

SAS Layer

The SAS Layer is a management infrastructure which manages SAS LLDDs. It sits between SCSI Core and SAS LLDDs. The layout is as follows: while SCSI Core is concerned with SAM/SPC issues, and a SAS LLDD+sequencer is concerned with phy/OOB/link management, the SAS layer is concerned with:

- SAS Phy/Port/HA event management (LLDD generates, SAS Layer processes),
- SAS Port management (creation/destruction),
- SAS Domain discovery and revalidation,
- SAS Domain device management,
- SCSI Host registration/unregistration,
- Device registration with SCSI Core (SAS) or libata (SATA), and
- Expander management and exporting expander control to user space.

A SAS LLDD is a PCI device driver. It is concerned with phy/OOB management, and vendor specific tasks and generates events to the SAS layer.

The SAS Layer does most SAS tasks as outlined in the SAS 1.1 spec.

The `sas_ha` struct describes the SAS LLDD to the SAS layer. Most of it is used by the SAS Layer but a few fields need to be initialized by the LLDDs.

After initializing your hardware, from the `probe()` function you call `sas_register_ha()`. It will register your LLDD with the SCSI subsystem, creating a SCSI host and it will register your SAS driver with the sysfs SAS tree it creates. It will then return. Then you enable your phys to actually start OOB (at which point your driver will start calling the `notify_*` event callbacks).

Structure descriptions

`struct sas_phy`

Normally this is statically embedded to your driver's phy structure:

```
struct my_phy {
    blah;
    struct sas_phy sas_phy;
    bleh;
};
```

And then all the phys are an array of `my_phy` in your HA struct (shown below).

Then as you go along and initialize your phys you also initialize the `sas_phy` struct, along with your own phy structure.

In general, the phys are managed by the LLDD and the ports are managed by the SAS layer. So the phys are initialized and updated by the LLDD and the ports are initialized and updated by the SAS layer.

There is a scheme where the LLDD can RW certain fields, and the SAS layer can only read such ones, and vice versa. The idea is to avoid unnecessary locking.

`enabled`

- must be set (0/1)

`id`

- must be set [0,MAX_PHYS]

`class, proto, type, role, oob_mode, linkrate`

- must be set

`oob_mode`

- you set this when OOB has finished and then notify the SAS Layer.

`sas_addr`

- this normally points to an array holding the sas address of the phy, possibly somewhere in your `my_phy` struct.

`attached_sas_addr`

- set this when you (LLDD) receive an IDENTIFY frame or a FIS frame, `_before_` notifying the SAS layer. The idea is that sometimes the LLDD may want to fake or provide a different SAS address on that phy/port and this allows it to do this. At best you should copy the sas address from the IDENTIFY frame or maybe generate a SAS address for SATA directly attached devices. The Discover process may later change this.

`frame_rcvd`

- this is where you copy the IDENTIFY/FIS frame when you get it; you lock, copy, set `frame_rcvd_size` and unlock the lock, and then call the event. It is a pointer since there's no way to know your hw frame size `_exactly_`, so you define the actual array in your phy struct and let this pointer point to it. You copy the frame from your DMAable memory to that area holding the lock.

`sas_prim`

- this is where primitives go when they're received. See sas.h. Grab the lock, set the primitive, release the lock, notify.
- port
- this points to the sas_port if the phy belongs to a port -- the LLDD only reads this. It points to the sas_port this phy is part of. Set by the SAS Layer.
- ha
- may be set; the SAS layer sets it anyway.
- lldd_phy
- you should set this to point to your phy so you can find your way around faster when the SAS layer calls one of your callbacks and passes you a phy. If the sas_phy is embedded you can also use container_of -- whatever you prefer.

struct sas_port

The LLDD doesn't set any fields of this struct -- it only reads them. They should be self explanatory.

phy_mask is 32 bit, this should be enough for now, as I haven't heard of a HA having more than 8 phys.

lldd_port

- I haven't found use for that -- maybe other LLDD who wish to have internal port representation can make use of this.

struct sas_ha_struct

It normally is statically declared in your own LLDD structure describing your adapter:

```
struct my_sas_ha {
    blah;
    struct sas_ha_struct sas_ha;
    struct my_phy phys[MAX_PHYS];
    struct sas_port sas_ports[MAX_PHYS]; /* (1) */
    bleh;
};
```

(1) If your LLDD doesn't have its own port representation.

What needs to be initialized (sample function given below).

pcidev

sas_addr

- since the SAS layer doesn't want to mess with memory allocation, etc, this points to statically allocated array somewhere (say in your host adapter structure) and holds the SAS address of the host adapter as given by you or the manufacturer, etc.

sas_port

sas_phy

- an array of pointers to structures. (see note above on sas_addr). These must be set. See more notes below.

num_phys

- the number of phys present in the sas_phy array, and the number of ports present in the sas_port array. There can be a maximum num_phys ports (one per port) so we drop the num_ports, and only use num_phys.

The event interface:

```
/* LLDD calls these to notify the class of an event. */
void sas_notify_port_event(struct sas_phy *, enum port_event, gfp_t);
void sas_notify_phy_event(struct sas_phy *, enum phy_event, gfp_t);
```

The port notification:

```
/* The class calls these to notify the LLDD of an event. */
void (*lldd_port_formed)(struct sas_phy *);
void (*lldd_port_deformed)(struct sas_phy *);
```

If the LLDD wants notification when a port has been formed or deformed it sets those to a function satisfying the type.

A SAS LLDD should also implement at least one of the Task Management Functions (TMFs) described in SAM:

```
/* Task Management Functions. Must be called from process context. */
int (*lldd_abort_task)(struct sas_task *);
int (*lldd_abort_task_set)(struct domain_device *, u8 *lun);
int (*lldd_clear_task_set)(struct domain_device *, u8 *lun);
int (*lldd_I_T_nexus_reset)(struct domain_device *);
int (*lldd_lu_reset)(struct domain_device *, u8 *lun);
int (*lldd_query_task)(struct sas_task *);
```

For more information please read SAM from T10.org

Port and Adapter management:

```

/* Port and Adapter management */
int (*lldd_clear_nexus_port)(struct sas_port *);
int (*lldd_clear_nexus_ha)(struct sas_ha_struct *);

```

A SAS LLDD should implement at least one of those.

Phy management:

```

/* Phy management */
int (*lldd_control_phy)(struct sas_phy *, enum phy_func);

```

lldd_ha

- set this to point to your HA struct. You can also use container_ if you embedded it as shown above.

A sample initialization and registration function can look like this (called last thing from probe()) *but* before you enable the phys to do OOB:

```

static int register_sas_ha(struct my_sas_ha *my_ha)
{
    int i;
    static struct sas_phy  *sas_phys[MAX_PHYS];
    static struct sas_port  *sas_ports[MAX_PHYS];

    my_ha->sas_ha.sas_addr = &my_ha->sas_addr[0];

    for (i = 0; i < MAX_PHYS; i++) {
        sas_phys[i] = &my_ha->phys[i].sas_phy;
        sas_ports[i] = &my_ha->sas_ports[i];
    }

    my_ha->sas_ha.sas_phy  = sas_phys;
    my_ha->sas_ha.sas_port = sas_ports;
    my_ha->sas_ha.num_phys = MAX_PHYS;

    my_ha->sas_ha.lldd_port_formed = my_port_formed;

    my_ha->sas_ha.lldd_dev_found = my_dev_found;
    my_ha->sas_ha.lldd_dev_gone = my_dev_gone;

    my_ha->sas_ha.lldd_execute_task = my_execute_task;

    my_ha->sas_ha.lldd_abort_task      = my_abort_task;
    my_ha->sas_ha.lldd_abort_task_set = my_abort_task_set;
    my_ha->sas_ha.lldd_clear_task_set = my_clear_task_set;
    my_ha->sas_ha.lldd_I_T_nexus_reset= NULL; (2)
    my_ha->sas_ha.lldd_lu_reset        = my_lu_reset;
    my_ha->sas_ha.lldd_query_task      = my_query_task;

    my_ha->sas_ha.lldd_clear_nexus_port = my_clear_nexus_port;
    my_ha->sas_ha.lldd_clear_nexus_ha = my_clear_nexus_ha;

    my_ha->sas_ha.lldd_control_phy = my_control_phy;

    return sas_register_ha(&my_ha->sas_ha);
}

```

2. SAS 1.1 does not define I_T Nexus Reset TMF.

Events

Events are **the only way** a SAS LLDD notifies the SAS layer of anything. There is no other method or way a LLDD to tell the SAS layer of anything happening internally or in the SAS domain.

Phy events:

```

PHYE_LOSS_OF_SIGNAL, (C)
PHYE_OOB_DONE,
PHYE_OOB_ERROR,      (C)
PHYE_SPINUP_HOLD.

```

Port events, passed on a _phy_:

```

PORTE_BYTES_DMAED,      (M)
PORTE_BROADCAST_RCVD,   (E)
PORTE_LINK_RESET_ERR,   (C)
PORTE_TIMER_EVENT,      (C)
PORTE_HARD_RESET.

```

Host Adapter event:

```

HAE_RESET

```

A SAS LLDD should be able to generate

- at least one event from group C (choice),
- events marked M (mandatory) are mandatory (only one),
- events marked E (expander) if it wants the SAS layer to handle domain revalidation (only one such).
- Unmarked events are optional.

Meaning:

HAE_RESET

- when your HA got internal error and was reset.

PORTE_BYTES_DMAED

- on receiving an IDENTIFY/FIS frame

PORTE_BROADCAST_RCVD

- on receiving a primitive

PORTE_LINK_RESET_ERR

- timer expired, loss of signal, loss of DWS, etc. [1]

PORTE_TIMER_EVENT

- DWS reset timeout timer expired [1]

PORTE_HARD_RESET

- Hard Reset primitive received.

PHYE_LOSS_OF_SIGNAL

- the device is gone [1]

PHYE_OOB_DONE

- OOB went fine and oob_mode is valid

PHYE_OOB_ERROR

- Error while doing OOB, the device probably got disconnected. [1]

PHYE_SPINUP_HOLD

- SATA is present, COMWAKE not sent.

[1] (1,2,3,4) should set/clear the appropriate fields in the phy, or alternatively call the inlined sas_phy_disconnected() which is just a helper, from their tasklet.

The Execute Command SCSI RPC:

```
int (*lldd_execute_task)(struct sas_task *, gfp_t gfp_flags);
```

Used to queue a task to the SAS LLDD. @task is the task to be executed. @gfp_mask is the gfp_mask defining the context of the caller.

This function should implement the Execute Command SCSI RPC,

That is, when lldd_execute_task() is called, the command go out on the transport *immediately*. There is *no* queuing of any sort and at any level in a SAS LLDD.

Returns:

- -SAS_QUEUE_FULL, -ENOMEM, nothing was queued;
- 0, the task(s) were queued.

```
struct sas_task {
    dev -- the device this task is destined to
    task_proto -- _one_ of enum sas_proto
    scatter -- pointer to scatter gather list array
    num_scatter -- number of elements in scatter
    total_xfer_len -- total number of bytes expected to be transferred
    data_dir -- PCI_DMA_...
    task_done -- callback when the task has finished execution
};
```

Discovery

The sysfs tree has the following purposes:

- a. It shows you the physical layout of the SAS domain at the current time, i.e. how the domain looks in the physical world right now.
- b. Shows some device parameters `_at_discovery_time_`.

This is a link to the tree(1) program, very useful in viewing the SAS domain: <ftp://mama.indstate.edu/linux/tree/>

I expect user space applications to actually create a graphical interface of this.

That is, the sysfs domain tree doesn't show or keep state if you e.g., change the meaning of the READY LED MEANING setting but it does show you the current connection status of the domain device.

Keeping internal device state changes is responsibility of upper layers (Command set drivers) and user space.

When a device or devices are unplugged from the domain, this is reflected in the sysfs tree immediately, and the device(s) removed from the system.

The structure `domain_device` describes any device in the SAS domain. It is completely managed by the SAS layer. A task points to a domain device, this is how the SAS LLDD knows where to send the task(s) to. A SAS LLDD only reads the contents of the `domain_device` structure, but it never creates or destroys one.

Expander management from User Space

In each expander directory in sysfs, there is a file called "smp_portal". It is a binary sysfs attribute file, which implements an SMP portal (Note: this is *NOT* an SMP port), to which user space applications can send SMP requests and receive SMP responses.

Functionality is deceptively simple:

1. Build the SMP frame you want to send. The format and layout is described in the SAS spec. Leave the CRC field equal 0.

`open(2)`

2. Open the expander's SMP portal sysfs file in RW mode.

`write(2)`

3. Write the frame you built in 1.

`read(2)`

4. Read the amount of data you expect to receive for the frame you built. If you receive different amount of data you expected to receive, then there was some kind of error.

`close(2)`

All this process is shown in detail in the function `do_smp_func()` and its callers, in the file "expander_conf.c".

The kernel functionality is implemented in the file "sas_expander.c".

The program "expander_conf.c" implements this. It takes one argument, the sysfs file name of the SMP portal to the expander, and gives expander information, including routing tables.

The SMP portal gives you complete control of the expander, so please be careful.