COMP3301 A2 Device Interface

COMP3301 S2 2023

Version $Revision: 493 $

Table of Contents

Notation used in this document

Unless otherwise noted, structure diagrams in this document show offsets from the beginning of the structure in hexadecimal bytes along the left hand side, and offsets from the beginning of each row in bytes along the top.

The structure diagrams shown in e.g. the PCI interface or Descriptor rings sections all have 64-bit (8-byte) rows.

1. Introduction

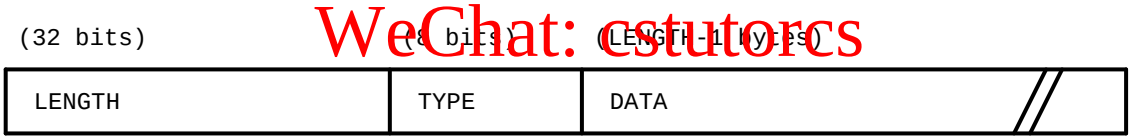
The COMP3301 A2 device is a virtual PCI device for transporting `ssh-agent` commands and responses to the VM hypervisor.

It supports:

- A simple descriptor ring DMA structure
- Dual-interrupt with 64-bit MSI-X support, with always-enabled coalescing
- Multiple concurrent operations

2. SSH agent messages

The `ssh-agent` messages which are transported via the device consist of a header followed by message-specific data. This is true of both command messages sent by the client and reply messages sent by the agent.



The header contains a 32-bit unsigned `LENGTH` field, which is the total length of the message excluding the bytes of the `LENGTH` field itself.

Following this is an 8-bit unsigned `TYPE` field containing the message type. Possible values for the message type are defined in `authfd.h` in the OpenSSH code and in the IETF draft `draft-miller-ssh-agent-04`. Some important examples:

Numeric value	Macro name	Purpose
11	`SSH2_AGENTC_REQUEST_IDENTITIES`	Request list of keys held in agent
12	`SSH2_AGENT_IDENTITIES_ANSWER`	Reply with list of keys held
13	`SSH2_AGENTC_SIGN_REQUEST`	Request for agent to sign some data with a key
14	`SSH2_AGENT_SIGN_RESPONSE`	Reply with calculated signature
5	`SSH_AGENT_FAILURE`	Reply indicating a generic failure

Numeric value	Macro name	Purpose
6	SSH_AGENT_SUCCESS	Reply indicating a generic success

The DATA field with the remaining contents of each message is opaque to the A2 device and its driver. The LENGTH and TYPE fields, however, are not.

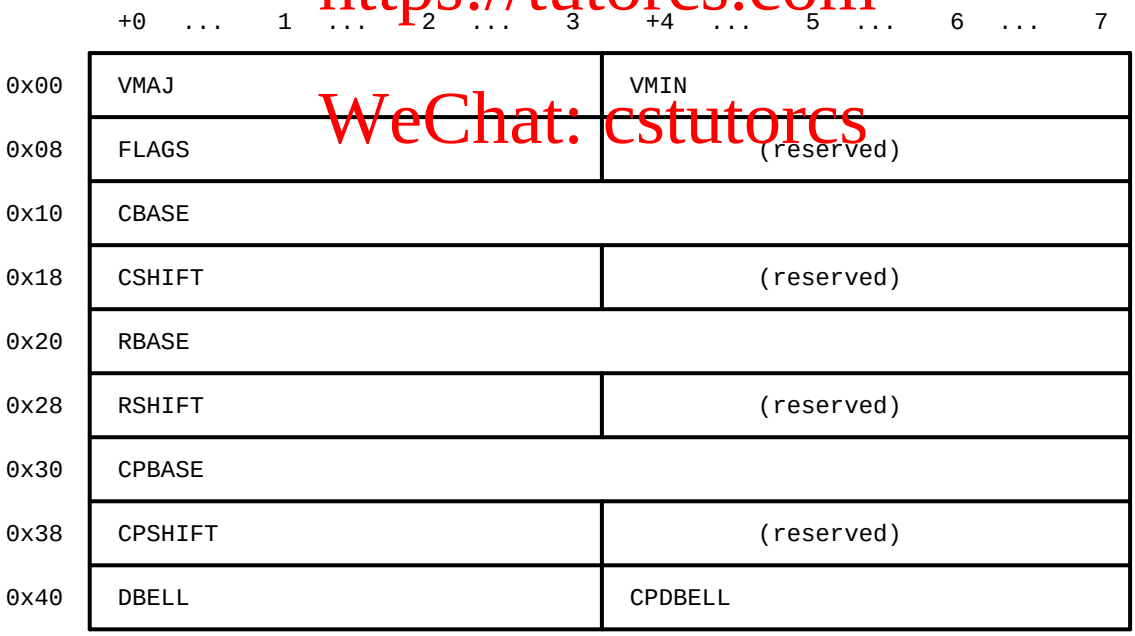
3. PCI interface

All COMP3301 devices use the PCI Vendor ID 0x3301. The A2 device should be recognised by its use of the device ID 0x0200 in the PCI configuration space Type-0 header.

The A2 device has two PCI BARs:

BAR index	Type-0 offset	Type	Contents
0	0x10	MEM64	Main memory BAR for operating the device
1	0x18	MEM32	MSI-X configuration table

The main BAR, of length 0x80, has the register structure shown below. All other bytes of the BAR not included in this diagram are reserved for future use (RFU). Reserved bytes, including those shown on the diagram must not be read from or written to by the device driver.



The registers VMAJ and VMIN are 32-bit, read-only, and contain the specification major and minor versions respectively. This specification is major 1, minor 0. You may use the content of this specification to communicate with any device with the same major number as this specification, and any minor number at least as high.

The register FLAGS contains error flags related to invalid operating conditions for the device. The section Error flags describes the possible values of this register. Once such a condition has been

reached, the device will fire MSI-X interrupt vector 1 (the second vector), then it will stop and need to be reset, which is described in Resetting the device.

The registers `CBASE` and `CSHIFT` contain the base linear (physical) address of the command descriptor ring structure for the device, and the bit shift to calculate its size in number of descriptors (which must be a power of 2). These are to be written by the device driver. See the section Descriptor rings for more details.

The registers `RBASE` and `RSHIFT` contain the base linear (physical) address of the reply descriptor ring structure for the device, and the bit shift to calculate its size. Like `CBASE` and `CSHIFT`, these are to be written by the device driver. See the section Descriptor rings for more details.

The registers `CPBASE` and `CPSHIFT` contain the base linear (physical) address of the completion descriptor ring structure for the device, and the bit shift to calculate its size. Like `CBASE` and `CSHIFT`, these are to be written by the device driver. See the section Descriptor rings for more details.

The `DBELL` register is written by the device driver to wake up the device when new entries on the command or reply descriptor rings have been written by the driver and are available for device use. This is described further in Enqueuing a new command and Retrieving a reply. The high bit (`0x80000000`) of the `DBELL` register must be set when writing the ring index of a reply descriptor, and unset when writing the ring index of a command descriptor.

Finally, the `CPDBELL` register is written by the device driver to notify the device that it has consumed entries on the completion ring and these are now available for re-use. Similar to `DBELL`, it is written with the ring index of the most recent entry consumed by the driver. The section Handling a completion has more detail.

All multi-byte integers processed by the device are stored in little-endian byte order. Integers within the `ssh-agent` protocol data payload use the rules for the `ssh-agent` protocol.

4. Flow of operation

At startup, the driver should check the contents of the `VMAJ` and `VMIN` fields for device compatibility.

Then, it should allocate linear contiguous memory for the command, reply and completion rings, and initialise all descriptors in each.

After initialising the descriptors, it should write the `CBASE`/`CSHIFT`, `RBASE`/`RSHIFT` and `CPBASE`/`CPSHIFT` registers in the BAR. This will begin device operation.

To enqueue a command, the driver should first make certain there are sufficient buffers available in the reply ring for the device to use.

Then it should write the command into the next available command ring descriptor and change its `OWNER` value, then write the ring index of the descriptor into the `DBELL` register.

Once the command has been fully issued and the buffers can be released, a command completion entry in the completion ring will be written, and the device will trigger an interrupt on MSI-X vector 0.

At each completion interrupt, the driver should look at the next completion ring descriptor to see if it has changed to host ownership. If it has, then it contains a valid completion which the driver should process. It continues processing and checking any subsequent completions available until it finds one set to device ownership. Then it writes the `CPDBELL` register to indicate that it has consumed the entries.

Once the command has finished executing and a reply is available, the device will write a reply completion entry in the completion ring and trigger its interrupt once again. At this point, the reply data can be read from the buffers supplied in the reply ring entry.

5. Descriptor rings

The descriptor rings for the A2 device are a region of contiguous linear memory allocated by the device driver. Each ring's address is written into the `CBASE`, `RBASE` or `CPBASE` register in BAR 0, and its size (as a bit shift value) is written into `CSHIFT`, `RSHIFT` or `CPSHIFT`.

Each ring consists of a power of two number of descriptors. For the command and reply rings, each entry is 64 bytes in length. For the completion ring, each entry is 32 bytes in length. The device driver is expected to initialise the entire ring before writing to the relevant `BASE` register (see Initial descriptor state below).

The size of the ring is written into the `SHIFT` register as a bit shift value. For example, the driver has allocated a command ring made up of a single 4k page, which has 64 descriptors. The driver would write the number `6` into the `CSHIFT` register for this ring, as `1 << 6 == 64`.

The `OWNER` field of each descriptor is set to indicate whether the host or device is currently in charge of each descriptor. A descriptor with `OWNER` set to the device ownership value on the command ring indicates a command ready for execution. On the completion ring, a descriptor with `OWNER` set to the host ownership value indicates a completion ready for the driver to pick up.

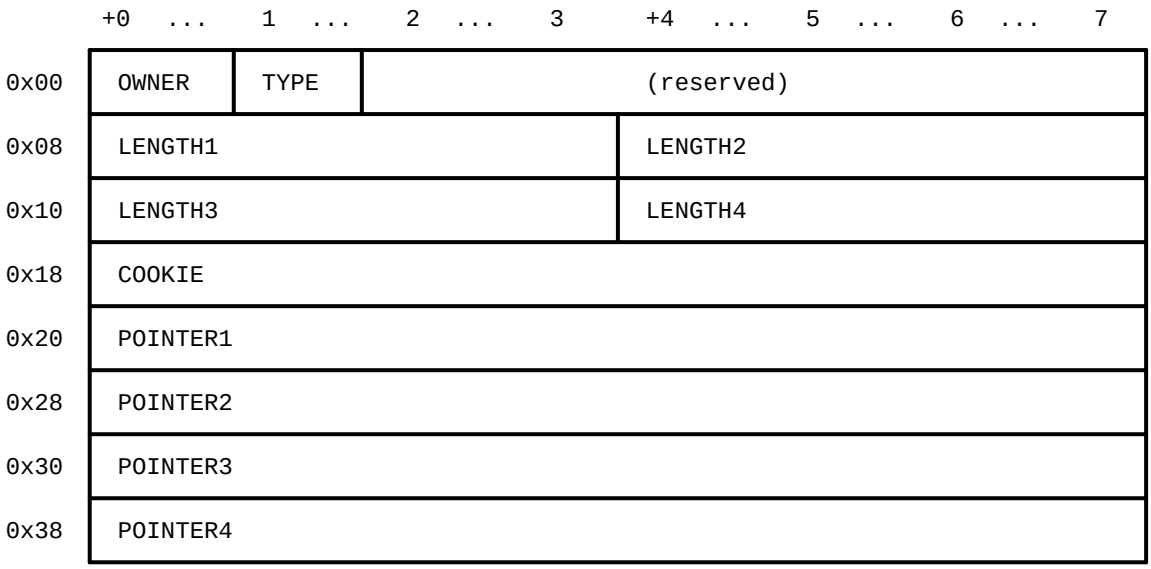
Descriptors in the ring are used in ascending order beginning at index 0 (the first descriptor). It is a circular buffer, meaning that when the end of the ring is reached, the producer "wraps" around to index 0 again (hence the term "ring"). The consumer is only ever expected to read from the next descriptor index after the last one they consumed.

On the command ring, as soon as a descriptor has been set to device ownership, the device may begin executing the command immediately (before any write to the `DBELL` register in BAR 0, since the device may be polling the ring). It is important to ensure that all writes to other fields of the descriptor are flushed out to memory before writing to the `OWNER` field.

Note that both command and reply descriptors may be completed out of order: entries in the completion ring are not guaranteed to come in the same order that the command and reply descriptors they were generated from were written, and replies may not be paired with a command added at the same time.

5.1. Command and reply descriptor

The descriptor structure for command and reply rings is shown below:



The `OWNER` field should be set either to the value `DEVICE_OWNER` (`0xAA`) or `HOST_OWNER` (`0x55`).

The `TYPE` field contains the `ssh-agent` message type (see SSH agent messages). It is unused on the reply ring (the message type is written into the completion ring entry instead).

The `COOKIE` field contains an opaque 64-bit unsigned number which is chosen by the device driver. When a completion is generated for a specific command or reply, the relevant `COOKIE` field on the completion descriptor will be set to match the one provided by the driver with the command or reply. This value should be used to correlate commands and replies with each other and with their completions.

The `LENGTH` and `POINTER` fields are scatter or gather pointers to buffers containing the command or reply data. Some commands and replies may not have any additional data, in which case all of the values of the `LENGTH` fields will be zero.

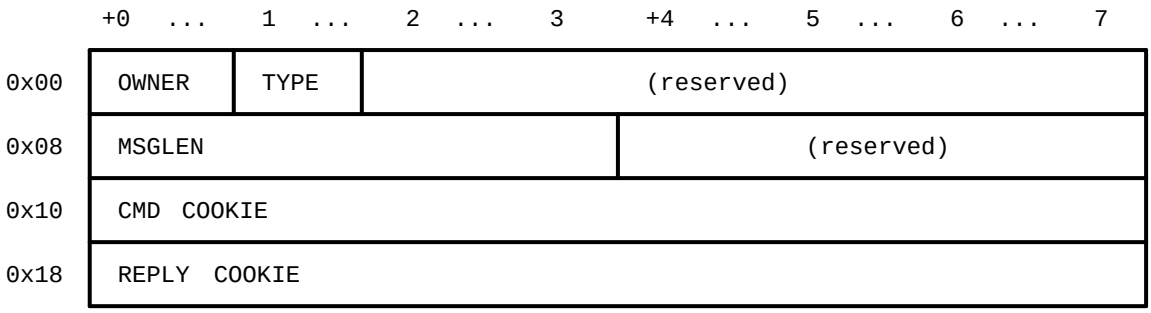
All pointer fields in descriptor rings contain linear (physical) addresses.

The scatter/gather pointers will be used in the order shown (`POINTER1`, then `POINTER2`, etc). The `LENGTH` fields for any unused pointers should be set to zero.

On the reply ring, descriptors ready for fulfillment must be given to the device with at least one scatter pointer.

5.2. Completion descriptor

The descriptor structure for the completion ring is shown below:



The `OWNER` field should be set either to the value `DEVICE_OWNER` (`0xAA`) or `HOST_OWNER` (`0x55`).

Two types of completion descriptor are possible: a command-only completion, which has `TYPE`, `MSGLEN` and `REPLY COOKIE` all set to 0, and only the `CMD COOKIE` filled out; and a reply completion, which has all fields set.

A given command always results in two completions: first a command-only completion at the point where the command is finished with its input data; and then a reply completion when the command has completely finished executing.

A reply completion with `MSGLEN = 0` is valid and indicates that no payload data was written to the reply buffers in question. A reply descriptor will still be consumed and its cookie copied into `REPLY COOKIE`, even though no data has been written to its payload buffers.

The `TYPE` field contains the `ssh-agent` message type (see SSH agent messages).

The `MSGLEN` field on a reply completion contains the total length of the received message. This should be used by the driver to compute how much of the provided reply buffers were written to by the device.

The `CMD COOKIE` field on both reply and command completions is written with the value of the `COOKIE` field from the command descriptor.

The `REPLY COOKIE` field on reply completions is written with the value of the `COOKIE` field on the reply descriptor. The `CMD COOKIE` and `REPLY COOKIE` fields do not have to match.

5.3. Initial descriptor state

The host device driver must initialise the command and reply descriptor rings by:

1. Setting the `OWNER` field on each descriptor to `HOST_OWNER`, and
2. Filling all other bytes with zero

It must also initalise the completion ring by:

1. Setting the `OWNER` field on each descriptor to `DEVICE_OWNER`, and
2. Filling all other bytes with zero

Initialisation of the rings must be done prior to writing the `BASE` and `SHIFT` registers.

5.4. Enqueuing a new command

Suppose the host driver wants to send a command, of type `11`. The command has a body payload, which is located at linear address `0xabcd1200` and is of length `0x10`. The next free descriptor in the command ring has index `5`

To enqueue the new command, the driver must take the following steps:

1. Locate the next free descriptor in the command ring
2. Check that the descriptor's owner field is set to `HOST_OWNER`
3. Generate a unique value for the `COOKIE` field so the driver can match the reply up later
4. Write the fields of the descriptor other than `OWNER`: `TYPE = 11`, `LENGTH1 = 0x10`, `POINTER1 = 0xabcd1200`, `COOKIE` and all the other `LENGTH` fields to zero
5. Perform a store memory barrier (if needed for the compiler/CPU in use) to prevent the previous stores crossing the next one
6. Write the `OWNER` field to `DEVICE_OWNER`
7. Perform a store memory barrier again
8. Write the index of the descriptor (`0x5`) to the `DBELL` register in the BAR

Once the command has been processed, the device will produce a command completion entry on the completion ring and trigger a MSI. Once the command completion entry has been read, the command data buffers may be released for other use (the device will not read from them again).

5.5. Retrieving a reply

Suppose the host driver is ready to receive a reply to a command. It has allocated two 4k pages for receiving up to 8k of data, located at `0xabcd1000` and `0xabcd5000` (they are linearly discontinuous). The next free descriptor in the reply ring has index `0x31`

To use this buffer to receive a response, the driver must take the following steps:

1. Locate the next free descriptor in the reply ring
2. Check that the descriptor's owner field is set to `HOST_OWNER`
3. Write the fields of the descriptor other than `OWNER`: `LENGTH1 = 0x1000`, `POINTER1 = 0xabcd1000`, `LENGTH2 = 0x1000`, `POINTER2 = 0xabcd5000`, all other `LENGTH` fields to zero
4. Perform a store memory barrier (if needed for the compiler/CPU in use) to prevent the previous stores crossing the next one
5. Write the `OWNER` field to `DEVICE_OWNER`
6. Perform a store memory barrier again
7. Write the index of the descriptor with the high bit set (`0x80000031`) to the `DBELL` register in the BAR

Then the driver must wait for a completion of this reply descriptor.

5.6. Handling a completion

Suppose that the driver has not handled a completion queue entry yet, and this is the first interrupt it received.

The driver must take the following steps:

1. Wait for an MSI on vector 0 from the device
2. Check the first completion ring descriptor's `OWNER` field has changed to `HOST_OWNER`
3. Perform a load memory barrier to prevent the load of the `OWNER` field crossing any subsequent loads
4. Read the `TYPE` and `MSGLEN` fields of the completion to determine the type of reply, and how much data was written into each of the buffers (in case of a reply completion)
5. Use the `CMD_COOKIE` field to correlate with the original command as needed
6. Use the `REPLY_COOKIE` field on reply completions to correlate the completion back to the reply descriptor entry and its buffers
7. Return the completion descriptor to device ownership by setting `OWNER` to `DEVICE_OWNER`
8. Check the subsequent completion ring descriptor's `OWNER` field and repeat until the next is marked `DEVICE_OWNER`
9. Write the index of the last completion ring descriptor it processed into the `CPDBELL` register (if only the first descriptor was valid in our example, then we would write `0`)

To prevent ring overflow (`OVF`) it is also recommended to periodically update `CPDBELL` while in the loop at step 8, if a significant number of completions are being processed.

6. Interrupt vectors

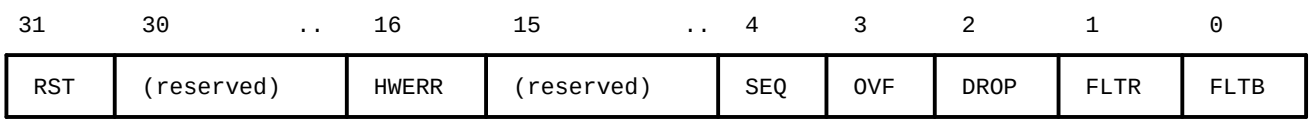
The device supports two MSI-X interrupt vectors:

MSI vector	When triggered
0	New completion ring entries available
1	Fatal error; bits in the `FLAGS` register can be used to diagnose

Legacy PCI interrupts and MSI are not supported.

7. Error flags

The `FLAGS` field in the BAR has the following bit structure:



- `FLTB` — the device encountered a page fault, bus fault or access violation while chasing a pointer provided by the driver in the BAR (e.g. `CBASE`)
- `FLTR` — the device encountered a page fault, bus fault or access violation while chasing a pointer provided by the driver in a descriptor ring
- `DROP` — the device dropped a reply due to insufficient availability of reply buffers
- `OVF` — the device failed to write a completion, because the next completion ring entry had not been changed back to `OWNER = DEVICE_OWNER` or the `CPDBELL` register had not been updated
- `SEQ` — the device detected an operation out of sequence (e.g. a write to the `DBELL` register before the ring base and shift registers have been set to valid values)
- `HWERR` — a miscellaneous hardware error occurred
- `RST` — set by the driver on write as part of the reset procedure (see Resetting the device). Always reads as zero.

Once any bit in the `FLAGS` register is set, the device will stop, and a reset will be required to resume service (see Resetting the device).

Encountering any condition which sets a bit in `FLAGS` will also trigger MSI-X interrupt vector 1, if enabled. This is a separate interrupt vector to that used for normal completion traffic.

As well as checking this register in the handler for MSI-X vector 1, a defensively written host driver could also perform a periodic check of the register, or check it at any time when it detects a "hang" where operations are not completing as normal.

Writes to the `FLAGS` register are allowed only as part of the reset procedure, and must be full 32-bit writes. Only the `RST` bit can be written: all other bits in a write will be ignored.

8. Resetting the device

A full device reset can be carried out in case of error. The steps required are:

1. Write the value `0x80000000` (`RST = 1`) to the `FLAGS` register in the BAR
2. Busy-poll the `FLAGS` register until its entire 32-bit value returns to `0`

To account for a hardware or bus fault, the driver may wish to limit the amount of time it busy-polls the register (or fall back to a periodic timer after some count of iterations).

All outstanding operations and DMA by the device will be abandoned, but some in-flight operations may still complete between the write to `FLAGS` and the point in time where it reads as `0` (and these may result in an MSI on vector 0 being issued). Once the `FLAGS` register has read `0`, all outstanding operations have either completed or been abandoned.