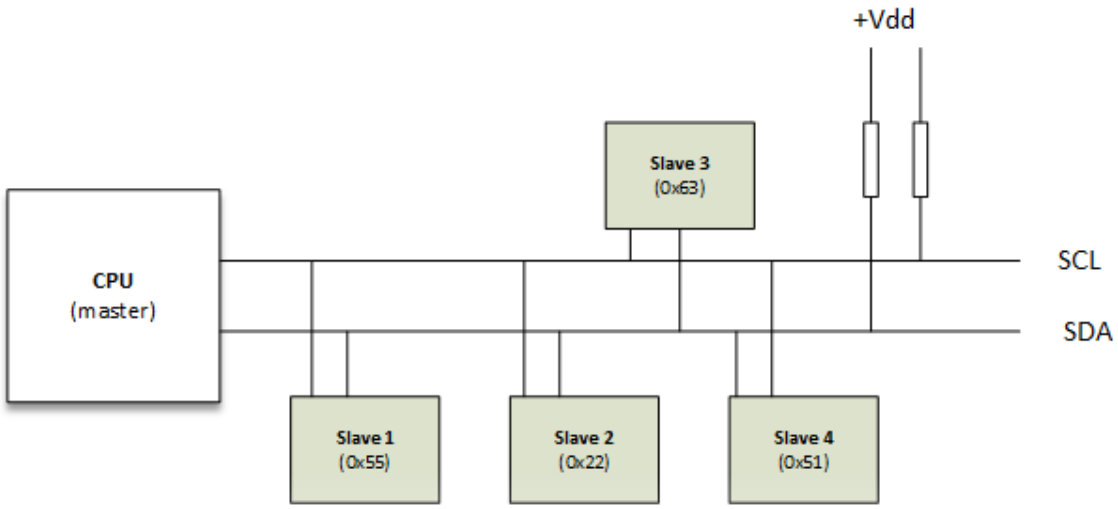
Table of Contents

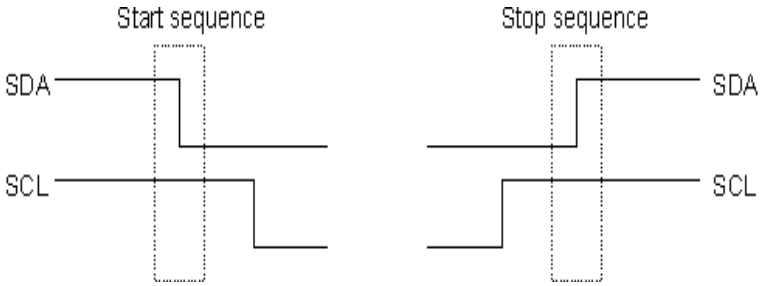
I2C Introduction:

The I2C bus, **invented by Philips (now NXP)** is a two-wire, **Serial Data** (**SDA**), **Serial Clock** (**SCL**) asynchronous serial bus. It is a multi-master bus, though multi-master mode is not widely used. Both SDA and SCL are open drain/open collector, meaning that each of these can drive its output low but neither of them can drive its output high without having pull-up resistors. SCL is generated by the master in order to synchronize data transfer(carried by SDA) over the bus. Both slave and master can send data (not at the same time of course), thus making SDA a bidirectional line. That said, the SCL signal is also bidirectional, **since the slave can stretch the clock by keeping the SCL line low.** The bus is controlled by the master, which in our case is a part of the SoC. This bus is frequently used in embedded systems to connect serial EEPROM, RTC chips, GPIO expander, temperature sensors, and so on:



The speed grades (standard mode: 100 kbit/s, full speed: 400 kbit/s, fast mode: 1 mbit/s, high speed: 3,2 Mbit/s) are maximum ratings

The start and stop sequences mark the beginning and end of a transaction with the slave device.



Clock Stretching

I2C client devices can slow down communication by *stretching* SCL: During an SCL low phase, any I2C device on the bus may additionally hold down SCL to prevent it from rising again, enabling it to slow down the SCL clock rate or to stop I2C communication for a while. This is also referred to as *clock synchronization*.

I2C Client : Driver

include/linux/i2c.h

include/uapi/linux/i2c.h

I2C client: Each device that is connected to the I2C bus on the system is represented using the ***struct i2c_client (defined in include/linux/i2c.h)***. The following are the important fields present in this structure.

Address: This field consists of the address of the device on the bus. This address is used by the driver to communicate with the device.

Name: This field is the name of the device which is used to match the driver with the device.

Interrupt number: This is the number of the interrupt line of the device.

I2C adapter: This is the ***struct i2c_adapter*** **which represents the bus on which this device is connected.** Whenever the driver makes requests to write or read from the bus, this field is used to identify the bus on which this transaction is to be done and also which algorithm should be used to communicate with the device.

I2C driver: For each device on the system, there should be a driver that controls it. For the I2C device, the corresponding driver is represented by ***struct i2c_driver (defined in include/linux/i2c.h).*** The following are the important fields defined in this structure.

***Driver.name*:** This is the name of the driver that is used to match the I2C device on the system with the driver.

Probe: This is the function pointer to the driver□s probe routine, which is called when the device and driver are both found on the system by the Linux device driver subsystem.

i2c driver : structure

```
struct i2c_driver {
        unsigned int class;
       /* Standard driver model interfaces */
        int (*probe)(struct i2c_client *, const struct i2c_device_id *);
        int (*remove)(struct i2c_client *);
       /* New driver model interface to aid the seamless removal of the
         * current probe()'s, more commonly unused than used second parameter.
         */
        int (*probe_new)(struct i2c_client *);

        struct device_driver driver;
        const struct i2c_device_id *id_table;
}
```

i2c client: structure

```
struct i2c_client {
        unsigned short flags;           /* div., see below              */
        unsigned short addr;            /* chip address - NOTE: 7bit    */
                                        /* addresses are stored in the  */
                                        /* _LOWER_ 7 bits               */
        char name[I2C_NAME_SIZE];
        struct i2c_adapter *adapter;    /* the adapter we sit on        */
        struct device dev;              /* the device structure         */
        int irq;                        /* irq issued by device         */
        struct list_head detected;
};
```

I2C Client Driver: API

```
static const struct of_device_id ov5640_dt_ids[] = {
        { .compatible = "ovti,ov5640" },
        { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, ov5640_dt_ids);


int i2c_register_driver(struct module *, struct i2c_driver *);
void i2c_del_driver(struct i2c_driver *);

/* use a define to avoid include chaining to get THIS_MODULE */
#define i2c_add_driver(driver)   i2c_register_driver(THIS_MODULE, driver)
```

```
static inline int i2c_master_send(const struct i2c_client *client, const char *buf, int count)
{
	return i2c_transfer_buffer_flags(client, (char *)buf, count, 0);
};

static inline int i2c_master_recv(const struct i2c_client *client, char *buf, int count)
{
	return i2c_transfer_buffer_flags(client, buf, count, I2C_M_RD);
};
```

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs,   int num);
```

```
struct i2c_msg {
	__u16 addr;	/* slave address			*/
	__u16 flags;
	__u16 len;	/* msg length			*/
	__u8 *buf;	/* pointer to msg data			*/
};
```

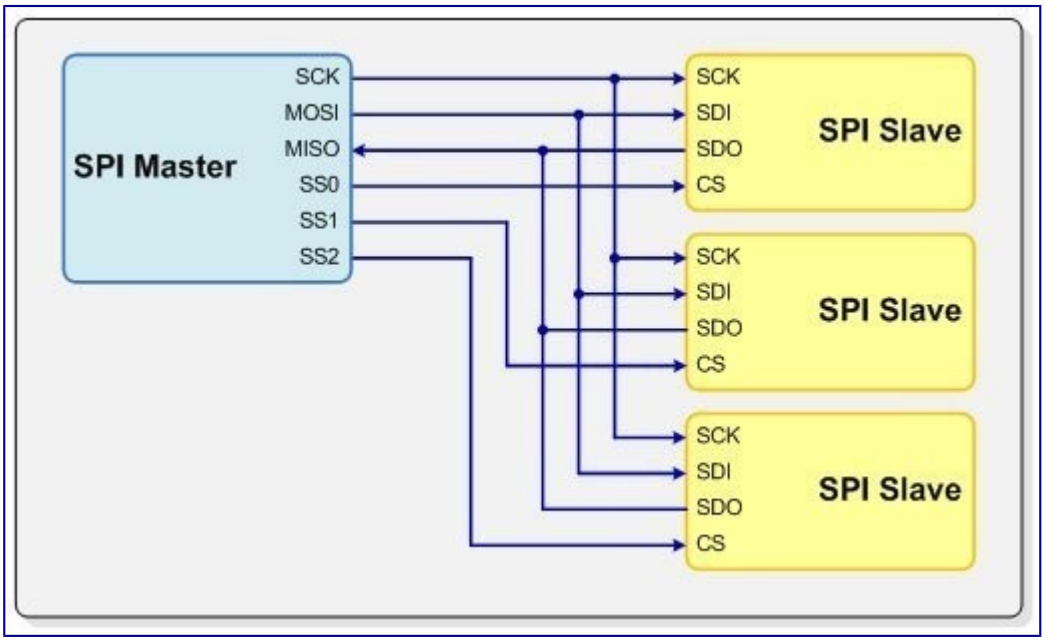
Device Tree Node:

```
&i2c1 {
	status = "okay";
	clock-frequency = <200000>;
	pinctrl-0 = <&i2c1_pins>;
	pinctrl-names = "default";

	eeprom@52 {
		compatible = "atmel,24c128";
		reg = <0x52>;
		pagesize = <32>;
	};

	bq27541@55 {
		compatible = "ti,bq27541";
		reg = <0x55>;
	};

  };
```

SPI Introduction

SPI (Serial Peripheral Interface) is an interface bus commonly used for communication with flash memory, sensors, real-time clocks (RTCs), analog-to-digital converters, and more. The Serial Peripheral Interface (SPI) bus was developed by Motorola to provide full-duplex synchronous serial communication between master and slave devices.

A standard SPI connection involves a master connected to slaves using the serial clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Slave Select (SS) lines. The SCK, MOSI, and MISO signals can be shared by slaves while each slave has a unique SS line.



SPI Device : Driver

include/linux/spi/spi.h

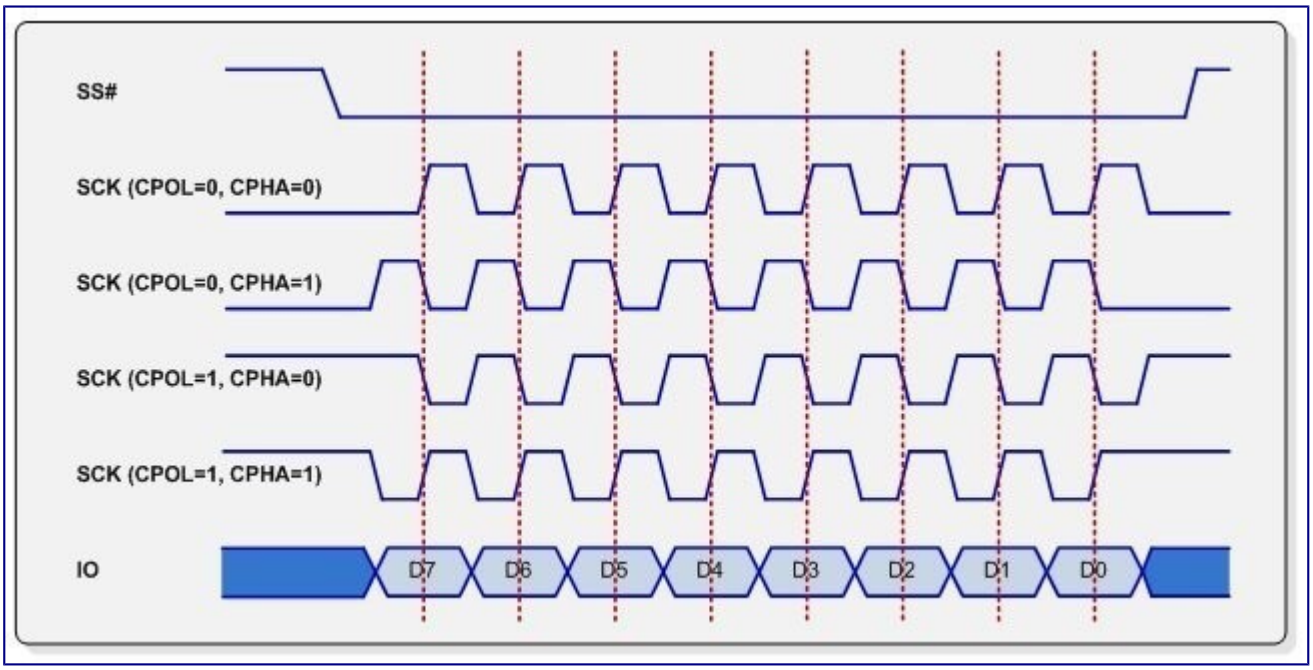
Modes:

```
#define SPI_CPHA                0x01
#define SPI_CPOL                0x02

#define SPI_MODE_0              (0|0)
```

```
#define SPI_MODE_1          (0|SPI_CPHA)
#define SPI_MODE_2          (SPI_CPOL|0)
#define SPI_MODE_3          (SPI_CPOL|SPI_CPHA)
```

SPI Mode: Polarity and Clock Phase

The SPI interface defines no protocol for data exchange, limiting overhead and allowing for high speed data streaming. Clock polarity (CPOL) and clock phase (CPHA) can be specified as ‘0’ or ‘1’ to form four unique modes to provide flexibility in communication between master and slave as shown in Figure .



SPI Driver : Structure

```
struct spi_driver {
        const struct spi_device_id *id_table;
        int                     (*probe)(struct spi_device *spi);
        int                     (*remove)(struct spi_device *spi);
        void                    (*shutdown)(struct spi_device *spi);
        struct device_driver    driver;
}
```

SPI Device : Structure

```
struct spi_device {
        struct device           dev;
        struct spi_controller   *controller;
        struct spi_controller   *master;        /* compatibility layer */
```

```
	u32			max_speed_hz;
	u8			chip_select;
	u8			bits_per_word;
	u16			mode;
	int			irq;
	void			*controller_state;
	void			*controller_data;
	char			modalias[SPI_NAME_SIZE];
	int			cs_gpio;	/* chip select gpio */
	bool			multi_die;	/* flash with multiple dies*/

	/* the statistics */
	struct spi_statistics	statistics;

};
```

SPI DRIVER : API

```
extern int __spi_register_driver(struct module *owner, struct spi_driver *sdrv);

/**
 * spi_unregister_driver - reverse effect of spi_register_driver
 * @sdrv: the driver to unregister
 * Context: can sleep
 */
static inline void spi_unregister_driver(struct spi_driver *sdrv)
{
	if (sdrv)
		driver_unregister(&sdrv->driver);
}

/* use a define to avoid include chaining to get THIS_MODULE */
#define spi_register_driver(driver)		__spi_register_driver(THIS_MODULE, driver)


static inline int  spi_read(struct spi_device *spi, void *buf, size_t len)
{
	struct spi_transfer	t = {
			.rx_buf		= buf,
			.len		= len,
		};

	return spi_sync_transfer(spi, &t, 1);
}

static inline int  spi_write(struct spi_device *spi, const void *buf, size_t len)
{
	struct spi_transfer	t = {
			.tx_buf		= buf,
			.len		= len,
```

```
        };

    return spi_sync_transfer(spi, &t, 1);
}

static inline int  spi_sync_transfer(struct spi_device *spi, struct spi_transfer *xfers,    unsigned int num_xfers)
{
    struct spi_message msg;

    spi_message_init_with_transfers(&msg, xfers, num_xfers);

    return spi_sync(spi, &msg);
}
```

Device Tree Node:

```
&spi_0 {
        pinctrl-0 = <&spi_0_pins>;
        pinctrl-names = "default";
        status = "ok";
        cs-gpios = <&tlmm 54 0>;

        mx25l25635e@0 {
            #address-cells = <1>;
            #size-cells = <1>;
            reg = <0>;
            compatible = "mx25l25635e";
            spi-max-frequency = <24000000>;
        };
    };
```