Linux I2C fault injection

The GPIO based I2C bus master driver can be configured to provide fault injection capabilities. It is then meant to be connected to another I2C bus which is driven by the I2C bus master driver under test. The GPIO fault injection driver can create special states on the bus which the other I2C bus master driver should handle gracefully.

Once the Kconfig option I2C_GPIO_FAULT_INJECTOR is enabled, there will be an 'i2c-fault-injector' subdirectory in the Kernel debugfs filesystem, usually mounted at /sys/kernel/debug. There will be a separate subdirectory per GPIO driven I2C bus. Each subdirectory will contain files to trigger the fault injection. They will be described now along with their intended use-cases.

Wire states

"scl"

By reading this file, you get the current state of SCL. By writing, you can change its state to either force it low or to release it again. So, by using "echo 0 > scl" you force SCL low and thus, no communication will be possible because the bus master under test will not be able to clock. It should detect the condition of SCL being unresponsive and report an error to the upper layers.

"sda"

By reading this file, you get the current state of SDA. By writing, you can change its state to either force it low or to release it again. So, by using "echo 0 > sda" you force SDA low and thus, data cannot be transmitted. The bus master under test should detect this condition and trigger a bus recovery (see I2C specification version 4, section 3.1.16) using the helpers of the Linux I2C core (see 'struct bus_recovery_info'). However, the bus recovery will not succeed because SDA is still pinned low until you manually release it again with "echo 1 > sda". A test with an automatic release can be done with the "incomplete transfers" class of fault injectors.

Incomplete transfers

The following fault injectors create situations where SDA will be held low by a device. Bus recovery should be able to fix these situations. But please note: there are I2C client devices which detect a stuck SDA on their side and release it on their own after a few milliseconds. Also, there might be an external device deglitching and monitoring the I2C bus. It could also detect a stuck SDA and will init a bus recovery on its own. If you want to implement bus recovery in a bus master driver, make sure you checked your hardware setup for such devices before. And always verify with a scope or logic analyzer!

"incomplete_address_phase"

This file is write only and you need to write the address of an existing I2C client device to it. Then, a read transfer to this device will be started, but it will stop at the ACK phase after the address of the client has been transmitted. Because the device will ACK its presence, this results in SDA being pulled low by the device while SCL is high. So, similar to the "sda" file above, the bus master under test should detect this condition and try a bus recovery. This time, however, it should succeed and the device should release SDA after toggling SCL.

"incomplete_write_byte"

Similar to above, this file is write only and you need to write the address of an existing I2C client device to it.

The injector will again stop at one ACK phase, so the device will keep SDA low because it acknowledges data. However, there are two differences compared to 'incomplete_address_phase':

a. the message sent out will be a write message
b. after the address byte, a 0x00 byte will be transferred. Then, stop at ACK.

This is a highly delicate state, the device is set up to write any data to register 0x00 (if it has registers) when further clock pulses happen on SCL. This is why bus recovery (up to 9 clock pulses) must either check SDA or send additional STOP conditions to ensure the bus has been released. Otherwise random data will be written to a device!

Lost arbitration

Here, we want to simulate the condition where the master under test loses the bus arbitration against another master in a multi-master setup.

"lose_arbitration"

This file is write only and you need to write the duration of the arbitration intereference (in Âµs, maximum is 100ms). The calling process will then sleep and wait for the next bus clock. The process is interruptible, though.

Arbitration lost is achieved by waiting for SCL going down by the master under test and then pulling SDA low for some time. So, the I2C address sent out should be corrupted and that should be detected properly. That means that the address sent out should have a

lot of '1' bits to be able to detect corruption. There doesn't need to be a device at this address because arbitration lost should be detected beforehand. Also note, that SCL going down is monitored using interrupts, so the interrupt latency might cause the first bits to be not corrupted. A good starting point for using this fault injector on an otherwise idle bus is:

```
# echo 200 > lose_arbitration &
# i2cget -y <bus_to_test> 0x3f
```

Panic during transfer

This fault injector will create a Kernel panic once the master under test started a transfer. This usually means that the state machine of the bus master driver will be ungracefully interrupted and the bus may end up in an unusual state. Use this to check if your shutdown/reboot/boot code can handle this scenario.

"inject_panic"

This file is write only and you need to write the delay between the detected start of a transmission and the induced Kernel panic (in Âµs, maximum is 100ms). The calling process will then sleep and wait for the next bus clock. The process is interruptible, though.

Start of a transfer is detected by waiting for SCL going down by the master under test. A good starting point for using this fault injector is:

```
# echo 0 > inject_panic &
# i2cget -y <bus_to_test> <some_address>
```

Note that there doesn't need to be a device listening to the address you are using. Results may vary depending on that, though.