# Surface Serial Hub Protocol

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 3**)

Unknown interpreted text role "c:type".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 3**)

Substitution definition contains illegal element <problematic>:

```
<problematic ids="id2" refid="id1">
    :c:type:`u8 <u8>`

.. |u8| replace:: :c:type:`u8 <u8>`
```

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 4**)

Unknown interpreted text role "c:type".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 4**)

Substitution definition contains illegal element <problematic>:

```
<problematic ids="id4" refid="id3">
    :c:type:`u16 <u16>`

.. |u16| replace:: :c:type:`u16 <u16>`
```

The Surface Serial Hub (SSH) is the central communication interface for the embedded Surface Aggregator Module controller (SAM or EC), found on newer Surface generations. We will refer to this protocol and interface as SAM-over-SSH, as opposed to SAM-over-HID for the older generations.

On Surface devices with SAM-over-SSH, SAM is connected to the host via UART and defined in ACPI as device with ID `MSHW0084`. On these devices, significant functionality is provided via SAM, including access to battery and power information and events, thermal read-outs and events, and many more. For Surface Laptops, keyboard input is handled via HID directed through SAM, on the Surface Laptop 3 and Surface Book 3 this also includes touchpad input.

Note that the standard disclaimer for this subsystem also applies to this document: All of this has been reverse-engineered and may thus be erroneous and/or incomplete.

All CRCs used in the following are two-byte `crc_ccitt_false(0xffff, ...)`. All multi-byte values are little-endian, there is no implicit padding between values.

## SSH Packet Protocol: Definitions

The fundamental communication unit of the SSH protocol is a frame (:c:type:`struct ssh_frame <ssh_frame>`). A frame consists of the following fields, packed together and in order:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 49**); *backlink*

Unknown interpreted text role "c:type".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 53**)

Unknown directive type "flat-table".

```
.. flat-table:: SSH Frame
   :widths: 1 1 4
   :header-rows: 1

   * - Field
     - Type
     - Description

   * - |TYPE|
     - |u8|
     - Type identifier of the frame.

   * - |LEN|
     - |u16|
     - Length of the payload associated with the frame.

   * - |SEQ|
     - |u8|
     - Sequence ID (see explanation below).
```

Each frame structure is followed by a CRC over this structure. The CRC over the frame structure (`TYPE`, `LEN`, and `SEQ` fields) is placed directly after the frame structure and before the payload. The payload is followed by its own CRC (over all payload bytes). If the payload is not present (i.e. the frame has `LEN=0`), the CRC of the payload is still present and will evaluate to `0xffff`. The `LEN` field does not include any of the CRCs, it equals the number of bytes inbetween the CRC of the frame and the CRC of the payload.

Additionally, the following fixed two-byte sequences are used:

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, line 84)**

Unknown directive type "flat-table".

```
.. flat-table:: SSH Byte Sequences
   :widths: 1 1 4
   :header-rows: 1

   * - Name
     - Value
     - Description

   * - |SYN|
     - ``[0xAA, 0x55]``
     - Synchronization bytes.
```

A message consists of `SYN`, followed by the frame (`TYPE`, `LEN`, `SEQ` and CRC) and, if specified in the frame (i.e. `LEN > 0`), payload bytes, followed finally, regardless if the payload is present, the payload CRC. The messages corresponding to an exchange are, in part, identified by having the same sequence ID (`SEQ`), stored inside the frame (more on this in the next section). The sequence ID is a wrapping counter.

A frame can have the following types (:c:type:`enum ssh_frame_type <ssh_frame_type>`):

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, line 103); *backlink***

Unknown interpreted text role "c:type".

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, line 106)**

Unknown directive type "flat-table".

```
.. flat-table:: SSH Frame Types
   :widths: 1 1 4
   :header-rows: 1

   * - Name
     - Value
     - Short Description
```

```
      * - |NAK|
        - ``0x04``
        - Sent on error in previously received message.

      * - |ACK|
        - ``0x40``
        - Sent to acknowledge receival of |DATA| frame.

      * - |DATA_SEQ|
        - ``0x80``
        - Sent to transfer data. Sequenced.

      * - |DATA_NSQ|
        - ``0x00``
        - Same as |DATA_SEQ|, but does not need to be ACKed.
```

Both `NAK`- and `ACK`-type frames are used to control flow of messages and thus do not carry a payload. `DATA_SEQ`- and `DATA_NSQ`-type frames on the other hand must carry a payload. The flow sequence and interaction of different frame types will be described in more depth in the next section.

## SSH Packet Protocol: Flow Sequence

Each exchange begins with `SYN`, followed by a `DATA_SEQ`- or `DATA_NSQ`-type frame, followed by its CRC, payload, and payload CRC. In case of a `DATA_NSQ`-type frame, the exchange is then finished. In case of a `DATA_SEQ`-type frame, the receiving party has to acknowledge receival of the frame by responding with a message containing an `ACK`-type frame with the same sequence ID of the `DATA` frame. In other words, the sequence ID of the `ACK` frame specifies the `DATA` frame to be acknowledged. In case of an error, e.g. an invalid CRC, the receiving party responds with a message containing an `NAK`-type frame. As the sequence ID of the previous data frame, for which an error is indicated via the `NAK` frame, cannot be relied upon, the sequence ID of the `NAK` frame should not be used and is set to zero. After receival of an `NAK` frame, the sending party should re-send all outstanding (non-ACKed) messages.

Sequence IDs are not synchronized between the two parties, meaning that they are managed independently for each party. Identifying the messages corresponding to a single exchange thus relies on the sequence ID as well as the type of the message, and the context. Specifically, the sequence ID is used to associate an `ACK` with its `DATA_SEQ`-type frame, but not `DATA_SEQ`- or `DATA_NSQ`-type frames with other `DATA`- type frames.

An example exchange might look like this:

```
tx: -- SYN FRAME(D) CRC(F) PAYLOAD CRC(P) -----------------------------
rx: ------------------------------------- SYN FRAME(A) CRC(F) CRC(P) --
```

where both frames have the same sequence ID (`SEQ`). Here, `FRAME(D)` indicates a `DATA_SEQ`-type frame, `FRAME(A)` an `ACK`-type frame, `CRC(F)` the CRC over the previous frame, `CRC(P)` the CRC over the previous payload. In case of an error, the exchange would look like this:

```
tx: -- SYN FRAME(D) CRC(F) PAYLOAD CRC(P) -----------------------------
rx: ------------------------------------- SYN FRAME(N) CRC(F) CRC(P) --
```

upon which the sender should re-send the message. `FRAME(N)` indicates an `NAK`-type frame. Note that the sequence ID of the `NAK`-type frame is fixed to zero. For `DATA_NSQ`-type frames, both exchanges are the same:

```
tx: -- SYN FRAME(DATA_NSQ) CRC(F) PAYLOAD CRC(P) ----------------------
rx: -------------------------------------------------------------------
```

Here, an error can be detected, but not corrected or indicated to the sending party. These exchanges are symmetric, i.e. switching `rx` and `tx` results again in a valid exchange. Currently, no longer exchanges are known.

## Commands: Requests, Responses, and Events

Commands are sent as payload inside a data frame. Currently, this is the only known payload type of `DATA` frames, with a payload-type value of `0x80` (:c:type:`SSH_PLD_TYPE_CMD <ssh_payload_type>`).

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, line 195);** ***backlink***

Unknown interpreted text role "c:type".

The command-type payload (:c:type:`struct ssh_command <ssh_command>`) consists of an eight-byte command structure, followed by optional and variable length command data. The length of this optional data is derived from the frame payload length given in the corresponding frame, i.e. it is `frame.len - sizeof(struct ssh_command)`. The command struct contains the following fields, packed together and in order:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 199**); ***backlink***

Unknown interpreted text role "c:type".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\surface_aggregator\[linux-master][Documentation][driver-api][surface_aggregator]ssh.rst`, **line 206**)

Unknown directive type "flat-table".

```
.. flat-table:: SSH Command
   :widths: 1 1 4
   :header-rows: 1

   * - Field
     - Type
     - Description

   * - |TYPE|
     - |u8|
     - Type of the payload. For commands always ``0x80``.

   * - |TC|
     - |u8|
     - Target category.

   * - |TID| (out)
     - |u8|
     - Target ID for outgoing (host to EC) commands.

   * - |TID| (in)
     - |u8|
     - Target ID for incoming (EC to host) commands.

   * - |IID|
     - |u8|
     - Instance ID.

   * - |RQID|
     - |u16|
     - Request ID.

   * - |CID|
     - |u8|
     - Command ID.
```

The command struct and data, in general, does not contain any failure detection mechanism (e.g. CRCs), this is solely done on the frame level.

Command-type payloads are used by the host to send commands and requests to the EC as well as by the EC to send responses and events back to the host. We differentiate between requests (sent by the host), responses (sent by the EC in response to a request), and events (sent by the EC without a preceding request).

Commands and events are uniquely identified by their target category (`TC`) and command ID (`CID`). The target category specifies a general category for the command (e.g. system in general, vs. battery and AC, vs. temperature, and so on), while the command ID specifies the command inside that category. Only the combination of `TC` + `CID` is unique. Additionally, commands have an instance ID (`IID`), which is used to differentiate between different sub-devices. For example `TC=3` `CID=1` is a request to get the temperature on a thermal sensor, where `IID` specifies the respective sensor. If the instance ID is not used, it should be set to zero. If instance IDs are used, they, in general, start with a value of one, whereas zero may be used for instance independent queries, if applicable. A response to a request should have the same target category, command ID, and instance ID as the corresponding request.

Responses are matched to their corresponding request via the request ID (`RQID`) field. This is a 16 bit wrapping counter similar to the sequence ID on the frames. Note that the sequence ID of the frames for a request-response pair does not match. Only the request ID has to match. Frame-protocol wise these are two separate exchanges, and may even be separated, e.g. by an event being sent after the request but before the response. Not all commands produce a response, and this is not detectable by `TC` + `CID`. It is the responsibility of the issuing party to wait for a response (or signal this to the communication framework, as is done in SAN/ACPI via the `SNC` flag).

Events are identified by unique and reserved request IDs. These IDs should not be used by the host when sending a new request. They are used on the host to, first, detect events and, second, match them with a registered event handler. Request IDs for events are chosen by the host and directed to the EC when setting up and enabling an event source (via the enable-event-source request). The EC then uses the specified request ID for events sent from the respective source. Note that an event should still be identified by its target category, command ID, and, if applicable, instance ID, as a single event source can send multiple different event types. In general, however, a single target category should map to a single reserved event request ID.

Furthermore, requests, responses, and events have an associated target ID (`TID`). This target ID is split into output (host to EC) and input (EC to host) fields, with the respecting other field (e.g. output field on incoming messages) set to zero. Two `TID` values are known: Primary (`0x01`) and secondary (`0x02`). In general, the response to a request should have the same `TID` value, however, the field (output vs. input) should be used in accordance to the direction in which the response is sent (i.e. on the input field, as responses are generally sent from the EC to the host).

Note that, even though requests and events should be uniquely identifiable by target category and command ID alone, the EC may require specific target ID and instance ID values to accept a command. A command that is accepted for `TID=1`, for example, may not be accepted for `TID=2` and vice versa.

## Limitations and Observations

The protocol can, in theory, handle up to `U8_MAX` frames in parallel, with up to `U16_MAX` pending requests (neglecting request IDs reserved for events). In practice, however, this is more limited. From our testing (although via a python and thus a user-space program), it seems that the EC can handle up to four requests (mostly) reliably in parallel at a certain time. With five or more requests in parallel, consistent discarding of commands (ACKed frame but no command response) has been observed. For five simultaneous commands, this reproducibly resulted in one command being dropped and four commands being handled.

However, it has also been noted that, even with three requests in parallel, occasional frame drops happen. Apart from this, with a limit of three pending requests, no dropped commands (i.e. command being dropped but frame carrying command being ACKed) have been observed. In any case, frames (and possibly also commands) should be re-sent by the host if a certain timeout is exceeded. This is done by the EC for frames with a timeout of one second, up to two re-tries (i.e. three transmissions in total). The limit of re-tries also applies to received NAKs, and, in a worst case scenario, can lead to entire messages being dropped.

While this also seems to work fine for pending data frames as long as no transmission failures occur, implementation and handling of these seems to depend on the assumption that there is only one non-acknowledged data frame. In particular, the detection of repeated frames relies on the last sequence number. This means that, if a frame that has been successfully received by the EC is sent again, e.g. due to the host not receiving an `ACK`, the EC will only detect this if it has the sequence ID of the last frame received by the EC. As an example: Sending two frames with `SEQ=0` and `SEQ=1` followed by a repetition of `SEQ=0` will not detect the second `SEQ=0` frame as such, and thus execute the command in this frame each time it has been received, i.e. twice in this example. Sending `SEQ=0`, `SEQ=1` and then repeating `SEQ=1` will detect the second `SEQ=1` as repetition of the first one and ignore it, thus executing the contained command only once.

In conclusion, this suggests a limit of at most one pending un-ACKed frame (per party, effectively leading to synchronous communication regarding frames) and at most three pending commands. The limit to synchronous frame transfers seems to be consistent with behavior observed on Windows.