Embedded Systems Programming

Quark I2C Interface
(Module 12)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014
Bit-Transfer on I2C Bus

- One clock pulse is generated for each data bit that is transferred.

- **Data Validity**
  - The data on the SDA line must be stable during the HIGH(1) period of the clock. The data line (SDA) can change data only when the clock signal (SCL) is LOW(0).

- **Wired-and function**
  - open-drain or open-collector
Data Transfer With 7-Bit Device Address

- After START condition (S), a slave address (7-bit) is sent.
- A read/write (R/W') direction is then sent (8th bit)
- Data transfer occurs, and then always terminated by STOP condition. However, repeated START conditions can occur.

![Diagram of data transfer sequence]

- SDA
- SCL
- START condition
- ADDRESS
- R/W
- ACK
- DATA
- ACK
- DATA
- ACK
- STOP condition
Master-Transmitter to Slave-Receiver Data Transfer

- In this, the transmission direction never changes. The set-up and transfer is straight-forward.

<table>
<thead>
<tr>
<th>S</th>
<th>SLAVE ADDRESS</th>
<th>R/W</th>
<th>A</th>
<th>DATA</th>
<th>A</th>
<th>DATA</th>
<th>A/Ā</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0' (write)</td>
<td>data transferred (n bytes + acknowledge)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- from master to slave: A = acknowledge (SDA LOW), Ā = not acknowledge (SDA HIGH)
- from slave to master: S = START condition, P = STOP condition
Master-Receiver and Slave-Transmitter Data Transfer

- Master initiates the data transfer by generating the START condition followed by the start byte (with read/write bit set to 1 i.e. read mode)
- After the first ack from the slave, the direction of data changes and the master becomes receiver and slave transmitter.
- The STOP condition is still generated by the master (master sends not-ACK before generating the STOP)
Example I2C Device – 24FC256 EEPROM

- 32K bytes in 512 pages of 64 bytes
- I2C interface with A2, A1, A0 address pins
- Page write operation:

![Page Write Operation Diagram]

- Random read operation:

![Random Read Operation Diagram]
Example: I2C and GPIO in Quark

- **A PCI device**: B:0, D:21, F:2
- **MMIO** –
  - use two base registers in configuration registers

<table>
<thead>
<tr>
<th>Offset Start</th>
<th>Offset End</th>
<th>Register ID</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10h</td>
<td>13h</td>
<td>BAR0</td>
<td>00000000h</td>
</tr>
<tr>
<td>14h</td>
<td>17h</td>
<td>BAR1</td>
<td>00000000h</td>
</tr>
</tbody>
</table>

- **I2C memory registers** – BAR0+offset
- **I2C Master mode operation**
  - Disable the I2C controller by writing 0 to IC_ENABLE.ENABLE.
  - Write to the IC_CON register
  - Write to the IC_TAR register.
  - Enable the I2C controller by writing a 1 in IC_ENABLE.
  - Write the transfer direction and data to be sent to the IC_DATA_CMD register.
Example: Quark GPIO IRQ Enable

- Allows each bit of Port A to be configured for interrupts.
- In drivers/mfd/intel_cln_gip_gpio.c,

```
#define PORTA_INT_EN   0x30 /* Interrupt enable */
#define PORTA_INT_MASK 0x34 /* Interrupt mask */
#define PORTA_INT_TYPE_LEVEL 0x38 /* Interrupt level*/

static void intel_cln_gpio_irq_enable(struct irq_data *d)
{
  __iomem *reg_inte = reg_base + PORTA_INT_EN;
  gpio = d->irq - irq_base;

  spin_lock_irqsave(&lock, flags);
  val_inte = ioread32(reg_inte);
  iowrite32(val_inte | BIT(gpio % 32), reg_inte);
  spin_unlock_irqrestore(&lock, flags);
}
```
I2C Controller in Quark

- **GIP controller:** Bus 0, Device 21, Function 2
- **PCI configuration registers**
  - Base Address Register (BAR0) — Offset 10h
  - Base Address Register (BAR1) — Offset 14h
- **I2C Controller Memory Mapped Registers — Based on BAR0**
  - register offsets are defined in /deriver/i2c/buses/i2c-designware-core.c
  - I2C algorithm
    
    ```c
    static struct i2c_algorithm i2c_dw_algo = {
        .master_xfer       = i2c_dw_xfer,
        .functionality     = i2c_dw_func,
    };
    ```
  - major steps of i2c_dw_xfer in i2c-designware-core.c,
    ```c
    i2c_dw_wait_bus_not_busy(dev);
    i2c_dw_xfer_init(dev); /* wait for tx to complete */
    wait_for_completion_interruptible_timeout
    ```
  - to write to control reg:
    ```c
    dw_writel(dev, 1, DW_IC_ENABLE); /* Enable the adapter */
    ```