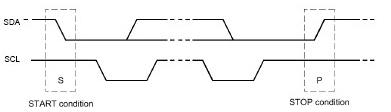
**I2C Protocol**

I²C bus is an abbreviation for Inter Integrated Circuit bus. It is also known as IIC and I2C.

I²C is a serial and synchronous bus protocol. In standard applications hardware and timing are often the same. The way data is treated on the I²C bus is to be defined by the manufacturer of the I²C master and slave chips.

In a simple I²C system there can only be one master, but multiple slaves. The difference between master and slave is that the master generates the clock pulse. The master also defines when communication should occur. For bus timing it is important that the slowest slave should still be able to follow the master’s clock. In other words the bus is as fast as the slowest slave.



Data can only occur after the master generates a **start condition.** A start condition is a high-to-low transition of the SDA line while SCL remains high. After each data transfer a **stop condition** is generated. A stop condition is a low-to-high transition of the SDA line while SCL remains high.

**Speed of Operation**

Standard Mode Speed: 100 kHz

Fast Mode Speed: 400 kHz

**Operation**

As said a data transfer can occur after a **start condition** of the master. The length of data sent over I²C is always 8 bit this includes a read/write direction bit, so you can effectively send 7 bits every time.

The most significant bit MSB is always passed first on the bus.

If the master writes to the bus the R/W bit = 0 and if the master reads the R/W bit = 1.

After the R/W bit the master should generate one clock period for an acknowledgement ACK.

Each receiving chip that is addressed is obliged to generate an acknowledge after the reception of each byte. A chip that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable LOW during the HIGH period of the acknowledge related clock pulse.

After an acknowledge there can be a stop condition, if the master wishes to leave the bus idle. Or a repeated start condition. A repeated start is the same as a start condition.

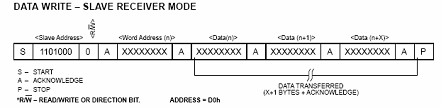
When the master reads from a slave it should acknowledge after each byte received. There are two reasons for the master not to acknowledge. The master sends a not acknowledge if data was not received correctly or if the master wishes the stop receiving.

**In other words if the master wishes to stop receiving, it sends a not acknowledge after the last received byte.**

The master can stop any communication on the bus **at any time** by sending a stop condition.

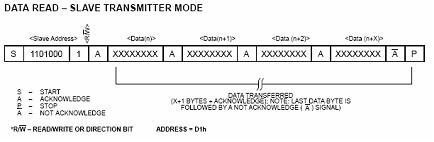
**Bus addressing**

Let’s say we have a slave chip with the address “1101000” and that the master wishes to write to that slave, the slave would then be in receiver mode, like this:

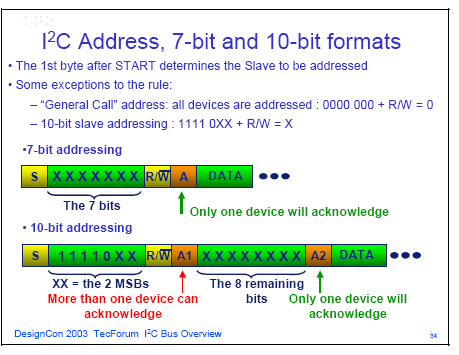


You can see here that the master always generates the start condition, then the master sends the address of the slave and a “0” for R/W. After that the master sends a command or word address. The function of that command or word address can be found in the data sheet of the slave addressed.

After that the master can send the data desired and stop the transfer with a stop condition.



Again the start condition and the slave address, only this time the master sends “1” for the R/W bit. The slave can then begin to send after the acknowledge. If the master wishes to stop receiving it should send a not acknowledge.



**Summary of flow**

The first thing that will happen is that the master will send out a start sequence.

This will alert all the slave devices on the bus that a transaction is starting and they should listen in incase it is for them. Next the master will send out the device address. The slave that matches this address will continue with the transaction, any others will ignore the rest of this transaction and wait for the next.

Having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has.

Having sent the I2C address and the internal register address  the master can now send the data byte (or bytes, it doesn't have to be just one). The master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte.

When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction.

**So to write to a slave device:**

1. Send a start sequence
2. Send the I2C address of the slave with the R/W bit low (even address)
3. Send the internal register number you want to write to
4. Send the data byte
5. Optionally, send any further data bytes.
6. Send the stop sequence.

**Read from a slave**

Before reading data from the slave device, you must tell it which of its internal addresses you want to read. So a read of the slave actually starts off by writing to it. This is the same as when you want to write to it: You send the start sequence, the I2C address of the slave with the R/W bit low (even address) and the internal register number you want to write to. Now you send another start sequence (sometimes called a restart) and the I2C address again - this time with the read bit set. You then read as many data bytes as you wish and terminate the transaction with a stop sequence.

1. Send a start sequence
2. Send I2C address with the R/W bit low (even address)
3. Send Internal address of the bearing register
4. Send a start sequence again (repeated start)
5. Send I2C address with the R/W bit high (odd address)
6. Read data byte from slave.
7. Send the stop sequence.

**I2C Bus Terminology**

1. **Transmitter** - the device that sends data to the bus. A transmitter can either be a device that puts data on the bus of its own accord (a ‘master-transmitter’), or in response to a request from data from another devices (a ‘slave-transmitter’).
2. **Receiver** - the device that receives data from the bus.
3. **Master** - the component that initializes a transfer, generates the clock signal, and terminates the transfer. A master can be either a transmitter or a receiver.
4. **Slave** - the device addressed by the master. A slave can be either receiver or transmitter.
5. **Multi-master** - the ability for more than one master to co-exist on the bus at the same time without collision or data loss.
6. **Arbitration** - the prearranged procedure that authorizes only one master at a time to take control of the bus.
7. **Synchronization** - the prearranged procedure that synchronizes the clock signals provided by two or more masters.
8. **SDA** - data signal line (Serial Data)
9. **SCL** - clock signal line (Serial Clcok)

**Terminology for Bus Transfer**

1. **F (FREE)** - the bus is free; the data line SDA and the SCL clock are both in the high state.
2. **S (START) or SR (Repeated START)** - data transfer begins with a start condition (not a start bit). The level of the SDA data line changes from high to low, while the SCL clock line remains high. When this occurs, the bus is ‘busy’.
3. **C (CHANGE)** - while the SCL clock line is low, the data bit to be transferred can be applied to the SDA data line by a transmitter. During this time, SDA may change its state, as along as the SCL line remains low.
4. **D (DATA)** - a high or low bit of information on the SDA data line is valid during the high level of the SCL clock line. This level must be maintained stable during the entire time that the clock remains high to avoid misinterpretation as a Start or Stop condition.
5. **P (STOP)** - data transfer is terminated by a stop condition, (not a stop bit). This occurs when the level on the SDA data line passes from the low state to the high state, while the SCL clock line remains high. When the data transfer has been terminated, the bus is free once again.

**Linux kernel Code Explanation wrt I2C**

The following are the logical software components as defined by the Linux I2C subsystem.

***The I2C algorithm driver****:* Each I2C bus adapter has its own way of interfacing with the processor and the I2C bus. In the above example both bus adapters use the PCF style of interfacing, which defines the registers that need to be implemented by the bus adapter and the implementation of the algorithms for transmitting and receiving data. The algorithm driver implements the basic data handshake routines (transmit and receive).

***The I2C adapter driver****:* This can be considered as a BSP layer for the I2C subsystem. The I2C adapter driver and the algorithm driver together drive the I2C buses on the system. In the above example we define two I2C adapter drivers for each of the two buses on the system. Both these adapter

drivers are bound to the PCF8584 algorithm driver.

***The I2C slave driver****:* The slave driver contains the routines to access a particular kind of slave device on the I2C bus. In our example we provide two slave drivers: one for accessing the EEPROMs on the first I2C bus and the other for accessing the RTC chip on the second I2C bus.

***The I2C client driver****:* One client driver is instantiated for hardware that needs to be accessed via the I2C bus. The slave and client drivers are bound together. In our example we need to define three client drivers: two EEPROM client drivers and one RTC client driver.

The I2C subsystem is located in the *drivers/i2c* directory of the kernel source tree. In that directory the buses subdirectory contains the various ***bus adapter drivers, algos*** contains the various algorithm drivers, and the chips directory contains the various slave and client drivers. The generic portion of the entire I2C subsystem is referred to as the I2C core and is implemented in the file ***drivers/i2c/i2c-core.c.***

**All the below structures are declared in include/linux/i2c.h**

**Algorithm structure members**

*functionality*: This is a pointer to a function that returns those features supported by the adapter such as what message types are supported by the I2C driver.

*master\_xfe*r: A function pointer which points to the function that implements the actual

I2C transmit and receive algorithm.

int (\*master\_xfer)(struct i2c\_adapter \*adap,struct i2c\_msg \*msgs, int num);

**Adapter structure members**

The algorithm driver by itself does not make sense unless it is bound by the I2C bus adapter driver. Each adapter driver is associated with a data structure i2c\_adapter (declared in the file include/ linux/i2c.h) that is instantiated by the adapter driver. The adapter driver calls the function i2c\_pcf\_add\_bus with a pointer to the i2c\_adapter structure. The important fields of the i2c\_adapter structure that are set up

by the adapter driver are:

struct i2c\_adapter {

struct module \*owner;

unsigned int id; //Each adapter is identified using a unique number. The different types

of algorithms are defined in the header file include/linux/i2c-id.h.

unsigned int class; //This indicates the type of I2C class devices that this driver supports.

const struct i2c\_algorithm \*algo; //The pointer to the i2c\_algorithm data structure

void \*algo\_data; //This is a pointer to the algorithm-specific private data structure(driver data)

}

**Client structure**

struct i2c\_client - represent an I2C slave device

struct i2c\_client {

unsigned short flags; //I2C\_CLIENT\_TEN indicates the device uses a ten bit chip address;

unsigned short addr; //Address used on the I2C bus connected to the parent adapter.

char name[I2C\_NAME\_SIZE];

struct i2c\_adapter \*adapter; /\* the adapter we sit on \*/

struct i2c\_driver \*driver; /\* and our access routines \*/

}

An i2c\_client identifies a single device (i.e. chip) connected to an \* i2c bus.

**The driver structure(slave/client driver)**

A i2c driver structure contains general access routines, and should be zero-initialized except for fields with data you provide. A client structure holds device-specific information like the driver model device node, and its I2C address.

static struct i2c\_driver foo\_driver = {

.driver = {

.name = "foo",

},

/\* iff driver uses driver model ("new style") binding model: \*/

.probe = foo\_probe,

.remove = foo\_remove,

/\* else, driver uses "legacy" binding model: \*/

.attach\_adapter = foo\_attach\_adapter,

.detach\_client = foo\_detach\_client,

/\* these may be used regardless of the driver binding model \*/

.shutdown = foo\_shutdown, /\* optional \*/

.suspend = foo\_suspend, /\* optional \*/

.resume = foo\_resume, /\* optional \*/

.command = foo\_command, /\* optional \*/

}

The name field is the driver name, and must not contain spaces. It should match the module name (if the driver can be compiled as a module), although you can use MODULE\_ALIAS (passing "foo" in this example) to add another name for the module. If the driver name doesn't match the module name, the module won't be automatically loaded (hotplug/coldplug).

**Attaching to an adapter**

Whenever a new adapter is inserted, or for all adapters if the driver is being registered, the callback attach\_adapter() is called. Now is the time to determine what devices are present on the adapter, and to register a client for each of them.

The attach\_adapter callback is really easy: we just call the generic detection function. This function will scan the bus for us, using the information as defined in the lists explained above. If a device is

detected at a specific address, another callback is called.

int foo\_attach\_adapter(struct i2c\_adapter \*adapter)

{

return i2c\_probe(adapter,&addr\_data,&foo\_detect\_client);

}

The i2c\_probe function will call the foo\_detect\_client function only for those i2c addresses that actually have a device on them (unless a `force' parameter was used). In addition, addresses that are already in use (by some other registered client) are skipped**.**

The i2c\_probe function is provided by the I2C core; this function uses the information in the data structure addr\_data to call the function detect() which fills up the new client structure and registers to the subsystem. The addr\_data structure is declared in the file include/linux/ i2c.h. The addr\_data function is used to do the following.

/\* i2c\_client\_address\_data is the struct for holding default client addresses for a driver and for the parameters supplied on the command line \*/

struct i2c\_client\_address\_data {

const unsigned short \*normal\_i2c; //Function in a normal mode that just picks up an I2C

device at a given address and detects its presence

const unsigned short \*probe; //Probe an I2C device at a given address using the adapter and detect its presence

const unsigned short \*ignore; //Ignore an I2C device at a given address

const unsigned short \* const \*forces; //Force an I2C device at a given address to be registered

as a client without detection

};

static const struct i2c\_client\_address\_data addr\_data = {

.normal\_i2c = normal\_i2c,

.probe = probe,

.ignore = ignore,

.forces = forces,

}

So we prefer to use the first argument and define the i2c address by referring the data sheet.So the i2c\_probe function will call the i2c\_probe\_address() and the latter will verify the address provided by the slave is valid or not by using some address range(7 bit addr) i.e

if (addr < 0x03 || addr > 0x77) {

dev\_warn(&adapter->dev, "Invalid probe address 0x%02x\n",

addr);

return -EINVAL;

}

And if address is valid, then the popular i2c\_detect() will be called.

So Now the detect function creates the i2c\_client data structure for the client and fills the clent structure like in the external uart(XR20M1172)

static int is7x2\_detect(struct i2c\_adapter \*adapter, int address, int kind)

{

struct i2c\_client \*new\_client;

struct is7x2\_data \*data; //current slave driver structure

new\_client = &data->client;

i2c\_set\_clientdata(new\_client, data);

new\_client->addr = address;

new\_client->adapter = adapter;

new\_client->driver = &is7x2\_driver;

new\_client->flags = 0;

/\* We can fill in the remaining client fields \*/

strlcpy(new\_client->name, name, I2C\_NAME\_SIZE);

/\* Tell the I2C layer a new client has arrived \*/

if ((err = i2c\_attach\_client(new\_client))) {

goto exit\_free;

xrm1172\_client = new\_client;

}

So once slave being detected by master,now comes the read and write operation through i2c.We will take example again of external uart(xr20m1172).For reading and writing of i2c we usually use one structure

i.e

struct i2c\_msg

& one function

i2c\_transfer()

We are filling the struct i2c\_msg with the required slave information and passes this structure to i2c\_transfer().

**struct i2c\_msg** {

\_\_u16 addr; /\* slave address \*/

\_\_u16 flags;

#define I2C\_M\_TEN 0x0010 /\* this is a ten bit chip address \*/

#define I2C\_M\_RD 0x0001 /\* read data, from slave to master \*/

#define I2C\_M\_NOSTART 0x4000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_REV\_DIR\_ADDR 0x2000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_IGNORE\_NAK 0x1000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_NO\_RD\_ACK 0x0800 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_RECV\_LEN 0x0400 /\* length will be first received byte \*/

\_\_u16 len; /\* msg length \*/

\_\_u8 \*buf; /\* pointer to msg data \*/

};

**i2c\_transfer()**

extern int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msg, int num);

This sends a series of messages. Each message can be a read or write, and they can be mixed in any way. The transactions are combined: no stop bit is sent between transaction. The i2c\_msg structure contains

for each message the client address, the number of bytes of the message and the message data itself.

The above function will in turn calls the master\_xfer(function pointer declared as a member of algo structure) which is pointing to i2c\_xfer defined in drivers/i2c/busses/i2c-s3c2410.c(according to arch).

static int s3c24xx\_i2c\_xfer(struct i2c\_adapter \*adap,struct i2c\_msg \*msgs, int num)

This function again calls i2c\_doxfer(),which starts the actual i2c\_transfer(i2c protocol implementation)

static int s3c24xx\_i2c\_doxfer(struct s3c24xx\_i2c \*i2c, struct i2c\_msg \*msgs, int num)

In this function following are the steps implemented

1. Reads the status Register of i2c and checks whether the bus is busy with any other slave or not.
2. Puts the message/data and length of bytes in the message structure of s3c24xx i.e

i2c->msg = msgs;

i2c->msg\_num = num;

1. The s3c24xx structure is

struct s3c24xx\_i2c {//only important members

struct i2c\_msg \*msg;

unsigned int msg\_num;

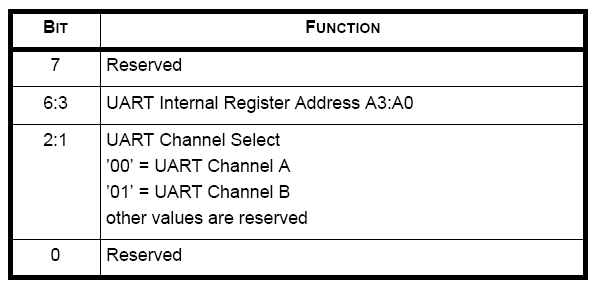
};

1. Setting the clocks in clock waveform register (in atmel/no info about s3c2440)
2. Configuring the i2c control register and setting the master enable bit and irq bit if required.
3. Then Checking the status register for read or write with the I2C\_M\_RD flag.
4. Checking for ACK after every transaction by reading the status register.
5. In atmel,i2c-at91.c,After master\_xfer,i2c\_xfer,xfer\_read & xfer\_write are available based on I2C\_M\_RD flag.
6. Then issues a stop bit.

Note:Start bit in atmel is not supported.

**Reading and Writing Example(External uart-xr20m1172)**

**This is according to uart-i2c protocol format which says**

****

static int is7x2\_reg\_write(u8 reg, char ch, u8 value)

{

int result;

struct i2c\_msg msg;

u8 \*temp;

struct i2c\_client \*client = xrm1172\_client;

temp = kmalloc(2, GFP\_KERNEL);

temp[0] = (reg<<3)|(ch<<1);

temp[1] = value;

msg.addr = client->addr;

msg.flags = 0;

msg.len = 2;

msg.buf = temp;

result = i2c\_transfer(client->adapter, &msg, 1);

if (result != 2 )

return -EFAULT;

kfree(temp);

return 0;

}

static u8 is7x2\_reg\_read(unsigned char reg, char ch)

{

struct i2c\_client \*client = xrm1172\_client;

struct i2c\_msg msg;

unsigned char \*temp;

int ret;

unsigned char \*temp1;

temp = kmalloc(2, GFP\_KERNEL);

temp[0] = ( reg << 3 ) | (ch<<1);

msg.addr = client->addr;

msg.flags = 0;

msg.len = 1;

msg.buf = &temp[0];

ret = i2c\_transfer(client->adapter, &msg, 1);

if (ret != 1) {

printk("%s: %s i2c\_transfer error\n", \_\_FILE\_\_, \_\_FUNCTION\_\_);

return -EFAULT;

}

msg.addr = client->addr;

msg.flags |= I2C\_M\_RD;

msg.len = 1;

msg.buf = &temp[1];

ret = i2c\_transfer(client->adapter, &msg, 1);

if (ret != 1) {

printk("%s: %s i2c\_transfer error\n",\_\_FILE\_\_, \_\_FUNCTION\_\_);

return -EFAULT;

}

rdata = temp[1];

kfree(temp);

return rdata;

}

**So in the init function of slave i.e**

static struct i2c\_driver is7x2\_driver = {

.driver = {

.name = "xrm1172pw",

},

.attach\_adapter = is7x2\_attach\_adapter,

.detach\_client = is7x2\_detach\_client,

};

static int \_\_init is7x2\_init(void)

{

i2c\_add\_driver(&is7x2\_driver);

}

static inline int i2c\_add\_driver(struct i2c\_driver \*driver)

{

return i2c\_register\_driver(THIS\_MODULE, driver);

}

The above function : An i2c\_driver is used with one or more i2c\_client (device) nodes to access i2c slave chips, on a bus instance associated with some i2c\_adapter. There are two models for binding the driver to its device: "new style" drivers follow the standard Linux driver model and just respond to probe() calls

issued if the driver core sees they match(); "legacy" drivers create device nodes themselves.

static void \_\_exit pcf8591\_exit(void) //this is the exit fucntion

{

i2c\_del\_driver(&pcf8591\_driver); i2c\_del\_driver - unregister I2C driver

driver: the driver being unregistered

} Context: can sleep

**Difference between SMBUS /I2C**

* The I²C bus and the SMBus are popular 2-wire buses that are essentially compatible with each other. Normally devices, both masters and slaves, are freely interchangeable between both buses. Both buses feature addressable slaves (although specific address allocations can vary between the two buses). The buses operate at the same speed, up to 100kHz, but the I²C bus has both 400kHz and 2MHz versions.
* Timeout and (as a consequence of timeout) minimum clock speed are the most important differences between the I²C bus and the SMBus.

I²C Bus = DC (no timeout)  
SMBus = 10kHz (35mS timeout)

* Timeout is where a slave device resets its interface whenever Clock goes low for longer than the timeout, typically 35mSec. Use of a timeout also dictates a minimum speed for the clock, because it can never go static. Thus, the SMBus has a minimum-clock-speed specification. By comparison, the I²C bus can go static indefinitely. In the I²C bus, either a master or a slave can hold the clock low as long as necessary to process data.
* SMbus is having different read/write api’s i.e

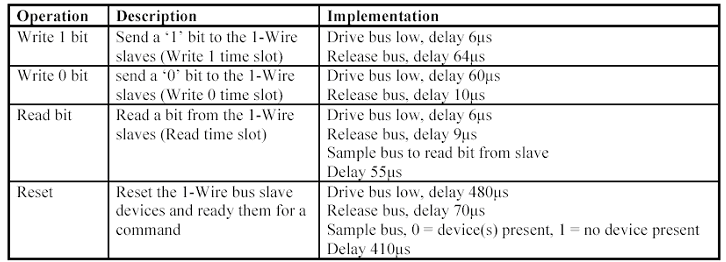
extern s32 i2c\_smbus\_xfer (),s32 i2c\_smbus\_read\_byte(),i2c\_smbus\_write\_byte(), etc…

**What is one Wire Communication**

The 1-wire protocol was invented by Dallas Semiconductors and needs only 1 wire for two-way communication. You also need power and ground of course. 1-Wire is similar in concept to [I²C](http://en.wikipedia.org/wiki/I%C2%B2C), but with lower data rates and longer range.

The basis of 1-Wire technology is a serial protocol using a single data line plus ground reference for communication. A 1-Wire master initiates and controls the communication with one or more 1-Wire slave devices on the 1-Wire bus. Each 1-Wire slave device has a unique, unalterable, factory-programmed, 64-bit ID (identification number), which serves as device address on the 1-Wire bus.

**1-Wire Operations**

****