

I2C Driver (Inter Integrated circuit)

Outline

- What is I²C (or I2C)?, Where is it Used ?
- Basic Description
- Electrical Wiring, Clock
- A Basic I2C Transaction
- How fast can I2C run?, Bus bit rate vs Useful data rate
- I2C Subsystem in Linux
- I2C Client Driver
- I2C Device Registration (Non DT and DT)

What is I²C (or I2C)?

- Inter-Integrated Circuit
- Pronounced “eye-squared-see”
- Two-wire serial bus protocol
- Invented by Philips in the early 1980’s
 - That division now spun-off into NXP

Where is it Used?

- Originally used by Philips inside television sets
- Now very common in peripheral devices intended for embedded systems use
 - Philips, National Semiconductor, Xicor, and Siemens , ...
- Also used in the PC world
 - Real time clock
 - Temperature sensors

Basic Description

- Two-wire serial protocol with addressing capability
- Speeds up to 3.4 Mbit/s
- Multi-master/Multi-slave

Electrical Wiring

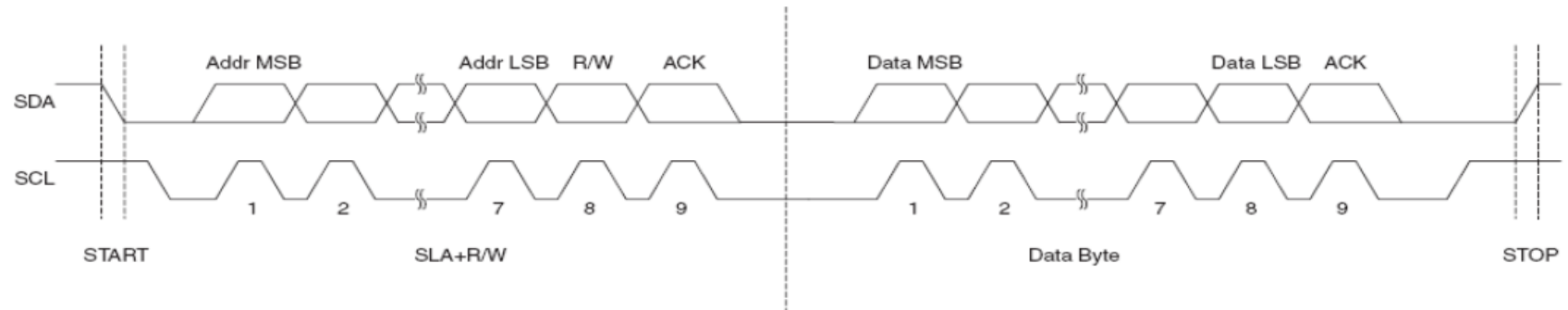
- Two lines
 - SDA (data)
 - SCL (clock)
- Open-collector
 - Very simple interfacing between different voltage levels

Clock

- Not a traditional clock
- Normally high (kept high by the pull-up)
- Pulsed by the master during data transmission (whether the master is transmitter or receiver)
- Slave device can hold clock low if it needs more time

A Basic I2C Transaction

- Master always initiates transactions
- Start Condition
- Address
- Data
- Acknowledgements
- Stop Condition



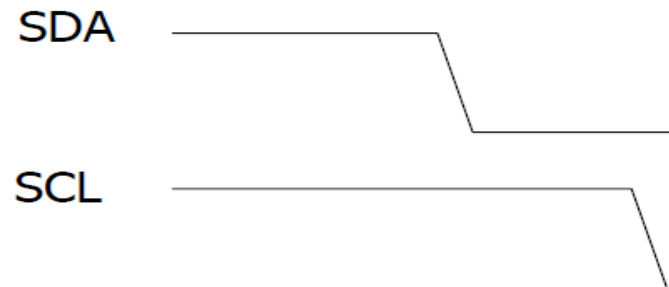
Source: ATmega8 Handbook

A Basic I2C Transaction

- Transmitter/Receiver differs from Master/Slave
- Master initiates transactions, slave responds
- Transmitter sets data on the SDA line, Receiver acknowledges
 - For a read, slave is transmitter
 - For a write, master is transmitter

Start Condition

- Master pulls SDA low while SCL is high
 - Normal SDA changes only happen while SCL is low

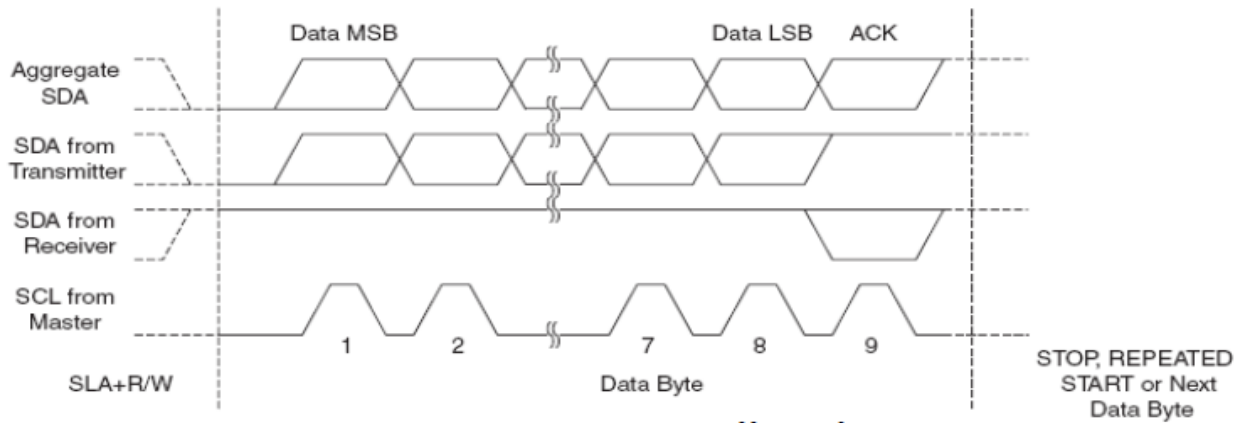


Address Transmission

- Data is always sampled on rising edge of clock
- Address is 7 bits
- An 8th bit indicates read or write
 - High for read, low for write
- Addresses assigned by Philips/NXP (for a fee)

Data transmission

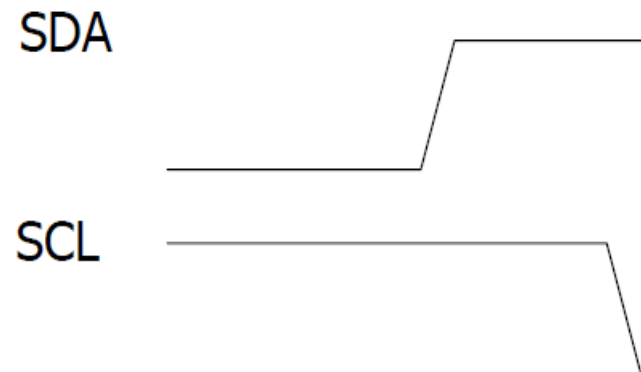
- Transmitted just like address (8 bits)
- For a write, master transmits, slave acknowledges
- For a read, slave transmits, master acknowledges
- Transmission continues with subsequent bytes until master creates stop condition



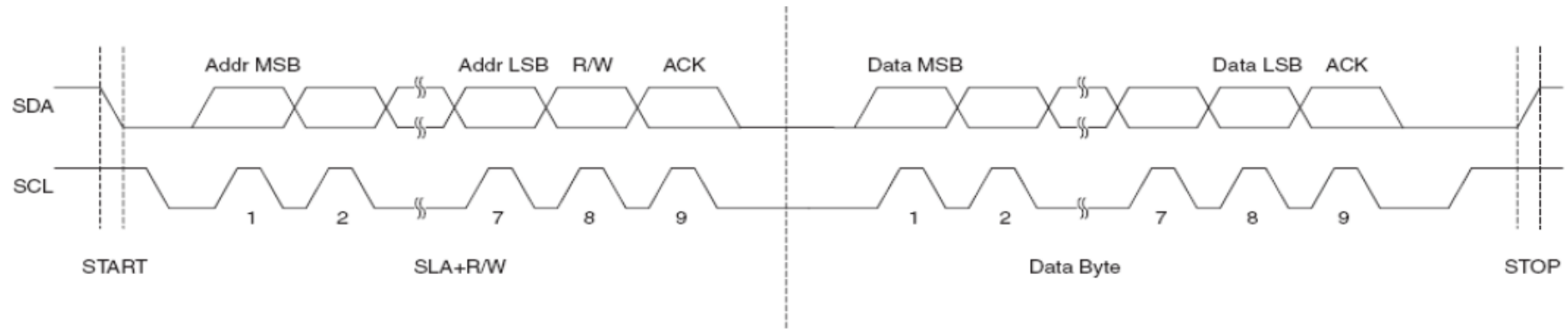
Source: ATmega8 Handbook

Stop Condition

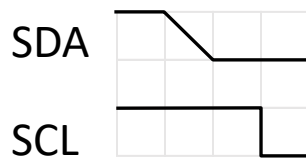
- Master pulls SDA high while SCL is high
- Also used to abort transactions



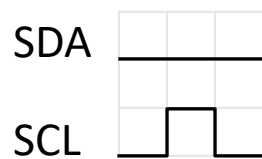
Another look at I2C



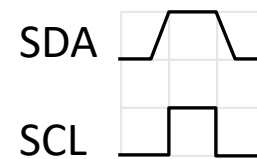
Source: ATmega8 Handbook



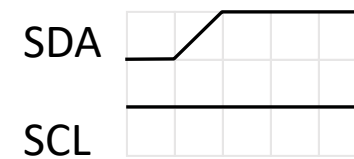
Start
condition



Sending 0

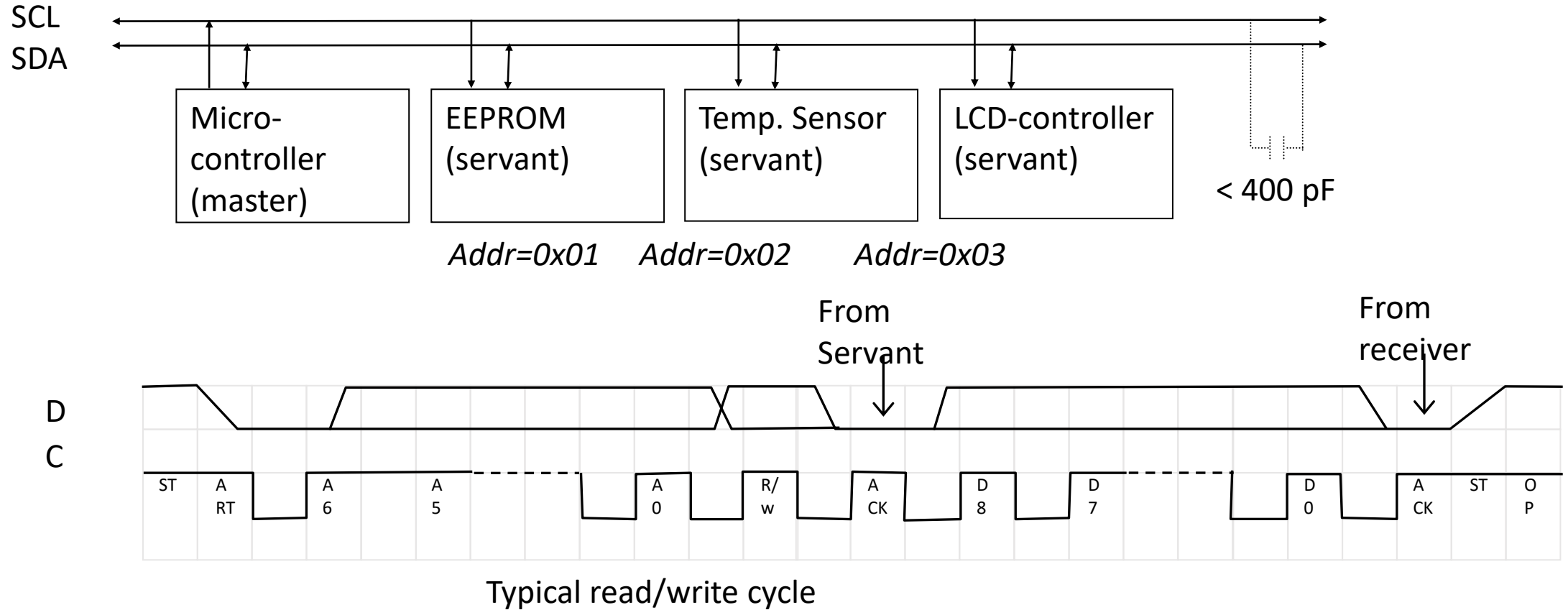


Sending 1



Stop
condition

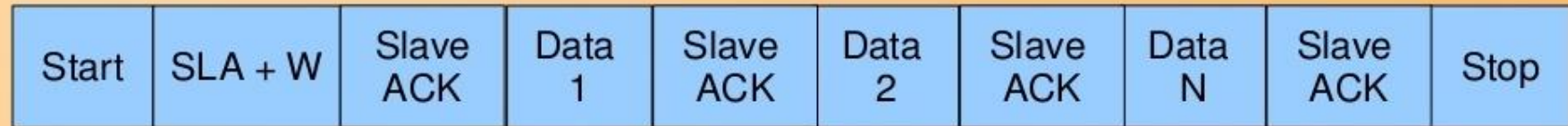
Another look at I2C



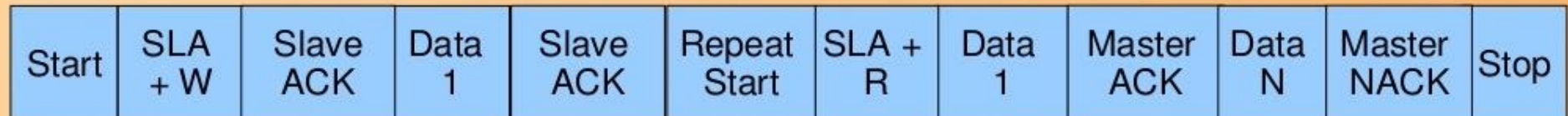
I2c Transactions

- ✧ Master begins the communication by issuing the start condition.
- ✧ Sends a 7/10 bit slave address followed by read/write bit
- ✧ The addressed slave ACK the transfer
- ✧ Transmitter transmits a byte of data and receiver issues ACK on successful receipt
- ✧ Master issues a STOP condition to finish the transaction

I2c Transactions

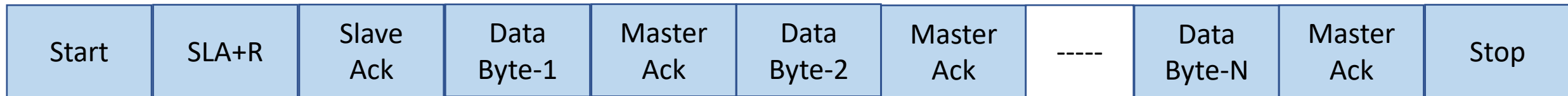


Master Write

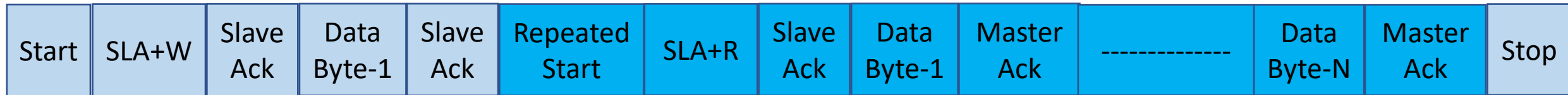


Master Read

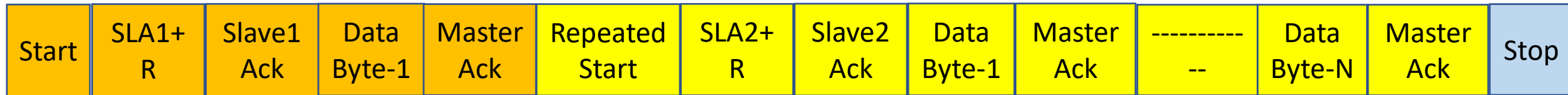
I2c Transactions (Read)



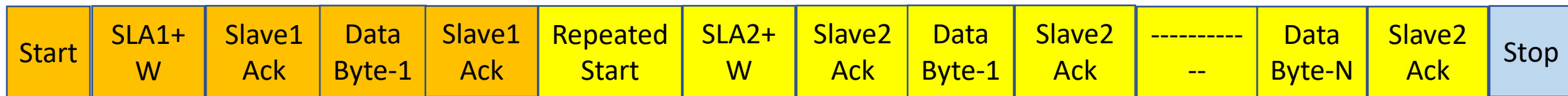
I2c Transactions (Combined Write-Read)



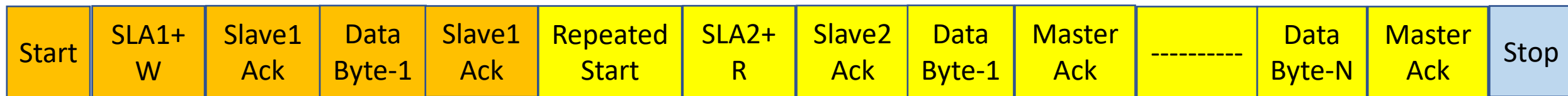
I2c Transactions (Combined Slave1-Read-Salave2-Read)



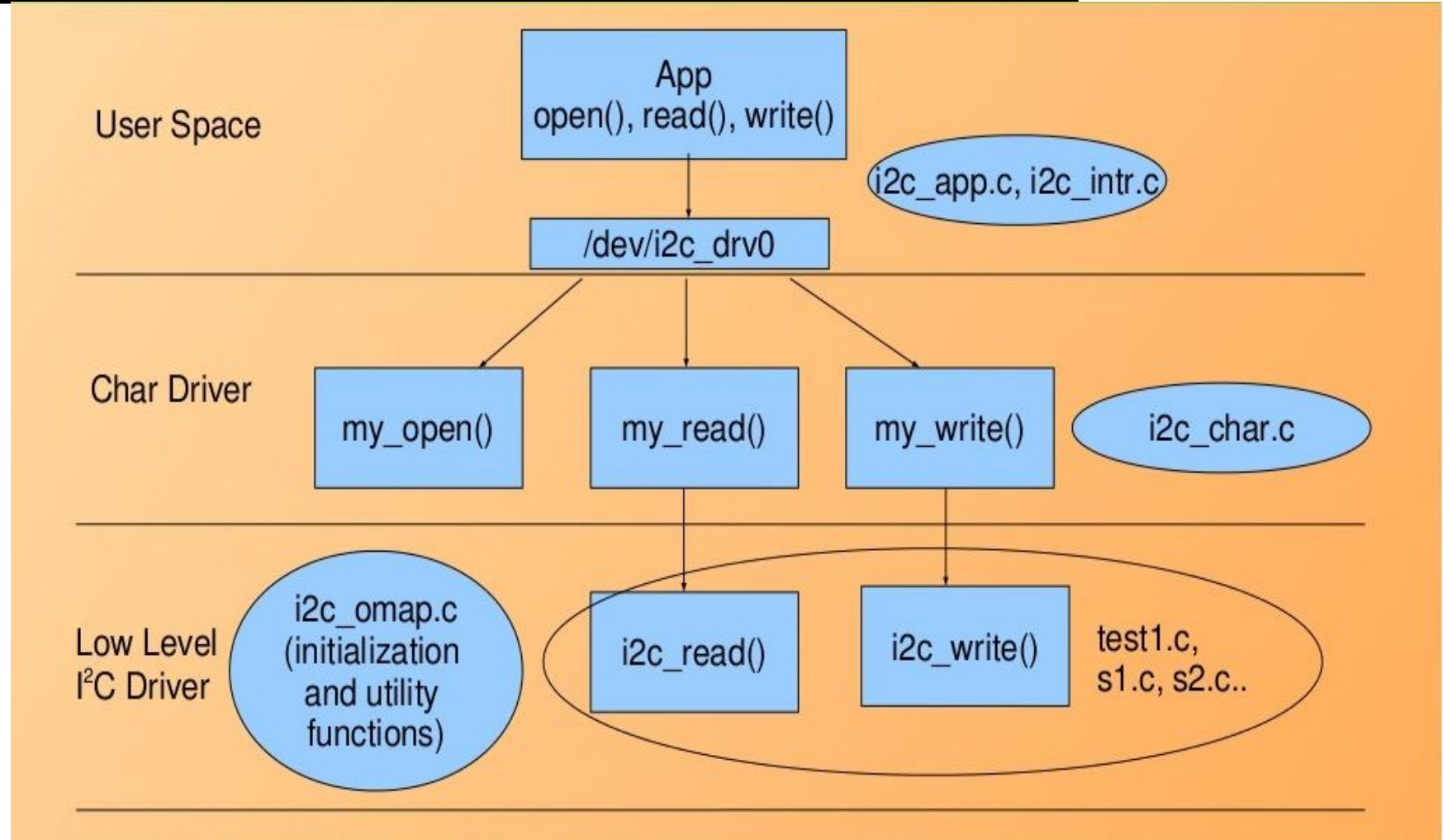
I2c Transactions (Combined Slave1-Write-Salave2-Write)



I2c Transactions (Combined Slave1-Write-Salave2-Read)



I2c Character Driver Framework



AM335X I2C Registers

- ★ I2C_SA_REGISTER (Slave Address Reg)
- ★ I2C_CON_REGISTER (Configuration Reg)
 - Bits for enabling the I²C module
 - Selecting the Fast / Standard mode of op
 - Selecting the Master / Slave config
 - Sending the Start / Stop conditions on the bus
- ★ I2C_DATA (RX/TX Data Reg)
- ★ I2C_BUF (FIFO Thresholds, DMA configuration)
- ★ I2C_CNT (Bytes in I²C data payload)
- ★ I2C_IRQ_STATUS_REG

AM335X I2C APIS

```

★ #include "i2c_char.h"

★ u16 omap_i2c_read_reg(struct omap_i2c_dev *i2c_dev,
    int reg)

★ void omap_i2c_write_reg(struct omap_i2c_dev *i2c_dev,
    int reg, u16 val)

★ u16 wait_for_event(struct omap_i2c_dev *dev)

★ void omap_i2c_ack_stat(struct omap_i2c_dev, u16 stat)

★ val = omap_i2c_read_reg(dev, OMAP_I2C_BUF_REG);
    val |= OMAP_I2C_BUF_TXFIF
    omap_i2c_write_reg(dev, OMAP_I2C_BUF_REG, val);

```

Writing to EEPROM

- ★ For writing at EEPROM offset 0x0060
 - Send the start condition.
 - Send the Slave address of EEPROM (0X50), followed by direction (Read/Write)
 - Send the EEPROM location higher byte, followed by lower byte
 - Send the actual data to be written
 - Send the Stop condition
 - START->0x50->0x00(offset High)->0x60 (offset Low)->0X95(Data)->STOP

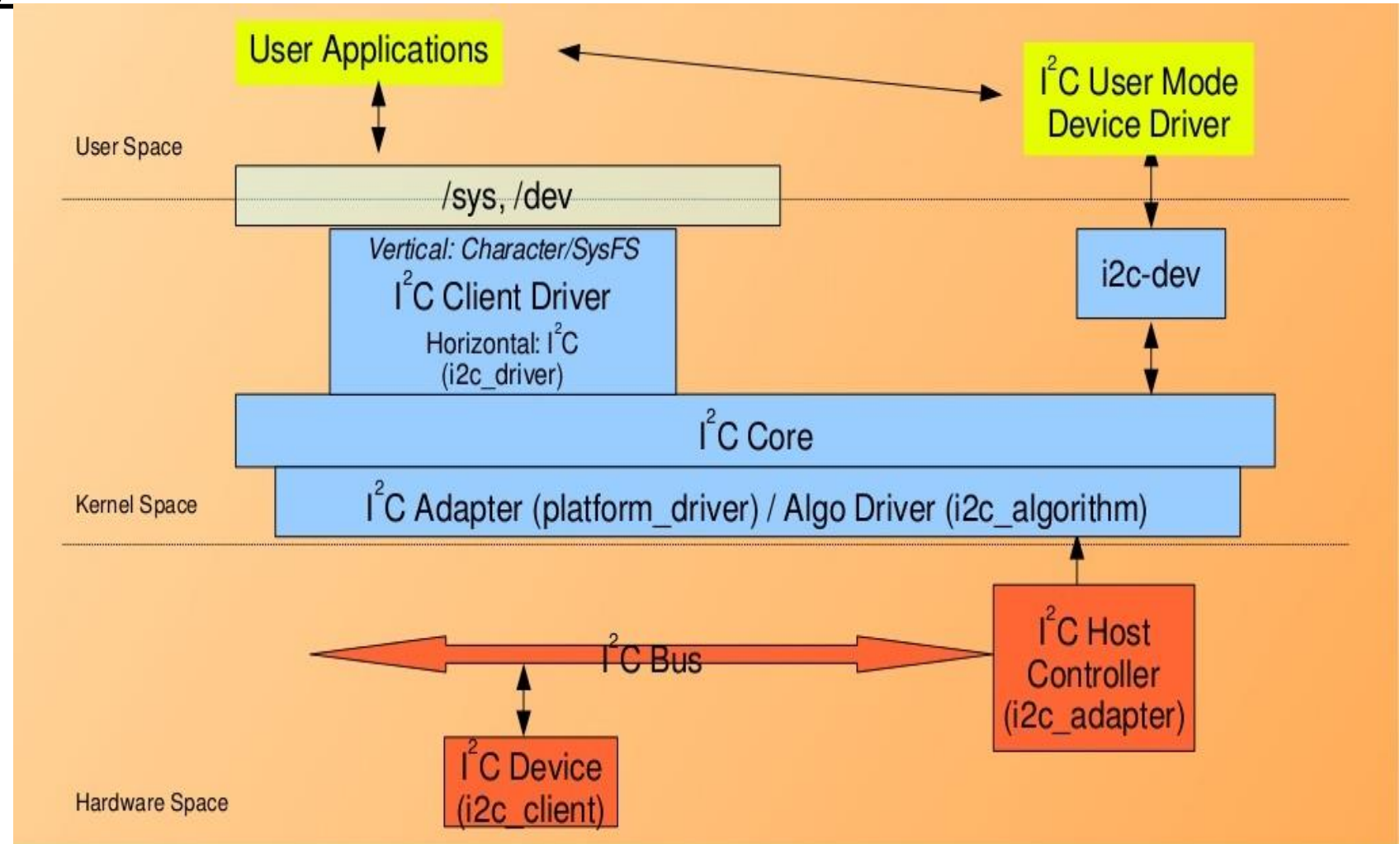
Reading From EEPROM

- ☆ Write the EEPROM offset say 0x0060 (read offset)
 - START->0x50->0x00(offset High)->0x60 (offset Low)->STOP
- ☆ Read the EEPROM data
 - Send the start condition.
 - Send the Slave address of EEPROM (0X50), followed by direction (Read)
 - Read the data
 - Send the Stop condition
 - START->0x50->Data(RX)->Data(RX)->STOP

I2C Subsystem

- ★ I2C subsystem provides
 - API to implement I²C controller driver
 - API to implement I²C device driver in kernel space
 - An abstraction to implement the client drivers independent of adapter drivers

I2C Subsystem



I2C Subsystem Details

★ i2c-adapter / i2-algo

- Controller-specific I²C host controller / adapter
- Also called as the I²C bus drivers

★ i2c-core

- Hides the adapter details from the layers above
- By providing the generic I²C APIs

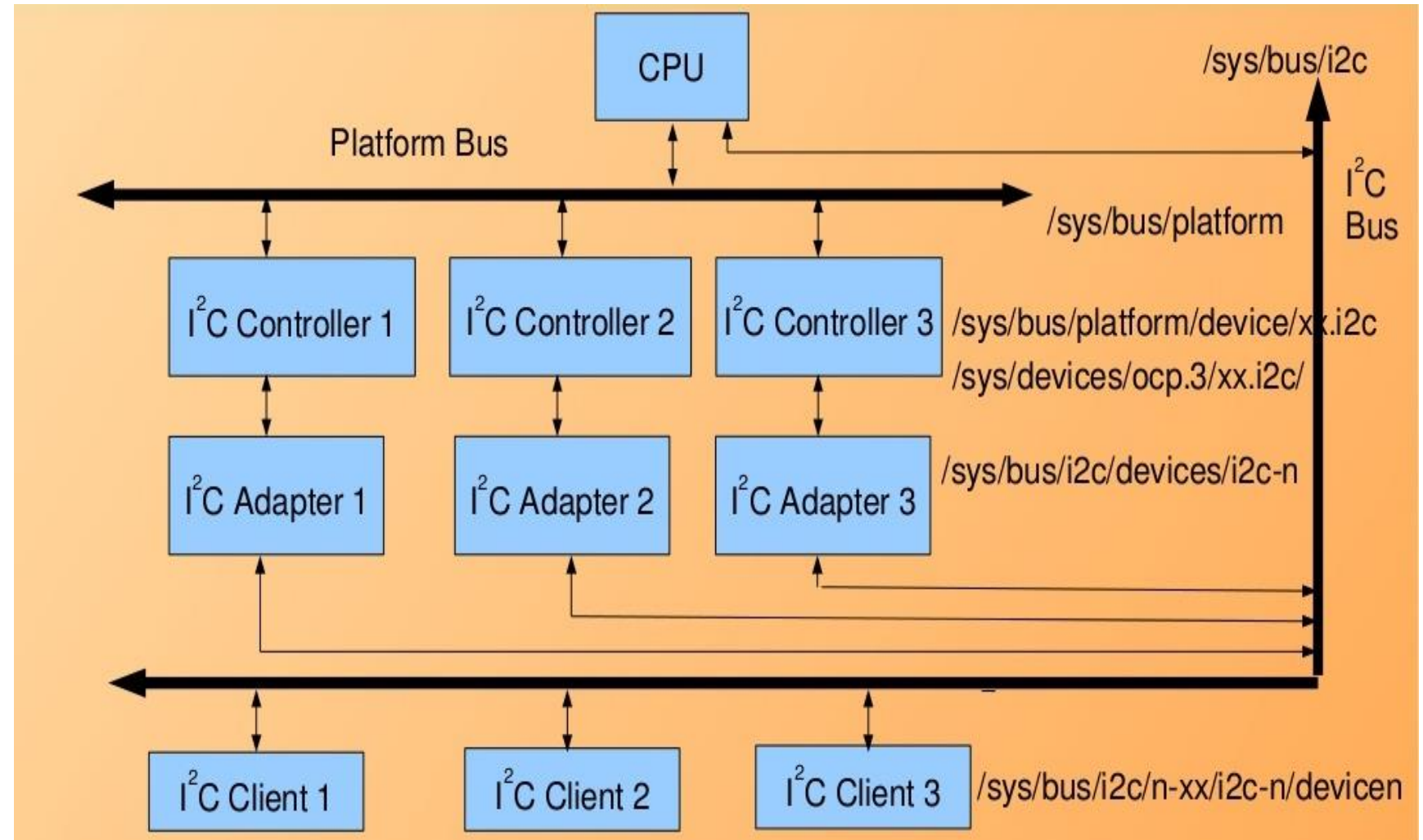
★ i2c-dev

- Provides device access in user space through /sys
- Enables implementation of User Mode Drivers

★ i2c-client

- Driver specific to an I²C device
- Implemented using i2c-core APIs

I2C Driver Model



SM Bus

- ★ Subset of I²C
 - ★ Using only SMBus commands makes it compatible with both adapters
 - ★ Part of I²C Core itself
 - ★ APIs for Device Access (Header: <linux/i2c.h>)
 - int ioctl(smbus_fp, cmd, arg);
 - s32 i2c_smbus_write_byte(client, val);
 - s32 i2c_smbus_read_byte(client);
 - s32 i2c_smbus_write_*_data(client, cmd, val);
 - s32 i2c_smbus_read_*_data(client, cmd);
- client – Pointer to struct i2c_client created by the probe function

I2C Client Driver

- ★ Typically a character driver vertical or /sys exposed
- ★ But actually depends on the device category
- ★ And the I²C driver horizontal
 - Registers with I²C Core (in the init function)
 - Unregisters from I²C Core (in the cleanup function)
 - And uses the transfer function from I²C Core for actual data transactions
 - int i2c_transfer
 - (struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
 - adap is the client->adapter

I2C Client Driver's Init & cleanup

- ★ Registering to the I²C Core using
 - `int i2c_add_driver(struct i2c_driver *);`
 - `struct i2c_driver` contains
 - probe function – called on device detection
 - remove function – called on device shutdown
 - `id_table` – Table of device identifiers
- ★ Unregistering from the I²C Core using
 - `void i2c_del_driver(struct i2c_driver *);`
- ★ Common bare-bone of init & cleanup
 - Just use `module_i2c_driver(struct i2c_driver);`

I2C Client Driver's Registration

```

★ struct i2c_driver dummy_driver = {
    .driver = {
        .name = "dummy_client",
        .owner = THIS_MODULE,
    },
    .probe = dummy_probe,
    .remove = dummy_remove,
    .id_table = dummy_ids,
}

★ static const struct i2c_device_id dummy_ids = {
    { "dummy_device", 0},
    {}
};

★ i2c_add_driver(dummy_driver);

```

I2C Adapter Driver

- ★ Registering to the I²C Core using
 - `int i2c_add_driver(struct i2c_driver *);`
 - `struct i2c_driver` contains
 - probe function – called on device detection
 - remove function – called on device shutdown
 - `id_table` – Table of device identifiers
- ★ Unregistering from the I²C Core using
 - `void i2c_del_driver(struct i2c_driver *);`
- ★ Common bare-bone of init & cleanup
 - Just use `module_i2c_driver(struct i2c_driver);`

Checking Adapter Capabilities

- ★ I²C client driver's probe would typically check for HC capabilities
- ★ Header: <linux/i2c.h>
- ★ APIs
 - ◆ u32 i2c_get_functionality
 - (struct *i2c_adapter);
 - ◆ int i2c_check_functionality
 - (struct *i2c_adapter, u32 func);

I2C Client Driver Examples

- ★ Path: <kernel_source>/drivers/
 - I²C EEPROM: AT24
 - misc/eeprom/at24.c
 - I²C Audio: Beagle Audio Codec cum Pwr Mgmt
 - mfd/twl4030-audio.c -> mfd/twl-core.c(plat driver)
 - I²C RTC: DS1307
 - rtc/rtc-twl.c -> mfd/twl-core.c; rtc/rtc-ds1307.c
- ★ Browse & Discuss any

Registering an I2c Client (Non DT)

- ★ On non-DT platforms, the struct `i2c_board_info` describes how device is connected to a board
- ★ Defined with `I2C_BOARD_INFO` helper macro
 - Takes as argument the device name and the slave address of the device on the bus.
- ★ An array of such structures is registered on per bus basis using the `i2c_register_board_info` during the platform initialization

Registering an I2c Client (Non DT)

```

★ static struct i2c_board_info my_i2c_devices [] = {
    {
        I2C_BOARD_INFO ("my_device", 0 x1D ),
        . irq = 70,
        .platform_data = &my_data },
    };

★ i2c_register_board_info (0, my_i2c_devices ,
    ARRAY_SIZE ( my_i2c_devices ));

```


Registering an I2C Client (DT)

- ★ In the device tree, the I²C devices on the bus are described as children of the I²C controller node
- ★ reg property gives the I²C slave address on the bus

Registering an I2C Client (DT)

```

★ i2c@49607899 {
    compatible = "dummy_adap";
    clock-frequency = <0x186a0>;
    #address-cells = <0x1>;
    #size-cells = <0x0>;

    my_dummy@0 {
        compatible = "dummy_device";
        reg = <0x40>;
    };
};

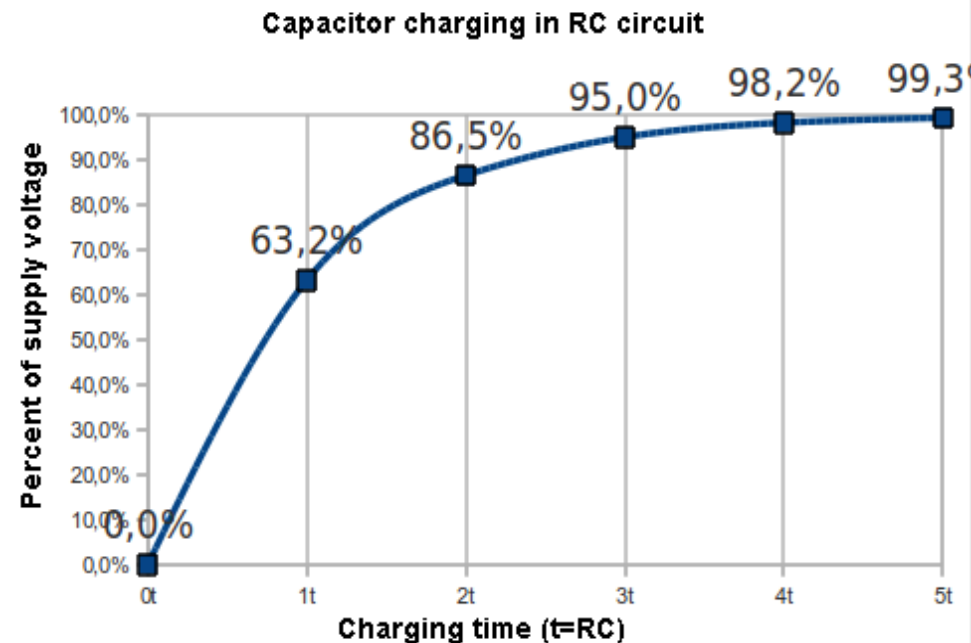
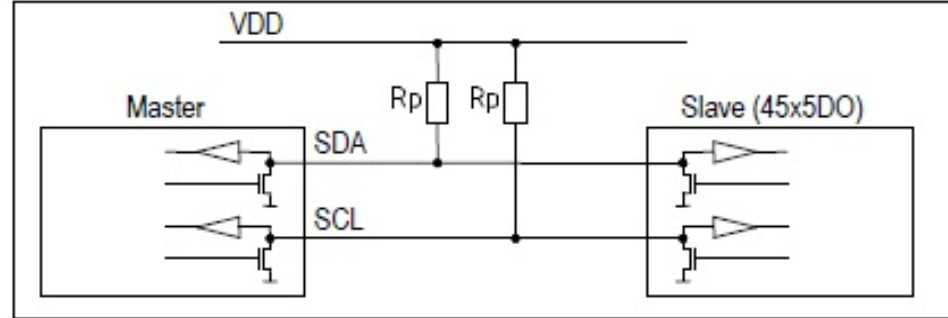
★ Registered internally by i2c_core based on info from DTB
i2c_new_device(struct i2c_adapter *, struct i2c_board_info *);

```

Registering an I2C Client (DT)

- Data on the I2C-bus can be transferred at rates of **up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.**

Exercise: How fast can I2C run?



- How fast can you run it?
- Assumptions
 - 0's are driven
 - 1's are "pulled up"
- Some working figures
 - $R_p = 10 \text{ k}\Omega$
 - $C_{cap} = 100 \text{ pF}$
 - $V_{DD} = 5 \text{ V}$
 - $V_{in_high} = 3.5 \text{ V}$
- Recall for RC circuit
 - $V_{cap}(t) = V_{DD}(1 - e^{-t/\tau})$
 - Where $\tau = RC$

Exercise: Bus bit rate vs Useful data rate

- An I2C “transactions” involves the following bits
 - <S><A6:A0><R/W><A><D7:D0><A><F>
- Which of these actually carries useful data?
 - <S><A6:A0><R/W><A><D7:D0><A><F>
- So, if a bus runs at 400 kHz
 - What is the clock period?
 - What is the data throughput (i.e. data-bits/second)?
 - What is the bus “efficiency”?

What all have we Learn?

- What is I²C (or I2C)?, Where is it Used ?
- Basic Description
- Electrical Wiring, Clock
- A Basic I2C Transaction
- How fast can I2C run?, Bus bit rate vs Useful data rate
- I2C Subsystem in Linux
- I2C Client Driver
- I2C Device Registration (Non DT and DT)

Any Queries?