Mentor®
A Siemens Business

Bus Monitor User Guide

UL-000247-TR (revision 29)

26 March 2021

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

IP Revision History

Always use the correct document revision for the IP version that has been licensed, as indicated below. Using a different document revision may lead to unpredictable behavior from the IP.

IP Version	Details	Document Revision
1 – 5	Documentation available on request	
6	Added cache coherent variants Trace improvements	UL-000247-TR (4)
7	Changed number of 32-bit counters per ***monitor_counter_data*** message	UL-000247-TR (6)
8	Added several features: • Exclusive range compare • Timestamps on min/max counter values • Option to match at end of transaction • Busy metric	UL-000247-TR (11)
9	Increased maximum AXI user and OCP info bus widths from 64 to 96 bits.	UL-000247-TR (15)
10	Added **transaction_differentiator** field to ***monitor_trace_data*** messages	UL-000247-TR (18)
11	Updated to second generation message indexing	UL-000247-TR (19)
12	Increased maximum AXI ID width from 16 to 24 bits	UL-000247-TR (22)
13	Added *clear_all_enables* function to ***set_enabled*** message for **operation**=11	UL-000247-TR (29)

The IP version can be determined from the **version** field in the ***discovery_response*** message. See Section 4.1 for more details on using the Discovery mechanism. The document revision of this document is shown on the title page.

Table of Contents

List of Figures

List of Tables

1 Preface

1.1 About this document

This document provides information on how to use the Tessent Embedded Analytics Bus Monitor in a System on Chip (SoC) design. It provides an overview of the functionality from a user's perspective, and focusses on how to configure the module in order to utilize its capabilities.

Intended audience

SoC developers and users who want to become familiar with how to use the Bus Monitor.

1.2 Using this document

This document is organized as follows:

- Introduction: Read this for a brief overview of the Bus Monitor.
- Functional operation: Read this to understand the functional capabilities offered by the Bus Monitor.
- Using the Bus Monitor: Read this to understand how to use the Bus Monitor. Contains usage examples.
- Messages: Provides a description of all the messages supported by the Bus Monitor. Organized alphabetically for easy reference

1.3 General conventions

Name Capitalization

Throughout this document various signals, parameters, messages and fields are defined which are written as case insensitive. Tessent Embedded Analytics product code is always written as lowercase with underscore separation between name tokens e.g. ‘word0_word1’.

Bit orientation

Throughout this document various tables identify the formats of signals, messages and similar. Such tables always begin with the lowest order bits, and where fields have multiple bits the left most bit is the most significant.

Byte orientation

UltraDebug® is internally little-endian; words and messages are stored and transmitted such that they are least significant bit and byte first enabling terms to be read inside a simulator without shuffling of bytes or reversing of bits.

Typographic conventions

The typographical conventions used in this document are:

bold	Used for message names
bold	Used for field names within messages
`monospace`	Used for source code examples and Verilog ports, parameters and module names
blue	Used for hyperlinks to make them easier to see

1.4 Related reading

UL-000528-TR Bus Monitor Integration Guide

UL-000566-TR An Introduction to the Tessent Embedded Analytics Architecture

UL-000589-TR Message Infrastructure User Guide

2 Introduction

The Bus Monitor analytic module provides a wide variety of monitoring functions. It can be used for debugging, reporting diagnostics, performance profiling, 'bare metal security' and other applications.

Each instance of the Bus Monitor can have up to four independently operational bus interfaces each parameterized at instantiation according to the needs of the application. Because of this, some features and capabilities described in this document may not be available on some Bus Monitor instances. The particular variant of a Bus Monitor instance, and the specific capabilities it supports can be determined via the Discovery Process (see Section 4.1). Additional detail on parameterization can be found in the *Integration Guide* (see Related reading).

Figure 2-1 shows the internal architecture of the Bus Monitor. Operation consists of three processes:

- Interpreting the bus protocol;
- Identifying transactions of interest using a filter;
- Taking action when transactions of interest occur:
  - Gather performance metric data (using counters);
  - Capture the transaction data (trace);
  - Issue a trigger (match message, real-time event or internal trigger).

These processes are discussed in more detail in the next section.

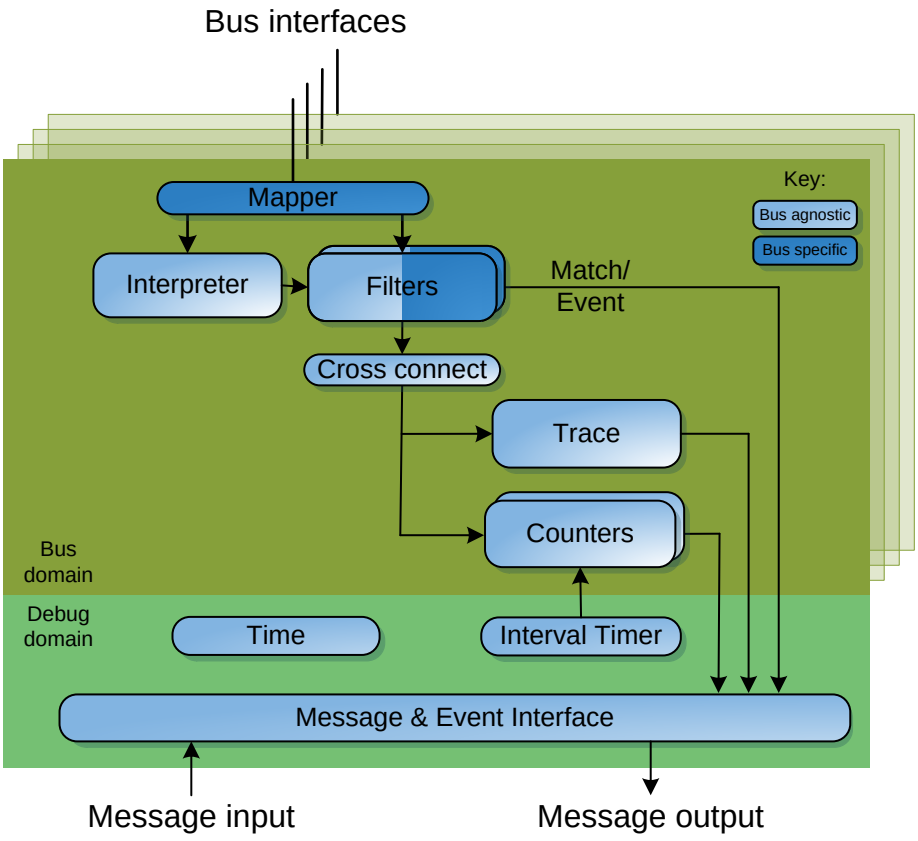

Figure 2-1 Bus Monitor block diagram

3 Functional operation

3.1 Bus protocol interpretation

The bus interpreter monitors the various bus interfaces in order to keep track of ongoing transactions.

3.1.1 Individual transaction accumulators

Individual transaction accumulators (ITAs) allow metrics to be accumulated for individual transactions prior to filtering (as such, ITAs are shared by all filters). This has three distinct uses:

- It enables threshold filtering. For example, to filter only transactions where the duration exceeds some threshold requires the duration of each transaction to be tracked;
- It enables min/max counting. For example, to record the longest single transaction data hesitancy observed over a timed interval;
- It enables metric counting for transactions where the matching criteria isn't fully resolved until after counting needs to start. For example, measuring data latency for transactions that end with an error response means the latency must be accumulated into an ITA. From there it will be transferred to one of the main counters (or not) when the results of the response phase match are known.

To use any of these capabilities, at least one ITA must be available, and it will need to be configured to accumulate the metric of interest; see Section 4.5.

3.2 Filtering transactions of interest

Monitored transactions are analysed by a configurable filter. Use the Discovery Process (see Section 4.1) to determine the filtering capabilities available to you. The fields which affect filtering behaviour are summarized in Table 3-1.

Table 3-1 Discovery fields that affect filtering behaviour

***discovery_response* field**	**Description**
filter_[signal]_*X*	Each of these fields determines which of the various buses can be filtered by filters in group 0 and group 1 for interface *X*: Bit 0: filtered by group 0 filters Bit 1: filtered by group 1 filters
filter_thres_width_*X*	Width of threshold counter for interface *X*. Bits 5:0 are filter group 0, 11:6 are filter group 1. Determines the maximum usable value of **count_threshold**.
filters_*X*	Reports the number of filters associated with interface *X*.
filters_g0_*X*	Reports the number of filters in group 0 for interface *X*. If this field is non-zero, the filters 0 to (**filters_g0_*X*** – 1) will be in group 0, and the remainder in group 1.
itas_*X*	Reports the number of individual transaction accumulators for interface *X*. Must be non-zero to use threshold filtering.
ita_width_*X*	Reports the width of individual transaction accumulators for interface *X*.

Filters are capable of selectively matching on any of the transaction fields, for example:

- address;
- data;
- transaction identifier;
- transaction type;
- burst length;
- burst type;
- etc.

The type of match function used is specific to each field, typical types include:

- match (equality) comparators
  - e.g. burst = B;
- inclusive (inside) range comparators
  - e.g. A ≤ address ≤ B;
- exclusive (outside) range comparators
  - e.g. not (A ≤ address ≤ B);
- multiple choice
  - e.g. response code = X, Y or Z

Transactions can also be filtered against a threshold, so only transactions where a particular metric (e.g. data latency) value is met or exceeded are considered to match, provided they have been assigned to an individual transaction accumulator (see Section 4.5). When using one of the latency metrics, which are signed, the threshold is also treated as signed.

A transaction is qualified if the filter matches all transaction fields for which filtering is enabled. For example, to match any read transaction with an address between A and B, the filter would be configured to match read transactions, any response, address matching would be enabled with A as the lower bound and B as the upper bound, and matching on all other fields would be disabled.

For burst transfers, the match decision is not based on the starting address alone; the transaction will be considered to match if any beat of the transaction is within the range, taking into consideration the address incrementing behaviour for the specified burst type.

3.2.1 Filter cascading

Where filters have been partitioned into two groups, filters in group 1 can be cascaded with a filter from group 0. In this configuration, both filters must match the transaction for it to be qualified. The advantage of this approach is that it allows the two groups of filters to have differing capabilities. For example, the group 0 filters may be able to match any bus fields, but there may only be one or two of these. There may be many group 1 filters, but these may only be able to match on a small subset of fields. This can provide the same or similar capability of many powerful filters without the expense.

3.3 Acting on transactions of interest

This section describes the various actions that can be taken on filtered transactions.

3.3.1 Issuing triggers

The filters can be used to trigger other activities both within the Bus Monitor, and in the wider system. Use the Discovery Process (see Section 4.1) to determine the triggering capabilities available to you. The fields which affect triggering behaviour are summarized in Table 3-2.

Table 3-2 Discovery fields that affect triggering behaviour

***discovery_response* field**	**Description**
filter_thres_width_*X*	Width of threshold counter for interface *X*. Bits 5:0 are filter group 0, 11:6 are filter group 1. Determines the maximum usable value of **count_threshold**.

The following options are available (in any combination ***set_bus_filter*** **issue_mode**):

- Generate any of the 256 Embedded Analytics real-time events when **issue_mode** bit 0 is set. These can be used to initiate various actions elsewhere in the system. If multiple filters schedule the same event on the same cycle, the event will only be issued once;
- Generate a ***monitor_match_data*** message when **issue_mode** bit 1 is set. This can be used to notify the debugger/user that the matching criteria has occurred;
- Generate an internal trigger. This can be used to enable or disable any of the filters, or to initiate a ***monitor_counter_data*** message.

Triggers can be issued at the earliest point in a transaction that the filtering criteria are all met (typically when the transaction starts), or alternatively when the transaction completes (see set_***bus_filter*** **threshold_mode**).

Each filter has a **count_threshold** field (see ***set_bus_filter***) that specifies exactly how many times it must match in order to issue a trigger, after which counting starts again. Furthermore, the trigger can be issued on the 1st or last of **count_threshold** matches. This mechanism can also be coupled to the interval timer, so that a trigger is issued only if the interval timer is running, and the threshold is reached before the timer expires (in this case, counting starts again at the start of each timer interval). See also Section 3.

3.3.2 Counting

Various transaction metrics can be counted using the counters. Use the Discovery Process (see Section 4.1) to determine the counting capabilities available to you. The fields which affect counting behaviour are summarized in Table 3-3.

Table 3-3 Discovery fields that affect counting behaviour

discovery_response field	Description
count_[metric]_*X*	When 1, the metric indicated in the field name can be counted for interface *X*.
counters_*X*	Reports the number of counters available for interface *X*.
counter_width_*X*	Reports the width of counter for interface *X*.
counters_minmax_*X*	Reports the number of counters that support min/max capability. If this field is non-zero, this capability will be available in counters 0 to (**counters_minmax_*X*** – 1).
itas_*X*	Reports the number of individual transaction accumulators for interface *X*.
ita_width_*X*	Reports the width of individual transaction accumulators for interface *X*.

Each counter can be independently configured to count a particular metric for transactions qualified by a specified filter.

The following bus transaction metrics can be collected using counters:

- Number of transactions;
- Duration;
- Bytes;
- Data Beats;
- Latency (data, response and total);
- Hesitancy (address, data, response and total);
- Bus cycles;
- Bus busy;
- Bus concurrency (min/max only – see Table 3-4).

These are illustrated in Figure 3-1 through Figure 3-6.

Bus cycles is simply the number of cycles of the bus clock that have occurred during the measurement interval. It is not affected by any filtering criteria, but the selected filter must be enabled.

Bus busy is the number of cycles when one or more transactions that match the filtering criteria are active.

Bus concurrency is the number of transactions that are in progress simultaneously.

Using these fundamental metrics some transaction statistics can be calculated, for example:

- Average Burst Size = Bytes / Transactions;
- Average Bytes per active cycle = Bytes / Beats;
- Average Data Latency = Data Latency / Transactions;
- Bus cycles utilisation = Transactions or Beats / Bus Cycles.

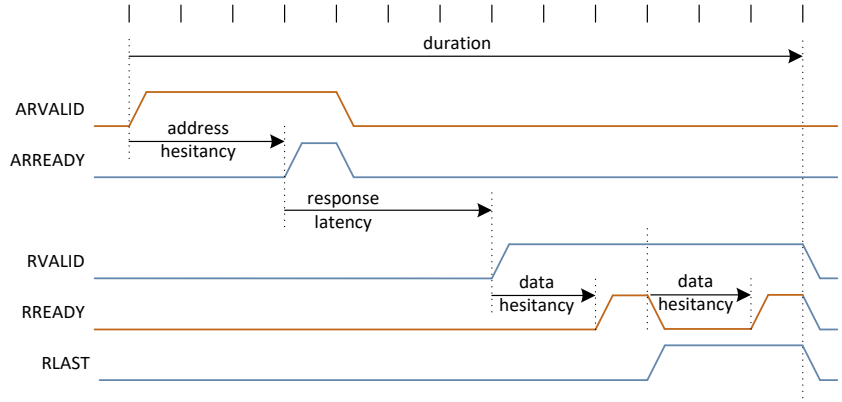

Figure 3-1 AXI read transaction metrics

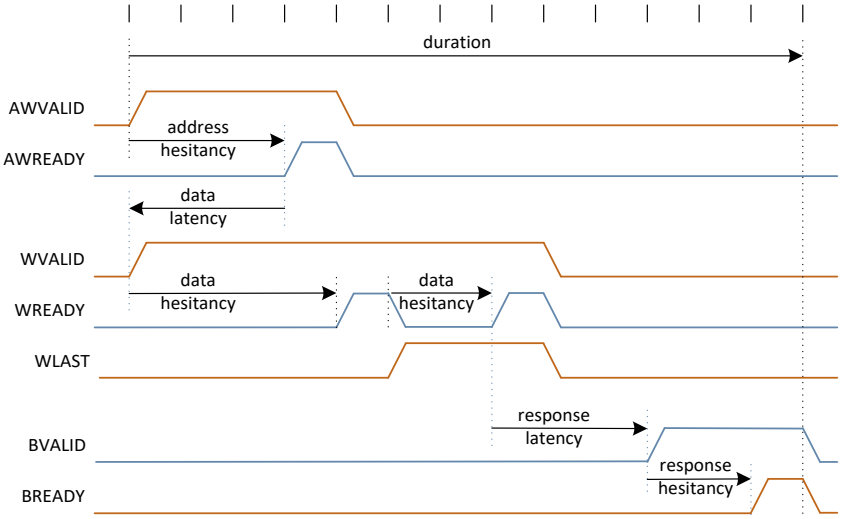

Figure 3-2 AXI write transaction metrics

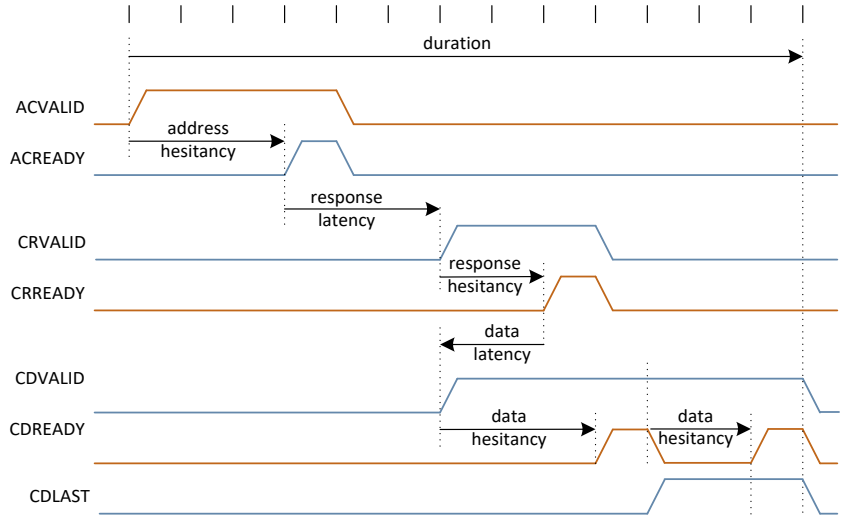

Figure 3-3 ACE snoop transaction metrics

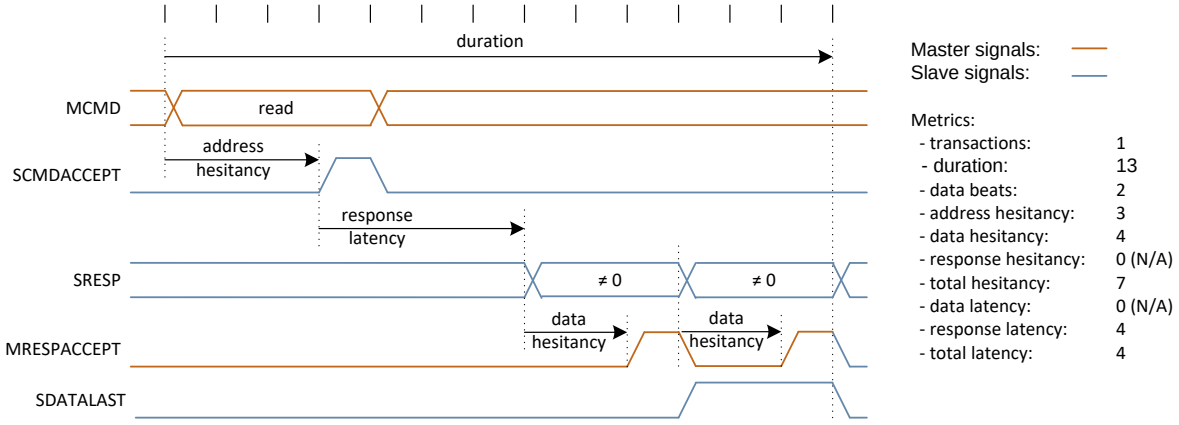

Figure 3-4 OCP read transaction metrics

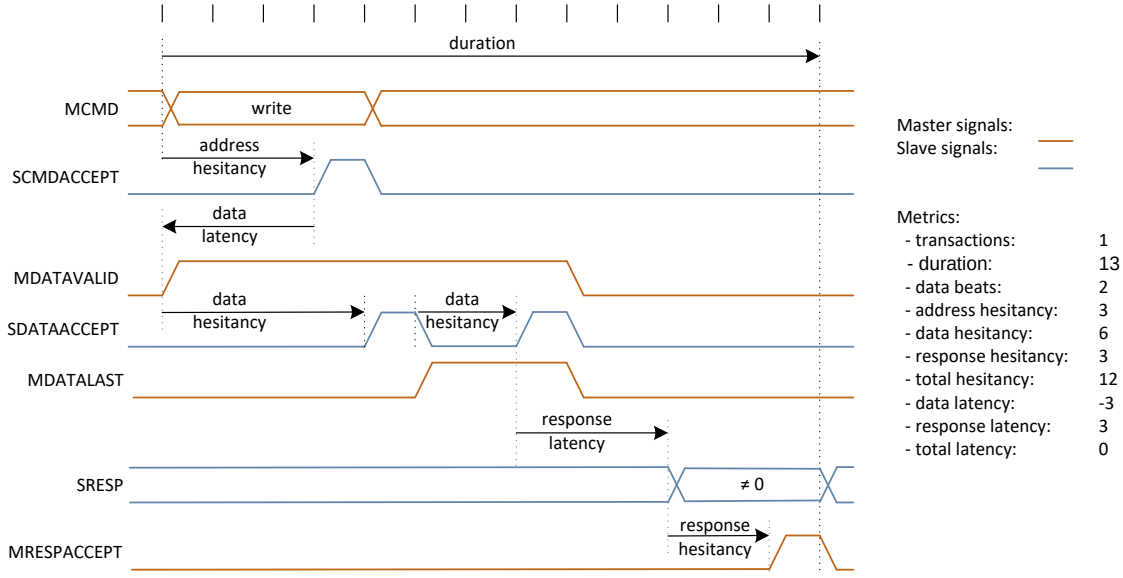

Figure 3-5 OCP write transaction metrics

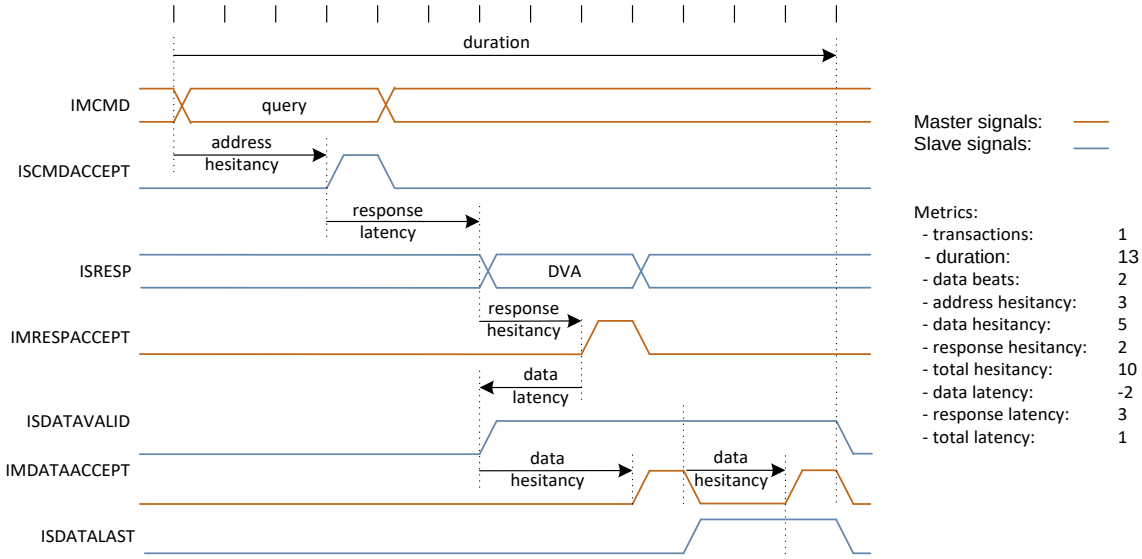

Figure 3-6 OCP3 split transaction query metrics

A note on data and response latency metrics: these are arranged so that the latency between master request accept and slave response is always measured as the response latency. Data latency records latency between two transaction phases from the same originator: address to write data from the master, or response to read data from the slave (where these can be separated).

Also note that the three latency metrics are reported as signed numbers.

As an alternative to accumulating a particular metric, some counters may be able to report the minimum or maximum value of the metric. For example, the largest or smallest individual transaction duration.

3.3.2.1 Individual transaction vs direct accumulation

Metrics can be gathered *directly*, or using *individual transaction accumulators* (see Section 3.1.1). The number of ITAs available is typically limited to one or two due to their high hardware cost. The number of metrics which can be simultaneously gathered using ITAs is limited to the number of ITAs that have been included. On the other hand, for direct accumulation, there is no limit (other than the number of counters). To use an ITA for a particular metric, set the required **ita_[metric]** bit using ***set_monitor***.

Using direct accumulation, the selected metric is accumulated directly into one of the counters as the transaction progresses, provided it matches the filtering criteria. For example, a counter configured to measure transaction duration will be incremented by 1 for each cycle that a matching transaction is in progress. Using individual transaction accumulation, the metric is first gathered into an ITA, and this is only added to a counter when the transaction completes provided it matches the filtering criteria.

Use an ITA if:

- Matching is not fully resolved until after metric counting has started. For example, measuring response latency for transactions that end with an error response. As can be seen from Figure 3-1 through Figure 3-6, the response latency occurs before the response is known, and the correct value will only be gathered if an ITA is used. Using direct accumulation, the counter will accumulate the response latency for all transactions, not just those that end with an error;
- You want to record the minimum or maximum value of a metric (but note an exception to this below).

If any data-phase matching is enabled, the three data-phase metrics (beats, bytes and data hesitancy) are gathered differently for the two accumulation methods:

- Using direct accumulation, metrics are gathered for only those data beats that match the filtering criteria;
- Using individual transaction accumulation, metrics are gathered for all beats in the transaction if any of the beats match the filtering criteria.

For duration and latency metrics, using min/max with direct accumulation generates alternative metrics, as shown in Table 3-4.

Table 3-4 Accumulation method and min/max

Min/Max Metric	Individual Transaction Accumulation	Direct Accumulation
Duration	Min/max individual duration	Min/max number of concurrently active transactions
Data, response or total latency	Min/max individual latency	Min/max number of simultaneously latent transactions

3.3.2.2 Outputting counter values

Counters typically accumulate over a period of time controlled by the interval timer, or alternatively the interval can be controlled directly by an internal trigger or external stimuli (for example cross-triggered from another module using a real-time event, or from external software using a ***monitor_snapshot*** message).

When requested, a snapshot of the counter values is taken and output. In each case, the **cause** field of the resultant ***monitor_counter_data*** indicates what caused the message to be produced:

- ***cause*** 00: the interval timer is selected to trigger the counters, and it has expired;
- ***cause*** 01: the interval timer is selected to trigger the counters, but it has been disabled whilst running;
- ***cause*** *10: at least one counter overflowed;*
- ***cause*** *11: output triggered by real-time event, internal trigger or* ***monitor_snapshot*** message.

When overflow occurs, no data is lost, as the value output is that prior to the overflow, and the amount that would have been added is carried forward to be output next time. In this way, the total count over a controlled measurement interval can be assembled in software by adding together any intermediate count values output with **cause** 10 to the count values output at the end of the interval.

Note: when the interval timer is selected as a trigger (by using ***set_monitor*** with **select** = 1 (*interface*) to set **counters_interval_trigger** to 1), the counters will not operate unless the interval timer is enabled.

Counters are reinitialized after being snapshot. They are not reinitialized when disabled (otherwise the final value would be lost). Nor are they reinitialized on a mode change (from counting to min or max mode, for example). When switching modes, the first ***monitor_counter_data*** message output after the switch will contain invalid count values and should be discarded. It is recommended that a ***monitor_snapshot*** message is sent after a mode switch to accomplish this.

For counters configured to report the min or max value observed, the timestamp is output (in gray coded format) in place of the 16 MSBs of the count value, provided the counter width as at least 17 bits wider than the individual transaction accumulators (i.e. ***discovery_response*** **counter_width_*X*** ≥ **ita_width_*X*** + 17). This allows the time at which the min or max value was recorded to be determined.

3.3.2.3 Count errors

Each counter has associated with it an **error** bit:

- For a metric gathered using an ITA, **error** will be set if the ITA saturated. In this case, the reported count will be smaller than it should be because the ITA wasn't able to record the full metric value for a transaction;
- If an ITA is specified for a metric when there are none available, **error** will be set. In this case, the reported count will be zero.
- For a metric gathered directly, **error** will be set if the match is not fully resolved until after accumulation may have started. This indicates that the selected match criteria are not suitable for use with direct accumulation, and an ITA should be used instead. The reported count may be higher than it should be;
- For a metric gathered directly, **error** will be set if the match criteria includes threshold matching. Threshold matching is not a suitable match criteria for use with direct accumulation, and an ITA should be used instead. As accumulation will only start once the threshold is reached, the reported count may be lower than it should be;

3.3.3 Data trace

The data trace capability allows data to be captured on a cycle-by-cycle basis. Use the Discovery Process (see Section 4.1) to determine the tracing capabilities available to you. The fields which affect trace behaviour are summarized in Table 3-5.

Table 3-5 Discovery fields that affect trace behaviour

***discovery_response* field**	**Description**
filter_[signal]_*X*	Each of these fields reports which of the various buses can be traced for interface *X*: Bit 2: traced using direct trace mode Bit 3: traced using deferred trace mode
trace_*X*	Reports whether trace is included for interface *X*: 0: no trace 1: trace without compression 2: trace with XOR compression
trace_addr_depth_*X*	Reports the depth of the address trace buffer associated with interface *X*.
trace_data_depth_*X*	Reports the depth of the data trace buffer associated with interface *X*.
trace_resp_depth_*X*	Reports the depth of the response trace buffer associated with interface *X*.
trace_ptime_width_*X*	The width of the **precise_time** trace field for interface *X*.
trace_shared_*X*	Reports whether read and write share trace buffers for interface *X*.

XOR compression is provided if **trace_*X*** in the ***discovery_response*** is 2. Rather than output the trace data, this outputs the XOR of the trace data with the previous trace data. This means only bits which change will be ones, which in conjunction with the standard message compression algorithm will tend to result in shorter ***monitor_trace_data*** messages, thereby reducing overall trace message bandwidth. See Section 3.3.3.4 for more details on how to decode the trace.

One or more filters are used to determine which data to capture, and independently one or more filters are used to determine when to trigger the output (see Section 3.3.3.3 for more details on trace modes). Additionally, output can be triggered asynchronously with respect to bus traffic on reception of an event or ***monitor_snapshot*** message, or when the interval timer expires.

Separate trace buffers are implemented for each bus channel when **trace_shared_*X*** in the ***discovery_response*** is 0, or for each bus phase (address, data, response) when it is 1, in which case read, write or snoop transactions can be traced at any one time, but not all simultaneously. This option reduces cost. Without sharing, an AXI variant will have 5 buffers (8 for the cache coherent variants which have snoop channels). With sharing there is a maximum of three buffers.

An *incomplete* mode is provided to aid with the diagnosis of bus lock-up. Normally, signals are captured as each bus phase completes (i.e. valid and ready both active). *Incomplete* mode allows signal capture for incomplete phases (i.e. valid active, but not ready), when the trigger condition is a ***monitor_snapshot*** message, an event or a filter exceeding a metric threshold (i.e. not a completing transaction phase). When the trigger is a transaction exceeding a metric threshold, the state of the triggering transaction is output via a ***monitor_trace_data*** message with a 'flag' **marker** for bus phases which have not yet started (see Section 3.3.3.4). This ensures the captured phases of the triggering transaction can be identified.

Individual enables are provided for each bus phase, so trace output is only generated for the bus phases of interest.

3.3.3.1 Triggering and filtering

One or more filters are used to determine when to trigger trace. Additionally, output can be triggered asynchronously with respect to bus traffic on reception of an event or ***monitor_snapshot*** message, or when the interval timer expires.

For all trace modes except streaming (see Section 3.3.3.3 for more details on trace modes), one or more filters can also optionally be used to determine which transactions to capture. These may be the same or different

to the filters used for triggering. When filtering is disabled, all transactions are captured. In streaming mode, the trigger condition alone determines which transactions are traced.

Note: When using deferred trace (see next section), one or more filters *must* be used to determine which transaction address phases to capture. Furthermore, if multiple filters are used, they must be configured to match on the same transaction phases. For example, you can use two filters to trace transactions to two different address ranges, but you can't use one filter that matches an address range, and another that matches transactions that have a particular response. This may result in incorrect data being output. Finally, the filtering condition for writes must include at least one address phase criteria if ***discovery_response*** **datanot1st** is 0.

3.3.3.2 Direct vs. deferred trace

Bus transactions are partitioned into address, data and response phases, which are spread out in time. Signals for each transaction phase are captured into separate buffers. Two options are provided for capturing address phase trace: *direct* and *deferred*. Data and response phases always use *direct* capture. Depending on parameterization, you may have only *direct* or *deferred*, or you may be able to select at run-time.

- *Direct* trace is available if bit 2 of any ***discovery_response*** fields with names beginning **filter_** is set;
- *Deferred* trace is available if bit 3 of any ***discovery_response*** fields with names beginning **filter_** is set.

Note: Any bus field for which neither of these bits is set cannot be traced.

Functionally, the two modes differ as follows:

- With *direct* capture, signals for a bus phase are captured into the buffer associated with that phase if the filtering condition matches *at that point in time.* For example, if a filter is configured to match on address and response, then during the address phase signals will be captured if the address matches, as the response isn't known at this time. This means that address phase signals will still be traced even if the transaction eventually fails the match criteria (because the response match fails);
- With *deferred* capture, signals for a bus phase are first captured into dedicated per-transaction storage. This is then only copied into the trace buffer when the filtering condition is fully resolved. So in the case of the above example, the address phase signals will not be traced for transactions which do not match the required response.

Whilst *deferred* capture offers better control of filtering when filtering is dependent on bus phases that haven't occurred yet, it adds a very significant extra hardware cost. On the other hand, the improved filtering control actually offers no advantage at all in many situations. The only situation in which *deferred* capture may be useful is where the Bus Monitor is attached at a point where the bus carries traffic from multiple sources (i.e. more than one ID is used), and also supports transaction re-ordering between different IDs. For example, at a bus master with multiple IDs, or at a sophisticated slave such as a memory management unit.

	1	2	3	4	5	6	7	8
Address	A_1	B_1	B_2	B_3		B_4		
Response			B_1 (ok)	B_2 (ok)	B_3 (ok)		B_4 (ok)	A_1 (err)

Address trace buffer (direct mode)								
	A_1	B1	B_2	B_3	B_3	B_4	B4	B_4
	-	A_1	B_1	B_2	B_2	B_3	B_3	B_3
	-	-	A_1	B_1	B_1	B_2	B_2	B_2
	-	-	-	A_1	A_1	B_1	B_1	B_1

Address trace buffer (deferred mode)								
	-	-	-	-	-	-	-	A_1
	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-

Response trace buffer								
	-	-	B_1	B_2	B_3	B_3	B_4	A_1
	-	-	-	B_1	B_2	B_2	B_3	B_4
	-	-	-	-	B_1	B_1	B_2	B_3
	-	-	-	-	-	-	B_1	B_2

Figure 3-7 Direct vs. deferred trace example

Figure 3-7 shows an example to illustrate the case where *deferred* capture may be advantageous. Suppose you want to trace the address of a transaction which completes with an error response. The bus has a concurrency of 4 (i.e. supports up to 4 transactions in progress at any one time). The trace buffers are 4 entries deep, and are configured for *to* mode (see next section). The first transaction uses ID A (noted A_1 in the diagram) and will ultimately complete with an error, but we obviously don't know that when it starts. After issuing an address for transaction A_1, three transactions follow on ID B (B_1, B_2 and B_3). These all complete without an error whilst transaction A_1 is still outstanding. As there's only 1 outstanding transaction in flight, it's possible to initiate another transaction. If this is also on ID B (transaction B_4), and it also completes before transaction A_1, the address trace buffer no longer contains address phase information for transaction A_1. As a result, when trace output is finally triggered by the completion of transaction A_1 in cycle 8, the address of that transaction cannot be reported. On the other hand, using *deferred* capture, loading address information into the trace buffer is deferred until the response match is known, and only A_1 is captured. Now when triggered in cycle 8, both address and response information for transaction A_1 are reported.

It can be seen that using *direct* capture would also have resulting in the address phase of A_1 being output, provided that A_1 completed before B_4 started. Alternatively, if you knew you were looking for an error in an ID A transaction, and filtered on ID, none of the B transactions would have been loaded into the address buffer, and A_1 address would still have been output when trace was triggered.

If transactions always complete in order, and the trace buffer is at least as deep as the bus concurrency, then *direct* trace will be sufficient.

3.3.3.3 Trace modes

There are 4 trace triggering modes:

- In *stream* mode, data is captured and output for each cycle the trigger matches, provided the buffer isn't full (there is no separate filter);
- In *from* mode, one entire buffer's worth of filtered data is captured from the cycle on which it is triggered;
- In *to* mode, up to one entire buffer's worth of filtered data is captured up to and including the cycle on which it is triggered;
- In *about* mode, up to one entire buffer's worth of filtered data is captured such that half a buffer's worth of data is captured before the cycle on which it is triggered, and half afterwards.

A *one-shot* mode can be used to limit the amount of trace:

- In *to*, *from* and *about* modes, trace is triggered on the first match only, so only one buffer's worth of data will be output, until the one-shot mechanism is re-primed;
- In *stream* mode, trace will continue to trigger until the trace buffer becomes full, after which a *status* flag message will be sent to indicate that tracing has stopped (see Section 3.3.3.4 for details on flag messages).

When not in *one-shot* mode, a ***message_lost*** message will be generated if the trace buffer can't be triggered:

- In *to*, *from* and *about* modes this will occur if a second trigger occurs whilst the buffer is still busy processing the previous trigger. The exception to this is if the condition that causes the second triggering is also captured as part of the data from the first trigger;
- In stream mode, this will occur if the buffer if full when the trigger occurs.

The choice of triggering mode depends in part on the trigger condition, and in particular when in a transaction it resolves. If the trigger condition is fully resolved at the beginning of the transaction (e.g. ID, address), or you are using *deferred* address capture, all modes can be used effectively. However, if the trigger condition doesn't resolve until later in the transaction (e.g. response, or a threshold metric such as the transaction duration) and you are using *direct* address capture, only *to* and *about* modes are useful; *stream* and *from* modes should be used with caution. The reason for this is as follows:

Using *direct* capture (see previous section), signals for a bus phase are captured into the buffer associated with that phase if the filtering condition matches *at that point in time.* For example, if a filter is configured to match on address and response, then during the address phase signals will be captured if the address matches. The

response isn't known at this time, and so this means that address phase signals may still be captured even if the transaction eventually fails the match criteria (because the response match fails). In contrast, triggering only occurs when the trigger condition is fully resolved. If this occurs during the response phase, then in *to* mode, the transaction which matched the trigger will be the last (newest) in the data output from the response phase buffer. The address for that transaction will be somewhere in the address phase buffer, provided the buffer is deep enough (typically greater than or equal to the maximum number of concurrent transactions supported by the bus). In contrast, if *from* mode were used, the address of the triggering transaction would not have been captured, as nothing is captured prior to the trigger.

Note that if write data can be accepted (valid and ready both true) before address, then filtering on address (or other address-phase only signals) will still require use of *to* or *about* modes to ensure the data phase signals are captured for those data beats which preceded the address.

The trigger condition will be met on a particular bus phase (which will depend on the match criteria). The buffer for that phase will be triggered immediately. The buffers for other phases will also be triggered immediately if that phase has completed. Buffers for bus phases which have not yet occurred will trigger when that phase happens for one of the transactions being traced. This is discussed in more detail in Section 3.3.3.5.

Some transactions do not include all bus phases. For example, OCP non-posted writes have address, data and response phases, but posted writes have only address and data phases. If the triggering transaction does not include a particular bus phase then the buffer for that phase will trigger on the next traced transaction which does include that phase. Furthermore, in *from* or *about* modes, the number of samples taken after triggering will take this into account. For example, suppose the trace buffer can hold 8 entries and is configured in *from* mode. Once triggered, the address buffer will output address phase information for 8 transactions. However, if only 3 of these had response phases, then the response buffer will only output information for those 3 transactions. If none of the traced transactions include the phase, then a *no data* flag message will be issued (see next section).

3.3.3.4 Markers

Trace is output via ***monitor_trace_data*** messages. These include a **marker** field to identify all messages associated with a particular trigger. In *stream* mode, the 'start', 'middle' and 'last' markers will indicate whether the bus phase captured is the first, middle or last phase of the transaction to occur. Typically, transactions will begin with an address phase, in which case all address trace messages will have a 'start' marker. However, in some configurations it is possible for write data to arrive before the address, in which case the first data trace message would have the 'start' marker and the associated address would have a 'middle' marker.

In the other triggered modes, the markers identify a group of trace messages associated with a particular trigger. In *to* and *about* modes, the amount of data prior to triggering is variable. Similarly, in *from* or *about* modes for bus phases which are not included in all transactions, the amount of data after triggering is variable. As such, the following ***monitor_trace_data*** marker sequences are all possible:

- Two or more cycles of filtered data: 'start', zero or more 'middle', then 'last';
- One cycle of filtered data: 'last';
- Zero cycles of filtered data: 'flag' message with 'no data' status.

monitor_trace_data messages with a 'flag' **marker** do not carry trace data. Instead, the payload is used to convey other information. There are three types of 'flag' message:

- *No data*. Indicates no filtered data captured as a result of the trigger (see Table 5-65);
- *Status*. Indicates the trace mode and enabled/disabled status, when they change (see Table 5-66). Any residual trace data in the buffer will be output before outputting this flag message. The first ***monitor_trace_data*** message received for any channel will always be a *status* flag message;
- *Transaction state*. Indicates which transaction phases have completed (see Table 5-67). Output when using *incomplete* mode, for bus phases which have yet to start for the transaction which triggered trace by exceeding the specified metric threshold.

If the trace mode or enable are changed, any residual trace data in the buffer will be output. Following this, an additional ***monitor_trace_data*** message with a 'flag' marker is issued with status to indicate the new trace mode and enabled/disabled state.

When parameterized to provide XOR compression, the **trace** field of ***monitor_trace_data*** messages will be encoded using XOR compression except for those carrying a flag marker. To recover the unencoded trace from all other ***monitor_trace_data*** messages, XOR with the trace field from the previous ***monitor_trace_data*** message for the same channel.

3.3.3.5 Trace transaction correlation

Because the order of transactions is not necessarily the same for all bus phases, the triggered modes do not guarantee that all transactions represented in one trace buffer will also be represented in the others. For example, suppose the trace buffers can each hold 4 entries, and are configured to trigger on an address phase match. Six read transactions A, B, C, D, E, F which all meet the filtering criteria occur (see Figure 3-8). The addresses are issued in that order, but the data is returned in the order A, C, D, E, F, B. Using *from* mode triggered by B, the address trace outputs B, C, D, E, but the data trace will output from whichever of the transactions B, C, D or E is returned first, and so will output C, D, E, F.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Address	A	B	C	D	E				F	
Tracker Index	0	1	2	3	0				0	
Data				A	C	D	E			F	B

Address trace buffer (direct mode)					
A(0)	B(1)	C(2)	D(3)	E(0)
-	A(0)	B(1)	C(2)	D(3)
-	-	A(0)	B(1)	C(2)
-	-	-	A(0)	B(1)

Data trace buffer										
-	-	-	-	C(2)	D(3)	E(0)	E(0)	E(0)	F(0)
-	-	-	-	-	C(2)	D(3)	D(3)	D(3)	E(0)
-	-	-	-	-	-	C(2)	C(2)	C(2)	D(3)
-	-	-	-	-	-	-	-	-	C(2)

Figure 3-8 Trace transaction correlation

So that the output from the different trace buffers can be correlated, the trace output includes a **tracker_index** field, which indicates which entry in the Bus Monitor's transaction tracking table is being used to track the transaction, and a **transaction_differentiator** bit. For *stream* and *from* modes, the combination of **tracker_index** and **transaction_differentiator** can never be the same for consecutive transactions, even if they are non-overlapping and use the same tracking table entry. For *to* and *about* modes, it is possible for a transaction with a 'last' marker and the following transaction with a 'first' marker to have the same **tracker_index** and **transaction_differentiator** , but these are differentiated by the marker.

In this case, the address trace outputs will carry indexes 1, 2, 3 and 0 respectively, whereas the data trace outputs will carry indexes 2, 3, 0, and 0. From this we can determine that the 2nd address output and the 1st data output are from the same transaction (C), and so on.

3.3.3.6 Trace timing correlation

As noted above, trace information for each distinct bus phase is output separately. The timestamp associated with each trace message indicates the UDB domain cycle in which the trace was captured. In most cases, this provides sufficient accuracy of the relative time between transaction phases. If you need to be able to determine the precise timing relationship between traced cycles in terms of bus clock domain cycles, you can use the **precise_time** field in the ***monitor_trace_data*** messages in conjunction with the timestamp (provided that **trace_ptime_width** in the ***discovery_response*** is not zero). The timestamp is relative to the Embedded Analytics system clock domain, which is typically asynchronous to the bus being traced, whereas the **precise_time** field is synchronous to the bus. Full details of how to combine these to follow.

3.4 System coordination

As well as being able to control what to monitor, and what actions to take, it's equally important to be able to control *when* actions are to be taken. This is controlled in a number of ways:

- There is a global **module_enable** bit for the entire Bus Monitor, which must be set in order for the Bus Monitor to perform any monitoring (via ***set_enabled***);
- Each filter, each counter and the trace unit all have individual enable bits;
- The interval timer can be used to determine when counting activity is to take place.

As described below, some of these capabilities can be controlled and observed in multiple different ways; where this is the case there is only one underlying hardware resource. For example, although there are three different ways to enable a filter, there is only one underlying enable bit.

3.4.1 Interval timer

The counters and trace can operate in conjunction with a programmable interval timer. Each time it expires (decrements to zero), the counters and/or trace will be output if they are configured to be triggered by the interval timer. The interval timer supports several modes of operation:

- In *continuous* mode, the interval timer is free running;
- In *windowed* mode, the interval timer is started upon reception of an event and runs until it expires, after which it will not restart until reception of another event;
- In *triggered* mode, the interval timer is started upon reception of an event, and thereafter runs continuously.

After the interval timer has been started in *windowed* or *triggered* mode, if the triggering event is received again before the timer expires, the timer will re-start immediately.

Note that in *triggered* or *windowed* modes, the interval timer is receptive to real-time events, but not internal triggers (see Section 3.3.1). However, it is receptive to real-time events generated by the filters, which achieves the same functionality.

3.4.2 Individual vs. collective filter control

The individual enables for each filter can be accessed individually or collectively:

- Each bit can be controlled individually via ***set_bus_filter***. This method groups the enable with other options associated with the filter, which is often convenient;
- Each bit can be set or cleared using external events or internal triggers. This method provides hardware level control from inside or outside the Bus Monitor;
- The filter enables can also be controlled collectively via ***set_enabled***. This allows multiple filters to be enabled or disabled simultaneously, which may be important when monitoring multiple activities simultaneously. The ***set_enabled*** message provides AND and OR operations to allow specified bits to be set or cleared without disturbing others.

3.4.3 Triggering from hardware

Tessent Embedded Analytics supports 256 real-time hardware events that can be used to control the Bus Monitor directly from hardware:

- Each filter can be enabled or disabled upon receipt of a user-specified event. Using the same event for enable and disable causes the filter to be enabled for 1 cycle only;
- In *windowed* or *triggered* mode, the interval timer is started upon receipt of a user-specified event.
- The time event (event 0x1) will reset the timer used to generate timestamps. This can be used to coordinate time across the wider system;
- Throttling (see below) is controlled via events: Throttle0 (0x7), Throttle1 (0x8) and Throttle_off (0x9).

3.4.4 Throttling behaviour

Throttling allows the amount of data produced by the Bus Monitor to be reduced (for example when other components in the system have a high priority bandwidth requirement). There are three throttling states:

- After reset, or after reception of event 0x9, the module is in the *unthrottled* state;
- After reception of event 0x7, the module is in the *throttle-0* state;
- After reception of event 0x8, the module is in the *throttle-1* state.

Throttling is achieved by comparing the throttle state of the module with the relevant **throttle_level** field:

- ***monitor_counter_data*** and ***monitor_match_data*** messages are throttled according to the level specified by the **counters_throttle_level** and **match_throttle_level** fields in the ***set_monitor*** message respectively;
- ***monitor_trace_data*** messages are throttled according to the **throttle_level** field via ***set_monitor_trace***.

The **throttle_level** fields support the following options:

- 00: Always throttled regardless of throttle state;
- 01: Never throttled regardless of throttle state;
- 10: Throttled in *throttle-1* state;
- 11: Throttled in *throttle-0* or *throttle-1* states.

When throttled:

- *Trace can be triggered once in to, from and about modes (see Section 3.3.3.3), but output via* ***monitor_trace_data*** messages *is deferred until throttling ends;*
- ***monitor_trace_data*** *messages are not generated if tracing in stream mode;*
- ***monitor_counter_data*** and ***monitor_match_data*** messages are not generated;

Further details on throttling and its application can be found in *An Introduction to the Tessent Embedded Analytics Architecture* (see Related reading).

3.4.5 Message flows

Flows are used to ensure downstream messages are routed to the required communicator, and to provide different priority levels. Further details can be found in *An Introduction to the Tessent Embedded Analytics Architecture* (see Related reading).

Downstream messages are issued on one of 4 flows:

- A system control response message is issued on the same flow as the upstream message that requested it;
- ***message_lost*** and ***event_lost*** messages are issued on the flow specified by the **sys_flow** field in the ***set_monitor*** message;
- ***monitor_match_data*** and ***monitor_counter_data*** messages are issued on the flow specified by the **match_flow** and **counter_flow** fields respectively in the ***set_monitor*** message;
- ***monitor_trace_data*** messages are issued on the flow specified by the **flow** field via ***set_monitor_trace***.

3.4.6 Timestamps

Timestamps can be optionally included in downstream messages. These provide useful information about when activities occur in relation to each other. If this information is not necessary, timestamps can be disabled, reducing the bandwidth required for downstream messages. Time stamps are enabled as follows:

- System control response messages, ***message_lost*** and ***event_lost*** messages have timestamps if **sys_timestamp** is set via the ***set_monitor*** message;
- ***monitor_match_data*** and ***monitor_counter_data*** messages have timestamps if **match_timestamp** and **counter_timestamp** respectively are set via ***set_monitor***;
- ***monitor_trace_data*** messages have timestamps if **timestamp** is set via ***set_monitor_trace***.

3.4.7 Lost messages and events

Under some circumstances the Bus Monitor may not be able to send events or messages:

- Loss of a ***monitor_trace_data*** message occurs if a transaction which should be traced cannot be because the trace buffer is full. This is typically a result of using *streaming* mode with a filter which matches a lot of transactions, or using the triggered modes with a trigger that matches more frequently than the time taken to empty the trace buffer;
- Loss of a ***monitor_match_data*** message occurs if the FIFO used to buffer these messages overflows. This may occur when a lot of matching transactions occur close together;
- Loss of a ***monitor_counter_data*** message occurs if one is scheduled when the previously requested one still hasn't been sent. This should not occur when using the interval timer unless the interval is unreasonably small, but could occur if using external events to trigger counter snapshots and the events occur close together;
- Loss of an event occurs if the FIFO used to buffer the events overflows. This may occur when a lot of matching transactions occur close together.

When one or more messages are lost, a ***message_lost*** message will be scheduled. This will indicate the source of the messages that have been lost, but not how many. There is a however a **high_loss** indication, which is set if the number of lost messages exceeds the number of successfully sent messages since the previous ***message_lost***.

When events are lost, an ***event_lost*** message will be scheduled.

3.4.8 Resource Allocation

The resource allocation mechanism provides a means for multiple debuggers (or other software APIs) to negotiate a claim to various Bus Monitor resources. Use the Discovery Process (see Section 4.1) to determine whether this capability is available to you. The fields which affect resource allocation behaviour are summarized in Table 3-6.

Table 3-6 Discovery fields that affect allocation behaviour

***discovery_response* field**	**Description**
alloc	Reports whether resource allocation is available (1) or not (0).

The approach taken is purely cooperative, relying on the various debuggers to claim capabilities before using them and to respect the claims of others.

If a debugger wishes to claim a resource it must negotiate its use by sending a ***manage_alloc*** message with **manage_op** set to *allocate*. The Bus Monitor will then grant the request, or decline it if the resource is already allocated, via ***alloc_response***.

Further details on resource allocation and its application can be found in *An Introduction to the Tessent Embedded Analytics Architecture* (see Related reading).

3.4.9 General purpose output

The general purpose output mechanism provides a means for directly controlling aspects of the target application from the Bus Monitor. Use the Discovery Process (see Section 4.1) to determine whether this capability is available to you. The fields which affect general purpose output behaviour are summarized in Table 3-7.

Table 3-7 Discovery fields that affect GPIO behaviour

discovery_response field	Description
gpio	Reports whether general purpose output is available (1) or not (0).
gpio_out_width	Number of general purpose outputs.

The `gpio_output` general purpose outputs are controlled using ***set_gpio***. Uses for these outputs are application specific.

4 Using the Bus Monitor

4.1 Discovery

The discovery mechanism is supported by all Embedded Analytics components, and provides a mechanism to uniquely identify the component being addressed, and details all of the capabilities that can be chosen via Verilog parameters upon instantiation. Sending a ***discovery_request*** message will result in a ***discovery_response*** message being returned.

The first 49 bits of the response (up to and including the vendor field) are standard for all components; fields after this are component specific. See Section 5.2.5 for details of the ***discovery_response*** message format.

It is important to use this mechanism to determine which of the capabilities described in this document are available to you.

4.2 Selecting triggers

A standard method is used throughout the Bus Monitor to configure triggering from either internal or external sources. This is used for enabling or disabling of filters, trace filtering and triggering, etc. In all cases it uses a pair of fields: **trigger** and **event_control** (the actual field names differ according to the message they belong to but are all variations of these names):

- When **event_control** is 0, **trigger** contains a bit-vector, wherein bit *N* corresponds to filter *N*. This allows one or more filters to be used as the trigger. For example, if **trigger** is set to 0x5 then the trigger will be active if either filter 0 or filter 2 match).
- When **event_control** is 1, trigger contains the number of an Embedded Analytics real-time event to use as the trigger.

When a filter is configured to be enabled and disabled by the same trigger condition, the filter will be enabled for 1 cycle only upon receipt of this trigger.

4.3 Flow and timestamp configuration

Before enabling any filters, counters or trace, the flow to use must be configured (see Section 3.4.5). Timestamps may be configured at the same time (see Section 3.4.6).

- Use ***set_monitor*** with **select** = 0 (*global*) to set up flow and timestamp settings for ***message_lost*** and ***event_lost***;
- Use ***set_monitor*** with **select** = 1 (*interface*) to set up flow and timestamp settings for ***monitor_match_data*** and ***monitor_counter_data*** data messages (see also Sections 4.5 through 4.7, which contain detail of how to set other fields in this message).
- Flow and timestamp settings for trace are configured via ***set_monitor_trace***; see Section 4.8.

4.4 Filtering

Relevant upstream messages		Relevant downstream messages
set_bus_filter	*get_bus_filter*	*bus_filter_response*
set_bus_match_ctrl	*get_bus_match_ctrl*	*bus_match_ctrl_response*
set_bus_match_data	*get_bus_match_data*	*bus_match_data_response*

To configure a filter, use the following procedure:

Use the ***set_bus_match_ctrl*** message with the following contents:

- Set the **interface** field to the interface you wish to configure;
- Set the **filter** field to the number of the filter you wish to access;
- Configure matching options as discussed in detail in Section 4.4.1;
- If this is a group 1 filter and you wish to cascade it with a group 0 filter, set **match_enable_group0_filter** to 1 and **match_value_group0_filter** to the number of the group 0 filter you wish to cascade with (see Section 3.2.1).

Then, use the ***set_bus_match_data*** message with the following contents:

- Set the **interface** field to the interface you wish to configure;
- Set the **filter** field to the number of the filter you wish to access;
- Set the **select** field to the required value;
- Configure matching options as discussed in detail in Section 4.4.1;
- Repeat for all **select** values.

Finally, use the ***set_bus_filter*** message with the following contents:

- Set the **interface** field to the interface you wish to configure;
- Set the **filter** field to the number of the filter you wish to access;
- Set **threshold_mode** and **count_threshold** as discussed in Section 4.4.1.3;
- To enable the filter, the **filter_enable** bit must be set to 1. Whether you set this at this time or later will depend on how you intend to use the filter; see Sections 3.4.2 and 3.4.3;
- To enable using an external event or internal trigger, set **event_enable_control** and **enable_trigger** as described in Section 4.2;
- To disable using an external event or internal trigger, set **event_disable_control** and **disable_trigger** as described in Section 4.2;
- The settings for other fields will depend on what actions are to be triggered by this filter and are addressed in Section 4.6.

4.4.1 Filter match configuration

Filter match configuration is split across two messages: ***set_bus_match_ctrl*** and ***set_bus_match_data***.

The **filter_*** ***discovery_response*** fields indicate which transaction fields can be filtered (bit 0 for group 0 filters, bit 1 for group 1 filters). Attempts to set message fields for unimplemented transaction filters will be ignored, and gets will return zero.

All enabled filtering conditions must be met in order for the filter to match. Most filter conditions have explicit enables (for example: when **match_enable_burst** is 1, a transaction must have the burst type specified by **match_value_burst** in order to match, whereas when **match_enable_burst** is 0, the burst type is ignored when determining whether a transaction matches). However, there are a few conditions that work differently:

- **match_transactions** and **match_value_resp** are multiple-choice vectors allowing any combination of the transaction type and response. They should be set to a non-zero value. *If you want to match any transaction type or any response, then set all bits in the respective field to 1*. **Note**: do not set either of these fields to zero as this will prevent any transactions from matching. Additional response matching considerations are discussed in Section 4.4.1.2.
- Where are range of values can be matched (e.g. **match_low_id** and **match_high_id**, the match can be either inside the range (between low and high), or outside.
- AXI user signal and OCP info signal matching (configured via ***set_bus_match_data***) all have masking so that you can control which specific bits of these buses are of interest. For example for the AXI `awuser` bus, the match function is

  match_value_awuser AND **match_mask_awuser** == `awuser` AND **match_mask_awuser**

  To disable matching on these fields, the respective mask value must be set to 0.

The following sections provide more detailed discussion of data, response, burst and threshold matching.

4.4.1.1 Data matching

Data matching is enabled by the **match_enable_data** bit in the ***set_bus_match_ctrl*** message, and the data value to match against is configured via ***set_bus_match_data*** with **select** = 16 (*data_0_127*)[1]. In addition, **match_mask_bytes** must be applied via ***set_bus_match_data*** with **select** = 1 (*data_byte_mask*):

- To match the entire data bus, set **match_mask_bytes** to all 1s;
- To match a value on (for example) just bits 15:0 of the data bus, set the 2 LSBs of **match_mask_bytes** to 1, and other bits to 0.

Furthermore, it's possible to look for a specific value of less than the full bus width in multiple positions on the bus. For example, suppose the data bus is 128 bits wide, you can look for a 32-bit value occurring in any of the four 32-bit aligned bus positions by

- Setting **match_size_data** in ***set_bus_match_ctrl*** to 010 (*32*);
- Replicating the desired data value in all four 32-bit words of *data_0_127*;
- Setting **match_mask_bytes** to all 1s.

As a variation on this, if you only cared about the value of the lowest byte of each 32-bit transaction, modify the above example to set only bits 0, 4, 8 and 12 of **match_mask_bytes** to 1.

The data value doesn't have to be replicated. You could also look for different values in each of the 4 positions. In this case a match would occur if at least one of the specified values occurred in its specified position.

Note that for AXI, the transfer size is part of the transaction (for OCP it's implicitly the same as the bus width). As such, if you wanted to apply the above examples to transactions of size 32-bits, you must also use **match_enable_size** and **match_value_size**. **match_size_data** is only used to determine the way the data bus matching is partitioned.

Note: for ACE-Lite and ACE, the read channel transactions MakeInvalid, CleanShared, CleanDirty, CleanUnique, DVMComplete, DVMMessage and Barrier do not transfer and data, and the value of the data during the transaction is not defined. To avoid unpredictable transaction matching when data matching is enabled, the bits of **match_transactions** in ***set_bus_match_ctrl*** corresponding to these transactions should be set to 0.

[1] If the bus is wider than 128 bits, use **select** = 17 (*data_128_255*), **select** = 18 (*data_256_383*), **select** = 19 (*data_384_511*), **select** = 20 (*data_512_639*), **select** = 21 (*data_640_767*), **select** = 22 (*data_768_895*) or **select** = 23 (*data_896_1023*) as required to configure the upper data bits.

4.4.1.2 Response matching

For reads, which have a response with every data beat, there are two forms of response matching, controlled by **match_resp_all** in ***set_bus_match_ctrl***.

- When **match_resp_all** is 0, a transaction is considered to match if any of the responses match;
- When **match_resp_all** is 1, the response for every beat must match in order for the transaction as a whole to be considered to match.

4.4.1.3 Burst transaction matching

If any of the beats within a burst transaction match, then the transaction as a whole is considered to match. More precise data/response matching of a specific beat or range of beats can be achieved using **match_enable_beat**, **match_low_beat** and **match_high_beat** (beats are numbered starting from zero).

When enabled by **match_enable_addr**, lower and upper bound addresses for all the beats within the burst are calculated from the starting address, burst length and address incrementing behaviour for the specified burst type. If inclusive matching is selected, then if any address between the lower and upper bounds is within the range specified by **match_low_addr** and **match_high_addr** then the transaction address will be considered to match. Exclusive matching inverts the match condition. The beat matching capabilities discussed in the previous paragraph cannot be used to make the address matching more precise.

4.4.1.4 Threshold matching

The filter can be configured to match only if one of the transaction metrics exceeds a threshold (for example, if the transaction duration is longer than a specified amount). To enable this, set **threshold_mode** to the metric you require (as outlined in Section 4.4), and **count_threshold** to the number of cycles at or above which you want to match. In addition the selected metric *must be configured to use an individual transaction accumulator* by setting the corresponding **ita_*** bit in the per interface ***set_monitor*** message (see Sections 3.3.2.1 and 4.5).

Note: latency metrics are signed quantities, and accordingly **count_threshold** is interpreted as signed when using one of these metrics.

The range and precision available for threshold matching is determined by the width of the ITAs, and the width of the **count_threshold** field (see **filter_thresh_width_*X*** and **ita_width_*X*** ***discovery_response*** fields). The range is determined by the ITA width. So for example, if this is 14 bits, the longest duration that can be recorded in an ITA is $2^{14} - 1$. If the width of **count_threshold** is less than the width of the ITAs, it is compared against the upper bits of the ITAs, reducing the precision. For example, if this is 8 bits, the 6 LSBs of the ITA are ignored, so the threshold can only be specified to a precision of 64 cycles. If the width of **count_threshold** is zero, the threshold is implicitly $2^{\text{ita\_width\_}X}$.

If threshold matching is not required, set **count_threshold** to 1 and **threshold_mode** to 1110 (*match_last*) or 1111 (*match_first*) - it doesn't matter which unless you want to issue a thresholded trigger; see Section 4.6.

4.5 Accumulation method

Relevant upstream messages		Relevant downstream messages
set_monitor	***get_monitor***	***monitor_response***

As discussed in Section 3.3.2.1, it's necessary to specify the accumulation method to use for each of the metrics of interest. Set the following fields in the ***set_monitor*** message to do this:

- Set **select** to 1 and **interface** to the interface you wish to configure;
- Set to 1 the **ita_*** bits corresponding to the metrics you wish to use an individual transaction accumulator for;
- See Sections 4.6 and 4.6.1 for details of other fields in this message.

The maximum number of **ita_*** bits set to 1 should not exceed the number of ITAs available (as reported via the **itas** *discovery_response* field). Metrics are associated with ITAs in reverse order of the bits in the message (bytes, beats … duration). Once all ITAs have been allocated, any other metrics that have their **ita_*** bit set cannot be counted. Any counter configured to count one of these metrics will remain at zero and its associated **error** bit in the *monitor_counter_data* message will be one.

4.6 Issuing match events, messages and internal triggers

Relevant upstream messages		Relevant downstream messages
set_bus_filter	*get_bus_filter*	*bus_filter_response*
set_monitor	*get_monitor*	*monitor_response*
		monitor_match_data

To issue a trigger (an event, ***monitor_match_data*** match message or internal trigger), set the following fields in the ***set_bus_filter*** message in addition to those outlined in Section 4.2:

- Set bits in the **issue_mode** field to enable issue of an event (bit 0), a message (bit 1) in any combination, as required (internal triggers are always enabled);
- If issuing events, set the **event** field to event number you wish to issue;
- If threshold matching is not required (see Section 4.4.1.3), the threshold counter can be used to limit the number of triggers issued. To do this, set the **count_threshold** field to the number of times the filter must match between issuing triggers, and set **threshold_mode** to *match_last* or *match_first* as discussed in Section 4.4.1.4.

If issuing messages, the following fields in the ***set_monitor*** message must have been set for this interface beforehand:

- Set **select** to 1 and **interface** to the interface you wish to configure;
- Set the **match_flow** field to the flow you wish to issue the message on;
- Set the **match_timestamp** bit to 1 if you want to include a timestamp with the message, or 0 otherwise;
- Set **match_throttle_level** according to the throttling behaviour required (see Section 3.4.4). **Note**: no ***monitor_match_data*** messages will be issued if this field is 00 (*always*);
- See Sections 4.5 and 4.6.1 for details of other fields in this message.

The trigger will be issued at the earliest point in a transaction that the match criteria are fully resolved. For example, if matching on ID only the trigger will be issued when the transaction starts, whereas if matching on a write with a particular response, the trigger will not be issued until the response is received (typically just before the end of the transaction).

4.6.1 Example: Issue an event when an address range is written within successfully

Firstly use *set_enabled* to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using *set_bus_match_ctrl* for the filter of your choice:

- Set **match_transactions** to 0x1 to enable filtering on writes;
- Set **match_enable_addr** to 0x2 (*inclusive*) to enable address matching within the range;
- Set **match_value_resp** to 0x3 to match OKAY or EXOKAY responses;
- Set all other fields to zero.

Using ***set_bus_match_data*** for the filter of your choice:

- With **select** = 0 (*address*), set **match_low_addr** and **match_high_addr** to the lower and upper address range bounds you wish to detect;
- Set fields for all legal values of **select** > 3 to zero (disables user/info matching and matching on cache coherent extensions if available).

Using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to *match_last* or *match_first* to disable threshold matching;
- Set **count_threshold** to 0x1 to issue on 1st match;
- Set **issue_mode** to 0x1 to select event issue;
- Set **event** to the event number you wish to issue;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Variations:

- Issue the event once every 10th occurrence of the write, starting with the first one by setting **count_threshold** to 0xA and **threshold_mode** to 0xF (*match_first*);
- Adjust the filtering criteria to match other conditions as required.

4.6.2 Example: Issue a match message when read data latency exceeds a threshold

Firstly use ***set_enabled*** to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using ***set_monitor*** with **select** = 1 (*interface*), configure the match message behaviour:

- Set **match_timestamp** if you want a timestamp added to the ***monitor_match_data*** message;
- Set **match_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **match_throttle_level** to a non-zero value;
- Set **ita_data_latency** to 1 to select an ITA for data latency;
- The values of the remaining fields are not important for this example.

Using ***set_bus_match_ctrl*** for the filter of your choice:

- Set **match_transactions** to 0x2 to enable filtering on reads;
- Set **match_value_resp** to all 1s so all responses will match;
- Set all other fields to zero.

Using ***set_bus_match_data*** for the filter of your choice:

- Set fields for all legal values of **select** > 3 to zero (disables user/info matching and matching on cache coherent extensions if available).

Finally, using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0x1 (*data_latency*) to enable data latency threshold matching;
- Set **count_threshold** to the latency threshold required;
- Set **issue_mode** to 0x2 to select message issue;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Variations:

- Adjust **threshold_mode** to select other metrics as required;
- Adjust the filtering criteria to match other conditions as required.

4.6.3 Example: Output counters when a value is written to a mailbox

Firstly use ***set_enabled*** to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using ***set_monitor*** with **select** = 1 (*interface*), configure the ***monitor_counter_data*** triggering:

- Set **counters_timestamp** if you want a timestamp added to the ***monitor_counter_data*** message;
- Set **counters_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **counters_throttle_level** to a non-zero value;
- Set **counters_event_control** to 0 to select internal triggering from filters;
- Set **counters_trigger** so that there is a 1 in the bit corresponding to the filter you are using;
- Set **counters_interval_trigger** to 0 to disable triggering from the interval timer;
- The values of the remaining fields are not important for this example.

Using ***set_bus_match_ctrl*** for the filter of your choice:

- Set **match_transactions** to 0x1 to enable filtering on writes;
- Set **match_enable_addr** to 0x2 (*inclusive*) to enable address matching within the range;
- Set **match_enable_data** to 1 to enable data matching;
- Set **match_value_resp** to all 0x3 so only non-error responses will match;
- Set all other fields to zero.

Using ***set_bus_match_data*** for the filter of your choice:

- With **select** = 0 (*address*), set **match_low_addr** and **match_high_addr** to the mailbox address;
- With **select** = 16 (*data_0_127*), set **data_0_127**[1] to the required data value to match;
- With **select** = 1 (data_byte_*mask*), set **match_mask_bytes** to the bytes to compare (all 1's for the whole bus);
- Set fields for all legal values of **select** > 3 to zero (disables user/info matching and matching on cache coherent extensions if available).

Finally, using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 0x1 to issue on 1st match;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Variations:

- Adjust **threshold_mode** to select other metrics as required;
- Adjust the filtering criteria to match other conditions as required.

4.7 Counting

Relevant upstream messages		Relevant downstream messages
set_monitor	*get_monitor*	*monitor_response*
set_monitor_counter	*get_monitor_counter*	*monitor_counter_response*
set_period	*get_period*	*period_response*
		monitor_counter_data

To count metrics for transactions matched by a filter, first configure the filter you wish to use as outlined in Section 4.4, then configure the accumulation method as outlined in Section 4.5, then configure the counter using the ***set_monitor_counter*** message, with the following contents:

- Set the **interface** and **counter** fields to the interface and counter you wish to access;
- Set the **filter** field to the filter you wish to use to match transactions;
- Set the **metric** field to the metric you want to count;
- Set the **counter_mode** field according to the function required (minimum and maximum modes may not be available on all counters; see Section 3.3.2).

The counter contents will ultimately be output via the ***monitor_counter_data*** message. Configure the properties of this message using the ***set_monitor*** message, with the following contents:

- Set **select** to 1 (*interface*) and **interface** to the interface you wish to configure;
- Set the **counters_flow** field to the flow you wish to issue the message on;
- Set the **counters_timestamp** bit to 1 if you want to include a timestamp with the message, or 0 otherwise;
- Set **counters_throttle_level** according to the throttling behaviour required (see Section 3.4.4). **Note**: no ***monitor_match_data*** messages will be issued if this field is 00 (*always*);
- Set the **counters_interval_trigger** bit to 1 if you want to use the interval timer to control when to count, and to initiate ***monitor_counter_data*** messages, or 0 otherwise;
- If not using the interval timer, set the **counters_trigger** and **counters_event_control** fields to the trigger you wish to use initiate ***monitor_counter_data*** messages as described in Section 4.2.

Finally, if you plan to use the interval timer, configure it using the ***set_period*** message, with the following contents:

- Set the **timer_interval** field to the length of interval you wish to use;
- Set the **timer_enable** field to 01 (*continuous*), 10 (*windowed*) or 11 (*triggered*); see Section 3.4.1 for further details;
- If using *windowed* or *triggered* modes, set the **timer_index_trigger** field to the event number you wish to use to start the timer.

4.7.1 Example: Determine bus efficiency

This example uses one filter and two counters. Efficiency is defined here as the percentage of cycles in which data is actually transferred, and is computed by dividing the total number of data transfers (beats) by the total number of bus cycles.

Firstly use *set_enabled* to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using *set_monitor* with **select** = 1 (*interface*), configure the ***monitor_counter_data*** and accumulation behaviour:

- Set **counters_timestamp** if you want a timestamp added to the ***monitor_counter_data*** message;
- Set **counters_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **counters_throttle_level** to a non-zero value;
- Set **counters_interval_trigger** to 1 to enable interval timer triggering;
- Individual transaction accumulation is not required so **ita_beats** can be to 0 (though 1 is also okay if the ITA is not required for something else);
- The values of the remaining fields are not important for this example.

Use ***set_monitor_counter*** to configure counter A to count the number of read beats:

- Set **filter** to the filter you wish to use;
- Set **metric** to 0x8 to count beats;
- Set **counter_mode** to 0;
- Set **counter_enable** to 1 to enable the counter.

Use ***set_monitor_counter*** to configure counter B to count the number of bus cycles:

- Set **filter** to the filter you wish to use;
- Set **metric** to 0xB (*bus_cycles*) to count bus cycles;
- Set **counter_mode** to 0;
- Set **counter_enable** to 1 to enable the counter.

Using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 1;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Finally, using ***set_period***, configure the interval timer:

- Set **timer_interval** to determine the time between issuing ***monitor_counter_data*** messages;
- Set **timer_enable** to 0x1 to enable the timer;
- Set other fields to zero.

To compute bus efficiency for each time interval, extract the counter A and counter B values from each ***monitor_counter_data*** message received, and then divide counter A by counter B.

Variations:

- For OCP, set **match_transactions** to 0xE to match all read-type transactions (Read/ReadEx/RealLinked)
- Adjust the filtering criteria to match other conditions as required. This will also require **ita_beats** to be 1 if the filtering criteria is not fully known at the start of the transaction;
- Change the counter metric (see Section 3.3.2 for other examples);
- Measure over a fixed interval triggered from elsewhere in the system by using ***set_period*** to set **timer_enable** to 0x2 (*windowed* mode) and **timer_index_trigger** to the real-time event you want to use to start the interval timer.

4.7.2 Example: Determine average burst length between two events generated elsewhere in the system

This example uses one filter and two counters to collect the metrics necessary to determine the average burst length between two real-time events. Dividing the number of data beats by the number of transactions gives the average burst length.

When using broadcast events it is unnecessary to configure any message engines in order for an event to be routed from one module to another. If there will be frequent event communication between two modules connected to the same message engine it is advisable to use scope-limited events to lower the amount of global event 'chatter'. When scope-limited events are used it is necessary to configure the event masking in the message engines. Further details can be found in *An introduction to the Tessent Embedded Analytics Architecture* and *Message Infrastructure User Guide* (see Related reading).

The event which marks the start of the measurement interval will enable the designated filter, which will in turn cause metric accumulation to commence. The event which marks the end of the measurement interval will disable the designated filter (stopping metric accumulation), and also trigger the issuing of a ***monitor_counter_data*** message to report the gathered counter values.

Firstly use ***set_enabled*** to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using ***set_monitor*** with **select** = 1 (*interface*), configure the ***monitor_counter_data*** triggering:

- Set **counters_timestamp** if you want a timestamp added to the ***monitor_counter_data*** message;
- Set **counters_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **counters_throttle_level** to a non-zero value;
- Set **counters_event_control** to 1 to select external triggering from events;
- Set **counters_trigger** to the event number that marks the end of the measurement interval;
- Set **counters_interval_trigger** to 0 to disable triggering from the interval timer;
- Individual transaction accumulation is not required so **ita_beats** and can be to 0 (though 1 is also okay if the ITA is not required for something else);
- The values of the remaining fields are not important for this example.

Use ***set_bus_match_ctrl*** and ***set_bus_match_data*** as outlined in Section 4.6.2 to match all reads.

Use ***set_monitor_counter*** to configure counter A to count the number of beats as described in the previous example.

Use ***set_monitor_counter*** to configure counter B to count transactions:

- Set **filter** to the filter you wish to use;
- Set **metric** to 0xA (*transactions*) to count bus transactions;
- Set **counter_mode** to 0;
- Set **counter_enable** to 1 to enable the counter.

Finally, using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 1;
- Set **filter_enable** to 0 to disable the filter;
- Set **event_enable_control** to 1 to select enabling from external events;
- Set **enable_trigger** to the event number that marks the start of the measurement interval;
- Set **event_disable_control** to 1 to select disabling from external events;
- Set **disable_trigger** to the event number that marks the end of the measurement interval;
- Set all other fields to zero.

To compute average burst length between the events, extract the counter A and counter B values from all ***monitor_counter_data*** messages received, sum all counter A values, sum all counter B values and then divide sum(counter B) by sum(counter A).

Variations:

- Adjust **match_transactions** to gather metrics for other transaction types (or set to all 1's for all transaction types).
- Adjust the filtering criteria to match other conditions as required. This will also require **ita_beats** to be 1 if the filtering criteria is not fully known at the start of the transaction;

4.7.3 Example: Generate bandwidth utilization histogram

This example uses four filters and four counters to collect the metrics necessary to determine the percentage of bus bandwidth utilized by each of four bus masters.

Firstly use ***set_enabled*** to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using ***set_monitor*** with **select** = 1 (*interface*), configure the ***monitor_counter_data*** and accumulation behaviour:

- Set **counters_timestamp** if you want a timestamp added to the ***monitor_counter_data*** message;
- Set **counters_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **counters_throttle_level** to a non-zero value;
- Set **counters_interval_trigger** to 1 to enable interval timer triggering;
- Individual transaction accumulation is not required so **ita_beats** can be to 0 (though 1 is also okay if the ITA is not required for something else);
- The values of the remaining fields are not important for this example.

For each of the four filters, repeat using ***set_bus_filter***:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 1;
- Set **filter_enable** to 0 to disable the filter;
- Set all other fields to zero.

For each of the filters 0 to 3, repeat using ***set_bus_match_ctrl***:

- Set **match_transactions** to all 1's to match all transactions;
- Set **match_value_resp** to all 1s so all responses will match;
- Set **match_enable_id** to 0x2 (*inclusive*) to enable ID matching within the range;
- Set **match_low_id** and **match_high_id** to the ID of one of the bus masters (different ID for each filter);
- Set all other fields to zero.

For each of the counters 0 to 3, use ***set_monitor_counter***:

- Set **filter** to 0 for counter 0, 1 for counter 1, and so on;
- Set **metric** to 0 (*duration*) to count duration;
- Set **counter_mode** to 0 (*count*);
- Set **counter_enable** to 1 to enable the counter.

Use ***set_period*** to configure the interval timer:

- Set **timer_interval** to determine the time between issuing ***monitor_counter_data*** messages;
- Set **timer_enable** to 0x1 (*continuous*) to enable the timer;
- Set other fields to zero.

Finally use ***set_enabled*** to enable the filters together:

- Set **operation** to 01 (*set*);
- Set **filter_enables** to 0xf to enable filters 0 – 3.

The counters will report the total duration of transactions for each bus master, from which a histogram can be graphed showing the proportion of bandwidth taken by each bus master.

Variations:

- Adjust **match_transactions** to gather metrics for just reads, or just writes, etc.
- Change **metric** to transactions to produce a histogram based on number of transactions rather than transaction duration;

4.7.4 Example: Measure clock gating efficiency

This example uses one filter and one counter. Efficiency is defined here as the percentage of cycles where the clock is gated off. The only prerequisite is that the frequency relationship between the bus clock and the UltraDebug (UDB) clock is fixed and known.

Firstly, use ***set_monitor*** with **select** = 1 (*interface*), to configure the ***monitor_counter_data*** behavior:

- Set **counters_timestamp** if you want a timestamp added to the ***monitor_counter_data*** message;
- Set **counters_flow** to the flow you wish to use for ***monitor_counter_data*** messages;
- Set **counters_throttle_level** to a non-zero value;
- Set **counters_interval_trigger** to 1 to enable interval timer triggering;
- The values of the remaining fields are not important for this example.

Use ***set_monitor_counter*** to configure a counter to count the number of bus cycles:

- Set **filter** to the filter you wish to use;
- Set **metric** to 0xC (*bus_cycles*) to count bus cycles;
- Set **counter_mode** to 0;
- Set **counter_enable** to 1 to enable the counter.

Using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 1;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Finally, using ***set_period***, configure the interval timer:

- Set **timer_interval** to determine the time between issuing ***monitor_counter_data*** messages;
- Set **timer_enable** to 0x1 to enable the timer;
- Set other fields to zero.

Finally, use ***set_enabled*** to enable the module:

- Set **operation** to 01 (*set*);
- Set **module_enable** to 1.

Counter values reported will be output every **timer_interval** UDB cycles. If the bus:UDB clock frequency ratio is *N*:1, then:

- If the bus clock is running continuously, the counter value will be *N* × **timer_interval**;
- If the bus clock is running X percent of the time, the counter value will be *N* × **timer_interval** × *X*/100;
- **Note**: if the bus clock is off permanently, the counter value will be 0.

Variations:

- Measure over a fixed interval triggered from elsewhere in the system by using ***set_period*** to set **timer_enable** to 0x2 (*windowed* mode) and **timer_index_trigger** to the real-time event you want to use to start the interval timer.

4.8 Data Trace

Relevant upstream messages		Relevant downstream messages
set_monitor_trace	***get_monitor_trace***	***monitor_trace_response***
		monitor_trace_data

To trace data when a qualifier is activated, first configure the filter(s) you wish to use as outlined in Section 4.4, then configure the trace unit using the ***set_monitor_trace*** message, with the following contents:

- Set the **trigger** and **event_control** fields to the trigger you wish to use initiate trace as described in Section 4.2;
- Set **interval_trigger** to 1 if you want to trigger trace from the interval timer;
- Set the **history_mode** field according to the trace mode you wish to use: 00 (*streaming*), 01 (*to*), 10 (*from*) or 11 (*about*);
- If you want to filter what is traced, set **filter_enable** to 1, and set the **filter** field to select the filter(s) you wish to filter with (data will only be traced when one or more of these filters is active). These fields are ignored for streaming mode, which captures trace data when the trigger is active only;
- Set **oneshot** to 1 if you only want to trigger trace the first time the trigger matches;
- Set the **flow** field to the flow you wish to issue the ***monitor_trace_data*** messages on;
- Set the **timestamp** bit to 1 if you want to include a timestamp with ***monitor_trace_data*** messages, or 0 otherwise;
- Set **throttle_level** according to the throttling behaviour required (see Section 3.4.4). **Note**: no ***monitor_trace_data*** messages will be issued if this field is 00;
- Set **trace_addr** to 01 to trace address using *direct* mode, or 11 to use *deferred* mode (see Section 3.3.3.2);
- Set **trace_data** and **trace_resp** to 1 if you want to trace data and response phases respectively;
- Set **trace_read** to 1 if you are tracing reads (necessary if using shared trace, and has no effect if not);
- Set the **trace_enable** bit to 1 to enable trace.

Data trace is configured via the ***set_monitor_trace*** message. The **trigger** field specifies which filters to use to trigger trace to be output. Trace will be triggered when the selected filter's match criteria (as specified in Section 4.4) is met. The **history_mode** field specifies whether the trigger point is at the beginning, end or centre of the trace window, or whether trace is captured as a stream. Except for *streaming* mode, the **filter** field specifies which filters to use to determine which transactions will be captured into the trace buffers.

Trace is output via ***monitor_trace_data*** messages. Tracing of qualified transactions is performed using separate buffers for each transaction, and the **channel** field in the ***monitor_trace_data*** message indicates which of these buffers the message is from (see Table 5-50). Each message contains trace data from a single cycle, though where the number of signals being traced exceeds the maximum message size, this will be split into a tranche of messages.

When using *streaming* mode, a trace message is output for each enabled transaction phase for every transaction that matches the trigger condition.

When using *to*, *from* or *about* modes and trace is triggered, some number of ***monitor_trace_data*** messages will be output for each transaction phase that is enabled. The number of messages will be determined by the buffer depth (as specified by the **trace_addr_depth_*X***, **trace_data_depth_*X*** and **trace_response_depth_*X*** ***discovery_response*** fields). The first, middle and last messages associated with the trigger can be determined from the **marker** field values (see Section 3.3.3.4) and correlating messages from the different bus phases to specific transactions can be determined using the **tracker_index** field values (see Section 3.3.3.5).

4.8.1 Example: Trace all bus transactions from one ID around a specific transaction

This example uses two filters (A and B): one to identify the specific transaction to trigger trace (a write to a particular address), and a second to filter which transactions are actually traced (in this case all transactions with a particular ID).

Firstly use ***set_enabled*** to enable the module but disable the filters:

- Set **operation** to 0 (*apply*);
- Set **module_enable** to 1.

Using ***set_bus_match_ctrl*** for filter A (used to trigger trace):

- Set **match_transactions** to 0x1 to enable filtering on writes;
- Set **match_enable_addr** to 0x2 (*inclusive*) to enable address matching within the range;
- Set **match_enable_id** to 0x2 (*inclusive*) to enable ID matching within the range;
- Set **match_high_id** and **match_low_id** to the ID you wish to match;
- Set **match_value_resp** to all 1s so all responses will match;
- Set all other fields to zero.

Using ***set_bus_match_data*** for the filter A:

- With **select** = 0 (*address*), set **match_low_addr** and **match_high_addr** to the address you wish to detect;
- If you have not done so previously, set fields for all legal values of **select** > 3 to zero (disables user/info matching and matching on cache coherent extensions if available).

Using ***set_bus_match_ctrl*** for the filter B (used to filter trace):

- Set **match_transactions** to all 1's to match all transactions;
- Set **match_enable_id** to 0x2 (*inclusive*) to enable ID matching within the range;
- Set **match_high_id** and **match_low_id** to the ID you wish to match;
- Set **match_value_resp** to all 1s so all responses will match;
- Set all other fields to zero.

Using ***set_bus_match_data*** for filter B:

- Set fields for all legal values of **select** > 3 to zero (disables user/info matching and matching on cache coherent extensions if available).

Using ***set_monitor_trace***:

- Set **timestamp** if you want a timestamp added to the ***monitor_trace_data*** message;
- Set **flow** to the flow you wish to use for ***monitor_trace_data*** messages;
- Set **throttle_level** to a non-zero value;
- Set **history_mode** to 0x3 to select *about* mode;
- Set **trigger** so that there is a 1 in the bit position corresponding to filter A;
- Set **filter_enable** to 1 to enable filtering;
- Set **filter** so that there is a 1 in the bit position corresponding to filter B;
- Set **trace_addr** to 0x1 to enable address tracing;
- Set **trace_data** to 1 to enable data tracing (if you want to trace the data);
- Set **trace_resp** to 1 to enable response tracing (if you want to trace the response);
- Set all other fields to zero.

Finally, using ***set_bus_filter*** for each of filters A and B:

- Set **threshold_mode** to 0xE (*match_last*) or 0xF (*match_first*) to disable threshold matching;
- Set **count_threshold** to 1;
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

Variations:

- Adjust filter A's **match_transactions** to gather metrics for other transaction types;
- Adjust **history_mode** so the triggering cycle is at the beginning or end of the sampling interval if required;
- Set **oneshot** to trigger trace the first time the triggering location is written only.

4.8.2 Example: Trace transaction responsible for bus deadlock

This example triggers trace if a transaction exceeds a specified duration, on the assumption that this indicates the bus has locked up (though it can equally be used to identify transactions taking much longer than expected). Only one filter is required.

Using ***set_bus_match_ctrl*** for the filter of your choice (used to trigger trace):

- Set **match_transactions** to all 1s to match all transactions;
- Set **match_value_resp** to all 1s so all responses will match;
- Set all other fields to zero.

Using ***set_monitor_trace***:

- Set **timestamp** if you want a timestamp added to the ***monitor_trace_data*** message;
- Set **flow** to the flow you wish to use for ***monitor_trace_data*** messages;
- Set **throttle_level** to a non-zero value;
- Set **history_mode** to 0x2 to select *to* mode;
- Set **trigger** so that there is a 1 in the bit position corresponding to filter A;
- Set **trace_addr** to 0x3 or 0x1 to enable address tracing (*deferred* if available, else *direct*);
- Set **trace_data** to 1 to enable data tracing (if you want to trace the data);
- Set **trace_resp** to 1 to enable response tracing;
- Set **incomplete** to enable sampling of an ongoing but incomplete transaction phase;
- Set all other fields to zero.

Finally, using ***set_bus_filter*** for the filter of your choice:

- Set **threshold_mode** to 0 (*duration*) to enable duration threshold matching;
- Set **count_threshold** to the duration threshold required (typically large);
- Set **filter_enable** to 1 to enable the filter;
- Set all other fields to zero.

When triggered, ***monitor_trace_data*** messages will be output for each bus phase that is enabled. If the triggering transaction has any phases that haven't started (for example, a read address phase has completed, but the slave never responds with the data, a ***monitor_trace_data*** message with a 'flag' marker will be output, reporting the transaction state of the triggering transaction (see

Table 5-67).

Variations:

- Adjust **threshold_mode** to trigger based on other metrics (for example, excessive address hesitancy).

4.9 Useful tips

This section provides help avoiding some common pitfalls.

1. How to avoid accidentally matching no transactions

The ***set_bus_match_ctrl*** fields **match_value_transactions** and **match_value_resp** fields are multiple-choice bit vectors, where each bit represents a transaction or response code to match. So if for example bit *N* of **match_value_resp** is 1, the filter will match all transactions where the response value is *N*. This approach allows the filter to be configured to match multiple transaction or response types.

However, a consequence of this approach is that if either of these fields is set to 0, the filter will not match any transactions at all.

A common mistake is to set **match_value_resp** to 0 when you don't care what the response is. In fact, if you want to match transactions regardless of the response, set the field to all 1s. Similarly, set **match_value_transactions** to all 1s if you want to match all transaction types.

2. Take care using min metrics

Metrics marked 'N/A' in Figure 3-1 through Figure 3-6 are recorded as zero. A consequence of this is that when using a counter in min mode to measure the minimum value for one of these metrics, the reported value will always be zero if the matching criteria matches transaction types for which the metric is listed as 'N/A'.

To avoid this, match only transaction types for which the metric is meaningful. For example, if measuring minimum data latency, don't match any read transactions.

3. Changing matching threshold

When issuing match messages or events on every *N*th match, (when ***set_bus_filter*** **threshold_mode** is 8 or more, and **count_threshold** is greater than 1, care should be taken when adjusting **count_threshold** to a smaller value. In this case, either the filter should be disabled and re-enabled, or **threshold_mode** toggled to a value less than 8. If this is not done, the counter will not be reinitialized, and there is a chance that the new configuration will not take effect until the counter has rolled over, which (depending on the size of the counter) may take a long time.

4.10 Troubleshooting

If the Bus Monitor isn't working as you expect, review these common symptoms and possible solutions.

Table 4-1 Troublingshooting guide

Symptom	Solutions
monitor_*_data messages not being generated.	Ensure that the relevant **throttle_level** field is not 0. Unless you are intentionally using throttling, this should be set to 1 (*never*). See Section 3.4.4. Ensure that the relevant **flow** field is set, to route the messages to the communicator you are using. See Section 4.3.
monitor_counter_data messages not being generated periodically.	Ensure the interval timer is programmed, and enabled. See Section 3.4.1. Ensure the interval timer is selected to trigger counter output, by using ***set_monitor*** with **select** = 1 (*interface*) to set **counters_interval_trigger** to 1. See Section 3.3.2.2.
monitor_counter_data messages reporting unexpected zero count values.	Ensure the counter is enabled, and the correct metric selected. See Section 4.7. Ensure the filter associated with the counter (***set_monitor_counter*** **filter** field) is also enabled, and transaction matching is correctly configured. See Section 4.4.1. If using min mode, ensure that only applicable transactions are matched (see Section 2). Ensure the bus is being clocked. See Section 4.7.4 for how to check this.
error bits set in ***monitor_counter_data*** messages.	See Section 3.3.2.3.
Filter not matching any transactions.	See Section 1.

5 Messages

This section provides details of the supported messages, partitioned into upstream and downstream. Each section contains a table of supported messages, ordered based on their control codes. This is followed by details of each message, listed alphabetically for easy reference.

5.1 Upstream messages

The Bus Monitor implements the upstream control messages listed in Table 5-1.

Table 5-1 Bus Monitor upstream system control messages

Control Code	Name	Description
0x00	*discovery_request*	Request discovery information
0x02	*debug_reset*	Soft reset of the entire module
0x03	*get_enabled*	Get module and filter enables
0x04	*set_enabled*	Set module and filter enables
0x05	*get_gpio*	Get GPIO configuration
0x06	*set_gpio*	Set GPIO configuration
0x07	*get_power*	Get power configuration
0x08	*set_power*	Set power configuration
0x1C	*get_msg_params*	Get standard message related parameters
0x1D	*get_period*	Get interval timer configuration
0x1E	*set_period*	Set the interval timer configuration
0x25	*get_bus_filter*	Get a bus performance monitor filter configuration
0x26	*set_bus_filter*	Set a bus performance monitor filter configuration
0x27	*get_bus_match_ctrl*	Get a bus filter match unit control configuration
0x28	*set_bus_match_ctrl*	Set a bus filter match unit control configuration
0x29	*get_bus_match_data*	Get a bus filter match unit data configuration
0x2A	*set_bus_match_data*	Set a bus filter match unit data configuration
0x2F	*manage_alloc*	Manage allocation of resources
0x30	*get_monitor*	Get bus monitor configuration
0x31	*set_monitor*	Set bus monitor configuration
0x32	*get_monitor_counter*	Get a bus monitor counter configuration
0x33	*set_monitor_counter*	Set a bus monitor counter configuration
0x34	*get_monitor_trace*	Get bus monitor trace configuration
0x35	*set_monitor_trace*	Set bus monitor trace configuration
0x36	*monitor_snapshot*	Take a trace or counters snapshot

5.1.1 debug_reset

Used to request a reset of the entire module. Initiates a *reset_response* message once the reset process is complete.

Table 5-2 debug_reset payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x02.

5.1.2 discovery_request

Used to discover the Bus Monitor capabilities parameterized at instantiation, returned via a *discovery_response* message.

Table 5-3 debug_reset payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x00.

5.1.3 get_bus_filter

Used to retrieve the configuration of a filter, which will be returned via *bus_filter_response*. Includes **filter** and **interface** fields to identify the filter and interface for which the configuration is requested.

Table 5-4 get_bus_filter payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x25.
filter	4	Selects filter to access.
interface	2	Selects interface.
reserved	2	Ignored.

5.1.4 get_bus_match_ctrl

Used to retrieve the matching configuration of a filter, which will be returned via *bus_match_ctrl_response*. Includes **filter** and **interface** fields to identify the filter and interface for which the configuration is requested.

Table 5-5 get_bus_match_ctrl payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x27.
filter	4	Selects match filter unit to access.
interface	2	Selects interface.
reserved	2	Ignored.

5.1.5 get_bus_match_data

Used to retrieve the data matching configuration of a filter, which will be returned via *bus_match_data_response*. Includes **filter** and **interface** fields to identify the filter and interface for which the configuration is requested, and a **select** field to further select the group of data fields to access.

Table 5-6 get_bus_match_data payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x29.
filter	4	Selects match filter unit to access.
interface	2	Selects interface.
select	5	Selects the group of data fields to access.
reserved	6	Ignored.

5.1.6 get_enabled

Used to retrieve the enabled state of the Bus Monitor, returned via ***enabled_response***.

Table 5-7 get_enabled payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x03.

5.1.7 get_gpio

Used to retrieve the GPIO configuration, returned via ***gpio_response***.

Table 5-8 get_gpio payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x05.

5.1.8 get_monitor

Used to retrieve global and per-interface Bus Monitor configuration, returned via ***monitor_response***. It includes a **select** field to choose global vs per interface configuration, and for the latter, an **interface** field to identify the interface for which the configuration is requested.

Table 5-9 get_monitor payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x30.
select	1	Global configuration when low, per-interface when high.
interface	2	Selects interface (ignored if **select** is 0).

5.1.9 get_monitor_counter

Used to retrieve the configuration of a counter, returned via ***monitor_counter_response***. Includes **counter** and **interface** fields to identify the counter and interface for which the configuration is requested.

Table 5-10 get_monitor_counter payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x32.
counter	4	Selects counter to access.
interface	2	Selects interface.
reserved	2	Ignored.

5.1.10 get_monitor_trace

Used to retrieve the configuration of the trace unit, returned via ***monitor_trace_response***. Includes an **interface** field to identify the interface for which the configuration is requested.

Table 5-11 get_monitor_trace payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x34.
interface	2	Selects interface

5.1.11 get_msg_params

Used to retrieve the message related parameters of the Bus Monitor, returned via ***msg_params_response***.

Table 5-12 get_msg_params payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x1C.

5.1.12 get_period

Used to retrieve the interval timer configuration, returned via ***period_response***.

Table 5-13 get_period payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x1D.

5.1.13 get_power

Used to retrieve the global power configuration, returned via ***power_response***.

Table 5-14 get_power payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x07.

5.1.14 manage_alloc

Used to negotiate resource allocation, response returned via ***alloc_response***.

Table 5-15 manage_alloc payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2F.
manage_op	2	Allocation operation: 00: *Get*. Requests allocation status of **allocation_index**. 01: *Allocate*. Resource identified by **allocation_index** will be allocated to **debugger_id** if currently unallocated. 10: *Release*. Deallocate resource identified by **allocation_index**. 11: *Force.* Resource identified by **allocation_index** will be allocated to **debugger_id** even if already allocated.
debugger_id	4	ID of debugger or other software API. Zero means unallocated.
reserved	2	Ignored.
allocation_index	16	Resource identifier. The following values are defined for the Bus Monitor: 0 **module_enable**, ***set_period***, ***set_power*** and ***set_monitor*** global fields 128*X* + 1 ***set_monitor*** fields for interface *X* 128*X* + 2 Trace unit for interface *X* 128*X* + 16 + *i* Filter *i* for interface *X* 128*X* + 64 + *i* Counter *i* for interface *X*

5.1.15 monitor_snapshot

Used to trigger trace, output via ***monitor_trace_data*** or snapshot counters, output via ***monitor_counter_data***.

Table 5-16 monitor_snapshot payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x36.
interface	2	Selects interface.
counters	1	Snapshot counters when 1.
trace	1	Trigger trace when 1.

5.1.16 set_bus_filter

Used to apply a configuration to a Bus Monitor filter. Includes **filter** and **interface** fields to identify the filter and interface to access and the configuration to apply.

Table 5-17 set_bus_filter payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x26.
filter	4	Selects filter to access.
interface	2	Selects interface.
filter_enable	1	Filter enable. Reset to 0.

threshold_mode	4[2]	Configures the threshold counter: 0xxx: transaction matches if selected metric is ≥ **count_threshold**: 0000 (*duration*): Duration 0001 (*data_latency*): Data latency 0010 (*resp_latency*): Response latency 0011 (*total_latency*): Total latency 0100 (*addr_hesitancy*): Address hesitancy 0101 (*data_hesitancy*): Data hesitancy 0110 (*resp_hesitancy*): Response hesitancy 0111 (*total_hesitancy*): Total hesitancy 1xxx: Issue triggers specified by **issue_mode**: 1010 (*match_end_last*): Issue on last of every **count_threshold** matches at end of transaction 1011 (*match_end_first*): Issue on 1st of every **count_threshold** matches at end of transaction 1100 (*match_timed*): Issue on last of every **count_threshold** matches in timed interval 1110 (*match_last*): Issue on last of every **count_threshold** matches as soon as match criteria is met 1111 (*match_first*): Issue on 1st of every **count_threshold** matches as soon as match criteria is met Others: reserved
issue_mode	3	Determines what to do when the filter matches. When set, each bit independently controls a different action: Bit 0: Issue an event Bit 1: Issue a ***monitor_match_data*** message (unless throttled) Bit 2: reserved When **threshold_mode** is *match_end_first*, *match_end_last*, *match_timed*, *match_first* or *match_last*, the issue rate can be reduced using **count_threshold** (see **threshold_mode**).
count_threshold	32	Threshold for use in conjunction with **threshold_mode** and **issue_mode**. When **threshold_mode** = *match_end_first*, *match_end_last*, *match_timed*, *match_first* or *match_last*, set to 1 to issue on every match; a value of 0 means the maximum threshold of $2^{\text{filter\_thres\_width\_}X}$ (***discovery_response*** field).
enable_trigger	16	Specifies triggers to enable the filter. Reset to 0. When **event_enable_control** is low, this field is a filter vector, allowing any of the filters to be specified. The enable will become set when filter *N*'s internal trigger activates if there is a 1 in bit *N* (see Section 3.3.1). When **event_enable_control** is high, the enable will be set upon reception of the event specified by bits 7:0. Other bits are ignored.
reserved	7	Ignored (zero in response message).
event_enable_control	1	Determines whether an event or filter can enable the filter (see **enable_trigger** for details).
disable_trigger	16	Specifies triggers to disable the filter. Reset to 0. When **event_disable_control** is low, this field is a filter vector, allowing any of the potential 16 filters to be specified, and when high bits 7:0 identify an event. Behaviour is as per **enable_trigger**.
reserved	7	Ignored (zero in response message).
event_disable_control	1	Determines whether an event or filter can disable the filter (see **enable_trigger** for details).

[2] MSB is not writable, read as 1 if ***discovery_response*** **itas** is 0, or **threshold_width** is 0. LSB is not writable, read as 0 if **threshold_width** is 0.

event	8	Event number to issue.
reserved	1	Ignored (zero in response message).

5.1.17 set_bus_match_ctrl

Used to apply the matching configuration of a filter. Includes **filter** and **interface** fields to identify the filter and interface to access and the configuration to apply.

The following tables show the message format, which is dependent on the supported bus protocol.

Table 5-18 set_bus_match_ctrl payload format - AXI/ACE

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x28.
filter	4	Selects filter to access.
interface	2	Selects interface.
match_transactions	31	Match all transactions for which the corresponding bit is set. Choice of transactions depends on the module variant; see Table 5-19.
match_enable_id	2	Compare ID with **match_low_id** and **match_high_id** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (*exclusive*): match if not between low and high bounds
match_high_id	24	See **match_enable_id**.
match_low_id	24	See **match_enable_id**.
match_enable_addr	2	Compare address with **match_low_addr** and **match_high_addr** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (*exclusive*): match if not between low and high bounds
match_enable_data	1	When set, match transactions with data matching **data_0_127** etc.
match_size_data	3	Configure the way data match lane comparator results are combined. Encoded as per **match_value_size**. See Section 4.4.1.1.
match_enable_beat	1	When set, only data beats between **match_low_beat** and **match_high_beat** (inclusive) may match. Note that the 1st beat in a transaction is beat 0. If **match_high_beat** is less than **match_low_beat**, all beats from **match_low_beat** to the end of the transaction may match.
match_high_beat	8	See **match_enable_beat**.
match_low_beat	8	See **match_enable_beat**.
match_enable_group0_filter	1	When set, match transactions when the group 0 filter given in **match_value_group0_filter** also matches.
match_value_group0_filter	4	See **match_enable_group0_filter**.
match_enable_length	1	When set, match transactions when the burst length matches **match_value_length**.
match_value_length	8	See match_enable_length.
match_enable_burst	1	When set, match transactions when the burst type matches **match_value_burst**.
match_value_burst	3	See **match_enable_burst**.
match_resp_all	1	When set, all beats in a read transaction must match the response specified via **match_value_resp** (and for ACE and ACE-Lite, **match_value_resp23**) for the transaction as a whole to match.
match_value_resp	4	Match all responses for which the corresponding bit is set.

		For AXI reads and writes: Bit 0: OKAY Bit 1: EXOKAY Bit 2: SLVERR Bit 3: DECERR For ACE snoop transactions: Bit 0: DataTransfer = 0, Error = 0 Bit 1: DataTransfer = 1, Error = 0 Bit 2: DataTransfer = 0, Error = 1 Bit 3: DataTransfer = 1, Error = 1
reserved	4	Ignored (zero in response message).
match_enable_size	1	When set, match transactions when the size matches $2^{\text{match\_value\_size}+3}$.
match_value_size	3	See **match_enable_size**.
match_enable_lock	1	When set, match transactions when the `lock` value matches **match_value_lock**.
match_value_lock	2	See **match_enable_lock**.
match_enable_cache	1	When set, match transactions when the `cache` value matches **match_value_cache**.
match_value_cache	4	See **match_ enable_cache**.
match_enable_prot	1	When set, match transactions when the `prot` value matches **match_value_prot**.
match_value_prot	3	See **match_enable_prot**.
match_enable_qos	1	When set, match transactions when the `qos` value matches **match_value_qos**.
match_value_qos	4	See **match_enable_qos**.
match_enable_region	1	When set, match transactions when the `region` value matches **match_value_region**.
match_value_region	4	See **match_enable_region**.

Table 5-19 match_transactions - AXI/ACE

Bit	Transaction (x*Nc*2 variants[3])	Transaction (x*Nc*1 variants[4])	Transaction (x*Nc*0 variants[5])
0	WriteNoSnoop	WriteNoSnoop	Write
1	ReadNoSnoop	ReadNoSnoop	Read
	Write Channel Transactions:	Write Channel Transactions:	reserved
2	WriteUnique / Barrier	WriteUnique / Barrier	
3	WriteLineUnique	WriteLineUnique	
4	WriteClean	WriteClean	
5	WriteBack	WriteBack	
6	Evict	Evict	
7	WriteEvict	WriteEvict	
	Read Channel Transactions:	Read Channel Transactions:	reserved
8	ReadOnce / Barrier[6]	ReadOnce / Barrier[6]	
9	ReadShared	ReadShared	
10	ReadClean	ReadClean	
11	ReadNotSharedDirty	ReadNotSharedDirty	
12	ReadUnique	ReadUnique	
13	CleanShared[6]	CleanShared[6]	
14	CleanInvalid[6]	CleanInvalid[6]	
15	CleanUnique[6]	CleanUnique[6]	
16	MakeUnique[6]	MakeUnique[6]	
17	MakeInvalid[6]	MakeInvalid[6]	
18	DVM Complete[6]	DVM Complete[6]	
19	DVM Message[6]	DVM Message[6]	
	Snoop Channel Transactions:	reserved	reserved
20	ReadOnce		
21	ReadShared		
22	ReadClean		
23	ReadNotSharedDirty		
24	ReadUnique		
25	CleanShared		
26	CleanInvalid		
27	MakeInvalid		
28	DVM Complete		
29	DVM Message		
30	reserved	reserved	reserved

[3] ***discovery_response*** **module_variant** 106, 107, 108, 109.
[4] ***discovery_response*** **module_variant** 102, 103, 104, 105.
[5] ***discovery_response*** **module_variant** 98, 99, 100, 101.
[6] These transactions do not include any valid data in their response. These bits should not be set if data matching is enabled (see Section 4.4.1.1), or when counting data bytes (otherwise each transaction will increment the byte count even though there are no valid bytes). Note: bit 8 can be set provided Barrier transactions are excluded via **match_mask_bar[0]** to 1 and **match_value_bar[0]** to 0. (see Table 5-28)

Table 5-20 set_bus_match_ctrl payload format - OCP

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x28.
filter	4	Selects filter to access.
interface	2	Selects interface.
match_transactions	31	Match all transactions for which the corresponding bit is set. Choice of transactions depends on the module variant; see Table 5-21.
match_enable_id	2	Compare TagID with **match_low_id** and **match_high_id** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_high_id	24	See **match_enable_id**.
match_low_id	24	See **match_enable_id**.
match_enable_addr	2	Compare address with **match_low_addr** and **match_high_addr** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_enable_data	1	When set, match transactions with data matching **data_0_127** etc.
match_size_data	3	Configure the way data match lane comparator results are combined. Encoded as per **match_value_size**. See Section 4.4.1.1.
match_enable_beat	1	When set, only data beats between **match_low_beat** and **match_high_beat** (inclusive) may match. Note that the 1st beat in a transaction is beat 0. If **match_high_beat** is less than **match_low_beat**, all beats from **match_low_beat** to the end of the transaction may match.
match_high_beat	8	See **match_enable_beat**.
match_low_beat	8	See **match_enable_beat**.
match_enable_group0_filter	1	When set, match transactions when the group 0 filter given in **match_value_group0_filter** also matches.
match_value_group0_filter	4	See **match_enable_group0_filter**.
match_enable_length	1	When set, match transactions when the burst length matches **match_value_length**.
match_value_length	8	See **match_enable_length**.
match_enable_burst	1	When set, match transactions when the burst type matches **burst_match_value**.
match_value_burst	3	See **match_enable_burst**.
match_resp_all	1	When set, all beats in a read transaction must match the response specified via **match_value_resp** for the transaction as a whole to match.
match_value_resp	4	Match all responses for which the corresponding bit is set: Bit 0: DVA Bit 1: FAIL Bit 2: ERR Bit 3: OK
reserved	4	Ignored (zero in response message).
match_enable_maddrspace	1	When set, match transactions when the **maddrspace** value matches **match_maddrspace_value**.
match_value_maddrspace	8	See **match_enable_maddrspace**.

match_enable_connid	2	Compare connection ID with **match_low_connid** and **match_high_connid** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_high_connid	16	See **match_enable_connid**.
match_low_connid	16	See **match_enable_connid**.
match_enable_threadid	2	Compare thread ID with **match_low_threadid** and **match_high_threadid** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_high_threadid	16	See **match_enable_threadid**.
match_low_threadid	16	See **match_enable_threadid**.

Table 5-21 match_transactions - OCP

Bit	Transaction (o*N*c2 variants[7])	Transaction (o*N*c1 variants[8])	Transaction (o*N*c0 variants[9])
0	Write	Write	Write
1	Read	Read	Read
2	ReadEx	ReadEx	ReadEx
3	ReadLinked	ReadLinked	ReadLinked
4	WriteNonPost	WriteNonPost	WriteNonPost
5	WriteConditional	WriteConditional	WriteConditional
6	Broadcast	Broadcast	Broadcast
	Main Port Transactions:	reserved	reserved
7	CohReadOwn		
8	CohReadShare		
9	CohReadDiscard		
10	CohReadShareAlways		
11	CohUpgrade		
12	CohWriteBack		
13	CohCopyBack		
14	CohCopyBackInv		
15	CohInvalidate		
16	CohWriteInvalidate		
17	CohCompletionSync		
	Intervention Port Transactions:	reserved	reserved
18	IntvReadOwn		
19	IntvReadShare		
20	IntvReadDiscard		
21	IntvReadShareAlways		
22	IntvUpgrade		
23	IntvWriteBack		
24	IntvCopyBack		
25	IntvCopyBackInv		
26	IntvInvalidate		
27	IntvInvalidateInv		
30:28	reserved	reserved	reserved

[7] *discovery_response* **module_variant** 118, 119, 120, 121.
[8] *discovery_response* **module_variant** 114, 115, 116, 117.
[9] *discovery_response* **module_variant** 110, 111, 112, 113.

5.1.18 set_bus_match_data

Used to apply data and other matching configuration too large to fit within ***set_bus_match_ctrl*** of a filter. Includes **filter** and **interface** fields to identify the filter and interface to access and the configuration to apply.

It also includes a **select** field to determine which fields to access. Some of these (**select** > 2) are protocol specific). The following tables show the message formats.

Table 5-22 Set bus match data message select fields – AXI/ACE

Select index	Name	Description
0	*address*	Address match fields; see Table 5-24.
1	*data_byte_mask*	Data byte comparator enable mask for `wdata` and `rdata`; see Table 5-26.
2 – 6	*reserved*	
7	*coherence*	Cache coherent match fields (x*N*c2 variants only); see Table 5-28.
8	*awuser*	Signal match and mask for `awuser` bus; see Table 5-27.
9	*wuser*	Signal match and mask for `wuser` bus; see Table 5-27.
10	*buser*	Signal match and mask for `buser` bus; see Table 5-27.
11	*aruser*	Signal match and mask for `aruser`; see Table 5-27.
12	*ruser*	Signal match and mask for `ruser` bus; see Table 5-27.
13 – 14	*reserved*	
16	*data_0_127*	Bits 127:0 of data (or less); see Table 5-25.
17	*data_128_255*	Bits 255:128 of data (where applicable); see Table 5-25.
18	*data_256_383*	Bits 383:256 of data (where applicable); see Table 5-25.
19	*data_384_511*	Bits 511:384 of data (where applicable); see Table 5-25.
20	*data_512_639*	Bits 639:512 of data (where applicable); see Table 5-25.
21	*data_640_767*	Bits 767:640 of data (where applicable); see Table 5-25.
22	*data_768_895*	Bits 895:768 of data (where applicable); see Table 5-25.
23	*data_896_1023*	Bits 1023:896 of data (where applicable); see Table 5-25.
24 – 31	*reserved*	

Table 5-23 Set bus match data message select fields - OCP

Select index	Name	Description
0	*address*	Address match fields; see Table 5-24.
1	*data_byte_mask*	Data byte comparator enable mask for `mdata` and `sdata`; see Table 5-26.
2 – 6	*reserved*	
7	*coherence*	Cache coherent match fields (o*N*c2 variants only); see Table 5-30.
8	*mreqinfo*	Match and mask for `mreqinfo` bus; see Table 5-29.
9	*mdatainfo*	Match and mask for `mdatainfo` bus; see Table 5-29.
10	*srespinfo*	Match and mask for `srespinfo` bus; see Table 5-29.
11	*sdatainfo*	Match and mask for `sdatainfo` bus; see Table 5-29.
12 – 15	reserved	
16	*data_0_127*	Bits 127:0 of data (or less); see Table 5-25.
17	*data_128_255*	Bits 255:128 of data (where applicable); see Table 5-25.
18	*data_256_383*	Bits 383:256 of data (where applicable); see Table 5-25.
19	*data_384_511*	Bits 511:384 of data (where applicable); see Table 5-25.
20	*data_512_639*	Bits 639:512 of data (where applicable); see Table 5-25.
21	*data_640_767*	Bits 767:640 of data (where applicable); see Table 5-25.
22	*data_768_895*	Bits 895:768 of data (where applicable); see Table 5-25.
23	*data_896_1023*	Bits 1023:896 of data (where applicable); see Table 5-25.
24 – 31	*reserved*	

Table 5-24 Set_bus_match_data payload format - address

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Set to 0 (*address*).
match_high_address	72	Address range match high address.
match_low_address	72	Address range match low address.

Table 5-25 Set_bus_match_data payload format - data

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	16 (*data_0_127*): data bits 127:0 17 (*data_128_255*): data bits 255:128 18 (*data_256_383*): data bits 383:256 19 (*data_384_511*): data bits 511:384 20 (*data_512_639*): data bits 639:512 21 (*data_640_767*): data bits 767:640 22 (*data_768_895*): data bits 895:768 23 (*data_896_1023*): data bits 1023:896
match_data	128	Data match value

Table 5-26 Set_bus_match_data payload format - data byte mask

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Set to 1 (*data_byte_mask*).
match_mask_bytes	128	Byte lane filtering for data buses. Bytes of data are compared to **match_data** only when the corresponding mask bit is 1.

Table 5-27 Set_bus_match_data payload format – AXI user

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Selects user bus to configure: 8 (*awuser*): `awuser` 9 (*wuser*): `wuser` 10 (*buser*): `buser` 11 (*aruser*): `aruser` 12 (*ruser*): `ruser`
match_value_*U*user[10]	96	User bus match value.
match_mask_*U*user[10]	96	User bus mask value. Set bits of the mask cause corresponding user bus bits to be compared with the match value, otherwise they are ignored. Reset to 0.

Table 5-28 Set_bus_match_data payload format – ACE cache coherency extensions

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Set to 7 (*coherence*).
match_mask_bar	2	Set bits of the mask cause the corresponding bits of `bar` to be compared with **match_value_bar**, otherwise they are ignored. Reset to 0.
match_value_bar	2	See **match_mask_bar**.
match_enable_domain	1	When set, match transactions when the `domain` value matches **match_value_domain**. Reset to 0.
match_value_domain	2	See **match_enable_domain**.
match_enable_unique	1	When set, match transactions when the `unique` value matches **match_value_unique**. Reset to 0.
match_value_unique	1	See **match_enable_unique**.
match_value_resp23	4	Match all read/snoop responses decoded from bits `resp[3:2]` for which the corresponding bit is set: 0: IsShared = 0, Dirty = 0 1: IsShared = 0, Dirty = 1 2: IsShared = 1, Dirty = 0 3: IsShared = 1, Dirty = 1 Reset to 1111.
match_value_resp4	2	Match all snoop responses decoded from bits `resp[4]` for which the corresponding bit is set: 0: WasUnique = 0 1: WasUnique = 1 Reset to 11.

[10] ***U*** = **aw**|**w**|**b**|**ar**|**r**

Table 5-29 Set_bus_match_data payload format – OCP info

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Selects info bus to configure: 8 (*mreqinfo*): `mreqinfo` 9 (*mdatainfo*): `mdatainfo` 10 (*srespinfo*): `srespinfo` 11 (*sdatainfo*): `sdatainfo`
match_value_*I*info[11]	96	Info bus match value.
match_mask_*I*info[11]	96	Info bus mask value. Set bits of the mask cause corresponding info bus bits to be compared with the match value, otherwise they are ignored. Reset to 0.

Table 5-30 Set_bus_match_data payload format – OCP cache coherency extensions

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2A.
filter	4	Selects filter to access.
interface	2	Selects interface.
select	5	Set to 7 (*coherence*).
match_enable_cohid	2	Compare coherent ID with **match_low_cohid** and **match_high_cohid** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_high_cohid	16	See **match_enable_cohid**.
match_low_cohid	16	See **match_enable_cohid**.
match_enable_cohfwdid	2	Compare coherent forward ID with **match_low_cohfwdid** and **match_high_cohfwdid** as follows: 0x (*disabled*): match disabled 10 (*inclusive*): match if between low and high bounds 11 (ex*clusive*): match if not between low and high bounds
match_high_cohfwdid	16	See **match_enable_cohfwdid**.
match_low_cohfwdid	16	See **match_enable_cohfwdid**.
match_enable_mcohcmd	1	When set, match transactions when the `mcohcmd` value matches **match_value_mcohcmd**. Reset to 0.
match_value_mcohcmd	1	See **match_enable_cohcmd**.
match_enable_mreqself	1	When set, match transactions when the `mreqself` value matches **match_value_mreqself**. Reset to 0.
match_value_mreqself	1	See **match_enable_mreqself**.
match_enable_scohstate	1	When set, match transactions when the `scohstate` value matches **match_value_scohstate**. Reset to 0.
match_value_scohstate	3	See **match_enable_scohstate**.

[11] ***I*** = **mreq**|**mdata**|**sdata**|**sresp**

5.1.19 set_enabled

Used to control the enabled state of the Bus Monitor.

Table 5-31 set_enabled payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x04.
operation	2	Identifies the operation to be performed on the configuration: 00: *Apply*: applies the ***_enable** fields as a new configuration. 01: *Set*: updates the configuration with the logical OR of the current configuration and the ***_enable** fields. 10: *Clear*: updates the configuration with the logical AND of the current configuration and the ***_enable** fields. 11: *Clear_all_enables*: Clears all enables (**module_enable**, **filter_enable**, **counter_enable** and **trace_enable** fields as accessible from the ***set_enabled***, ***set_bus_filter***, ***set_monitor_counter*** and ***set_monitor_trace*** messages respectively). Note: The ***_enable** field values are ignored in this case.
reserved	6	Ignored (zero in response message).
module_enable	1	Global module enable. Reset to 0. Note: This field is don't care if **operation** = '11'.
filter_enable	64	One enable per filter for each interface: 15:0 – enables for interface 0 31:16 – enables for interface 1 47:32 – enables for interface 2 63:48 – enables for interface 3 Reset to 0. Note: This field is don't care if **operation** = '11'.

5.1.20 set_gpio

Used to configure the GPIO interface.

Table 5-32 set_gpio payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x06.
gpio	232	The value to be driven on the general purpose output bus. Reset value determined at instantiation.

5.1.21 set_monitor

Used to apply global and per-interface Bus Monitor configuration. It includes a **select** field to choose global vs per interface configuration, and for the latter, an **interface** field to identify the interface for which the configuration is requested.

Table 5-33 set_monitor payload format – global

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x31.
select	1	Set to 0 (*global*).
sys_timestamp	1	Include timestamp in downstream system control messages when set. Reset to 0.
sys_flow	2	The flow to assign to all internally initiated system control messages (***message_lost***, ***event_lost***).

Table 5-34 set_monitor payload format – per interface

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x31.
select	1	Set to 1 (*interface*).
interface	2	Selects interface.
reserved	1	Ignored (zero in response message).
match_timestamp	1	Include timestamp in ***monitor_match_data*** messages when set.
match_flow	2	The flow to assign to all ***monitor_match_data*** messages.
match_throttle_level	2	For ***monitor_match_data*** messages: 00 (*always*): Always throttled 01 (*never*): Never throttled (ignore throttling events) 10 (*one*): Throttled after throttle1 event 11 (*nonzero*): Throttled after throttle1 or throttle0 event Resume on throttle_off event
counters_timestamp	1	Include timestamp in ***monitor_counter_data*** messages when set.
counters_flow	2	The flow to assign to all ***monitor_counter_data*** messages.
counters_throttle_level	2	For ***monitor_counter_data*** messages. Coding as per **match_throttle_level**.
counters_trigger	16	Specifies triggers to snapshot the counters and issue a ***monitor_counter_data*** message. Reset to 0. When **counters_event_control** is low, this field is a filter vector, allowing any number of filters to be specified. The snapshot will occur when filter *N*'s internal trigger activates if there is a 1 in bit *N* (see Section 3.3.1). When **counters_event_control** is high, the snapshot will occur upon reception of the event specified by bits 7:0. Other bits are ignored.
reserved	6	Ignored (zero in response message).
counters_interval_trigger	1	When set, counters are snapshot when the interval timer expires. Reset to 0.
counters_event_control	1	Determines whether an event or filter can snapshot the counters (see **counters_trigger** for details). Reset to 0.

ita_duration	1	Each bit determines whether the respective transaction metric should be assigned to an individual transaction accumulator or not (see Sections 3.1.1 and 4.5). Reset to 0.
ita_data_latency	1	
ita_response_latency	1	
ita_total_latency	1	
ita_address_hesitancy	1	
ita_data_hesitancy	1	
ita_response_hesitancy	1	
ita_total_hesitancy	1	
ita_beats	1	
ita_bytes	1	
tracker_disable	3	Disables the bus transaction tracker when set. Reset to 0. Disabling trackers should be approached with caution. Re-enabling when the bus is not idle can result in a corrupted tracker state. Bit 0: AXI write tracker / OCP tracker Bit 1: AXI read tracker / OCP intervention tracker Bit 2: ACE snoop tracker / read as zero for OCP
tracker_overflow	3	Status field in ***monitor_response***; reserved in ***set_monitor***. If 1, indicates that the transaction tracker has overflowed since last ***monitor_response*** message. This can only happen if the actual number of concurrent transactions the bus can support is larger than indicated by the relevant ***_concurrent** *discovery_response* field (i.e. parameterization at hardware instantiation is incorrect). Bit 0: AXI write tracker / OCP tracker Bit 1: AXI read tracker / OCP intervention tracker Bit 2: ACE snoop tracker / read as zero for OCP

5.1.22 set_monitor_counter

Used to apply the configuration of a counter. Includes **counter** and **interface** fields to identify the counter and interface for which the configuration is requested.

Table 5-35 set_monitor_counter payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x33.
counter	4	Selects counter to access.
interface	2	Selects interface.
counter_enable	1	Counter enable. Reset to 0.
filter	4	Selects the filter used to activate the counter.
metric	4	Metric to accumulate: 0000 (*duration*): Duration 0001 (*data_latency*): Data latency 0010 (*resp_latency*): Response latency 0011 (*total_latency*): Total latency 0100 (*addr_hesitancy*): Address hesitancy 0101 (*data_hesitancy*): Data hesitancy 0110 (*resp_hesitancy*): Response hesitancy 0111 (*total_hesitancy*): Total hesitancy 1000 (*data_beats*): Data beats 1001 (*data_bytes*): Data bytes 1010 (*transactions*): Transactions 1011 (*bus_cycles*): Bus cycles 1100 (*busy_cycles*): Busy cycles 1101: reserved 111x: reserved
counter_mode	2	Operating mode for counter. Reset to 0.

		00 (*count*): Counter will accumulate the selected metric. 01 (*min*): Counter will record minimum value of selected metric since counter last output. 10 (*max*): Counter will record maximum value of selected metric since counter last output. 11: reserved When using min or max modes, note that: • metrics must usually also be assigned to an individual transaction accumulator (see Section 3.3.2.1). • The 16 MSBs of the counter value will usually be replaced with the timestamp (see Section 3.3.2.2),

5.1.23 set_monitor_trace

Used to apply the configuration of the trace unit. Includes an **interface** field to identify the interface for which the configuration is requested.

Table 5-36 set_monitor_trace payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x35.
interface	2	Selects interface.
trace_enable	1	Trace enable. Reset to 0.
timestamp	1	Include timestamp in ***monitor_trace_data*** messages when set.
history_mode	2	Controls the trace mode: 00 (*stream*): Trace when the trigger is active; 01 (*to*): Trace to the trigger; 10 (*from*): Trace from the trigger; 11 (*about*): The trigger is in the middle of the trace window.
flow	2	The flow to assign to all ***monitor_trace_data*** messages.
throttle_level	2	For ***monitor_trace_data*** messages: 00 (*always*): Always throttled 01 (*never*): Never throttled (ignore throttling events) 10 (*one*): Throttled after throttle1 event 11 (*nonzero*): Throttled after throttle1 or throttle0 event Resume on throttle_off event
trigger	16	Specifies triggers to trigger trace operation. When **event_control** is low, this field is a filter vector, allowing any number of filters to be specified. Trace will trigger when filter *N* activates[12] if there is a 1 in bit *N* When **event_control** is high, trace will trigger upon reception of the event specified by bits 7:0. Other bits are ignored.
reserved	6	Ignored (zero in response message).
interval_trigger	1	When set, trace is triggered when the interval timer expires.
event_control	1	Determines whether an event or filter triggers trace (see **trigger** for details).

[12] Note: this is not an internal trigger (see Section 3.3.1). The filter activates when all enabled matching criteria have been met. It is not dependent on ***set_bus_filter*** **threshold_mode** or **count_threshold** values (it behaves as if **threshold_mode** is 0xe or 0xf and **count_threshold** is 1).

filter	16	Specifies filters to determine when trace is captured in *to*, *from* or *about* modes: When **filter_enable** is high, trace data is captured when filter *N* activates[12] if there is a 1 in bit *N*. When **filter_enable** is low, trace capture is not filtered, and all trace data is captured, except address signals in deferred mode, which won't be captured unless it matches an enabled filter (see Section 3.3.3.1 for further details).
filter_enable	1	Determines whether capture of trace data is filtered (see **filter** for details).
oneshot	1	When set, trace *to*, *from* and *about* modes will only trigger once until a 1 is written to this bit again (this repriming will only be effective if all trace from a previous trigger has been output). In *stream* mode it, tracing will stop once the buffer fills.
incomplete	1	Trace incomplete transaction phase. When set, a trigger caused by exceeding a metric threshold, an event, or from a ***monitor_snapshot*** message will cause transactions phases in progress but not yet complete to be captured immediately rather than waiting for completion. This is to aid the diagnosis of bus lock-up.
trace_addr[13]	2	Enable address/cmd trace in the required mode: 00 (*disabled0*): Don't trace 01 (*direct*): Direct mode trace (channels 0, 1 & 5) 10 (disabled2): Don't trace 11 (*deferred*): Deferred mode trace (channels 8, 9 & 13) **Note:** to switch between direct and deferred mode, first disable trace, then re-enable with the desired new mode.
trace_data	1	Enable data trace (channels 2, 3 & 6).
trace_resp	1	Enable response trace (channels 4 & 7).
trace_source[14]	2	Determines which set of channels to trace when trace sharing is available (**trace_shared_X** ***discovery_response*** field is 1): 00 (*read*): channels 0/8 & 2 (x*N* variants) channels 0/8, 3 & 4 (o*Nc2* variants) 01 (*write*): channels 1/9, 3 & 4 (x*N* variants) channels 0/8, 3 & 4 (o*Nc2* variants) 10 (*snoop*): channels 5/13, 6 & 7 (c2 variants only) **Note:** the trace source cannot be changed whilst trace is enabled. To change source, first disable trace, then re-enable with the desired new source.

[13] The MSB is not writable if only direct or only deferred trace is supported. It will read as 0 if direct trace is the only available option, or 1 if deferred trace is the only available option.
[14] Only writable if the **trace_shared_X** ***discovery_response*** field is 1; read as zero otherwise. MSB only writable for c2 variants.

5.1.24 set_period

Used to configure the interval timer.

Table 5-37 set_period payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x1E.
timer_enable	2	Controls the interval timer. 00 (d*isabled*): Timer is disabled 01 (*continuous*): Timer is free-running 10 (*Windowed*): Timer is triggered once 11 (*Triggered*): Timer is triggered and keeps running
reserved	6	Ignored (zero in response message).
timer_interval	64	The timer period, in UDB cycles.
timer_index_trigger	8	The event index to trigger the timer when configured in *windowed* or *triggered* modes.

5.1.25 set_power

Used to apply the power configuration.

Table 5-38 set_power payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x08.
clk_disable	1	Enables internal clock gating (and reduces power consumption) when set to 1. Set to 0 only under guidance from Tessent Embedded Analytics.

5.2 Downstream messages

The Bus Monitor implements the downstream messages as shown by Table 5-39. The system control messages are listed in Table 5-40.

Table 5-39 Bus Monitor downstream messages

Message Type	Name	Description
0x0	Various	System control messages (See Table 5-40 for list of supported messages).
0x1	*monitor_counter_data*	Metric counter values
0x2	*monitor_match_data*	Filter match details
0x3	*monitor_trace_data*	Trace values

Table 5-40 Bus Monitor downstream system control messages

Control Code	Name	Description
0x00	*discovery_response*	Discovery information
0x02	*message_lost*	Indicates that a ***monitor_*_data*** message could not be sent
0x03	*return*	Reports that a message could not be fully processed
0x04	*enabled_response*	Reports module and filter enables
0x06	*gpio_response*	Reports the GPIO configuration
0x08	*power_response*	Reports the power configuration
0x13	*event_lost*	Indicates that a scheduled event could not be sent.
0x19	*reset_response*	Sent when all actions resulting from *debug_reset* have been completed
0x1C	*msg_params_response*	Reports standard message related parameters
0x1E	*period_response*	Reports the interval timer configuration.
0x26	*bus_filter_response*	Reports a filter configuration
0x28	*bus_match_ctrl_response*	Reports a bus match configuration
0x2A	*bus_match_data_response*	Reports a bus filter match data configuration
0x2F	*alloc_response*	Reports status in response to *manage_alloc*.
0x31	*monitor_response*	Reports the bus monitor configuration
0x33	*monitor_counter_response*	Reports a counter configuration
0x35	*monitor_trace_response*	Reports the trace configuration

5.2.1 alloc_response

Generated in response to ***manage_alloc***.

Table 5-41 alloc_response payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x2F.
return_code	2	Response to requested allocation operation: 00: *Success*. Resource identified by **allocation_index** was allocated to **debugger_id** prior to allocation request. Generated in response to *get*, *force* or successful *allocate* **manage_op** values. 01: *reserved*. 10: *Declined*. Resource identified by **allocation_index** is allocated to **debugger_id**. Generated in response to *allocate* **manage_op** value if **debugger_id** is non-zero (i.e. resource already allocated). 11: *Error.* Resource identified by **allocation_index** does not exist. Generated in response to ***manage_alloc*** with an invalid **allocation_index**.
debugger_id	4	ID of debugger or other software API. Zero means unallocated. Reset to 0.
reserved	2	Read as zero.
allocation_index	16	Resource identifier. The following values are defined for the Bus Monitor: 0 **module_enable**, ***set_period***, ***set_power*** and ***set_monitor*** global fields 128*X* + 1 ***set_monitor*** fields for interface *X* 128*X* + 2 Trace unit for interface *X* 128*X* + 16 + *i* Filter *i* for interface *X* 128*X* + 64 + *i* Counter *i* for interface *X*

5.2.2 bus_filter_response

Generated in response to ***get_bus_filter***. Payload is as per ***set_bus_filter***.

5.2.3 bus_match_ctrl_response

Generated in response to ***get_bus_match_ctrl***. Payload is as per ***set_bus_match_ctrl***.

5.2.4 bus_match_data_response

Generated in response to ***get_bus_match_data***. Payloads are as ***set_bus_match_data***.

5.2.5 discovery_response

Generated in response to ***discovery_request***.

The ***discovery_response*** message provides information about this analytic module. The response is spread across 3 to 9 tranches. The number of tranches sent will depend on the module variant. The 1st tranche contains information about the entire module. This is followed by a pair of tranches per interface. The 1st in each pair is bus protocol independent, the second is bus protocol specific.

Table 5-42 discovery_response payload format – first tranche

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x00.
tranche	4	Number of tranches still to come.
type	4	0
error	1	0
module_variant	10	Identifies the analytic module. The following variants are defined: Not cache coherent: 98: 1 AXI interface (ust_bus_mon_x1c0_m) 99: 2 AXI interfaces (ust_bus_mon_x2c0_m) 100: 3 AXI interfaces (ust_bus_mon_x3c0_m) 101: 4 AXI interfaces (ust_bus_mon_x4c0_m) 110: 1 OCP 2.1 interfaces (ust_bus_mon_o1c0_m) 111: 2 OCP 2.1 interfaces (ust_bus_mon_o2c0_m) 112: 3 OCP 2.1 interfaces (ust_bus_mon_o3c0_m) 113: 4 OCP 2.1 interfaces (ust_bus_mon_o4c0_m) Cache coherence aware: 102: 1 ACE-Lite interface (ust_bus_mon_x1c1_m) 103: 2 ACE-Lite interfaces (ust_bus_mon_x2c1_m) 104: 3 ACE-Lite interfaces (ust_bus_mon_x3c1_m) 105: 4 ACE-Lite interfaces (ust_bus_mon_x4c1_m) 114: 1 OCP 3 interfaces (ust_bus_mon_o1c1_m) 115: 2 OCP 3 interfaces (ust_bus_mon_o2c1_m) 116: 3 OCP 3 interfaces (ust_bus_mon_o3c1_m) 117: 4 OCP 3 interfaces (ust_bus_mon_o4c1_m) Fully cache coherent: 106: 1 ACE interface (ust_bus_mon_x1c2_m) 107: 2 ACE interfaces (ust_bus_mon_x2c2_m) 108: 3 ACE interfaces (ust_bus_mon_x3c2_m) 109: 4 ACE interfaces (ust_bus_mon_x4c2_m) 118: 1 OCP 3 interfaces (ust_bus_mon_o1c2_m) 119: 2 OCP 3 interfaces (ust_bus_mon_o2c2_m) 120: 3 OCP 3 interfaces (ust_bus_mon_o3c2_m) 121: 4 OCP 3 interfaces (ust_bus_mon_o4c2_m)
minor_revision	4	Identifies the minor revision.
version	9	Identifies the module version.
vendor	9	Set to 1 to identify the vendor as Tessent Embedded Analytics.
alloc	1	Indicates whether resource allocation is supported or not.
bypass	1	Indicates whether retiming registers on the message interfaces are bypassed or not.
ds_msg_sz	4	Width of downstream message interface is $2^{\mathbf{ds\_msg\_sz}}$.
gpio	1	Indicates whether the GPIO interface is implemented or not.
gpio_out_width	8	Width of the `gpio_output` bus is **gpio_out_width** + 1.
level_shift	1	Indicates whether level shifters are included between UDB and bus clock domains.
timer_width	3	Width of the interval timer: 000: reserved 001: 16 010: 24 011: 32 100: 40 101: 48 110: 56 111: 64
us_msg_sz	4	Width of upstream data bus to message engine is $2^{us\_msg\_sz}$.

Table 5-43 discovery_response payload format – even tranches 2 to 8

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x00.
tranche	4	Number of tranches still to come.
type	4	1
count_duration_*X*	1	Duration metric gathering implemented for interface *X*.
count_dlatency_*X*	1	Data hesitancy metric gathering implemented for interface *X*.
count_rlatency_*X*	1	Response hesitancy metric gathering implemented for interface *X*.
count_tlatency_*X*	1	Total latency metric gathering implemented for interface *X*.
count_ahesitancy_*X*	1	Address hesitancy metric gathering implemented for interface *X*.
count_dhesitancy_*X*	1	Data hesitancy metric gathering implemented for interface *X*.
count_rhesitancy_*X*	1	Response hesitancy metric gathering implemented for interface *X*.
count_thesitancy_*X*	1	Total hesitancy metric gathering implemented for interface *X*.
count_beats_*X*	1	Beats metric gathering implemented for interface *X*.
count_bytes_*X*	1	Bytes metric gathering implemented for interface *X*.
reserved	1	0
counters_*X*	7	The number of counters for interface *X*.
counter_width_*X*	3	Width of the counters: 000: No counters 001: 16 010: 24 011: 32 100: 48 101: 64
counters_minmax_*X*	7	The number of counters supporting min/max mode for interface *X*.
ds_cdc_depth_*X*	8	Depth of FIFOs used to cross from bus interface *X* clock to debug clock domain is **ds_cdc_depth_*X*** + 1.
filters_*X*	4	Number of filters associated with interface *X* is **filters_*X*** + 1.
filters_g0_*X*	5	Number of filters in group 0 for interface *X*.
filter_thres_width_*X*	12	Width of threshold counter for interface *X*. Bits 5:0 are filter group 0 11:6 are filter group 1.
itas_*X*	4	Number of individual transaction accumulators for interface *X*.
ita_width_*X*	5	Width of individual transaction accumulators for interface *X* is **ita_width_*X*** + 1.
pipeline_*X*	4	Additional internal pipelining for interface *X* when not zero.
test_*X*_mtype	5	Memory type for trace buffers associated with interface *X*: 1: inferred (i.e. flip flop based) 0, 2 – 15: reserved for Tessent Embedded Analytics use 16 – 31: user defined memory macro type.
test_in_*X*_width	9	Width of the `ust_test_in_X_ip` bus is 8 * (**test_in_*X*_width** + 1).
test_out_*X*_width	9	Width of the `ust_test_out_X_ip` bus is 8 * **test_out_*X*_width** + 1).
trace_*X*	2	Indicates whether trace is included for interface *X*: 0: no trace 1: trace without compression 2: trace with XOR compression
trace_addr_depth_*X*	4	Depth of the address trace buffer associated with interface *X* is 0 if trace_addr_depth_X is 15 or $2^{\text{trace\_addr\_depth\_}X}$ otherwise
trace_data_depth_*X*	4	Depth of the data trace buffer associated with interface *X* is 0 if trace_data_depth_X is 15 or $2^{\text{trace\_data\_depth\_}X}$ otherwise
trace_resp_depth_*X*	4	Depth of the response trace buffer associated with interface *X* is 0 if trace_resp_depth_X is 15 or $2^{\text{trace\_resp\_depth\_}X}$ otherwise
trace_ptime_width_*X*	5	The width of the precise time trace field for interface *X*.
trace_shared_*X*	1	Indicates whether read and write share trace buffers for interface *X*.

us_cdc_depth_*X*	8	Depth of the upstream event FIFO to cross from debug clock to bus interface clock domain is automatically sized if 0, or is **us_cdc_depth_X** + 1 for non-zero values.
tracker_type_*X*	2	Transaction tracker type for interface *X*. 0: Linked list. No restrictions on transaction ID. 1: Unique ID. Each transaction in progress must have a different ID.

Table 5-44 discovery_response payload format – AXI: odd tranches 3 to 9 tranche

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x00.
tranche	4	Number of tranches still to come.
type	4	2
axi4_*X*	1	Indicates AXI3 (0) or AXI4 (1) for interface *X*.
axi_*X*_addr	7	Width of the AXI *X* `address` bus is **axi_*X*_addr** + 1.
axi_*X*_aruser	7	Width of the AXI *X* `aruser` bus is **axi_*X*_aruser** + 1
axi_*X*_awuser	7	Width of the AXI *X* `awuser` bus is **axi_*X*_awuser** + 1
axi_*X*_buser	7	Width of the AXI *X* `buser` bus is **axi_*X*_buser** + 1
axi_*X*_bypass	4	AXI bus input register slice bypass for interface *X*.
axi_*X*_data	3	Width of the AXI *X* `data` bus is $2^{(\text{axi\_}X\text{\_data}+3)}$: 000: 8 bits 001: 16 bits 010: 32 bits : 111: 1024 bits
axi_*X*_id	5	Width of the AXI *X* `id` bus is **axi_*X*_id** + 1.
axi_*X*_ruser	7	Width of the AXI *X* `ruser` bus is **axi_*X*_ruser** + 1
axi_*X*_wuser	7	Width of the AXI *X* `wuser` bus is **axi_*X*_wuser** + 1
axi2udb_*X*_sync_stages	3	Number of synchronizer stages from AXI to UDB clock domain.
filter_addr_*X*	4	Each of these fields determines which of the various buses can be filtered by filters in group 0 and group 1, and which can be traced for interface *X*: Bit 0: filtered by group 0 filters Bit 1: filtered by group 1 filters Bit 2: traced using direct trace mode Bit 3: traced using deferred trace mode Fields not associated with the transaction address phase do not have a bit 3, as they can only be captured using direct trace.
filter_aruser_*X*	4	
filter_awuser_*X*	4	
filter_bar_*X*	4	
filter_beat_*X*	3	
filter_burst_*X*	4	
filter_buser_*X*	3	
filter_cache_*X*	4	
filter_domain_*X*	4	
filter_data_*X*	3	
filter_id_*X*	4	
filter_len_*X*	4	
filter_lock_*X*	4	
filter_prot_*X*	4	
filter_qos_*X*	4	
filter_region_*X*	4	
filter_resp_*X*	3	
filter_ruser_*X*	3	
filter_size_*X*	4	
filter_unique_*X*	4	
filter_wuser_*X*	3	
r_concurrent_*X*	8	Number of concurrent read transactions supported by interface *X* is **r_concurrent_*X*** + 1.

s_concurrent_*X*	8	Number of concurrent snoop transactions supported by interface *X* is **s_concurrent_*X*** + 1.
s_data_*X*	1	Indicates that the snoop data channel is implemented for interface *X* when 1.
udb2axi_*X*_sync_stages	3	Number of synchronizer stages from UDB to AXI clock domain.
w_concurrent_*X*	8	Number of concurrent write transactions supported by interface *X* is **w_concurrent_*X*** + 1.
w_datanot1st_*X*	1	Indicates that data for a write cannot arrive before the address for interface *X* when 1.

Table 5-45 discovery_response payload format – OCP: odd tranches 3 to 9

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x00.
tranche	4	Number of tranches still to come.
type	4	2
concurrent_*X*	8	Number of concurrent main port transactions supported by interface *X* is **concurrent_*X*** + 1.
i_concurrent_x	8	Number of concurrent intervention port transactions supported for interface X is **i_concurrent_x** + 1
ocp_*X*_burstblock	1	2D blocks supported for interface *X*.
ocp_*X*_bypass	4	OCP bus input register slice bypass for interface *X*.
ocp_*X*_byteen	1	OCP *X* address phase byte strobes provided when 1.
ocp_*X*_cohfwdid	4	Width of the OCP *X* `mcohfwdid` and `scohfwdid` buses is **ocp_*X*_cohfwdid** + 1.
ocp_*X*_cohid	4	Width of the OCP *X* `mcohid` and `scohid` buses is **ocp_*X*_cohid** + 1.
ocp_*X*_databyteen	1	OCP *X* data handshake phase byte strobes provided when 1.
ocp_*X*_datahandshake	1	Data handshake phase supported for interface *X*.
ocp_*X*_data	8	Width of the OCP *X* `data` bus is **ocp_*X*_data** + 1.
ocp_*X*_intportwritedata	1	OCP writeback data uses intervention port when 1.
ocp_*X*_intportsplittranx	1	OCP intervention port splits read response and data when 1.
ocp_*X*_maddr	7	Width of the OCP *X* `maddr` bus is **ocp_*X*_maddr** + 1.
ocp_*X*_maddrspace	4	Width of the OCP *X* `maddrspace` bus is **ocp_*X*_maddrspace** + 1.
ocp_X_mblockheight	3	Width of the OCP *X* `mblockheight` bus is **ocp_*X*_mblockheight** + 1.
ocp_X_mblockstride	7	Width of the OCP *X* `mblockstride` bus is **ocp_*X*_mblockstride** + 1.
ocp_*X*_mburstlength	3	Width of the OCP *X* `mburstlength` bus is **ocp_*X*_mburstlength** + 1.
ocp_*X*_mconnid	4	Width of the OCP *X* `mconnid` bus is **ocp_*X*_mconnid** + 1.
ocp_*X*_mdatainfo	7	Width of the OCP *X* `mdatainfo` bus is **ocp_*X*_mdatainfo** + 1.
ocp_*X*_mdatalast	1	OCP *X* `mdatalast` provided when 1.
ocp_*X*_mreqinfo	7	Width of the OCP *X* `mreqinfo` bus is **ocp_*X*_mreqinfo** + 1.
ocp_*X*_sdatainfo	7	Width of the OCP *X* `sdatainfo` bus is **ocp_*X*_sdatainfo** + 1.
ocp_*X*_srespinfo	7	Width of the OCP *X* `srespinfo` bus is **ocp_*X*_srespinfo** + 1.
ocp_*X*_sresplast	1	OCP *X* `sresplast` provided when 1.
ocp_*X*_tagid	5	Width of the OCP *X* `mtagid`, `mdatatagid` and `sdatatagid` buses is **ocp_*X*_tagid** + 1.
ocp_*X*_threadid	4	Width of the OCP *X* `mthreadid`, `mdatathreadid` and `sdatathreadid` buses is **ocp_*X*_threadid** + 1.
ocp_*X*_writeresp	1	OCP *X* writes all expect a response when 1.
ocp2udb_*X*_sync_stages	3	Number of synchronizer stages from OCP to UDB clock domain.
filter_addr_*X*	4	Each of these fields determines which of the various buses can be filtered by filters in group 0 and group 1, and which can be traced for interface *X*: Bit 0: filtered by group 0 filters Bit 1: filtered by group 1 filters
filter_addrspace_*X*	4	
filter_beat_*X*	3	
filter_burst_*X*	4	

filter_burstlen_*X*	4	Bit 2: traced using direct trace mode Bit 3: traced using deferred trace mode Fields not associated with the transaction request phase do not have a bit 3, as they can only be captured using direct trace. **filter_data_*X*[3]** and **filter_mdatainfo_*X*[3]** are only relevant if data is part of the request phase (**ocp_*X*_datahandshake** = 0).
filter_cohcmd_*X*	4	
filter_cohfwdid_*X*	4	
filter_cohid_*X*	4	
filter_cohstate_*X*	4	
filter_connid_*X*	4	
filter_data_*X*	4	
filter_tagid_*X*	4	
filter_mreqinfo_*X*	4	
filter_mdatainfo_*X*	3	
filter_resp_*X*	3	
filter_sdatainfo_*X*	3	
filter_srespinfo_*X*	3	
filter_threadid_*X*	4	
udb2ocp_*X*_sync_stages	3	Number of synchronizer stages from UDB to OCP clock domain.

5.2.6 enabled_response

Generated in response to ***get_enabled***. Payload format is as ***set_enabled***, except that the operation field is always returned as 0.

5.2.7 event_lost

Generated whenever an event could not be sent; see Section 3.4.7.

Table 5-46 event_lost payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x13.

5.2.8 gpio_response

Generated in response to ***get_gpio***. Payload format is as ***set_gpio***.

5.2.9 message_lost

Generated whenever a ***monitor_match_data***, ***monitor_counter_data*** or ***monitor_trace_data*** message could not be sent; see Section 3.4.7.

Table 5-47 message_lost payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x02.
interface	2	Interface for which messages have been lost.
high_loss	1	Indicates number of messages lost is greater than the number successfully sent since the last ***message_lost*** message from this module.
lost_stream	16	Identifies the source of the lost messages. Bit 0 Indicates ***monitor_match_data*** match message has been lost Bit 1 Indicates a ***monitor_counter_data*** counter message has been lost Bits 15:2 Indicates a ***monitor_trace_data*** message has been lost from trace channel *N* – 2 (where *N* is the bit which is set)

5.2.10 monitor_counter_data

This message reports count values; see Section 3.3.2.

Table 5-48 monitor_counter_data payload format

Field name	Bits	Description
msg_type	2	Set to 0x1.
reserved	4	Read as zero.
interface	2	Interface to which the counters belong.
tranche	4	The number of tranches still to come.
cause	2	Indicates what caused this set of count values to be output: 00: Interval timer expiry 01: Interval timer reconfigured so interval incomplete 10: Counter overflow (snapshot occurs before overflow so no data is lost) 11: Other trigger (***monitor_snapshot*** message, event or a filter match)
counter0	See[15]	The value from counter 0.
error0	1	Error flag associated with counter 0 (see section 0).
...	...	...
counter*N*	See[15]	The value from the *N*th counter in the channel. Max value of *N* is: 16 bit counters: 12 24 bit counters: 8 32 bit counters: 6 if (**index_length** % 8) < 4, or 5 otherwise[16] 48 bit counters: 3 64 bit counters: 2
error*N*	1	Error flag associated with counter *N*.

5.2.11 monitor_counter_response

Generated in response to ***get_monitor_counter***. Payload format is as ***set_monitor_counter***.

5.2.12 monitor_match_data

This message is generated by filters that are configured to do so (see Section 4.6).

Table 5-49 monitor_match_data payload format

Field name	Bits	Description
msg_type	2	Set to 0x2.
reserved	4	Read as zero.
interface	2	Interface with which the match is associated
reserved	4	Read as zero.
match	16	A bit vector reporting what matched. A 1 in bit *N* indicates filter *N* matched.

5.2.13 monitor_response

Generated in response to ***get_monitor***. Payload formats are as per ***set_monitor***.

[15] The width of this field is 16, 24, 32, 48 or 64 as determined by the **counter_width_*X*** ***discovery_response*** field.

[16] index_length is the number of bits used for message indexes in the system – see *An Introduction to the Tessent Embedded Analytics Architecture* (Related reading)

5.2.14 monitor_trace_data

Generated by the trace unit (see Section 3.3.3), or in response to ***monitor_snapshot***. Traced information for bus transactions is spread across multiple messages, as follows:

- Signals for each bus channel are gathered and output separately, and can be identified from the channel field as shown in Table 5-50;
- If the number of signals traced is more than can be accommodated in a single message, it will be spread across multiple tranches.

Table 5-50 Trace Buffer Channel Assignment

Channel	Trace Buffer (AXI)	Trace Buffer (OCP)
0	Read address, *direct* capture	Main port request, *direct* capture
1	Write address, *direct* capture	reserved
2	Read data	reserved
3	Write data	Main port data handshake[17]
4	Write response	Main port response[18]
5	Snoop address, *direct* capture	Intervention port request, *direct* capture
6	Snoop data	Intervention port read data[19]
7	Snoop response	Intervention port response[19]
8	Read address, *deferred* capture	Main port request, *deferred* capture
9	Write address, *deferred* capture	Intervention port request, *deferred* capture
10 – 12	reserved	reserved
13	Snoop address, *deferred* capture	reserved
14 – 15	reserved	reserved

The overall ***monitor_trace_data*** payload format is shown in Table 5-51. Provided the **marker** is not 'flag', the message will contain traced signal values as shown in the following tables. For each buffer, the trace payload size will vary according to whether or not the various input buses are to be traced (as specified by bit 2 of the respective **filter_*_X** ***discovery_response*** field for *direct* capture, or bit 3 for *deferred* capture), and the width of those buses. Depending on the total payload size, multiple tranches may be required to output all the data captured from a single cycle. The payloads are outlined in Table 5-52 through Table 5-64. The final column in these tables shows the ***discovery_response*** field values that indicate whether each field is included or not. The **trace** field is formed by concatenating the included bus signals (there are no read-as-zero bits for signals that aren't included).

[17] Write data is provided via channel 0 (and channel 3 is unused) if the data handshake signal group is unused (***discovery_response*** **ocp_*X*_datahandshake** not set).
[18] Combined read data and response provided via channel 4
[19] Combined read data and response provided via channel 7 (and channel 6 is reserved) unless configured for split transactions (***discovery_response*** **ocp_*X*_intportsplittranx** is set), in which case channel 6 is used for read data and channel 7 for the separate response.

Table 5-51 monitor_trace_data payload format

Field name	Bits	Description
msg_type	2	Set to 0x3.
channel	4	Trace buffer, per Table 5-50.
interface	2	Interface with which the channel is associated.
tranche	4	The number of tranches still to come.
marker	2	Places this trace message within a wider context: 00 (*first*): Start of trace 01 (*middle*): Not start or end of trace 10 (*last*): End of trace 11 (*flag*): Flag message
trace	234[20]	Trace data captured in a single clock cycle. When **marker** = 11, there is no trace data, and this field records status information, according to the value of bits [1:0] (**flag_type**): 00: *No data*; see Table 5-65; 01: *State change*; see Table 5-66; 10: *Transaction state*; see Table 5-67.

Table 5-52 Read and write address trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**[rw]_concurrent_*X*** + 1)	**[rw]_concurrent_*X*** > 0
`a[rw]addr X`	**axi_*X*_addr** + 1	**filter_addr_*X*[3/2]**
`a[rw]bar X`	2	**filter_bar_*X*[3/2]**
`a[rw]burst X`	2	**filter_burst_*X*[3/2]**
`a[rw]cache X`	4	**filter_cache_*X*[3/2]**
`a[rw]domain X`	2	**filter_domain_*X*[3/2]**
`a[rw]id X`	**axi_*X*_id** + 1	**filter_id_*X*[3/2]**
`a[rw]len X`	4 + (**axi4_*X*** * 4)	**filter_len_*X*[3/2]**
`a[rw]lock X`	2 - **axi4_*X***	**filter_lock_*X*[3/2]**
`a[rw]prot X`	3	**filter_prot_*X*[3/2]**
`a[rw]qos X`	4	**filter_qos_*X*[3/2]**
`a[rw]region X`	4	**filter_region_*X*[3/2]**
`a[rw]size X`	3	**filter_size_*X*[3/2]**
`a[rw]snoop X`	4 (read) or 3 (write)	**module_variant** = 102 – 109
`awunique X`	1	**filter_unique_X[3/2]** (write only)
`a[rw]user X`	**axi_*X*_a[rw]user** + 1	**filter_a[rw]user_*X*[3/2]**

[20] If the total width of all traceable signals is greater than 234 bits, this is split across multiple tranches. If the downstream message interface is 64-bits or wider (***discovery_response*** **ds_msg_size** ≥ 6), the width of this field is limited to 210 bits in order to achieve better utilization of the message interface.
[21] If this is 0, the field is not included (i.e. width is effectively treated as 0).

Table 5-53 Write data trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**w_concurrent_*X*** + 1)	**w_concurrent_*X*** > 0
`wdata` *X*	decode(**axi_*X*_data**)[22]	**filter_data_*X*[2]**
`wid` *X*	**axi_*X*_id** + 1	**(filter_id_*X*[2]** | **filter_id_*X*[3])** & !**axi4_*X***
`wlast` *X*	1	1
`wstrb` *X*	decode(**axi_*X*_data**)/8	**filter_data_*X*[2]**
`wuser` *X*	**axi_*X*_wuser** + 1	**filter_wuser_*X*[2]**

Table 5-54 Read data trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(r**_concurrent_*X*** + 1)	**r_concurrent_*X*** > 0
`rdata` *X*	decode(**axi_*X*_data**)[22]	**filter_data_*X*[2]**
`rid` *X*	**axi_*X*_id** + 1	**filter_id_*X*[2]** | **filter_id_*X*[3]**
`rlast` *X*	1	1
`rresp` *X*	2 (**module_variant** = 98 - 105), 4 otherwise	**filter_resp_*X*[2]**
`ruser` *X*	**axi_*X*_ruser** + 1	**filter_ruser_*X*[2]**

Table 5-55 Response trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**w_concurrent_*X*** + 1)	**w_concurrent_*X*** > 0
`bid` *X*	**axi_*X*_id** + 1	**filter_id_*X*[2]** | **filter_id_*X*[3]**
`bresp` *X*	2	**filter_resp_*X*[2]**
`buser` *X*	**axi_*X*_buser** + 1	**filter_buser_*X*[2]**

[22] See ***discovery_response*** Table 5-44.

Table 5-56 Snoop address trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**s_concurrent_*X*** + 1)	**s_concurrent_*X*** > 0
`acaddr X`	**axi_*X*_addr** + 1	**filter_addr_*X*[3/2]**
`acprot_ip X`	3	**filter_prot_*X*[3/2]**
`acsnoop X`	4	1

Table 5-57 Snoop data trace payload interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**s_concurrent_*X*** + 1)	**s_concurrent_*X*** > 0
`cddata X`	decode(**axi_*X*_data**) [22]	**filter_data_*X*[2]**
`cdlast X`	1	1

Table 5-58 Snoop response trace payload for interface *X* (AXI)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**s_concurrent_*X***+ 1)	**s_concurrent_*X*** > 0
`crresp X`	5	**filter_resp_*X*[2]**

Table 5-59 Main port request phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**concurrent_*X*** + 1)	**concurrent_*X*** > 0
maddr *X*	**ocp_*X*_maddr** + 1	**filter_addr_*X*[3/2]**
maddrspace *X*	**ocp_*X*_maddrspace** + 1	**filter_addrspace_*X*[3/2]**
mblockheight *X*	**ocp_*X*_blockheight** + 1	**filter_addr_*X*[3/2]** & **ocp_*X*_burstblock**
mblockstride *X*	**ocp_*X*_blockstride** + 1	**filter_addr_*X*[3/2]** & **ocp_*X*_burstblock**
mburstlength *X*	**ocp_*X*_burstlength** + 1	**filter_burstlength_*X*[3/2]**
mburstseq *X*	3	**filter_burst_*X*[3/2]**
mburstsinglereq *X*	1	**filter_burst_*X*[3/2]**
mbyteen *X*	(**ocp_*X*_data** + 1)/8	**filter_data_*X*[3/2]** & **ocp_*X*_byteen**
mcmd *X*	3 (**module_variant** = 50 – 53) 5 (**module_variant** = 85 – 88) 5 (**module_variant** = 93 – 96)	1
mcohcmd *X*	1	**filter_cohcmd_*X*[3/2]**
mcohfwdid *X*	**ocp_*X*_cohfwdid** + 1	**filter_cohfwdid_*X*[3/2]**
mcohid *X*	**ocp_*X*_cohid** + 1	**filter_cohid_*X*[3/2]**
mconnid *X*	**ocp_*X*_connid** + 1	**filter_connid_*X*[3/2]**
mdata *X*	**ocp_*X*_data** + 1	**filter_data_*X*[3/2]** & !**ocp_*X*_datahandshake**
mdatainfo *X*	**ocp_*X*_mdatainfo** + 1	**filter_mdatainfo_*X*[3/2]** & !**ocp_*X*_datahanshake**
mreqinfo *X*	**ocp_*X*_mreqinfo** + 1	**filter_mreqinfo_*X*[3/2]**
mtagid *X*	**ocp_*X*_tagid** + 1	**filter_tagid_*X*[3/2]**
mthreadid *X*	**ocp_*X*_threadid** + 1	**filter_threadid_*X*[3/2]**

Table 5-60 Data handshake phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**concurrent_*X*** + 1)	**concurrent_*X*** > 0
mdata *X*	**ocp_*X*_data** + 1	**filter_data_*X*[2]**
mdatabyteen *X*	(**ocp_*X*_data** + 1)/8	**filter_data_*X*[2] & ocp_X_databyteen**
mdatainfo *X*	**ocp_*X*_mdatainfo** + 1	**filter_mdatainfo_*X*[2]**
mdatalast *X*	1	**ocp_X_mdatalast**
mtagid *X*	**ocp_*X*_tagid** + 1	**filter_tagid_*X*[2]** | **filter_tagid_*X*[3]**
mthreadid *X*	**ocp_*X*_threadid** + 1	**filter_threadid_*X*[2]** | **filter_threadid_*X*[3]**

Table 5-61 Main port response phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**concurrent_*X*** + 1)	**concurrent_*X*** > 0
`scohfwdid X`	**ocp_*X*_cohfwdid** + 1	**filter_cohfwdid_*X*[2]**
`scohid X`	**ocp_*X*_cohid** + 1	**filter_cohid_*X*[2]**
`scohstate X`	3	**filter_cohstate_*X*[2]**
`sdata X`	**ocp_*X*_data** + 1	**filter_data_*X*[2]**
`sdatainfo X`	**ocp_*X*_sdatainfo** + 1	**filter_sdatainfo_*X*[2]**
`sresp X`	2 (**module_variant** = 50 – 53) 3 (**module_variant** = 85 – 88) 3 (**module_variant** = 93 – 96)	**filter_sresp_*X*[2]**
`srespinfo X`	**ocp_*X*_srespinfo** + 1	**filter_srespinfo_*X*[2]**
`sdatalast X`	1	**ocp_*X*_sdatalast**
`stagid X`	**ocp_*X*_tagid** + 1	**filter_tagid_*X*[2]** | **filter_tagid_*X*[3]**
`sthreadid X`	**ocp_*X*_threadid** + 1	**filter_threadid_*X*[2]** | **filter_threadid_*X*[3]**

Table 5-62 Intervention port request phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**i_concurrent_*X*** + 1)	**i_concurrent_*X*** > 0
`imaddr X`	**ocp_*X*_maddr** + 1	**filter_addr_*X*[3/2]**
`imaddrspace X`	**ocp_*X*_maddrspace** + 1	**filter_addrspace_*X*[3/2]**
`imburstlength X`	**ocp_*X*_burstlength** + 1	**filter_burstlength_*X*[3/2]**
`imburstseq X`	3	**filter_burst_*X*[3/2]**
`imcmd X`	5	1
`imcohfwdid X`	**ocp_*X*_cohfwdid** + 1	**filter_cohfwdid_*X*[3/2]**
`imcohid X`	**ocp_*X*_cohid** + 1	**filter_cohid_*X*[3/2]**
`imconnid X`	**ocp_*X*_connid** + 1	**filter_connid_*X*[3/2]**
`imreqinfo X`	**ocp_*X*_mreqinfo** + 1	**filter_mreqinfo_*X*[3/2]**
`imreqself X`	1	**filter_mreqself_*X*[3/2]**
`imtagid X`	**ocp_*X*_tagid** + 1	**filter_tagid_*X*[3/2]**
`imthreadid X`	**ocp_*X*_threadid** + 1	**filter_threadid_*X*[3/2]**

Table 5-63 Intervention port response phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**i_concurrent_*X*** + 1)	**i_concurrent_*X*** > 0
`iscohfwdid X`	**ocp_*X*_cohfwdid** + 1	**filter_cohfwdid_*X*[2]**
`iscohid X`	**ocp_*X*_cohid** + 1	**filter_cohid_*X*[2]**
`iscohstate X`	3	**filter_cohstate_*X*[2]**
`isdata X`	**ocp_*X*_data** + 1	**filter_data_*X*[2]** & !**ocp_*X*_intportsplittranx**
`sdatainfo X`	**ocp_*X*_sdatainfo** + 1	**filter_sdatainfo_*X*[2]** & !**ocp_*X*_intportsplittranx**
`isresp X`	3	**filter_sresp_*X*[2]**
`isrespinfo X`	**ocp_*X*_srespinfo** + 1	**filter_srespinfo_*X*[2]**
`isresplast X`	1	!**ocp_*X*_intportsplittranx**
`istagid X`	**ocp_*X*_tagid** + 1	**filter_tagid_*X*[2]** | **filter_tagid_*X*[3]**
`isthreadid X`	**ocp_*X*_threadid** + 1	**filter_threadid_*X*[2]** | **filter_threadid_*X*[3]**

Table 5-64 Intervention port read data phase trace payload for interface *X* (OCP)

Bus signal	Bits	Enabling condition[21]
Bus clock precise timetag extension (not a signal)	**trace_ptime_width_*X***	**trace_ptime_width_*X*** > 0
Transaction differentiator (not a signal)	1	1
Tracker transaction index (not a signal)	$\log_2$(**i_concurrent_*X*** + 1)	**i_concurrent_*X*** > 0
`iscohfwdid X`	**ocp_*X*_cohfwdid** + 1	**filter_cohfwdid_*X*[2]**
`iscohid X`	**ocp_*X*_cohid** + 1	**filter_cohid_*X*[2]**
`iscohstate X`	3	**filter_cohstate_*X*[2]**
`isdata X`	**ocp_*X*_data** + 1	**filter_data_*X*[2]** & !**ocp_*X*_intportsplittranx**
`sdatainfo X`	**ocp_*X*_sdatainfo** + 1	**filter_sdatainfo_*X*[2]** & !**ocp_*X*_intportsplittranx**
`isdatalast X`	1	1
`istagid X`	**ocp_*X*_tagid** + 1	**(filter_tagid_*X*[2]** | **filter_tagid_*X*[3])** & !**ocp_X_intportsplittranx**
`isthreadid X`	**ocp_*X*_threadid** + 1	**(filter_threadid_*X*[2]** | **filter_threadid_*X*[3])** & !**ocp_X_intportsplittranx**

When the **marker** field is 'flag', the payload for all of the channels takes on one of three different formats:

Table 5-65 Flag (no data) trace payload

Field name	Bits	Description
flag_type	2	Set to 00 to indicate this is a *no data* flag type.
reserved	rest	Remaining payload bits all zero

Table 5-66 Flag (status change) trace payload

Field name	Bits	Description
flag_type	2	Set to 01 to indicate this is a *status change* flag type.
enable	1	Trace is enabled and not throttled when 1.
history_mode	2	Reports the current ***set_monitor_trace*** **history_mode**.
reserved	rest	Remaining payload bits all zero

Table 5-67 Flag (transaction state) trace payload

Field name	Bits	Description
flag_type	2	Set to 10 to indicate this is a *transaction state* flag type
index	See[23]	Tracker transaction index
aaccepted	1	Address/request phase completed when 1
daccepted_1st	1	First data phase completed when 1
daccepted_last	1	Last data phase completed when 1
raccepted	1	Response phase completed when 1
reserved	rest	Remaining payload bits all zero

5.2.15 monitor_trace_response

Generated in response to ***get_monitor_trace***. Payload format is as ***set_monitor_trace***.

5.2.16 msg_params_response

Generated in response to ***get_msg_params***.

Table 5-68 message_params_response payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x1c.
msg_index	24	The number of message indexes needed by the hierarchy.
instance_id	16	User-defined identifier for the module instance.

5.2.17 period_response

Generated in response to ***get_period***. Payload format is as per ***set_period***.

5.2.18 power_response

Generated in response to ***get_power***. Payload format is per ***set_power***.

[23] Number of bits will be the same as for the transaction tracker when **marker** is not 'flag' (e.g. $\log_2$(**w_concurrent_*X***), etc.)

5.2.19 reset_response

Generated when all actions resulting from ***debug_reset*** have been completed.

Table 5-69 reset_response payload format

Field name	Bits	Description
msg_type	2	Set to 0x0.
control_code	6	Set to 0x19.

5.2.20 return

Generated in response to receipt of an unsupported or unrecognised message, or when the message addresses functionality that is not implemented. The return response includes a return code and the message's first two payload bytes.

Table 5-70 return message payload format

Field name	Bits	Details
msg_type	2	Set to 0x0.
control_code	6	Set to 0x03.
return_code	8	The reason for returning the message: 0x00: Unsupported: The message control code is unsupported/unrecognised (i.e any system control message not listed in Table 5-1 or Table 5-40). 0x05: Invalid sub-unit: A module sub-unit addressed by a field within the message does not exist. For example, a ***set_monitor_counter*** message that selects counter 8 when ***discovery_response*** **counters_*X*** is 6. 0x06: Invalid select: A message sub-format select field value is invalid. For example, a ***set_bus_match_data*** message where the select field is one of the reserved values.
returned_header	8	Zero
returned_payload	16	The first two bytes of the returned message payload.