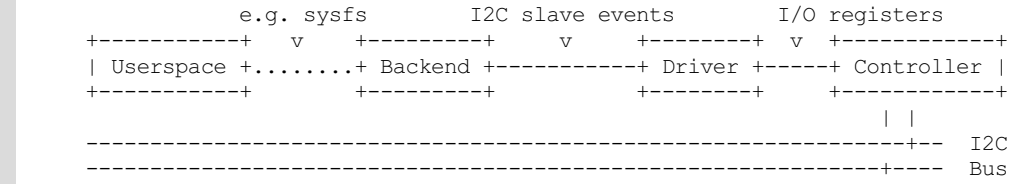
Linux I2C slave interface description

by Wolfram Sang <wsa@sang-engineering.com> in 2014-15

Linux can also be an I2C slave if the I2C controller in use has slave functionality. For that to work, one needs slave support in the bus driver plus a hardware independent software backend providing the actual functionality. An example for the latter is the slave-eeprom driver, which acts as a dual memory driver. While another I2C master on the bus can access it like a regular EEPROM, the Linux I2C slave can access the content via sysfs and handle data as needed. The backend driver and the I2C bus driver communicate via events. Here is a small graph visualizing the data flow and the means by which data is transported. The dotted line marks only one example. The backend could also use a character device, be in-kernel only, or something completely different:



Note: Technically, there is also the I2C core between the backend and the driver. However, at this time of writing, the layer is transparent.

User manual

I2C slave backends behave like standard I2C clients. So, you can instantiate them as described in the document 'instantiating-devices'. The only difference is that i2c slave backends have their own address space. So, you have to add 0x1000 to the address you would originally request. An example for instantiating the slave-eeprom driver from userspace at the 7 bit address 0x64 on bus 1:

```
# echo slave-24c02 0x1064 > /sys/bus/i2c/devices/i2c-1/new_device
```

Each backend should come with separate documentation to describe its specific behaviour and setup.

Developer manual

First, the events which are used by the bus driver and the backend will be described in detail. After that, some implementation hints for extending bus drivers and writing backends will be given.

I2C slave events

The bus driver sends an event to the backend using the following function:

```
ret = i2c_slave_event(client, event, &val)
```

'client' describes the I2C slave device. 'event' is one of the special event types described hereafter. 'val' holds an u8 value for the data byte to be read/written and is thus bidirectional. The pointer to val must always be provided even if val is not used for an event, i.e. don't use NULL here. 'ret' is the return value from the backend. Mandatory events must be provided by the bus drivers and must be checked for by backend drivers.

Event types:

- I2C_SLAVE_WRITE_REQUESTED (mandatory)

  'val': unused

  'ret': always 0

Another I2C master wants to write data to us. This event should be sent once our own address and the write bit was detected. The data did not arrive yet, so there is nothing to process or return. Wakeup or initialization probably needs to be done, though.

- I2C_SLAVE_READ_REQUESTED (mandatory)

  'val': backend returns first byte to be sent

  'ret': always 0

Another I2C master wants to read data from us. This event should be sent once our own address and the read bit was detected. After returning, the bus driver should transmit the first byte.

- I2C_SLAVE_WRITE_RECEIVED (mandatory)

  'val': bus driver delivers received byte

  'ret': 0 if the byte should be acked, some errno if the byte should be nacked

Another I2C master has sent a byte to us which needs to be set in 'val'. If 'ret' is zero, the bus driver should ack this byte. If 'ret' is an errno, then the byte should be nacked.

- I2C_SLAVE_READ_PROCESSED (mandatory)

  'val': backend returns next byte to be sent

  'ret': always 0

The bus driver requests the next byte to be sent to another I2C master in 'val'. Important: This does not mean that the previous byte has been acked, it only means that the previous byte is shifted out to the bus! To ensure seamless transmission, most hardware requests the next byte when the previous one is still shifted out. If the master sends NACK and stops reading after the byte currently shifted out, this byte requested here is never used. It very likely needs to be sent again on the next I2C_SLAVE_READ_REQUEST, depending a bit on your backend, though.

- I2C_SLAVE_STOP (mandatory)

  'val': unused

  'ret': always 0

A stop condition was received. This can happen anytime and the backend should reset its state machine for I2C transfers to be able to receive new requests.

Software backends

If you want to write a software backend:

- use a standard i2c_driver and its matching mechanisms
- write the slave_callback which handles the above slave events (best using a state machine)
- register this callback via i2c_slave_register()

Check the i2c-slave-eeprom driver as an example.

Bus driver support

If you want to add slave support to the bus driver:

- implement calls to register/unregister the slave and add those to the struct i2c_algorithm. When registering, you probably need to set the I2C slave address and enable slave specific interrupts. If you use runtime pm, you should use pm_runtime_get_sync() because your device usually needs to be powered on always to be able to detect its slave address. When unregistering, do the inverse of the above.
- Catch the slave interrupts and send appropriate i2c_slave_events to the backend.

Note that most hardware supports being master _and_ slave on the same bus. So, if you extend a bus driver, please make sure that the driver supports that as well. In almost all cases, slave support does not need to disable the master functionality.

Check the i2c-rcar driver as an example.

About ACK/NACK

It is good behaviour to always ACK the address phase, so the master knows if a device is basically present or if it mysteriously disappeared. Using NACK to state being busy is troublesome. SMBus demands to always ACK the address phase, while the I2C specification is more loose on that. Most I2C controllers also automatically ACK when detecting their slave addresses, so there is no option to NACK them. For those reasons, this API does not support NACK in the address phase.

Currently, there is no slave event to report if the master did ACK or NACK a byte when it reads from us. We could make this an optional event if the need arises. However, cases should be extremely rare because the master is expected to send STOP after that and we have an event for that. Also, keep in mind not all I2C controllers have the possibility to report that event.

About buffers

During development of this API, the question of using buffers instead of just bytes came up. Such an extension might be possible, usefulness is unclear at this time of writing. Some points to keep in mind when using buffers:

- Buffers should be opt-in and backend drivers will always have to support byte-based transactions as the ultimate fallback anyhow because this is how the majority of HW works.
- For backends simulating hardware registers, buffers are largely not helpful because after each byte written an action should be immediately triggered. For reads, the data kept in the buffer might get stale if the backend just updated a register because of internal processing.
- A master can send STOP at any time. For partially transferred buffers, this means additional code to handle this exception. Such code tends to be error-prone.