Audio Stream in SoundWire

An audio stream is a logical or virtual connection created between

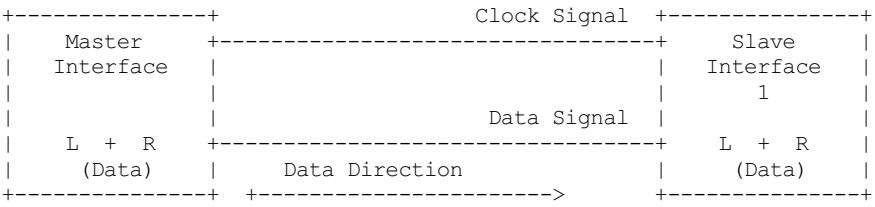
1. System memory buffer(s) and Codec(s)
2. DSP memory buffer(s) and Codec(s)
3. FIFO(s) and Codec(s)
4. Codec(s) and Codec(s)

which is typically driven by a DMA(s) channel through the data link. An audio stream contains one or more channels of data. All channels within stream must have same sample rate and same sample size.

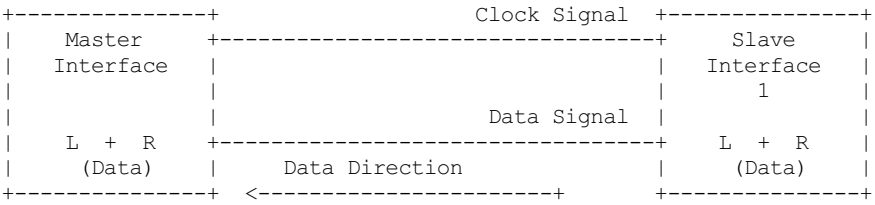
Assume a stream with two channels (Left & Right) is opened using SoundWire interface. Below are some ways a stream can be represented in SoundWire.

Stream Sample in memory (System memory, DSP memory or FIFOs)

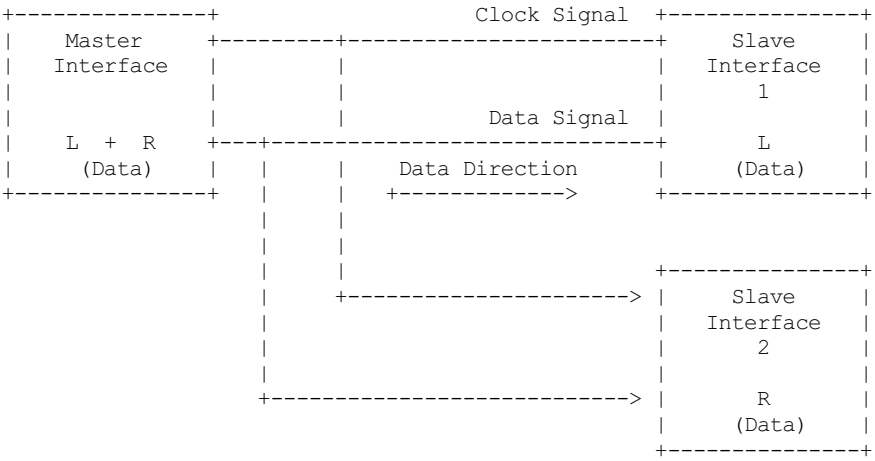
Example 1: Stereo Stream with L and R channels is rendered from Master to Slave. Both Master and Slave is using single port.



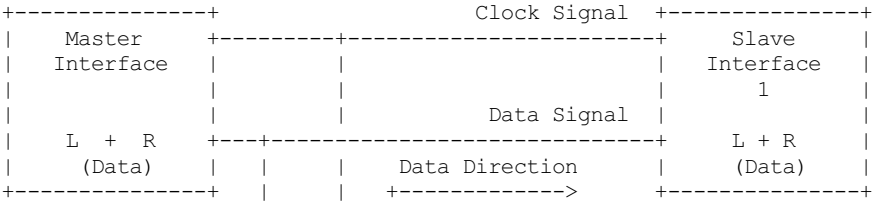
Example 2: Stereo Stream with L and R channels is captured from Slave to Master. Both Master and Slave is using single port.

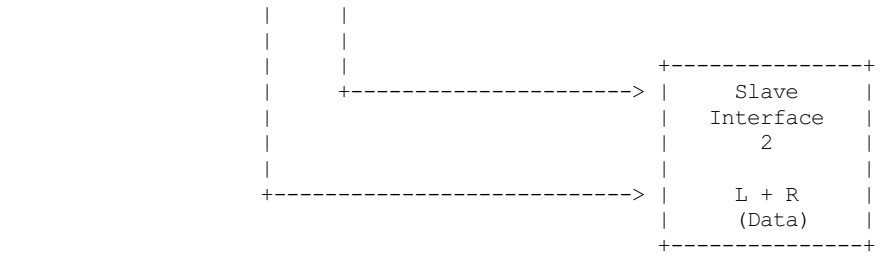


Example 3: Stereo Stream with L and R channels is rendered by Master. Each of the L and R channel is received by two different Slaves. Master and both Slaves are using single port.

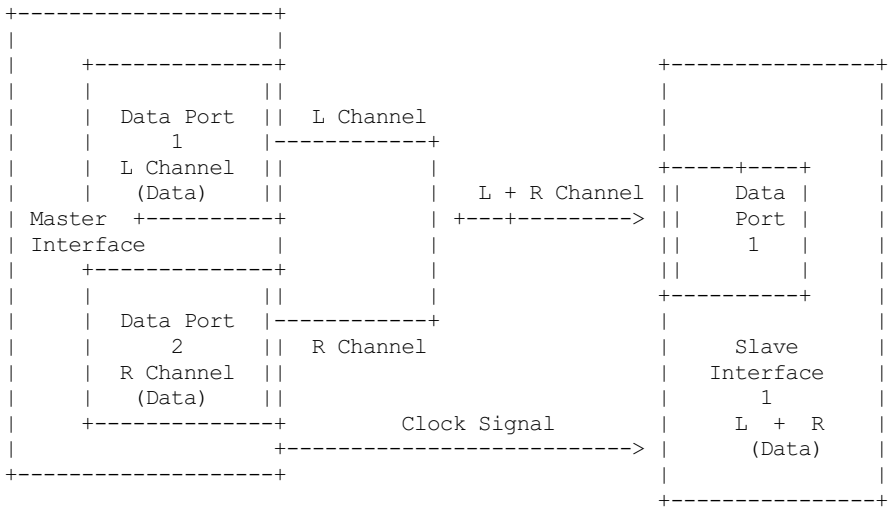


Example 4: Stereo Stream with L and R channels is rendered by Master. Both of the L and R channels are received by two different Slaves. Master and both Slaves are using single port handling L+R. Each Slave device processes the L + R data locally, typically based on static configuration or dynamic orientation, and may drive one or more speakers.

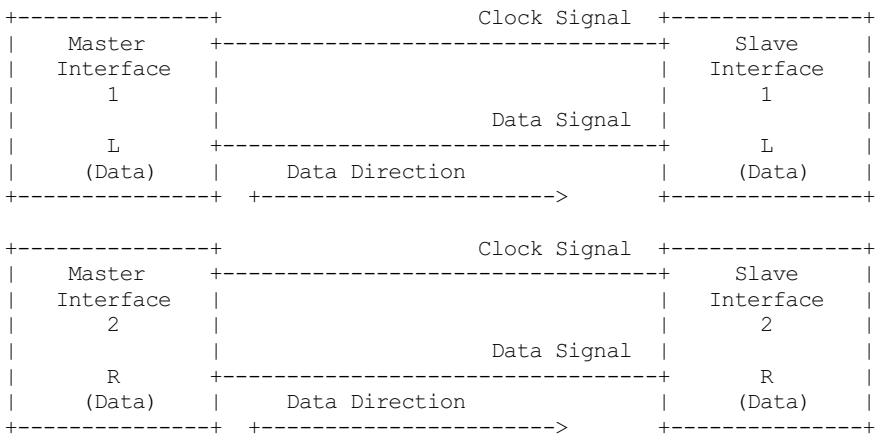




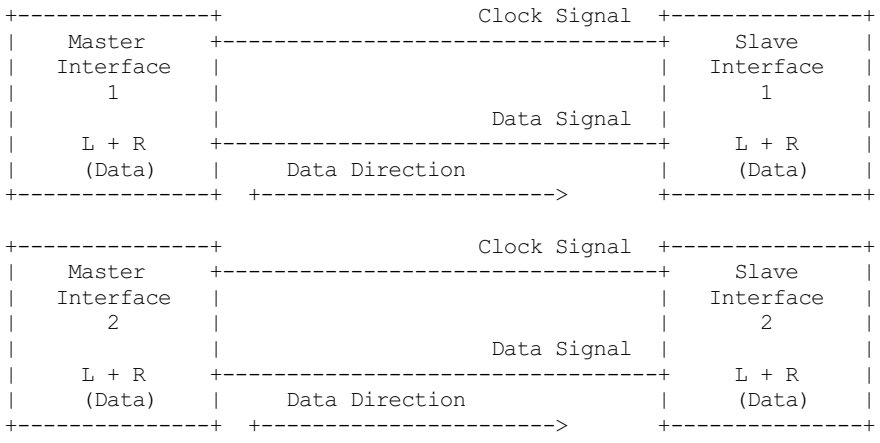
Example 5: Stereo Stream with L and R channel is rendered by two different Ports of the Master and is received by only single Port of the Slave interface.



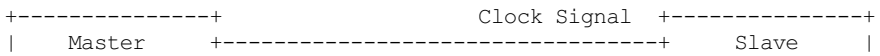
Example 6: Stereo Stream with L and R channel is rendered by 2 Masters, each rendering one channel, and is received by two different Slaves, each receiving one channel. Both Masters and both Slaves are using single port.

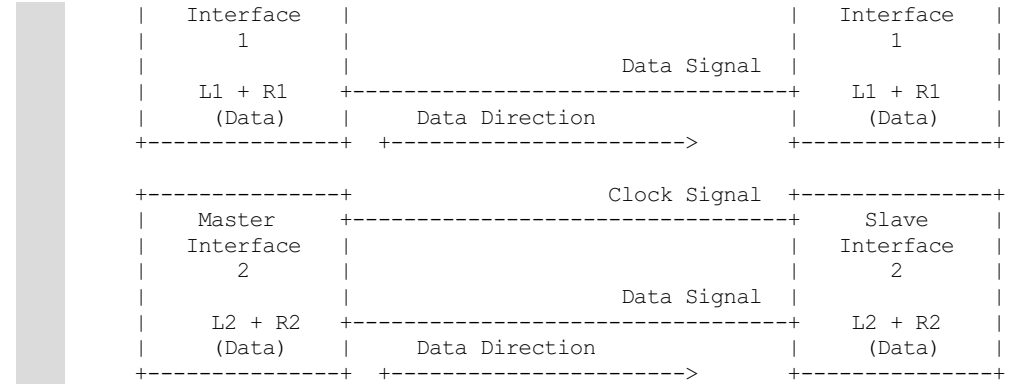


Example 7: Stereo Stream with L and R channel is rendered by 2 Masters, each rendering both channels. Each Slave receives L + R. This is the same application as Example 4 but with Slaves placed on separate links.



Example 8: 4-channel Stream is rendered by 2 Masters, each rendering a 2 channels. Each Slave receives 2 channels.





Note1: In multi-link cases like above, to lock, one would acquire a global lock and then go on locking bus instances. But, in this case the caller framework(ASoC DPCM) guarantees that stream operations on a card are always serialized. So, there is no race condition and hence no need for global lock.

Note2: A Slave device may be configured to receive all channels transmitted on a link for a given Stream (Example 4) or just a subset of the data (Example 3). The configuration of the Slave device is not handled by a SoundWire subsystem API, but instead by the snd_soc_dai_set_tdm_slot() API. The platform or machine driver will typically configure which of the slots are used. For Example 4, the same slots would be used by all Devices, while for Example 3 the Slave Device1 would use e.g. Slot 0 and Slave device2 slot 1.

Note3: Multiple Sink ports can extract the same information for the same bitSlots in the SoundWire frame, however multiple Source ports shall be configured with different bitSlot configurations. This is the same limitation as with I2S/PCM TDM usages.

SoundWire Stream Management flow

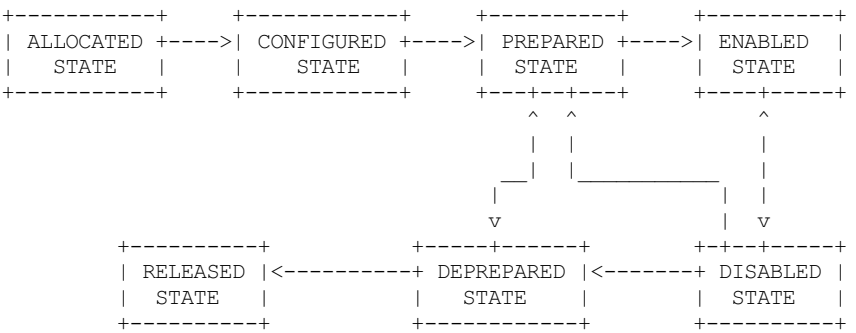
Stream definitions

1. Current stream: This is classified as the stream on which operation has to be performed like prepare, enable, disable, de-prepare etc.
2. Active stream: This is classified as the stream which is already active on Bus other than current stream. There can be multiple active streams on the Bus.

SoundWire Bus manages stream operations for each stream getting rendered/captured on the SoundWire Bus. This section explains Bus operations done for each of the stream allocated/released on Bus. Following are the stream states maintained by the Bus for each of the audio stream.

SoundWire stream states

Below shows the SoundWire stream states and state transition diagram.



NOTE: State transitions between `SDW_STREAM_ENABLED` and `SDW_STREAM_DISABLED` are only relevant when then INFO_PAUSE flag is supported at the ALSA/ASoC level. Likewise the transition between `SDW_DISABLED_STATE` and `SDW_PREPARED_STATE` depends on the INFO_RESUME flag.

NOTE2: The framework implements basic state transition checks, but does not e.g. check if a transition from DISABLED to ENABLED is valid on a specific platform. Such tests need to be added at the ALSA/ASoC level.

Stream State Operations

Below section explains the operations done by the Bus on Master(s) and Slave(s) as part of stream state transitions.

SDW_STREAM_ALLOCATED

Allocation state for stream. This is the entry state of the stream. Operations performed before entering in this state:

1. A stream runtime is allocated for the stream. This stream runtime is used as a reference for all the operations

performed on the stream.

2. The resources required for holding stream runtime information are allocated and initialized. This holds all stream related information such as stream type (PCM/PDM) and parameters, Master and Slave interface associated with the stream, stream state etc.

After all above operations are successful, stream state is set to `SDW_STREAM_ALLOCATED`.

Bus implements below API for allocate a stream which needs to be called once per stream. From ASoC DPCM framework, this stream state maybe linked to .startup() operation.

```
int sdw_alloc_stream(char * stream_name);
```

The SoundWire core provides a sdw_startup_stream() helper function, typically called during a dailink .startup() callback, which performs stream allocation and sets the stream pointer for all DAIs connected to a stream.

SDW_STREAM_CONFIGURED

Configuration state of stream. Operations performed before entering in this state:

1. The resources allocated for stream information in SDW_STREAM_ALLOCATED state are updated here. This includes stream parameters, Master(s) and Slave(s) runtime information associated with current stream.
2. All the Master(s) and Slave(s) associated with current stream provide the port information to Bus which includes port numbers allocated by Master(s) and Slave(s) for current stream and their channel mask.

After all above operations are successful, stream state is set to `SDW_STREAM_CONFIGURED`.

Bus implements below APIs for CONFIG state which needs to be called by the respective Master(s) and Slave(s) associated with stream. These APIs can only be invoked once by respective Master(s) and Slave(s). From ASoC DPCM framework, this stream state is linked to .hw_params() operation.

```
int sdw_stream_add_master(struct sdw_bus * bus,
              struct sdw_stream_config * stream_config,
              struct sdw_ports_config * ports_config,
              struct sdw_stream_runtime * stream);

int sdw_stream_add_slave(struct sdw_slave * slave,
              struct sdw_stream_config * stream_config,
              struct sdw_ports_config * ports_config,
              struct sdw_stream_runtime * stream);
```

SDW_STREAM_PREPARED

Prepare state of stream. Operations performed before entering in this state:

0. Steps 1 and 2 are omitted in the case of a resume operation, where the bus bandwidth is known.
1. Bus parameters such as bandwidth, frame shape, clock frequency, are computed based on current stream as well as already active stream(s) on Bus. Re-computation is required to accommodate current stream on the Bus.
2. Transport and port parameters of all Master(s) and Slave(s) port(s) are computed for the current as well as already active stream based on frame shape and clock frequency computed in step 1.
3. Computed Bus and transport parameters are programmed in Master(s) and Slave(s) registers. The banked registers programming is done on the alternate bank (bank currently unused). Port(s) are enabled for the already active stream(s) on the alternate bank (bank currently unused). This is done in order to not disrupt already active stream(s).
4. Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect.
5. Ports of Master(s) and Slave(s) for current stream are prepared by programming PrepareCtrl register.

After all above operations are successful, stream state is set to `SDW_STREAM_PREPARED`.

Bus implements below API for PREPARE state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .prepare() operation. Since the .trigger() operations may not follow the .prepare(), a direct transition from `SDW_STREAM_PREPARED` to `SDW_STREAM_DEPREPARED` is allowed.

```
int sdw_prepare_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_ENABLED

Enable state of stream. The data port(s) are enabled upon entering this state. Operations performed before entering in this state:

1. All the values computed in SDW_STREAM_PREPARED state are programmed in alternate bank (bank currently unused). It includes programming of already active stream(s) as well.
2. All the Master(s) and Slave(s) port(s) for the current stream are enabled on alternate bank (bank currently unused) by programming ChannelEn register.

3. Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect and port(s) associated with current stream are enabled.

After all above operations are successful, stream state is set to `SDW_STREAM_ENABLED`.

Bus implements below API for ENABLE state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .trigger() start operation.

```
int sdw_enable_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_DISABLED

Disable state of stream. The data port(s) are disabled upon exiting this state. Operations performed before entering in this state:

1. All the Master(s) and Slave(s) port(s) for the current stream are disabled on alternate bank (bank currently unused) by programming ChannelEn register.
2. All the current configuration of Bus and active stream(s) are programmed into alternate bank (bank currently unused).
3. Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect and port(s) associated with current stream are disabled.

After all above operations are successful, stream state is set to `SDW_STREAM_DISABLED`.

Bus implements below API for DISABLED state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .trigger() stop operation.

When the INFO_PAUSE flag is supported, a direct transition to `SDW_STREAM_ENABLED` is allowed.

For resume operations where ASoC will use the .prepare() callback, the stream can transition from `SDW_STREAM_DISABLED` to `SDW_STREAM_PREPARED`, with all required settings restored but without updating the bandwidth and bit allocation.

```
int sdw_disable_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_DEPREPARED

De-prepare state of stream. Operations performed before entering in this state:

1. All the port(s) of Master(s) and Slave(s) for current stream are de-prepared by programming PrepareCtrl register.
2. The payload bandwidth of current stream is reduced from the total bandwidth requirement of bus and new parameters calculated and applied by performing bank switch etc.

After all above operations are successful, stream state is set to `SDW_STREAM_DEPREPARED`.

Bus implements below API for DEPREPARED state which needs to be called once per stream. ALSA/ASoC do not have a concept of 'deprepare', and the mapping from this stream state to ALSA/ASoC operation may be implementation specific.

When the INFO_PAUSE flag is supported, the stream state is linked to the .hw_free() operation - the stream is not deprepared on a TRIGGER_STOP.

Other implementations may transition to the `SDW_STREAM_DEPREPARED` state on TRIGGER_STOP, should they require a transition through the `SDW_STREAM_PREPARED` state.

```
int sdw_deprepare_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_RELEASED

Release state of stream. Operations performed before entering in this state:

1. Release port resources for all Master(s) and Slave(s) port(s) associated with current stream.
2. Release Master(s) and Slave(s) runtime resources associated with current stream.
3. Release stream runtime resources associated with current stream.

After all above operations are successful, stream state is set to `SDW_STREAM_RELEASED`.

Bus implements below APIs for RELEASE state which needs to be called by all the Master(s) and Slave(s) associated with stream. From ASoC DPCM framework, this stream state is linked to .hw_free() operation.

```
int sdw_stream_remove_master(struct sdw_bus * bus,
              struct sdw_stream_runtime * stream);
int sdw_stream_remove_slave(struct sdw_slave * slave,
              struct sdw_stream_runtime * stream);
```

The .shutdown() ASoC DPCM operation calls below Bus API to release stream assigned as part of ALLOCATED state.

In .shutdown() the data structure maintaining stream state are freed up.

```
void sdw_release_stream(struct sdw_stream_runtime * stream);
```

The SoundWire core provides a sdw_shutdown_stream() helper function, typically called during a dailink .shutdown() callback, which clears the stream pointer for all DAIS connected to a stream and releases the memory allocated for the stream.

Not Supported

1. A single port with multiple channels supported cannot be used between two streams or across stream. For example a port with 4 channels cannot be used to handle 2 independent stereo streams even though it's possible in theory in SoundWire.