	Control	Yes
0x013C + 0x0014 × <i>m</i>	Counters_M_AlarmMode	Control	Yes
0x0140 + 0x0014 × <i>m</i>	Counters_M_Val	Status	No

NOTE These registers are only available if parameter *statisticsCollection* is set to **True** (Architecture: Observability: Probe).

od10310v1

RELATED PARAMETERS

allowFilterOnUser 67

Counters_M_PortSel

Register *Counters_M_PortSel* controls which probe port is connected to counter *m* of the probe, when the selected event type for that counter, specified with register *Counters_M_Src*, is one of the following:

IDLE	XFER	BUSY
WAIT	PKT	BYTE
PRESS0	PRESS1	PRESS2

The register can be changed at any time, with the change effective immediately.

The number of bits in register *Counters_M_PortSel* is:

$\text{ceil}(\log_2(\text{number of probe points to which the probe is connected}))$

The register is available only if parameter *statisticsCollection* is set to **True** (Architecture: Observability: Probe).

od10311v1

Counters_M_Src

Register *Counters_M_Src* specifies, for counter m , the type of event source used to increment the counter.

The register is available only if parameter *statisticsCollection* is set to **True** (Architecture: Observability: Probe).

NOTE Settings for *Counters_M_Src* that cannot be implemented, such as nonexistent pressures or filters, are equivalent to event type OFF.

od10312v1
Preliminary

RELATED REFERENCES

FlexNoC packet formats	43
statisticsCollection	71

The following table shows the event source type associated with each counter m specified by register *Counters_M_Src*.

<i>Counters_M_Src</i>	Event source type	Event description
9'h000	OFF	Counter disabled.
9'h001	CYCLE ⁸	Probe clock cycles.
9'h002	IDLE	Idle cycles during which no packet data is observed.
9'h003	XFER	Transfer cycles during which packet data is transferred.
9'h004	BUSY	Busy cycles during which the packet data is made available by the transmitting agent but the receiving agent is not ready to receive it.
9'h005	WAIT	Wait cycles during a packet in which the transmitting agent suspends the transfer of packet data.
9'h006	PKT	Packets.
9'h007	LUT ¹	Packets selected by the LUT.
9'h008	BYTE ²	Total number of payload bytes.
9'h009	PRESS0 ⁹	Clock cycles with pressure level > 0.
9'h00A	PRESS1 ⁹	Clock cycles with pressure level > 1.
9'h00B	PRESS2 ⁹	Clock cycles with pressure level > 2.
9'h00C	FILT0 ¹	Packets selected by Filter 0.
9'h00D	FILT1 ¹	Packets selected by Filter 1.
9'h00E	FILT2 ¹	Packets selected by Filter 2.
9'h00F	FILT3 ¹	Packets selected by Filter 3.
9'h010	CHAIN ³	Carry from counter $2m$ to counter $2m + 1$.
9'h011	LUT_BYTE_EN ^{1,4,6}	Enabled payload byte in packets selected by the LUT.
9'h012	LUT_BYTE ^{1,2}	Total number of payload bytes in packets selected by the LUT.
9'h013	FILT_BYTE_EN ^{1,4,5,6}	Enabled payload byte in packets selected by the associated filter.
9'h014	FILT_BYTE ^{1,2,5}	Total number of payload bytes in packets selected by the associated filter.
9'h015 to 9'h01F	Reserved	
9'h020 to 9'h11F	EXT_EVENT ^{7,8}	Clock cycles while external event signal is 1.
9'h120 to 9'h1FF	Reserved	

Event source type table notes

od17504v1
Preliminary

Table note 1

When selecting this event type, the source probe port is specified using register *TracePortSel* and not *Counters_M_PortSel*, because the filters are a trace function.

od17485v1
Preliminary

Table note 2

The counter increments by $(Len1 + 1)$ for read packets (*Opc* = RD, RDW, RDL, or RDX) and write packets (*Opc* = WR, WRW, or WRC). For interleaved reads, response fragments other than the first one (*Status* = CONT) are not counted because *Len1* of the initial fragment specifies the total length of the response.

More information on packet header fields *Len1*, *Opc*, and *Status* is available in related reference documentation.

od17486v1
Preliminary

RELATED REFERENCES

Packet header fields45

Table note 3

Only odd-numbered counters can be set to CHAIN. Only two counters can be chained together.

od17487v1
Preliminary

Table note 4

To select this event type, set parameter *allowFilterOnEnabledBytes* to True and ensure that the value of serialization parameter *nBytePerWord* is greater than zero. For multi-port probes, *allowFilterOnEnabledBytes* can only be set to True if the serialization is the same across all ports.

od17488v1
Preliminary

RELATED PARAMETERS

allowFilterOnEnabledBytes.....67

Table note 5

The mapping between filters and counters is hardwired. If there are *n* filters, the output of filter *i* is connected to counter *j* according to the expression:

$$i = j \% n \text{ for } j \geq n$$

The following table shows an example of filter-to-counter mapping for a packet probe with three filters and seven statistics counters.

Counter <i>j</i>	Filter <i>i</i>
------------------	-----------------

Counter j	Filter i
0	0
1	1
2	2
3	0
4	1
5	2
6	0

Filter outputs that are mapped to odd counters are restricted: such counters cannot be chained. To overcome this, an alternative mapping with filter outputs connected to even counters only can be selected at run time by programming register *MainCtl* field *FiltByteAlwaysChainableEn* to 1.

The following table shows the alternative mapping for this configuration.

Counter j	Filter i
0	0
1	not applicable
2	1
3	not applicable
4	2
5	not applicable
6	0

With this mapping, count events of this type, for all filter outputs, can be counted using a chained counter, the source of the associated odd counter being set to CHAIN.

NOTE When *FiltByteAlwaysChainableEn* is set to 1, setting the source of odd counters to this type has no effect.

Table note 6

The counter increments each time a payload byte has its bit **BE**, or *byte enable*, set to 1 in a packet selected by the associated filter. More information on bit **BE** is available in related reference documentation.

FILT_BYTE_EN and LUT_BYTE_EN are of use only when filtering write packets probed on the request path, or filtering read packets probed on the response path, because the BE bits are only valid in these cases.

In the case of responses due to read interleaving, it is not necessary to probe on the response path with FILT_BYTE_EN or LUT_BYTE_EN to count the number of enabled read bytes. The total number of read bytes for the packet can be obtained by using FILT_BYTE or LUT_BYTE when probing on the request path.

od17489v1
Preliminary

od17490v1
Preliminary

RELATED REFERENCES

Be 50

Table note 7

A value in the range of EXT_EVENT will count probe clock cycles while an "external event" is asserted high. Such events can be driven from signals external to the NoC or from transaction histogram bin outputs, or both, when parameter *transactionProfiling* is set to True.

Event type EXT_EVENT is only available if either parameter *nExternalEvent* is set to a value greater than 0 or parameter *transactionProfiling* is set to True.

The following table shows how the selection of external events using *Counters_M_Src*, depends on parameters *nExternalEvent*, *transactionProfiling*, *nObservable*, and *nComparators*.

<i>transactionProfiling</i>	<i>Counters_M_Src</i>		<i>External event</i>
	<i>Lower limit</i>	<i>Upper limit</i>	
False	0x20	0x20 + (nExternalEvent - 1)	Signal external to NoC
	0x20	0x20 + (nExternalEvent - 1)	Signal external to NoC
True	0x20 + nExternalEvent	0x20 + nExternalEvent + (nObservable x (nComparators + 1) - 1)	Transaction profiling histogram output

where an event external to the NoC is driven by signal *ExtEvent* (*Counters_M_Src* - 0x20), and the port and bin number of the mapped transaction profiling histogram output is defined by:

```
Port = floor((((Counters_M_Src - (0x20 + nExternalEvent)) / (nComparators + 1))
Bin = ((Counters_M_Src - (0x20 + nExternalEvent)) % (nComparators + 1))
```

The example developed in following table shows the mapping of statistic counter source *m* to external events for a packet probe, which is instantiated in a transaction probe, whose parameters *nExternalEvent*, *nObservable*, and *nComparators* have values 3, 2, and 3 respectively.

<i>Counters_M_Src</i>	<i>External event source</i>
9'h020	External event signal ExtEvent.0
9'h021	External event signal ExtEvent.1
9'h022	External event signal ExtEvent.2
9'h023	Transaction profiling observed Port 0 / Bin 0
9'h024	Transaction profiling observed Port 0 / Bin 1
9'h025	Transaction profiling observed Port 0 / Bin 2
9'h026	Transaction profiling observed Port 0 / Bin 3
9'h027	Transaction profiling observed Port 1 / Bin 0
9'h028	Transaction profiling observed Port 1 / Bin 1
9'h029	Transaction profiling observed Port 1 / Bin 2
9'h02A	Transaction profiling observed Port 1 / Bin 3

Counters_M_Src	External event source
9'h02B – 9'h11F	OFF

RELATED PARAMETERS

nExternalEvent	72
nObservable	74
nComparators.....	76

RELATED SIGNALS

ExtEvent	58
----------------	----

od17491v1
Preliminary

Table note 8

When *transactionProfiling* is set to **True**, these are the only event types available.

Table note 9

Pressure level is encoded by the sideband request signal *Press*. This signal is implemented when parameter *usePress* is set to **True**; the number of bits used to encode the pressure level is set by parameter *nUrgencyLevel*.

More information is available in related reference documentation ([../PDF/dmpFlexNoC_QoS.pdf](#)).

The example developed in following table illustrates which packets, over a range of pressure levels, are counted depending on the value of *PRESSn*. In this example, *nUrgencyLevel* set to 4.

Request Press				Pressure level	Counters_M_Src		
Bit 3	Bit 2	Bit 1	Bit 0		PRESS0	PRESS1	PRESS2
0	0	0	0	0			
0	0	0	1	1			
0	0	1	1	2	Clock cycles counted		
0	1	1	1	3		Clock cycles counted	
1	1	1	1	4			Clock cycles counted

od17492v1
Preliminary**Counters_M_AlarmMode**

Register *Counters_M_AlarmMode* is a 2-bit register that is present when parameter *statisticsCounterAlarm* is set to **True**. The register is available only if parameter *statisticsCollection* is set to **True** (Architecture: Observability: Probe).

The register defines statistics alarm counter behavior as follows:

- 0 – OFF** The comparison is disabled.
- 1 – MIN** If the value of the counter is less than the *StatAlarmMin* register at the dump period, the *StatAlarmStatus* bit is set.
- 2 – MAX** If the value of the counter is greater than the *StatAlarmMax* register at the dump period, the *StatAlarmStatus* bit is set.

od17493v1
Preliminary

3 – MIN_MAX If the value of the counter is less than the *StatAlarmMin* register or greater than the *StatAlarmMax* register at the dump period, the corresponding *StatAlarmStatus* bit is set.

If the alarm is programmed for a chained counter, then:

- § Even counters, which store the lower bits of the chained counter, generate alarms. The alarm of odd counters is masked.
- § Only the alarm mode of the even counters is used. The alarm mode of the odd counter is ignored.
- § The comparison with *StatAlarmMin* or *StatAlarmMax*, or both, is made against the composite value contained in the two counters.

od10313v1
Preliminary

Counters_M_Val

Registers *Counters_M_Val* contain the statistics counter values. The values in the registers may be dumped into statistics packets by the ATB endpoint of an observer, either periodically or under manual control. Alternatively, the values may be read directly via software.

Register width, in bits, is determined at design-time by the parameter *wStatisticsCounter*. The register is available only if parameter *statisticsCollection* is set to True (Architecture: Observability: Probe).

od10314v1

Transaction filter registers

The FlexNoC *transaction filter* unit has the following global configuration registers.

Address offset	Name	Type	Retention
0x0008	Mode	Control	Yes
0x000C	AddrBase_Low	Control	Yes
0x0010	AddrBase_High	Control	Yes
0x0014	AddrWindowSize	Control	Yes
0x0018	SrcIdBase	Control	Yes
0x001C	SrcIdMask	Control	Yes
0x0020	Opcode	Control	Yes
0x0024	UserBase	Control	Yes
0x0028	UserMask	Control	Yes
0x002C	SecurityBase	Control	Yes
0x0030	SecurityMask	Control	Yes
0x0034	UserBaseHigh	Control	Yes
0x0038	UserMaskHigh	Control	Yes

Constraints on register availability

The following registers are available only with certain FlexArtist software configurations, or with specific licenses:

AddrBase_High Is available only if the generic address width of the associated NIU is greater than 32 bits.

UserBaseHigh and *UserMaskHigh* Are available only if the number of user flags, set by datapath network parameter *wUser*, is greater than 32.

SrcIdBase and *SrcIdMask* Are available only with specific licenses.

od10315v1
Preliminary

Mode

Register *Mode* is a 1-bit register that specifies the transaction filter delay counting mode of the probed NIU. The following table describes the delays and events which start and stop the clock counter.

Value	Mode	Events
0	HANDSHAKE	Start event = begin request Stop event = request accepted
1	LATENCY	Start event = begin request Stop event = begin response (for request and response with matching context ID)

od7746v1

RELATED REFERENCES

Measuring latency 40

RELATED REGISTERS

Mode 112

SrcIdBase

Register *SrcIdBase* is available only with specific licenses. The register has null width.

od10316v1

SrcIdMask

Register *SrcIdMask* is available only with specific licenses.

The register has null width.

od10317v1

AddrBase_Low

Register *AddrBase_Low* specifies the values of the lower address bits used, in conjunction with register *AddrWindowSize*, to filter packets.

The overall address base is the concatenation of *AddrBase_Low* and *AddrBase_High*:

Addr Base = (AddrBase_High, AddrBase_Low)

The number of bits in the register is equal to the generic address width of the associated NIU, determined by parameter *wAddr* (Specification: Interface: NIU: generic) up to a maximum size of 32 bits.

od10318v1

AddrBase_High

Register *AddrBase_High* specifies the values of the upper address bits used, in conjunction with the address mask (see register *AddrWindowSize*) filter packets.

The register is available only if the generic address width of the associated NIU is greater than 32 bits. The number of bits in the register is equal to $(wAddr - 32)$, where $wAddr$ is the generic address width (Specification: Interface: NIU: generic).

AddrWindowSize

od10319v1
Preliminary

Register *AddrWindowSize* specifies the encoded address mask used to filter packets. The effective mask value is equal to:

$$\text{Address Mask} = \sim(2^{\text{AddrWindowSize}} - 1)$$

A packet is selected if:

$$\text{Packet Address \& Address Mask} = \text{Address Base \& Address Mask}$$

The number of bits in this register is equal to $\text{ceil}[\log_2(wAddr + 1)]$, where $wAddr$ is the generic address width (Specification: Interface: NIU: generic).

od10320v1

Opcode

Register *Opcode* allows filter criteria to be based on the type of generic transaction.

Generic transaction types are described in the "Generic Interface Protocol Features" section of related concept documentation ([../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf](#)).

Bit range	Field name
0	RdEn
1	WrEn

od10321v1
Preliminary

RdEn

Set register field *RdEn* to 1 to select the following generic transaction types: RD (Read), RDX (Exclusive Read) and RDL (Read Linked).

Setting the field to 1 enables the transaction filter to send *start* and *stop* events.

WARNING Because a *stop* event must correspond to *each* start event, tenure counters must be enabled prior to setting the field to 1. Similarly, the field must be set to 0 prior to disabling the counters.

TYPE Control

BITS (0)

od10322v1

WrEn

Set register field *WrEn* to 1 to select the following generic transaction types: WR (Write), WRC (Write Conditional).

Setting the field to 1 enables the transaction filter to send *start* and *stop* events.

WARNING Because a *stop* event must correspond to *each* start event, tenure counters must be enabled prior to setting the field to 1. Similarly, the field must be set to 0 prior to disabling the counters.

TYPE Control

BITS (1)

od10323v1

UserBase

Register *UserBase* specifies the *values* to be matched against the lower bits in the packet header field *User* of transport packets. Bit *UserBase*(*n*) is mapped to *User*(*n*), where *n* is a value ranging from 0 to (*wUser* – 1) and *wUser* is the datapath network parameter.

This register is used in conjunction with register *UserMask*.

The overall match string is the concatenation of *UserBase* and *UserBaseHigh*:

User Base = (UserBaseHigh, UserBase)

Register size is determined by *wUser*.

od10324v1

UserMask

Register *UserMask* specifies *which* lower bits of packet header field *User* are used to filter transport packets. Bit *User*(*n*) is added to the filter when bit *UserMask*(*n*) is set to 1, where *n* is a value ranging from 0 to (*wUser* – 1) and *wUser* is the datapath network parameter.

The overall user mask is the concatenation of *UserMask* and *UserMaskHigh*:

User Mask = (UserMaskHigh, UserMask)

A packet is selected if it satisfies:

PacketUserBits & UserMask = UserBase & UserMask

Register size is determined by *wUser*.

od10325v1

SecurityBase

Register *SecurityBase* specifies the *values* to be matched against the packet header field *Security* of transport packets. Bit *SecurityBase*(*n*) is mapped to *Security*(*n*), where *n* is a value ranging from 0 to (*wSecurity* – 1) and *wSecurity* is the datapath network parameter.

This register is used in conjunction with register *SecurityMask*.

Register size is determined by *wSecurity*.

od10326v1

SecurityMask

Register *SecurityMask* specifies *which* bits of packet header field *Security* are used to filter transport packets. Bit *Security*(*n*) is added to the filter when bit *SecurityMask*(*n*) is set to 1, where *n* is a value ranging from 0 to (*wSecurity* – 1) and *wSecurity* is the datapath network parameter.

A packet is selected if it satisfies:

PacketSecurity & SecurityMask = SecurityBase & SecurityMask

Register size is determined by *wSecurity*.

od10327v1

UserBaseHigh

Register *UserBaseHigh* specifies the *values* to be matched against the upper bits in the packet header field *User* of transport packets. Bit *UserBaseHigh*(*n* – 32) is mapped to *User*(*n* – 32), where *n* is a value ranging from 32 to (*wUser* – 1) and *wUser* is the datapath network parameter.

The register is available only if the number of user flags, set by datapath network parameter *wUser*, is greater than 32. The number of register bits is then equal to (*wUser* – 32).

od10328v1

UserMaskHigh

Register *UserMaskHigh* specifies *which* upper bits of packet header field *User* are used to filter transport packets. Bit *User*(*n* – 32) is added to the filter when bit *UserMaskHigh*(*n* – 32) is set to 1, where *n* is a value ranging from 32 to (*wUser* – 1) and *wUser* is the datapath network parameter.

The register is available only if the number of user flags, set by datapath network parameter *wUser*, is greater than 32. The number of bits is then equal to (*wUser* – 32).

od10329v1

Transaction Profiler registers

The complete set of FlexNoC *transaction profiler* unit registers is shown in the following table.

Address offset	Name	Type	Retention
0x0008	En	Control	Yes
0x000C	Mode	Control	Yes
0x0010	ObservedSel_0	Control	Yes
0x0014	ObservedSel_1	Control	Yes
0x0018	ObservedSel_2	Control	Yes
0x001C	ObservedSel_3	Control	Yes
0x0020	nTenureLines_0	Control	Yes
0x0024	nTenureLines_1	Control	Yes
0x0028	nTenureLines_2	Control	Yes
0x002C	Thresholds_0_0	Control	Yes
0x0030	Thresholds_0_1	Control	Yes
0x0034	Thresholds_0_2	Control	Yes
0x0038	Thresholds_0_3	Control	Yes
0x003C	Thresholds_1_0	Control	Yes
0x0040	Thresholds_1_1	Control	Yes
0x0044	Thresholds_1_2	Control	Yes
0x0048	Thresholds_1_3	Control	Yes

Address offset	Name	Type	Retention
0x004C	Thresholds_2_0	Control	Yes
0x0050	Thresholds_2_1	Control	Yes
0x0054	Thresholds_2_2	Control	Yes
0x0058	Thresholds_2_3	Control	Yes
0x005C	Thresholds_3_0	Control	Yes
0x0060	Thresholds_3_1	Control	Yes
0x0064	Thresholds_3_2	Control	Yes
0x0068	Thresholds_3_3	Control	Yes
0x006C	OverflowStatus	Status	No
0x0070	OverflowReset	Control	Yes
0x0074	PendingEventMode	Control	Yes
0x0078	PreScaler	Control	Yes

od10330v1

En

Register *En* comprises a 1-bit register field that must be set to 1 to enable the transaction profiling counters. The counters are reset when *En* is set to 1.

The counters must be disabled, by clearing this register to 0, before programming other registers in the transaction profiler unit.

od8131v1
Revised

Mode

Register *Mode* specifies the event mode for each observed port of the referenced transaction profiler. The number of bits in the register is equal to parameter *nObservable* of the referenced transaction probe; event types can be selected independently per port.

The following table describes event modes.

Value	Mode	Behavior
0	DELAY ¹	Counts the number of cycles between start and stop events. Counter is reset on stop event. All transaction profiling counters allocated to the observed port may be used.
1	PENDING ²	Counts the number of outstanding transactions. Start event increments counter; stop event decrements counter. Only one transaction profiling counter from those allocated to the observed port is used.

Table notes

- 1 The delay counting mode, defining the start and stop events, of the observed NIU can be set to either **HANDSHAKE** or **LATENCY** via its Transaction Filter register *Mode*.
- 2 The delay counting mode, defining the start and stop events, of the observed NIU must be set to **LATENCY** via its Transaction Filter register *Mode*.

NOTE The mapping of probed NIU to port number of the transaction probe is displayed in the FlexNoC FlexArtist Information Pane when the probe is selected in the design editor *Architecture* view (Architecture: Observability: Probe).

od10332v1

RELATED REGISTERS

Mode 108

ObservedSel_i

Set register *ObservedSel_i* to specify the transaction probe point to be observed by port *i* of the transaction probe, where *i* is in the range 0 to (*nObservable* – 1). The number associated with each transaction probe point is displayed in the FlexArtist FlexNoC Information pane when the probe is selected in the design editor *Architecture* view (Architecture: Observability: Probe).

The width of *ObservedSel_i* is:

$\text{ceil}(\log_2(\text{number of transaction probe points}))$

od10333v1

nTenureLines_i

Each register of the register set *nTenureLines_i* specifies the number of groups of transaction probe counters, also known as *tenure lines*, to be allocated to a given observed transaction port.

The register index *i* is in the range 0 to (*nObservable* – 2). The width of each register *nTenureLines_i* is:

$\text{ceil} \log_2(n\text{Counters}/4)$

NOTE Since the maximum number of tenure lines is 8, the number of tenure lines to be allocated to transaction port *i*, where $i = (n\text{Observable} - 1)$ is implicitly equal to the number of tenure lines that remain unallocated. Hence, there is no register for the this observed transaction port.

od10334v1
Preliminary

Thresholds_i_j

Registers *Thresholds_i_j*, where *i* and *j* are indices, are used to specify the comparator thresholds.

Index *i*, which identifies the transaction port that is being observed, is in the range:

$0 \leq i \leq n\text{Observable} - 1$

Index *j*, which identifies the comparator associated with the threshold, is in the range:

$0 \leq j \leq n\text{Comparators} - 1$

The number of registers is equal to *nObservable* × *nComparators* unless the number of elements in both *delayThresholds* and *pendingThresholds* is set to 1, in which case no registers are created.

If the number of elements in both *delayThresholds* and *pendingThresholds* is non-zero, each register should be set to an index number of the relevant parameter array being used for the measurement. In this case, the register width is determined by:

$\text{ceil} [\log_2 (\max(\text{number of elements in } \textit{delayThresholds}, \text{number of elements in } \textit{pendingThresholds}))]$

If the number of elements in both parameter arrays *delayThresholds* and *pendingThresholds* is zero (Architecture: Observability: Probe: transactionProfiling), threshold values are directly programmed into the registers *Thresholds_Lj*. In this case, the register width is equal to *wCounters* (Architecture: Observability: Probe: transactionProfiling).

RELATED PARAMETERS

<i>delayThresholds</i>	75
<i>pendingThresholds</i>	75
<i>nObservable</i>	74
<i>nComparators</i>	76

od17418v1
Revised

OverflowStatus

Bit *n* of register *OverflowStatus* is set to 1 if a start event occurs on observed port *n* and either of the following conditions occurs:

- § All tenure lines allocated to the observed port are already in use.
- § No tenure lines have been allocated to the observed port.

This register can be read at the end of the measurement period to determine whether any transactions were missed because transaction profiling counters were not available. The register, which can only be read by software, cannot be used to drive an interrupt.

Bits set in *OverflowStatus* can be cleared to 0 by setting the corresponding bits in register *OverflowReset* to 1. The number of bits in *OverflowStatus* is equal to the setting for parameter *nObservable* (Architecture: Observability: Probe: transactionProfiling).

RELATED PARAMETERS

<i>nObservable</i>	74
--------------------------	----

od7758v1

OverflowReset

Register *OverflowReset* is a pulse register which clears set bits in register *OverflowStatus*. Writing value 1 to bit *n* clears bit *n* in register *OverflowStatus* to 0. [od10337v1 Preliminary]

PendingEventMode

Register *PendingEventMode* is a 1-bit register that determines when the transaction profiler generates events corresponding to pending transactions.

The register only affects event generation when the profiler is monitoring pending transactions, that is, when register *Mode* is set to PENDING.

Event behavior is described in the following table.

Value	Mode	Behavior
0	CYCLE	Pending transactions generate an event on each clock cycle if transaction profiling counter is greater than zero.
1	STOP	Pending transactions generate event only at <i>stop</i> event. The stop event is defined in the transaction filter register <i>Mode</i> .

RELATED REGISTERS

Mode 108

od10338v1
Revised

Prescaler

Register *Prescaler* specifies a pre-scaler value that divides the measured latency values on the transaction probe point before incrementing the bin counters. A register value of n sets the prescaler ratio to $n + 1$.

The 8-bit register can accept prescaler values in the range 1–255. A value of 0 (default) disables the prescaler.

IMPORTANT The prescaler function is only active when the transaction profiler is in delay mode. It is not used when counting pending transactions.

CAUTION To avoid losing events, before changing *Prescaler*, it is recommended to disable the transaction probe by setting register *En* to 0.

od13971v1
Preliminary

Universal Probe registers

With minor exceptions, the register set of a given universal probe is the aggregation of the registers of its constituent *packet probe* unit and optionally, *transaction filters* and *transaction profiler*, depending on the probe configuration.

More information about these constituent units is available in related reference documentation.

RELATED REFERENCES

Transaction Profiler registers 111

od14819v1

Observers

This section provides detailed technical reference information for the FlexNoC Observability technology *Observer* unit.

od10340v1
Preliminary

Error logging features and codes

Errors that occur in packets observed by probes can be logged.

Main features

- § Error filtering can be based on selected error codes, user and security flags and sideband signals.
- § An optional *fault* interrupt can be asserted on an error.
- § Can be used in conjunction with trace probe filtering to monitor a specific port.

od10341v1

FlexNoC error codes

The following table lists all error codes that are relevant to FlexNoC interconnect operation.

Code	Value	Source	Type
SLV	0	Target	Target error detected by slave.
DEC	1	Initiator NIU	Address decode error.
UNS	2	Target NIU	Unsupported request.
DISC	3	Power Disconnect	Disconnected target or domain.
SEC	4	Initiator NIU or Firewall	Security violation.
HIDE	5	Firewall	Hidden security violation, reported as OK to initiator.
TMO	6	Target NIU	Time-out.
RSV	7	None	Reserved.

od8413v1
Preliminary

ATB formatting

FlexNoC *ATB formatting* technology supplies trace data and statistics, collected by the observation network, to an ATB debug port.

Main features

The main features are:

- § AMBA™ 3 ATB compliant.
- § Supports ATB data widths from 8 to 128 bits.
- § Fixed 64-bit cells carrying trace headers and payload, statistics, and synchronization information.
- § Supports any transport packet format.
- § Time-stamp and overflow extraction for STPv2 support.

Reference documents

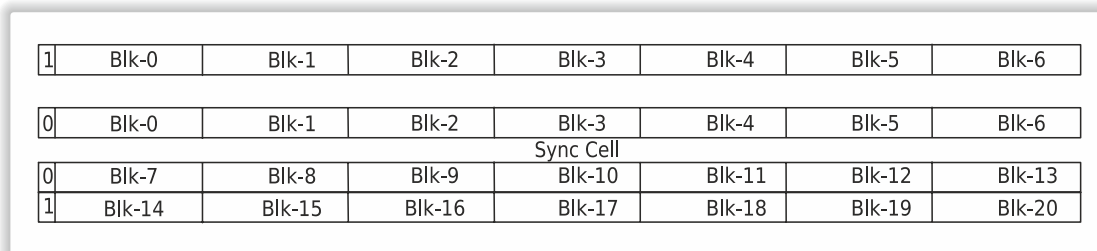
AMBA™3 ATB Protocol Specification v1.0 ARM IHI0032A2006
<http://www.arm.com>,
[http://www.arm.com/community/display_company.php?CompanyId=5299& display=10](http://www.arm.com/community/display_company.php?CompanyId=5299&display=10))

od10342v1
Preliminary

ATB packet formats

FlexNoC *ATB formatting* technology converts packets received from the observation network into a stream of 64-bit cells, with cells being grouped into packets. Each cell contains seven *Blocks* of 9 bits. The MSB of a cell is the *Last* bit, which is set to 1 if the cell is the last one of the current packet.

The following figure shows a one-cell packet and a three-cell packet with a synchronization cell inserted (see "Synchronization cell insertion" on page 120).



ob6108v1

Packet formats for ATB transmission

The first blocks of a packet contain:

- § A 2-bit *Packet Type* field (0 for Trace/Error, 1 for Statistic, 2 and 3 Reserved).
- § A *Probed* field (0 to 7 bits) that identifies the probe that generated the packet.
- § A *Header*, which depends on packet type.
- § Padding bits of value "0", following *Header*, may be present to preserve block alignment.

In a trace packet, *Header* is the probed FlexNoC packet header. Succeeding blocks contain the payload bytes of the trace packet. The MSB of these 'Payload' blocks is set to 1 to indicate that the block contains a valid value.

In an error packet, *Header* is the logged FlexNoC packet header. There is no payload.

In a statistic packet, *Header* comprises a 1-bit field *Width*, which indicates the number of blocks used to contain the statistic counters (0 for 1 block, 1 for 2 blocks). Succeeding blocks contain the statistic counter values. The MSB of each "counter" block is set to 1 to indicate that the block contains a valid value.

The types of blocks which may follow *Header* are summarized in the following table.

Block name	Value	Description
Pad	9'h000	Used to complete the last cell of a packet.
Null	9'h001	Used when bit Be (byte enable) of a payload byte is 0.
Error	9'h002	Used when a read data value belongs to a FlexNoC packet payload which has its bit WE (word error) set.
Reserved	9'h003 – 9'h07F	Reserved for future use.
Illegal	9'h080 – 9'h0FF	Guarantees uniqueness of synchronization cell.
Payload	9'h100 – 9'h1FF	Data byte. Byte value = Value – 9'h100

More information about packet payload fields *Be* and *WordErr* is available in related documentation (../PDF/dmpFlexNoC_Observability.pdf).

ATB packet examples

The following figures show examples of various packet-type structures.

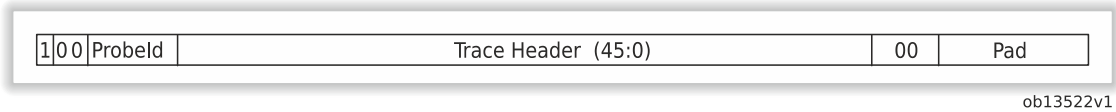


Figure 1. One-cell trace packet with a 4-bit Probeld, a 46-bit FlexNoC packet header and no payload.

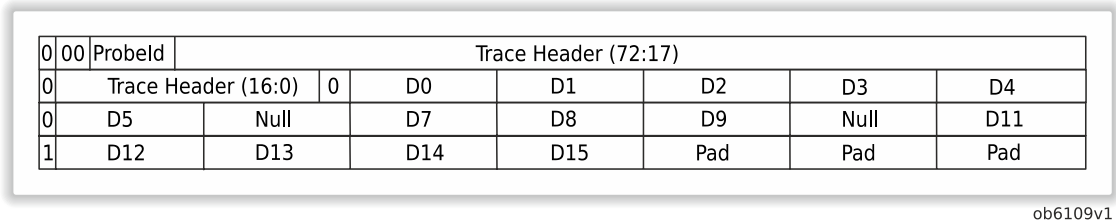


Figure 2. Four-cell trace packet with a 5-bit Probeld, a 73-bit FlexNoC packet header and a 16-byte payload with bytes 6 and 10 set to Null.

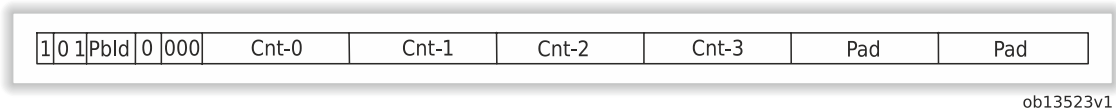


Figure 3. One-cell statistic packet with a 3-bit Probeld, and four 8-bit counters.

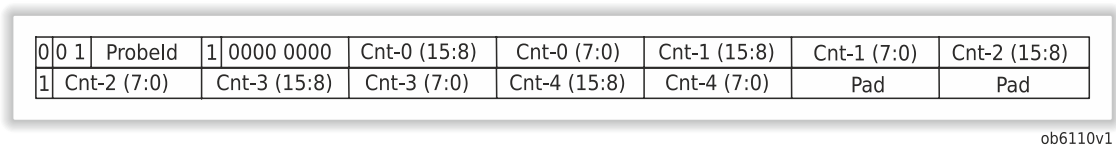


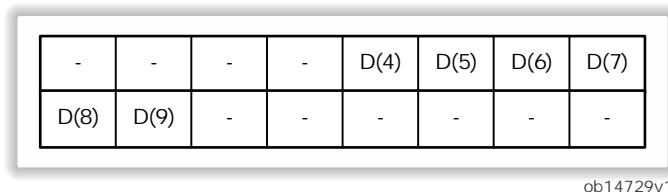
Figure 4. Two-cell statistic packet with a 7-bit Probeld, and five 16-bit counters.

Trace packet payload byte ordering

The order of payload bytes in a trace packet is the same as that in the FlexNoC packet. The first payload byte in the trace packet is that corresponding to the address given in FlexNoC packet header field *Addr*. Following payload bytes are those at successive addresses. In the case where the burst is wrapping, the address increments at $\text{modulo}(\text{Len1} + 1)$, where FlexNoC packet header field *Len1* gives the number of payload bytes in the transaction.

This is illustrated in the following examples.

A 6-byte incrementing read is performed, starting at address 0x4, and based on an *nBytePerWord* setting of 8 at the packet probe point. This is represented in the transport as shown in the following figure.



The resulting trace packet is shown in the following figure. The cell is completed with two **Pad** blocks.

0	00	Probeld	Trace Header				000	D(4)
1		D(5)	D(6)	D(7)	D(8)	D(9)	Pad	Pad

ob14748v1

Figure 5. A 6-byte (*INCR*) read starting at address 0x4.

The second case is a 10-byte incrementing write, from address 0x2. Bytes at addresses 0x4, 0x8 and 0xB do not contain valid data, so their byte enables (*Be*) are set to 0. The transport representation, based on a *nBytePerWord* of 8 at the probed link, is shown in the following figure.

-	-	D(2)	D(3)	-	D(5)	D(6)	D(7)
-	D(9)	D(A)	-	-	-	-	-

ob14731v1

The corresponding trace packet is shown in the following figure. **Null** blocks replace the bytes for which *Be* is 0. **Pad** blocks terminate the second cell.

0	00	Probeld	Trace Header				000	D(2)
0		D(3)	Null	D(5)	D(6)	D(7)	Null	D(9)
1		D(A)	Null	Pad	Pad	Pad	Pad	Pad

ob14749v1

Figure 6. An 10-byte (*INCR*) write starting at address 0x2, with some bytes not valid (*Be*=0).

In the final example, a 16-byte wrapping read starting at address 0xC is performed, with the response being interrupted after 12 bytes. The transport representation, based on a *nBytePerWord* of 8 at the probed link, is shown in the following figure.

-	-	-	-	D(C)	D(D)	D(E)	D(F)
D(0)	D(1)	D(2)	D(3)	D(4)	D(5)	D(6)	D(7)

ob14707v1

This leads to the trace packet shown in the following figure. Despite the interruption of the read, there are no bytes which are marked with invalid data since the interruption occurs on the *nBytePerWord* boundary. Hence, there are no **Null** blocks in the trace packet. However, it does contain **Pad** blocks in order to complete the last cell.

0	00	Probeld	Trace Header				000	D(C)
0		D(D)	D(E)	D(F)	D(0)	D(1)	D(2)	D(3)
1		D(4)	D(5)	D(6)	D(7)	Pad	Pad	Pad

ob13521v1

Figure 7. A 16-byte (*WRAP*) read starting at address 0xC but interrupted after the 12th byte

Synchronization cell insertion

In order to allow the ATB socket to align itself on the 64-bit cell boundary, synchronization cells must be sent. A cell is sent when the ATB endpoint is enabled, via register *AtbEn*, and periodically thereafter, determined by register *SyncPeriod*.

The value of the cell is 0x2D96EA874B15F0C3. This value is unique and cannot occur in other cells once synchronization has been established.

A synchronization cell may occur within a packet.

In the case where the disconnect protocol of the ATB endpoint is set to *ARTERIS* and the ATB slave retracts a power disconnect request by asserting signal *SlvRdy* from 0 back to 1 before the master drives signal *SocketConn* to 0, an additional sync cell may possibly be sent. More information about power disconnect signals is available in related reference documentation ([./PDF/dmpFlexNoC_QoS.pdf](#)).

If the ATB parameter *wData* is set to 128, the synchronization cell may occur in either the first cell of the ATB word (bits 0 to 63) or the second cell (bits 64 to 127). Whichever cell is not used for the synchronization cell can contain a non-synchronization ATB cell.

od6100v2
Revised

RELATED REFERENCES

Power disconnect signals 128

ATB serialization

The number of ATB words required to transfer a cell depends on the setting for ATB parameter *wData* (Specification: Interface: Observation: debugOutput).

If the setting is less than or equal to 64, the 64-bit cells are split into $64/wData$ ATB words. If *wData* is set to 128, then two cells can be transferred per ATB word. The first cell occupies bits 0 to 63, and the second cell occupies bits 64 to 127.

Because all bytes of the ATB word carry valid cell data, signal *ATBytes* is set to "-1":

```
ATBytes[m-1:0] = m'b1
```

Serialization is performed LSB first for coherency with 128-bit mode.

od10345v1

ATB flush control

Because all FlexNoC packet transmissions that begin must unconditionally finish, the FlexNoC implementation of the ATB flush control function complies with Chapter 4.2.1 “Behavior of trace sources that do not store trace locally” of the ARM AMBA™ 4 ATB Protocol Specification ATBv1.0 and ATBv1.1, 28 March 2012, Issue B

(<http://infocenter.arm.com/help/topic/com.arm.doc.ih0032b/index.html>)

The flush control algorithm is summarized in the following table:

<i>AfValid</i>	<i>AtValid</i>	<i>AtReady</i>	<i>AfReady next cycle</i>
0	don't care	don't care	0

<i>AfValid</i>	<i>AtValid</i>	<i>AtReady</i>	<i>AfReady next cycle</i>
1	0	don't care	1
1	1	0	0
1	1	1	1

od6103v1
Preliminary

STPv2 stream generation

The FlexNoC STPv2 adapter in the observer encapsulates ATB packets and generates synchronization sequences to provide a MIPI-STPv2 compliant STP stream. Time stamps can optionally be included in the stream.

Main features

- § Compliant with AMBA-3 ATB and MIPI STPv2 specifications.
- § Supports ATB data widths from 8 to 64 bits.
- § Supports STPv2 opcodes ASYNC, VERSION, NULL, NULL_TS, D64, D64MTS, and MERR.
- § Synchronization sequence generation with optional time stamp.
- § Encapsulation of probe time stamps.
- § Supports all time stamp length encodings.

Reference documents

MIPI® Alliance Specification for System Trace Protocol (STP), version 2.00.01, 24 February 2010.-STPv2 specification.

Synchronization sequence generation

od10346v1





The synchronization sequence permits the SoC debug system to identify the STPv2 packet boundary. A time stamp can be optionally inserted in the sequence to which the SoC debug system may resynchronize itself.



The synchronization sequence is sent when one of the following events occurs:

- § After a reset.
- § On the rising edge of input signal *SyncReq*.
- § When the number of clock cycles since the last synchronization sequence is between $2^{AsyncPeriod}$ and 2^{23} inclusive, and register *AsyncPeriod* is greater than 0.
- § When the STPv2 adapter power disconnect interface, if present, is reconnected.

NOTE The synchronization sequences are inserted instead of ATB synchronization cells.

The STPv2 adapter can issue synchronization sequences either with or without time stamps. The examples in figures that follow are represented according to the following key.

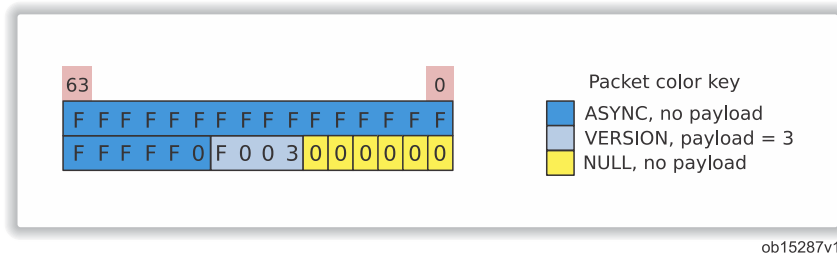
STPv2 packets		
	Type	Payload
	ASync	None
	VERSION	3
	NULL	None
	NULL_TS	Timestamp

Timestamps	
	Timestamp size
	Timestamp (STPv2NAT)

ob15289v1

STPv2 synchronization sequence without time stamp

Synchronization sequences are sent without a time stamp when the observer STPv2 parameter *wTimeStamp* is set to 0. In this case, the sequence comprises the STPv2 packets *ASync* and *VERSION*, as well as *NULL* packets for padding. Such a sequence is shown in the following figure.

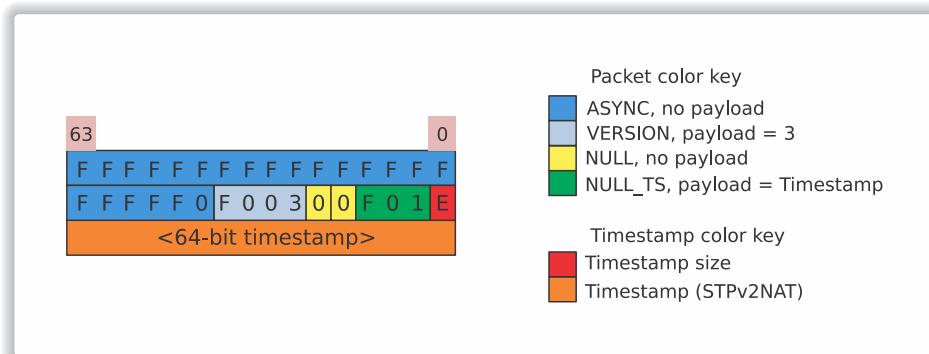


NOTE The sequence in the figure assumes a data width of 64 for the ATB debug port, as set by parameter *wData* (Specification: Interface: Observation: debugOutput).

STPv2 synchronization sequence with time stamp

If *wTimeStamp* is greater than zero, synchronization sequences are sent with time stamps. The sequence comprises the STPv2 packets *ASync*, *VERSION*, *NULL*, and *NULL_TS*. The time stamp value, received on observer input signal *TimeStamp* and extended to 64 bits, is sent in the payload of *NULL_TS*.

The synchronization sequence with time stamp is shown in the following figure.



ob15288v1

In the figure:

- § Regardless of the width of the time stamp, specified by *wTimeStamp*, the time stamp is extended to 64 bits.
- § All 16 nibbles of the time stamp are significant, indicated by the time stamp size nibble being set to 0xE.
- § The protocol version is set to 3 in the payload of packet *VERSION*. This indicates that the time stamp is encoded using the uncompressed STPv2NAT format.

RELATED SIGNALS

SyncReq 127

RELATED REGISTERS

AsyncPeriod 138

ob10347v1
Preliminary

Encapsulation of ATB packets

The STPv2 adapter encapsulates 64-bit ATB cells into the payload of 64-bit STPv2 data packets. The first ATB cell of an observation packet is encapsulated into a *D64MTS* STPv2 packet; *D64* STPv2 packets are used for subsequent cells of the same observation packet. If observer parameter *wTimeStamp* is set to a value greater than 0, a *Timestamp Size* nibble and the time stamp are appended to the *D64MTS* packet. If *wTimeStamp* is 0, only the *Timestamp Size* nibble, set to a value of 0, is appended.

If an overflow packet is received by the observer, an *MERR* STPv2 packet is sent, indicating an STPv2 master error, prior to the encapsulation of the probed data into *D64MTS* and *D64* packets.

Time stamp compression

When *wTimeStamp* is greater than 0, the number of bits sent in the *Timestamp* field of *D64MTS* packets depends on the difference between the time stamp from the current observation packet and the previous time stamp value sent in the STPv2 stream. The previous time stamp may be either itself the time stamp of the previous observation packet or the absolute time stamp sent in an STPv2 synchronization sequence.

Only nibbles which are different between the current and previous time stamps are sent, left-aligned, in the *Timestamp* field of the *D64MTS* packet. The field is preceded by a *Timestamp Size* nibble, which indicates the field size, in nibbles. The encoding of *TimeStamp Size* is defined in Section 7.2 of MIPI STP specification version 2.00.01.

The following table shows several examples of time stamp compression values.

Time stamp value		STP Timestamp	
Previous	Current	Timestamp Size	Compressed Timestamp
0x0000 0000 1111 2222	0x0000 0000 1111 2222	0	--
0x0000 0000 1111 2222	0x0000 0000 1111 2234	2	0x34
0x0000 0000 1234 5678	0x0000 0000 1254 5601	6	0x545601
0x0123 0000 0000 0000	0x0226 0000 0002 0030	16 ^a	0x0226 0000 0002 0030
0x0000 0000 0012 3459	0x0000 0000 0012 3457	1	0x7 ^b

Table notes

- a The Timestamp Size nibble is encoded for 16 nibbles even though only 15 nibbles are different because there is no encoding for 15 (see MIPI STP specification).
- b It is possible for the time stamp value of the packet being processed to be less than the time stamp of the previous packet. This can occur if an observation packet, which has an earlier time stamp than a packet from another probe, is delayed due to buffering in the observation network.

Examples of STP encapsulation of ATB packets

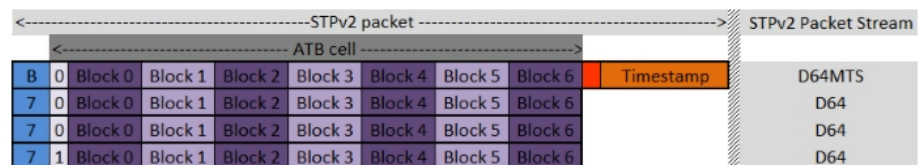
Several examples of ATB encapsulation are shown below, for which the following legend used is:

Legend	Bit width
STPv2 packet opcode	4
Bit Last of ATB cell	1
ATB blocks	9
Timestamp size	4
Timestamp (STPv2NAT)	variable

STPv2 encapsulation with timestamp of 1-cell ATB packet

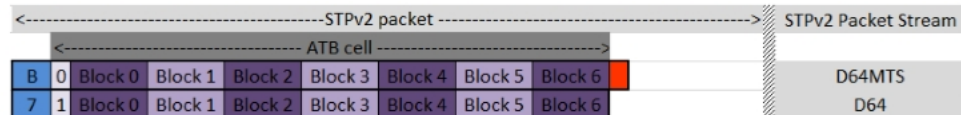


STPv2 encapsulation with timestamp of 4-cell ATB packet



STPv2 encapsulation without timestamp of 2-cell ATB packet

Although the STPv2 Adapter is configured without timestamps, the Timestamp Size nibble is still appended to the *D64MTS* packet. Its value is 0, indicating that no time stamp is present in the packet.



od15275v1
Preliminary

Flush Control

STPv2 flush operations comply with the STP specification: all handling of packets is unconditionally completed once transmission of the packets has begun.

od8609v1

Power disconnection of the observer output

The observer debug interface supports the following power disconnect protocols: Arteris proprietary, OCP, and SYSTEM.

More information about these protocols and the mechanisms to initiate a power disconnect is available in related reference documentation ([./PDF/dmpFlexNoC_Power_Management.pdf](#)).

A request to disconnect the observer debug interface may be initiated by either the SoC power management unit (master disconnect) or the SoC debug module (slave disconnect). When a disconnect is requested, and before the disconnect is acknowledged, the observer completes the generation of ATB packets currently being processed and their encapsulation into an STPv2 stream.

Once disconnected, all incoming packets to the observer received from the observation network are discarded. The observer ceases ATB packet generation and STPv2 encapsulation.

When the ATB socket is either disconnected or reset, all output signals of the observer debug interface are all forced to 0, except *AFReady*, which is forced to 1.

od10348v1

Observer signals

od10349v1
Preliminary

Error logging signals

od10350v1
Preliminary

Fault

Output port signal *Fault* is asserted when an error occurs in the probed packet and the error logging function has been enabled.

It can be used to drive an interrupt signal, either directly or via a FlexNoC sideband manager.

Fault ports are implemented when parameter *errorLoggers* is set to a value greater than 0, in which case the ports are enumerated **Fault_0** to **Fault_(*errorLoggers* – 1)**.

DIRECTION Output

RELATED PARAMETERS	RELATED REGISTERS
<i>errorLoggers</i> 128	<i>FaultEn</i> 133

od10351v1

Sideband signals for user-defined security filtering

When parameter *errorLogger* is set to User-defined filtering, as many as 128 sideband signals can be added. Each signal must be specified in terms of name, input or output direction, and width up to 32 bits.

Debug interface signals

od12103v1

ATB endpoint signals

od15378v1
Preliminary

The ATB endpoint comprises the following signals.

ATValid

Signal *ATValid* is asserted to 1 when a valid ATB word is present. If the signal is 0, all other *AT*-prefixed signals should be ignored.

DIRECTION Output

od10352v1

ATReady

Signal *ATReady* is driven to 1 by the ATB slave to indicate that it is ready to accept data.

DIRECTION Input

od8338v1
Preliminary

ATId

Signal *ATId* identifies the source of the ATB data. The signal is constant and its value is set in register *AtbId*.

WIDTH (6:0)

DIRECTION Output

od8339v1
Preliminary

ATBytes

Output signal *ATBytes* indicates the number of bytes available in the ATB word being transferred, minus 1. Signal width is equal to $\log_2(wData / 8)$.

When ATB parameter *wData* is set to 128:

- § If *ATBytes* (3:0) is 4'b0111, the ATB word contains one cell, because it contains a synchronization cell.
- § If *ATBytes* (3:0) is 4'b1111, the ATB word contains two cells.

od8340v1

ATData

Output signal *ATData* is the ATB word which can convey header, payload, statistic counter values, and synchronization information. Its width is equal to ATB parameter *wData*.

od8341v1

AFValid

Input signal *AFValid* is driven to 1 by the ATB slave to request flushing of all buffers.

od8342v1

AFReady

Output signal *AFReady* is driven to 1 by the ATB endpoint to acknowledge that all buffers have been flushed.

od8343v1
Preliminary

STPv2 encapsulation signals

od8344v1
Preliminary

The following additional signals are added to the observer debug interface when STPv2 encapsulation is used.

SyncReq

When input signal *SyncReq* transitions from low to high, an STPv2 synchronization sequence is sent.

The signal is created when the observer parameter *debugOutput* is set to STPv2 (Specification: Interface: Observation: debugOutput).

The signal must be synchronous to the observer clock, and must remain high for at least one clock cycle.

More information on synchronization sequence generation is available in related reference documentation.

od10360v1

RELATED REFERENCES

Synchronization sequence generation
..... 121

od8598v1

TimeStamp

Input signal *TimeStamp* carries the time stamp which is inserted into the STPv2 synchronization sequence. It is also used to calculate the compressed time stamp value for insertion into STPv2 *D64MTS* packets.

The signal value must be absolute and encoded in natural unsigned format (STPv2NAT).

Observer sub-parameter *wTimeStamp* sets the width of *TimeStamp* (Specification: Interface: Observation: debugOutput). If set to 0, the signal is not created, and synchronization sequences are sent without time stamps. The parameter is available when the debug interface is set to STPv2.

NOTE A common time stamp signal source can be used for all observers that belong to the same clock regime.

Power disconnect signals

Power disconnect protocol signals are generated on the ATB observer endpoint if the power domain of the ATB target, set by parameter *IPpowerDomain* (Specification: Interface: Observation: debugOutput: power) is of type *CLOCK_ONLY* or *SUPPLY*, and is different from the observer power domain.

od10363v1

FlexNoC power disconnect protocol signals

FlexNoC power disconnect protocol signals *SocketConn* and *SlvRdy* are implemented if sub-parameter *disconnect* (Specification: Interface: Observation: debugOutput: power) is set to *ARTERIS*.

More information about the FlexNoC power disconnect protocol is available in related reference documentation ([../PDF/dmpFlexNoC_Power_Management.pdf](#)).

od14766v1

OCP power disconnect signals

OCP power disconnect signals *MConnect* and *SConnect* are implemented if sub-parameter *disconnect* (Specification: Interface: Observation: debugOutput: power) is set to *OCP*.

More information about the OCP power disconnect is available from the OCP organization (<http://www.ocpip.org/membership/information/wheel/specification/>).

od14118v1

Observer parameters

od10366v1
Preliminary

clock

Parameter *clock* sets the clock for the referenced FlexArtist design element.

Clocks are created and configured in the *Clock* page of the design editor *Specification* view *Domain* section.

LOCATION Various places in FlexArtist editors

VALUES List of available clocks

od4827v1

Error logging parameters

The following parameters are available to configure error logging.

od15377v1

errorLoggers

Parameter group *errorLoggers* sets the number of error loggers for the endpoint of the referenced observer unit.

The number of available sub-parameters available depends on the kind of filtering set for each logger: *Standard filtering*, or *User-defined filtering*.

LOCATION Specification: Interface: Observation

VALUES 0–8

od8394v1

Error logging standard filtering parameters

The following standard filtering parameters are available.

od10372v1
Preliminary

ignoredErrorCodes

Parameter *ignoredErrorCodes* specifies the number and type of error codes which will be masked from triggering an error. For each error code, the error type can be selected from: SLV, DEC, UNS, DISC, SEC, HIDE, TMO, RSV_7.

LOCATION Specification: Interface: Observation: errorLoggers

od10373v1
Preliminary

flagFiltering

Parameter *flagFiltering* defines a match pattern for supported interface signals, as well as user and security flags. When the pattern matches, the packet is logged in the error logger.

LOCATION Specification: Interface: Observation: errorLoggers

VALUES x, 0, 1

od10374v1

Error logging user-defined filtering parameters

The following user-defined filtering parameters are available.

od10376v1
Preliminary

coreName

Parameter *coreName* specifies the name of the customer cell which will contain the filtering logic associated with the error logger. By default, *coreName* is set to the name of the observer and the suffix *_Core*.

LOCATION Specification: Interface: Observation: errorLoggers

od10377v1
Preliminary

interfaceFlags

Parameter *interfaceFlags* lists the interconnect protocol, user and security flags that will be made available at the port interface of the customer cell.

LOCATION Specification: Interface: Observation: errorLoggers

od10378v1
Preliminary

sidebands

Parameter *sidebands* sets a list of sideband signals connected to the referenced design element.

The signals are characterized by sub-parameters *name*, *direction*, and *width*.

In certain cases, sub-parameter *async* can be used to specify that the signal is asynchronous with the design element core.

LOCATION Architecture: Datapath Transport: Firewall: version, other places in FlexArtist software

VALUES 0-128

od5529v2

forward

When set to **True**, parameter *forward* allows error packets that match the filter criteria of the *current* error logger to be forwarded to the *next* error logger in the chain.

NOTE Error packets that do not match the filter criteria are always forwarded to the next stage.

LOCATION Specification: Interface: Observation: errorLoggers

VALUES False, True

observeUserBits

When set to **True**, parameter *observeUserBits* enables the user-bit information contained in probed datapath packets to be observed.

LOCATION Specification: Interface: Observation

VALUES True (default), False.

od10375v1
Preliminary

Debug interface parameters

od7456v1

The following parameters are available to configure the observer debug interface.

debugOutput

Parameter *debugOutput* determines whether the observer includes ATB formatting and STPv2 encapsulation.

LOCATION Specification: Interface: Observation

VALUES None, ATB, STPv2

od10380v1

wData

Parameter *wData* sets the width of ATB signal *ATData*.

LOCATION Specification: Interface: Observation: debugOutput

VALUES ATB only: 8, 16, 32, 64, 128. With STPv2 encapsulation: 8, 16, 32, 64

od10367v1

IPpowerDomain

Parameter *IPpowerDomain* indicates the power domain of the SoC debug module that is connected to the observer debug interface.

If the referenced power domain differs from the power domain of the observer, a disconnect interface is created at the observer debug interface. This interface is identical to a target NIU disconnect interface.

When the interface is disconnected, the observer stops ATB packet generation and STPv2 encapsulation.

NOTE Power domains are specified in the [Power and Voltage](#) page of the design editor [Specification](#) view [Domain](#) section.

LOCATION Specification: Interface: Observation: debugOutput: power

VALUES List of available power domains

od14745v1
Preliminary

od10382v1

disconnect

Parameter *disconnect* sets the power disconnect protocol to be used at the observer debug interface.

NOTE Parameter *disconnect* is only available if at least one power domain has been defined in the [Power and Voltage](#) page of the FlexArtist design editor Specification view [Domain](#) section.

A full description of the disconnect protocol is available in related documentation ([../PDF/dmpFlexNoC_Power_Management.pdf](#)).

LOCATION Specification: Interface: Observation: debugOutput

VALUES None, ARTERIS, OCP, SYSTEM

RELATED REFERENCES

Power disconnect signals 128
 Power disconnection of the observer
 output 125

od10383v1

wTimeStam p

Observer parameter *wTimeStam p* sets the width of *TimeStam p*, the observer STP adapter input signal.

If set to 0, the signal is not created. As a result, no time stamp is appended to the STPv2 synchronization sequence, and the Timestamp Size nibble of *D64MTS* STPv2 packets is set to 0.

This parameter is only available when observer parameter *debugOutput* (Specification: Interface: Observation) is set to *STPv2*.

LOCATION Specification: Interface: Observation: debugOutput

VALUES 0 (default)–48

RELATED PARAMETERS

debugOutput 130

RELATED SIGNALS

TimeStam p 127

od11521v1
Preliminary**timeStam pOnProbes**

When parameter *timeStam pOnProbes* is set to *True*, an input signal *TimeStam p* is added to each packet probe and transaction probe connected to the referenced observer. This signal stamps a time reference directly in the packet generated by the probe before it is sent to the observer.

When set to *True*, time stamping is more accurate because packets are stamped when created, rather than once observation packets reach the observer. This advantage, however, costs additional time stamp signals and logic created for each probe, and greater size of observation network packets.

When the parameter is set to *False*, probes connected to the referenced observer do not have a time stamp input. Instead, the observation packets are time stamped in the observer itself, using its dedicated *TimeStam p* signal.

NOTE This mode of time stamping is backward-compatible with PDDs created with FlexNoC releases prior to v.2.11.2

The parameter is available only for observers whose parameter *debugOutput* is set to STPV2 (Specification: Interface: Observation).

LOCATION Specification: Interface: Observation: debugOutput

VALUES True, False

od17188v1

synthesisInfo

Parameter *synthesisInfo* sets user-defined directives such as functions, commands, or arguments that are included in FlexNoC scripts exported for execution in third-party synthesis environments.

LOCATION Specification: Domain: Clock, Specification: Interface: Ports, other locations in FlexArtist editors

VALUES Plain text character string (512 characters max.)

od10369v1

comment

Possible use for parameter *comment* includes preferred or optimal settings, cross-references to dependencies, specifications, or other external references, synthesis constraints, personal reminders, internal codes or project names, and so on. You can use any standard alphanumeric US-ASCII character in the range **a–z, A–Z, 0–9**, plus the standard US-ASCII typographical symbols, including the space character.

Comments are entered either in single-line text-boxes, limited to 100 characters, or in multiple-line dialog boxes, which can store up to 65,336 characters: roughly the equivalent of 43 standard US-letter or A4-size printed pages.

The parameter has no effect on interconnect configuration or FlexNoC FlexArtist operations.

LOCATION Various places in the FlexNoC FlexArtist editors, usually in or near the last column in tabular displays.

VALUES plain-text character string

od2310v1

Observer registers

Observer functions, such as error logging, ATB formatting, and STPV2 encapsulation, are run-time programmable via registers. For a given function, in addition to the specific registers for that function, the register set includes standard identification registers

(../PDF/trpFlexNoC_NIU_Transaction_Handling.pdf).

The base address for each function is specified in parameter *base* (Architecture: Service: Registers).

Reset values of read–write registers can be specified in parameter *rstVal* (Structure: Parameters: Control Setting).

Setting *rstVal* to Set by jumper is not supported for ATB formatting registers.

WARNING Although *rstVal* of read–write registers for error logging and STP encapsulation functions may be set to Set by jumper, it is strongly recommended to only change input values at boot time

od10385v1

Error Logging registers

Registers related to error logging are summarized in the following table.

Address offset	Name	Type	Retention
0x0008	FaultEn	Control	Yes
0x000C	ErrVld	Status	No
0x0010	ErrClr	Pulse	No
0x0014	ErrLog0	Status	No
0x0018	ErrLog1	Status	No
0x001C	ErrLog2	Status	No
0x0020	ErrLog3	Status	No
0x0024	ErrLog4	Status	No
0x0028	ErrLog5	Status	No
0x002C	ErrLog6	Status	No
0x0030	ErrLog7	Status	No
0x0034	ErrLog8	Status	No
0x0038	StallEn	Control	Yes

od17097v1

FaultEn

When 1-bit register *FaultEn* is set to 1, error reporting signal *Fault* is enabled and asserted when register *ErrVld* is 1. When *FaultEn* is cleared to 0, signal *Fault* is driven to 0.

BITS (0)

TYPE Control

od10387v1

RELATED SIGNALS

Fault 125

ErrVld

One-bit register *ErrVld* indicates that an error is logged in the *Errlog* registers.

BITS (0)

TYPE Status

od10388v1

ErrClr

Writing 1 to one-bit register *ErrClr* clears register *ErrVld*. Reading the register has no effect.

NOTE The written value is not stored in *ErrClr*. A read always returns 0.

TYPE Pulse

BITS (0)

od10440v1

ErrLog0

Register *ErrLog0* logs fields from the FlexNoC packet transport protocol header of the logged packet (see "Packet header fields" on page 45).

The MSB field indicates the transport protocol version number.

The following table shows the field assignments. All other bits are reserved and read as 0.

Bit range	Field name
0	Lock
4:1	Opc
10:8	ErrCode
27:16	Len1
31	Format

TYPE Status

od10390v1
Preliminary

Lock

Register field *Lock* contains the packet header bit *Lock*.

TYPE Status

BIT (0)

od10391v1
Preliminary

Opc

Register field *Opc* contains the packet header field *Opc*.

TYPE Status

ACCESS read

BITS (4:1)

od14682v1

Len1

Register field *Len1* contains packet header field *Len1*.

TYPE Status

ACCESS read

BITS (27:16)

od14680v1

ErrCode

Register field *ErrCode* contains packet header field *ErrCode*.

If the packet field does not exist, the register field contains 0.

TYPE Status

ACCESS read

BITS (10:8)

od10393v1

Format

Register field *Format* specifies the version of the FlexNoC packet transport protocol, which is identical to the FlexNoC software version.

The field is set to 1 for FlexNoC versions 2.7 and later.

TYPE Status

ACCESS read

BIT (31)

od10395v1

ErrLog1

Register *Errlog1* contains the least significant part of the FlexNoC packet transport protocol packet header field *Routeld* of the logged error. Up to 32 bits will be stored, depending on the width of *Routeld*. The field is right-aligned in the register, and unused bits are read as 0.

The *Routeld* value can be used to identify the initiator–target mapping associated with the logged error, in order to calculate the complete address.

When decoding *Routeld* sub-field *Seqld*, bit index 0 corresponds to the value for packet header field *Opc* when:

- § The transaction comes from an interface with a protocol, such as AHB, AXI, or AMBA, that allows response disorder based on transaction opcodes.
- § The NoC supports other interface protocols that do not.

A description of the sub-fields comprising *Routeld* is provided in related reference documentation.

TYPE Status

od3207v2

RELATED REFERENCES

Routeld45

ErrLog2

Register *Errlog2* contains the most significant part of the FlexNoC packet transport protocol packet header field *Routeld* of the logged error. Up to 16 MSBs of *Routeld* are stored, depending on the width of *Routeld*. The field is right-aligned in the register, and unused bits are read as 0.

The *Routeld* value can be used to identify the initiator–target mapping associated with the logged error, in order to calculate the complete address.

A description of the sub-fields comprising *Routeld* is provided in related reference documentation.

This register is only present if the width of *Routeld* is greater than 32 bits.

TYPE Status

od10397v1

RELATED REFERENCES

Routeld45

ErrLog3

Register *Errlog3* contains the least significant part of the FlexNoC packet transport protocol packet header field *Addr* of the logged error. Up to 32 bits will be stored, depending on the width of *Addr*. The field is right-aligned in the register and unused bits are read as 0.

Because *ErrLog3* only contains the transport address offset within the particular mapped region, its value must be used in conjunction with the local addresses of the initiator–target mapping, in order to reconstitute the complete transaction address. The mapping is identified using *Routeld*, logged in *ErrLog1* and optionally *ErrLog2*.

TYPE Status

od10398v1
Preliminary

RELATED REFERENCES

Reconstructing transaction addresses
.....29

ErrLog4

Register *Errlog4* contains the most significant part of the FlexNoC packet transport protocol packet header field *Addr* of the logged error. Up to 16 MSBs of *Addr* are stored, depending on the width of *Addr*. The field is right-aligned in the register and unused bits are read as 0.

Because *ErrLog4* only contains the transport address offset within the particular mapped region, its value, concatenated with *ErrLog3*, must be used in conjunction with the local addresses of the initiator–target mapping in order to reconstitute the complete transaction address. The mapping is identified using *Routeld*, logged in *ErrLog1*, and optionally, *ErrLog2*.

This register is only present if the width of *Addr* is greater than 32 bits.

TYPE Status

od10399v1
Preliminary

RELATED REFERENCES

Reconstructing transaction addresses
.....29

ErrLog5

Register *Errlog5* contains the least significant part of the FlexNoC *packet transport* protocol packet header field *User* of the logged error.

Up to 32 bits will be stored, depending on the width of *User*. The field is right-aligned in the register, and unused bits are read as 0.

TYPE Status

od10400v1

ErrLog6

Register *Errlog6* contains the midrange part of field *User* contained in the packet header of the *packet transport* protocol. Bit *User* (32) up to bit *User* (63) will be stored, depending on the width of *User*. The field is right-aligned in the register, and unused bits are read as 0.

The register is only present if the width of *User* is greater than 32 bits.

TYPE Status

ErrLog7

od10401v1
Preliminary

Register *Errlog7* contains the FlexNoC *packet transport* protocol packet header field *Security* of the logged error. The field is right-aligned in the register and unused bits are read as 0.

TYPE Status

od10402v1

ErrLog8

Register *Errlog8* contains the most significant part of the FlexNoC packet transport protocol packet header field *User* of the logged error. Bits *User*(64) up to *User*(95) are stored, depending on the width of *User*. The field is right-aligned in the register, and unused bits are read as 0.

The register is present only if the width of *User* is greater than 64 bits.

TYPE Status

od10403v1
Preliminary

StallEn

Register *StallEn* controls the stall mode of the FlexNoC error logger.

When set to 1, error packets (*status* = **ERR**) are stalled, that is, queued at the input to the error logger. The packets are only passed into the error logger once the currently logged error packet has been cleared by setting register *ErrClr* to 1.

When set to 0, only the first error packet is stored in the error logger. Subsequent error packets are lost.

NOTE The number of error packets which may be queued is determined by the storage capacity of the observation network.

REGISTER ADDRESS OFFSET 0x38

WIDTH 1 bit

TYPE Read-Write Control

RETENTION Yes

od17096v1

ATB endpoint registers

Registers which are used to control the ATB endpoint are summarized in the following table.

Address Offset	Name	Type	Retention
0x0008	AtbId	Control	Y
0x000C	AtbEn	Control	Y
0x0010	SyncPeriod	Control	Y

AtbId

od10404v1
Preliminary

Register *AtbId* is a 7-bit control register that sets the ATB ID for the observer on signal *ATId*.

Each observer within the NoC must have a unique ID. This value, together with the unique probe ID, allows the probe source to be identified in the ATB traffic.

NOTE Values 0x00 and 0x70 to 0x7F are reserved for CoreSight.

TYPE Control

BITS 6:0

AtbEn

When set to 1, register *AtbEn* enables ATB output.

NOTE Setting register *AtbEn* to 0 has an equivalent effect to a power disconnect on this socket: the transmission of an ATB packet in progress will be completed before the ATB output is disabled.

TYPE Control

BITS 0

od10405v1
Legacy
Archive

SyncPeriod

Register *SyncPeriod* sets the period, in terms of the number of ATB cells sent, between insertion of ATB synchronization cells.

The following table summarizes the relationship between *SyncPeriod* and the delay between synchronization cells.

<i>SyncPeriod</i> value	Period of sync cell insertion
0	No sync cell insertion
1-2	8
3-22	$2^{(\text{SyncPeriod} + 1)}$
23-31	Not supported

This register is not present when setting parameter *debugOutput* of the observer to STPv2, since ATB synchronization cells are not generated when using the STPv2 adapter.

WIDTH (4:0)

TYPE Control

od10406v1

STPv2 encapsulation registers

Registers relative to STPv2 encapsulation are summarized in the following table.

Address offset	Name	Type	Retention
0x0008	AsyncPeriod	Control	Y
0x000C	STPv2En	Control	Y

AsyncPeriod

Register *AsyncPeriod* is a 5-bit control register that defines the period, in observer clock cycles, between transmission of STPv2 synchronization sequences.

The following table summarizes the relationship between *AsyncPeriod* and the delay between synchronization sequences.

od18471v1
Preliminary

od10408v1
Preliminary

AsyncPeriod value	Period between synchronization sequences
0	No sync sequence inserted due to AsyncPeriod
1-22	$2^{\text{AsyncPeriod}}$
23-31	2^{23}

Control field. Bits 4:0

od8600v1

STPV2En

When set to 1, register *STPV2En* enables the STPv2 output.

When set to 0, the observer debug interface is disabled: neither STPv2 packets nor synchronization sequences are sent.

By default, the reset value is 1 so that STPv2 stream generation is enabled. If it should be enabled explicitly by software, the reset value can be changed to 0 by setting its parameter *rstVal* (Structure: Parameters: Control Setting).

NOTE Setting register *STPV2En* to 0 has an equivalent effect to a power disconnect on this interface: the STPv2 encapsulation and transmission of an ATB packet will be completed before the observer debug output is disabled.

Control register. Bit 0

od10031v1

Glossary

activate

1. Used in conjunction with run-time programming of NoC registers that are accessible through the service network, to enable a particular feature.
2. In the context of read or write operations to SDRAM memory, opens a row of a selected bank for subsequent access.

address mapping

The projection, in whole or part, of a *local physical address* space, or *range*, of an initiator or target socket, into a virtual, infinite byte space, called the *global address* space.

Address ranges within an address map are typically characterized by a start address, a size, an expected destination port, an offset, and an attribute authorizing read, write, or read-write operations.

advanced mode

The second of four FlexNoC software operating modes. This mode adds advanced or enhanced functionality to basic operating mode features. Access to modes is entirely controlled by product licensing.

ATB cell

A string of 64 bits comprising seven 9-bit blocks and an MSB (bit Last).

ATB packet

A group of one or more ATB cells. The final cell of the packet is indicated by its MSB (bit Last) set to 1.

basic mode

The first of four FlexNoC software operating modes. This mode includes standard end-user features and functionality. Other operating modes are end-user *advanced*, and Arteris internal *experimental* and *developer*. Access to modes is entirely controlled by product licensing.

clock regime

A element in FlexNoC interconnect designs that does not necessarily have a direct counterpart in FlexNoC FlexArtist RTL export output, but whose role is crucial to ensuring that the interconnect synthesis process receives the information it needs to create instances of FlexNoC hardware IP library components.

compact

In the context of address space, the most common type of mapping whose mask format is expressed as 2^{n-1} . Mappings whose masks are formatted differently belong to the category *striped*.

configure

In FlexNoC technology, typically used in conjunction with design-time configuration of interconnect registers by setting parameters with FlexNoC FlexArtist software. These registers are typically accessible through the service network, and run-time programmable.

conversion

1. In FlexNoC FlexArtist software, a group of parameters used to specify in detail how back-and-forth conversions are handled between Arteris external or third-party protocol *specific* interfaces and the Arteris internal FlexNoC *generic* interface, as required by initiator or target NIUs.
2. The process that converts these protocols at the periphery of the FlexNoC interconnect.

datapath packet

An NTP packet in the datapath network.

design time, n.; design-time, adj.

In FlexNoC technology, typically used in conjunction with the configuration of interconnect registers by setting parameters with FlexNoC FlexArtist software. These registers are typically accessible through the service network, and run-time programmable.

Designer

The Qualcomm Technologies, Inc. proprietary interconnect design software developed by the Qualcomm Interconnect Technology Center (QITC) as part of the QNoC hardware IP and software product line. The software is used to implement the hardware IP in QNoC interconnect designs that can be exported for integration with third-party SoC development environments.

Renamed to *Creator* in 2016.

dissolve

In the FlexNoC FlexArtist architecture editor, an action that effectively removes a module from the design project hierarchy by promoting the module's constituent elements to the next higher level.

edge

(deprecated). A link between two nodes in a data structure. The preferred term in this context is *wire*.

FlexArtist

An Arteris, Inc. trademark for the Arteris-proprietary interconnect design software, part of the Arteris-proprietary hardware and software product line marketed under the Arteris registered trademark name *FlexNoC*, up to and including FlexNoC version 2.10.4. Interconnect designs developed with the software can be exported for integration with third-party SoC development environments. Certain Arteris assets, excluding Arteris trademarks but including the hardware and software technology upon which FlexNoC version 2.10.4 was based, were acquired by Qualcomm Technologies, Inc. in October 2013. Subject to Qualcomm license agreements, Arteris is allowed to continue to market and distribute under its own trademark subsequent versions of FlexNoC 2.10.4 based on the technology acquired and henceforth developed by Qualcomm.

FlexNoC

Arteris, Inc. product-line trademark for Arteris proprietary hardware and software technology developed by Arteris until October 2013 and including FlexNoC version 2.10.4. Certain Arteris assets, excluding Arteris trademarks but including the hardware and software technology upon which FlexNoC version 2.10.4 was based, were acquired by Qualcomm Technologies, Inc. in October 2013. Subject to Qualcomm license agreements, Arteris is allowed to continue to market and distribute under its own trademark subsequent versions of FlexNoC 2.10.4 based on the technology acquired and henceforth developed by Qualcomm.

FlexVerifier

An Arteris, Inc. trademark for Arteris-proprietary verification software, part of the Arteris-proprietary hardware and software product line marketed under the Arteris registered trademark name *FlexNoC*, up to and including FlexNoC version 2.10.4.

The software creates a complete test environment for interconnect designs developed with the Arteris-proprietary software FlexArtist. Certain Arteris assets, excluding Arteris trademarks but including the hardware and software technology upon which FlexNoC version 2.10.4 was based, were acquired by Qualcomm Technologies, Inc. in October 2013. Subject to Qualcomm license agreements, Arteris is allowed to continue to market and distribute under its own trademark subsequent versions of FlexNoC 2.10.4 based on the technology acquired and henceforth developed by Qualcomm.

generic side

In an NIU, one of two principle parts that handle incoming and outgoing transactions within the NoC core. Transactions with third-party socket interfaces situated at the periphery of the NoC are handled by the part called the *specific* side.

global address space

A virtual infinite byte space, associated with an interconnect, into which an initiator or target socket projects, in whole or part, its local, physical address space. Each such projection is called a *mapping*.

interconnect composition

In Arteris technology, an assembly of any number of Arteris interconnect designs with compatible specification, architecture, and structure information that constitutes an SoC-level interconnect.

The corresponding graphical user interface object in FlexArtist software is called a *NoC Composition*.

interconnect synthesis

FlexNoC FlexArtist internal engine that generates interconnect architectures based on FlexArtist specifications, and infers physical structures based on FlexNoC hardware IP library units. Also referred to as *interconnect*, *internal*, or *NoC* synthesis, none of which should be confused with the RTL synthesis performed by third-party tools.

invalid state

In FlexNoC FlexArtist software, the condition of a configurable object, such as a specification or export option, or object attribute, that cannot be used to carry out various commands. Objects in an invalid state are typically associated with a red cross, or check mark symbol:



link

1. In the FlexNoC FlexArtist architecture editor, a four-part graphical object built from three basic geometric shapes: a cone, a square, and a line segment. These shapes are purely *abstract* visual representations of FlexArtist internal protocol link functions, and do not correspond to *actual* physical objects in the design. A link object comprises a central blue square, called the link *main* element. An orange *input cone* element, which by convention appears to the left of the main element, represents data collection operations at the entrance to the link. An orange *output cone* to the right of the main element represents data distribution operations at the point of exit from the link. These three elements are joined by line segments that represent data conduits, called *routes*.
2. In the Arteris *NoC Solution* legacy product line, a packet-handling feature of the Arteris network transport and transaction protocol (NTTP).

main element

In the FlexNoC FlexArtist architecture editor, a constituent element of a link object. A main element represents data handling operations that are inferred from the corresponding specification being referenced by the link.

MISR

(multiple-input signature register).

module

In FlexNoC FlexArtist software, an HDL-compliant structural representation of FlexNoC hardware IP library components. The representation can be instantiated within another such representation, or exported from FlexArtist software for use with standalone tools such as FlexNoC FlexVerifier, or with third-party environments.

multiple-input signature register

(MISR). Arteris FlexNoC unit that generates signature on specific traffic.

NIDEN

(non invasive debug enable). A type of security signal or pin.

NoC transaction and transport protocol

(NTTP). The FlexNoC proprietary packet-based protocol for interconnect operation. The preferred terms for this internal designation are, formally, the FlexNoC *generic interface protocol* for FlexNoC interconnects, or, informally, the *generic interface*.

non invasive debug enable

(NIDEN). A type of security signal or pin.

NTTP

(NoC transaction and transport protocol).

object

In the context of FlexNoC FlexArtist software, a manipulable graphical user interface (GUI) element that typically represents a container for information stored in FlexNoC PDD files. Examples of such objects are architectures, exports, scenarios, specifications, structures, and so on.

observation packet

An NTTP packet in the observation network.

observation subsystem

A set of FlexNoC units comprising probes, observation network components, and an observer, that processes information from *probe points* and forwards it to the observer or reports it to the system using registers and interrupt signals.

observer

A FlexNoC unit used to either log errors, or encapsulate traced packets of statistics packets in order to forward them to a third-party interface.

operating mode

In the context of using FlexNoC software, a state of operation that makes particular features and functionality available, depending on licensing configurations and intended Arteris internal or customer use.

Two are primarily intended for end-users: basic mode, and advanced mode.

origin object

In FlexArtist software, a graphical user interface element that establishes an *affiliate* relationship with other objects which thus inherit certain of the origin object's properties. Typical examples are FlexArtist architecture objects, affiliated with a specification object (the origin), or structure objects, affiliated with an architecture object. Object affiliation can be changed in software with the [Change origin](#) command. Note the difference with a *parent* element, which denotes in FlexArtist software a hierarchical relationship that is visually manifest in the structural organization of the project tree.

parameter group

In FlexArtist software, a collection of two or more related or dependent parameters, represented by a single expandable column heading in a table, or label for an expandable data entry or display field in the Graphical User Interface (GUI) or dialog box.

parameter location string

(PLS). A colon-separated text string of names for subordinate parts in the FlexNoC software graphical user interface (GUI). Such strings appear in FlexNoC print and software help system documentation published by TWT.

The string serves as a memory aid for the location of a parameter in relation to FlexNoC software editors and their constituent views, pages, and parameter groups in which the parameter is found. The string does not contain the name of the parameter itself, which is usually conveyed by the nearest preceding heading or mention.

When placed in a separate paragraph, the string is preceded by the inline heading "Location "; when placed within another paragraph, typically near the end of a sentence to which it applies, the string is placed between parentheses. For example, the location of parameter *x* can be indicated by the shorthand (Architecture: Details: Probe: statisticsCollection) which expands to "Parameter *x* is located in the *statisticsCollection* parameter group displayed in the *Probe* page of the *Details* view of the FlexArtist *architecture editor*".

parameterizable value

Deprecated term for a parameter value that can be set at design time.

PDD

(project description document).

PLS

(parameter location string).

probe point

An intrusive or non-intrusive data collection device that provides observation subsystem units with information.

procedure

1. In the FlexArtist exploration scripting language, a hierarchical object that is referenced in a script process, and defines the following actions in the process: Enqueue, Dequeue, Space, Repeat, Sequence, Choice.
2. In software documentation, a series of steps to follow in order to accomplish a specific task with the product.

project description document

(PDD). A human readable XML-compliant file, recognizable by its file-name extension *.pdd*, in which FlexNoC interconnect design information is stored for use with FlexNoC FlexArtist software.

pseudo switch

In FlexNoC FlexArtist software, a type of packet-transport connection, sometimes referred to as a pseudo *crossbar*, that is used to connect a set of output and input cone objects. Such a connection in the architecture editor is displayed with a characteristic gray applied to routes running between a group of cones, and circumscription by a single domain.

QITC

(Qualcomm Interconnect Technology Center). Located near Paris, France.

QNoC

The Qualcomm interconnect technology initially acquired from Arteris, Inc. in October, 2013. Henceforth developed by the engineering teams that were also acquired from Arteris, the technology is released under the product name "QNoC" to internal Qualcomm engineering groups. Parts of the technology are also delivered under license to Arteris, Inc, which is allowed to market the Qualcomm technology under Arteris trademarks for a duration fixed by the acquisition agreement.

Qualcomm Interconnect Technology Center

(QITC). The Qualcomm, Incorporated entity located near Paris, France, that develops interconnect hardware IP and design configuration, exploration, and verification software.

request path

In the [datapath](#) page of the FlexNoC FlexArtist architecture editor [Topology](#) view, the left part of the dual display of NIU initiators, which are mirrored on either side of a central axis comprising NIU targets.

response path

In the [datapath](#) page of the FlexNoC FlexArtist architecture editor [Topology](#) view, the right part of the dual display of NIU initiators, which are mirrored on either side of a central axis comprising NIU targets.

route

In the FlexArtist architecture editor GUI, a continuous line, comprising one or more black segments, that runs between NIU initiator and target objects. The line, which changes color when a segment passes through a pseudo switch zone in the design, is an abstract representation of network traffic on physical routes.

When a cone object is selected in the editor, all lines that pass through the selected cone are highlighted. Certain segments are highlighted to indicate potential placement points for new or unused links.

The relative thickness of segments in a line represents the serialization of traffic in terms of throughput, or relative carrying capacity—also known as *width*.

Display filters in the architecture editor toolbar determine which serialization characteristic is displayed.

run time (n.); run-time (adj.)

In FlexNoC technology, typically used in conjunction with programming of interconnect registers that are accessible through the service network.

sector

1. In the FlexArtist architecture editor, designates a refined view of the interconnect that emphasizes a particular aspect of the design, such as clock, voltage, or power domains.
2. In the context of the FlexNoC last level cache controller, a group of naturally aligned cache lines.

secure privileged invasive debug enable (SPIDEN). A type of security signal or pin.

segment

1. In the context of FlexNoC FlexArtist software, a linear graphical element in the FlexArtist architecture editor that either joins cones and main element objects to form a link, or directly joins initiator and target objects. The segment represents a data conduit, called a route.
2. In the context of address translation, a set of addresses.

span

In the context of address space, designates the size of the smallest possible compact mapping, expressed as $2^{\log_2(\text{mask})}$.

SPIDEN

(secure privileged invasive debug enable). A type of security pin.

STPv2 packet

A string of one or more 4-bit nibbles conveying an STP opcode and optional payload.

striped

Designates a type of address space mapping that screens or reveals certain regular or irregular intervals within an otherwise attainable or unattainable address space range. This feature is available in FlexNoC software advanced operating mode only.

structure information file

A file exported with FlexArtist software that contains information relative to FlexArtist structure objects. Upon export, FlexNoC FlexArtist proposes a default file name based on the convention *StructureObjectName.info*.

tactical port

An external port required by a system-on-chip. The port is added to FlexNoC interconnect designs during the specification stage of the FlexNoC workflow using FlexNoC FlexArtist software. The port, which has no effect on internal interconnect operations, must be connected manually on the Netlister page view of the FlexArtist structure editor.

target

1. In FlexNoC technology, a third-party core that processes transactions issued by a third-party initiator and routed through the FlexNoC interconnect.
2. An IP or network interface unit (NIU) that receives requests from the interconnect and transmits responses.
3. In the FlexArtist exploration scripting language, an optional object that designates one of four possible behavioral models for interconnect flows to destination units: Instantaneous, SRAM, DRAM, and DRAM Scheduler. A given model can be assigned to any number of flows from various sockets. A flow with no assigned target model receives instantaneous responses to requests.

tenure line

A group of four counters allocated, via a configuration register, as a resource to an observed port.

untimed

In FlexNoC exploration, designates an interconnect model that contains no internal timing delay, resulting in requests and responses being instantaneously transported from initiators and targets whenever possible. Consequently, clocking, bus width, bandwidth, and so forth, are not taken into account, and clock input ports are simply ignored.

valid state

In FlexNoC FlexArtist software, the condition of a configurable object, such as a specification or export option, that can be used to carry out various commands. Objects in a valid state are typically associated with a green check mark symbol:



Verifier

The Qualcomm Technologies, Inc. proprietary interconnect verification software developed by the Qualcomm Interconnect Technology Center (QITC) as part of the Qualcomm proprietary hardware IP and software product line called QNoC. The software is used to create a complete test environment for interconnect designs developed with QNoC Designer software.

wire

In the [Topology](#) view of the FlexNoC FlexArtist architecture editor, a line segment connecting two architecture objects, or *elements*. The relative width of the segment reveals certain characteristics of data flows. These characteristics can be selected with FlexNoC FlexArtist display options.

word

In the context of NTP, the amount of data transferred within one clock cycle. A word may comprise one or more cells, or a fraction of a cell.

Index

A

Acronyms and Initialisms

ATB • 1, 116, 120, See also STP

Adding and connecting probes • 12

C

Codes

error codes, hardware • 115, 116

Collecting statistics • 37

Concepts

error processing • 2

error sources • 2

errors, handling • 1, 116

observation and security filtering • 66

observation power management • 10, 66

processing errors • 2

sources, errors • 2

Configuration • 27

creating an observation subsystem • 11

registers, configuring • 87, 95, 101

statistics alarm, configuring and managing • 38

trace alarm, configuring and managing • 37

Configuring and managing the statistics alarm • 20

Configuring probes for security filtering • 15

Configuring probes for transaction profiling • 40

Counting packets over a fixed period • 39

D

Documentation

legal • 2

E

Error Codes

error codes, hardware • 115, 116

hardware error codes • 116, See also messages,
error

Errors

error logging • 1, 29, 116

error messages • 116

error probes • 128

error processing • 2

error sources • 2

errors, handling • 1, 116

logging errors • 1, 29, 116

processing errors • 2

sources, errors • 2

Examples

run-time programming, measuring latency • 40

F

FlexNoC packet formats • 56

L

Logging

error logging • 1, 29, 116

logging errors • 1, 29, 116

M

Modes

modes, filtering • 40, 108

modes, transaction filter • 40, 108

O

Observation Network Units • 66

ATB • 1, 116, 120, See also STP

measuring latency, transaction probe • 40, 108

modes, filtering • 40, 108

modes, transaction filter • 40, 108

packet probes • 20, 53

probes, packet • 20, 53

Operation

measuring latency, transaction probe • 40, 108

P

Packet header fields • 134

Packet Header Fields Directory

Addr • 47

Debug • 49

Echo • 49

ErrCode • 49

LastFrag • 50

LastWord • 50

Len • 47

Len1 • 47

Lock • 45

Opc • 46

Resilience • 49

Routeld • 46

Security • 48

SeqId • 46, 135

Status • 47

Urgency • 45

User • 48

Packet Payload Fields Directory

Be • 50

Byte • 50

LastWord • 50

- WordErr • 50
 - Packet tracing • 32
 - Packet Transport Units
 - packet header fields • 45
 - Addr • 47
 - Be • 50
 - Byte • 50
 - Debug • 49
 - Echo • 49
 - ErrCode • 49
 - LastWord • 50
 - Len • 47
 - Len1 • 47
 - Lock • 45
 - Opc • 46
 - Resilience • 49
 - Routeld • 46
 - Security • 48
 - Status • 47
 - Urgency • 45
 - User • 48
 - WordErr • 50
 - packet payload fields • 50
 - packet probes • 20, 53
 - packets, tracing • 32
 - probes, packet • 20, 53
 - Parameters
 - allowFilterOnEnabledBytes • 67
 - allowPayloadTracing • 68
 - allowStatisticsSuspend • 72
 - async • 84
 - clock • 70
 - comment • 132
 - coreName • 78, 84, 85, 129
 - crossTrigger • 69
 - delayThresholds • 75
 - direction • 84
 - disconnect • 131
 - errorLoggers • 128
 - errorLogging • 128
 - flagFiltering • 77, 129
 - forward • 130
 - ignoredErrorCodes • 129
 - interfaceFlags • 78, 129
 - IPpowerDomain • 79, 130
 - keepAddressAndFlagsInResponsePacketsForErrorProbes • 69
 - mode • 70
 - module • 66
 - name • 84
 - nComparators • 76
 - nCounters • 73
 - nExternalEvent • 72
 - nFilter • 67
 - nObservable • 74
 - nStatisticsCounter • 71
 - observation • 66
 - observer • 66
 - observeUserBits • 130
 - pendingThresholds • 75
 - portClocks • 82
 - portNames • 81, 82
 - probe • 70
 - securityBitMatch • 83, 85
 - securityFiltering • 77
 - sidebands • 79
 - statisticsCollection • 71
 - statisticsCounterAlarm • 72
 - synthesisInfo • 132
 - timeStampOnProbes • 131
 - traceProbe • 66
 - tracing • 68
 - transactionStatPipe • 66
 - useNIDENinput • 78
 - userFlagsKeptInResponsePacketsForErrorProbes • 70
 - useSPIDENinput • 78
 - useSPNIDENinput • 78
 - useUserFilter • 67
 - wAddress • 80, 82
 - wCounters • 74
 - wData • 130
 - wId • 80, 81
 - width • 84
 - wLength • 80
 - wPrio • 80
 - wSecurity • 81, 82
 - wStatisticsCounter • 72
 - wTId • 82
 - wTimeStamp • 131
 - wUser • 81, 82
 - Power Management
 - observation power management • 10, 66
 - Preambles • 47
 - Probing on response paths • 17
 - Programming
 - run-time programming, measuring latency • 40
 - Protocols
 - ATB • 1, 116, 120, See also STP
 - STP • 121
- ## R
- Reconstructing transaction addresses • 16
 - Registers
 - Active • 91
 - AddrBase_High • 108
 - AddrBase_Low • 108
 - AddrWindowSize • 109
 - AlarmEn • 88
 - AsyncPeriod • 138
 - AtbEn • 138
 - AtbId • 137
 - CfgCtl • 90
 - Counters_M_AlarmMode • 106
 - Counters_M_PortSel • 101
 - Counters_M_Src • 102
 - Counters_M_Val • 107
 - En • 112
 - ErrClr • 133
 - ErrCode • 134
 - ErrEn • 88
 - ErrLog0 • 134
 - ErrLog1 • 135
 - ErrLog2 • 135
 - ErrLog3 • 136
 - ErrLog4 • 136
 - ErrLog5 • 136
 - ErrLog6 • 136
 - ErrLog7 • 137
 - ErrLog8 • 137
 - ErrVId • 133
 - FaultEn • 133
 - FiltByteAlwaysChainableEn • 90
 - FilterLut • 91
 - Filters_N_AddrBase_High • 97
 - Filters_N_AddrBase_Low • 96
 - Filters_N_Length • 99
 - Filters_N_Opcode • 98
 - Filters_N_RouteldBase • 96
 - Filters_N_RouteldMask • 96
 - Filters_N_SecurityBase • 97
 - Filters_N_SecurityMask • 97
 - Filters_N_Status • 99
 - Filters_N_Urgency • 100
 - Filters_N_UserBase • 100

- Filters_N_UserBaseHigh • 100
- Filters_N_UserMask • 100
- Filters_N_UserMaskHigh • 101
- Filters_N_WindowSize • 97
- Format • 135
- GlobalEn • 90
- IntrusiveMode • 89
- Lock • 134
- LockEn • 98
- MainCtl • 87
- Mode • 108, 112
- nTenureLines_i • 113
- ObservedSel_i • 113
- Opc • 134
- Opcode • 109
- OverflowReset • 114
- OverflowStatus • 114
- PayloadEn • 88
- PendingEventMode • 114
- Prescaler • 115
- RdEn • 98, 109
- ReqEn • 99
- RspEn • 99
- SecurityBase • 110
- SecurityMask • 110
- StallEn • 137
- StatAlarmClr • 94
- StatAlarmEn • 94
- StatAlarmMax • 93
- StatAlarmMin • 93
- StatAlarmStatus • 94
- StatCondDump • 89
- StatEn • 88
- StatGo • 93
- StatPeriod • 92
- STPV2En • 139
- SyncPeriod • 138
- Thresholds_i_j • 113
- TraceAlarmClr • 92
- TraceAlarmEn • 92
- TraceAlarmStatus • 92
- TraceEn • 88
- TracePortSel • 91
- UrgEn • 99
- UserBase • 110
- UserBaseHigh • 111
- UserMask • 110
- UserMaskHigh • 111
- WrEn • 98, 109
- Reporting
 - error logging • 1, 29, 116
 - logging errors • 1, 29, 116
- StatSuspend • 58
- SyncReq • 127
- TimeStamp • 59, 127
- TraceAlarm • 58
- User • 63
- Valid • 61
- Software
 - parameters, software • 66, 128
- Subsystems
 - creating an observation subsystem • 11
- Synchronization cell insertion • 117

S

- Security
 - observation and security filtering • 66
- Signals
 - Address • 62
 - AFReady • 127
 - AFValid • 127
 - ATBytes • 126
 - ATData • 127
 - ATId • 126
 - ATReady • 126
 - Length • 62
 - Opcode • 62
 - Rsp_Ready • 65
 - Rsp_TId • 65
 - Rsp_Valid • 65
 - Security • 63
 - StatAlarm • 58

