# I$^2$C and SMBus Subsystem

I$^2$C (or without fancy typography, “I2C”) is an acronym for the “Inter-IC” bus, a simple bus protocol which is widely used where low data rate communications suffice. Since it’s also a licensed trademark, some vendors use another name (such as “Two-Wire Interface”, TWI) for the same bus. I2C only needs two signals (SCL for clock, SDA for data), conserving board real estate and minimizing signal quality issues. Most I2C devices use seven bit addresses, and bus speeds of up to 400 kHz; there’s a high speed extension (3.4 MHz) that’s not yet found wide use. I2C is a multi-master bus; open drain signaling is used to arbitrate between masters, as well as to handshake and to synchronize clocks from slower clients.

The Linux I2C programming interfaces support the master side of bus interactions and the slave side. The programming interface is structured around two kinds of driver, and two kinds of device. An I2C “Adapter Driver” abstracts the controller hardware; it binds to a physical device (perhaps a PCI device or platform_device) and exposes a **`struct i2c_adapter`** representing each I2C bus segment it manages. On each I2C bus segment will be I2C devices represented by a **`struct i2c_client`**. Those devices will be bound to a **`struct i2c_driver`**, which should follow the standard Linux driver model. There are functions to perform various I2C protocol operations; at this writing all such functions are usable only from task context.

The System Management Bus (SMBus) is a sibling protocol. Most SMBus systems are also I2C conformant. The electrical constraints are tighter for SMBus, and it standardizes particular protocol messages and idioms. Controllers that support I2C can also support most SMBus operations, but SMBus controllers don’t support all the protocol options that an I2C controller will. There are functions to perform various SMBus protocol operations, either using I2C primitives or by issuing SMBus commands to i2c_adapter devices which don’t support those I2C operations.

## `int i2c_master_recv(const struct i2c_client *client, char *buf, int count)`

issue a single I2C message in master receive mode

**Parameters**

`const struct i2c_client *client`
Handle to slave device

`char *buf`
Where to store data read from slave

`int count`
How many bytes to read, must be less than 64k since msg.len is u16

**Description**

Returns negative errno, or else the number of bytes read.

## `int i2c_master_recv_dmasafe(const struct i2c_client *client, char *buf, int count)`

issue a single I2C message in master receive mode using a DMA safe buffer

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`char *buf`
: Where to store data read from slave, must be safe to use with DMA

`int count`
: How many bytes to read, must be less than 64k since msg.len is u16

**Description**

Returns negative errno, or else the number of bytes read.

## `int i2c_master_send(const struct i2c_client *client, const char *buf, int count)`

issue a single I2C message in master transmit mode

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`const char *buf`
: Data that will be written to the slave

`int count`
: How many bytes to write, must be less than 64k since msg.len is u16

**Description**

Returns negative errno, or else the number of bytes written.

## `int i2c_master_send_dmasafe(const struct i2c_client *client, const char *buf, int count)`

issue a single I2C message in master transmit mode using a DMA safe buffer

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`const char *buf`
: Data that will be written to the slave, must be safe to use with DMA

`int count`
: How many bytes to write, must be less than 64k since msg.len is u16

**Description**

Returns negative errno, or else the number of bytes written.

## struct i2c_device_identity

i2c client device identification

**Definition**:

```
struct i2c_device_identity {
    u16 manufacturer_id;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS                0;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_1              1;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_2              2;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_3              3;
#define I2C_DEVICE_ID_RAMTRON_INTERNATIONAL             4;
#define I2C_DEVICE_ID_ANALOG_DEVICES                    5;
#define I2C_DEVICE_ID_STMICROELECTRONICS                6;
#define I2C_DEVICE_ID_ON_SEMICONDUCTOR                  7;
#define I2C_DEVICE_ID_SPRINTEK_CORPORATION              8;
#define I2C_DEVICE_ID_ESPROS_PHOTONICS_AG               9;
#define I2C_DEVICE_ID_FUJITSU_SEMICONDUCTOR            10;
#define I2C_DEVICE_ID_FLIR                             11;
#define I2C_DEVICE_ID_O2MICRO                          12;
#define I2C_DEVICE_ID_ATMEL                            13;
#define I2C_DEVICE_ID_NONE                         0xffff;
    u16 part_id;
    u8 die_revision;
};
```

**Members**

`manufacturer_id`
: 0 - 4095, database maintained by NXP

`part_id`
: 0 - 511, according to manufacturer

`die_revision`
: 0 - 7, according to manufacturer

## enum i2c_driver_flags

Flags for an I2C device driver

**Constants**

`I2C_DRV_ACPI_WAIVE_D0_PROBE`
: Don’t put the device in D0 state for probe

## struct i2c_driver

represent an I2C device driver

**Definition**:

```
struct i2c_driver {
    unsigned int class;
    int (*probe)(struct i2c_client *client);
    void (*remove)(struct i2c_client *client);
    void (*shutdown)(struct i2c_client *client);
    void (*alert)(struct i2c_client *client, enum i2c_alert_protoc
    int (*command)(struct i2c_client *client, unsigned int cmd, vc
    struct device_driver driver;
    const struct i2c_device_id *id_table;
    int (*detect)(struct i2c_client *client, struct i2c_board_infc
    const unsigned short *address_list;
    struct list_head clients;
    u32 flags;
};
```

**Members**

`class`
: What kind of i2c device we instantiate (for detect)

`probe`
: Callback for device binding

`remove`
: Callback for device unbinding

`shutdown`
: Callback for device shutdown

`alert`
: Alert callback, for example for the SMBus alert protocol

`command`
: Callback for bus-wide signaling (optional)

`driver`
: Device driver model driver

`id_table`
: List of I2C devices supported by this driver

`detect`
: Callback for device detection

`address_list`
: The I2C addresses to probe (for detect)

`clients`
List of detected clients we created (for i2c-core use only)

`flags`
A bitmask of flags defined in **`enum i2c_driver_flags`**

**Description**

The driver.owner field should be set to the module owner of this driver. The driver.name field should be set to the name of this driver.

For automatic device detection, both **detect** and **address_list** must be defined. **class** should also be set, otherwise only devices forced with module parameters will be created. The detect function must fill at least the name field of the i2c_board_info structure it is handed upon successful detection, and possibly also the flags field.

If **detect** is missing, the driver will still work fine for enumerated devices. Detected devices simply won't be supported. This is expected for the many I2C/SMBus devices which can't be detected reliably, and the ones which can always be enumerated in practice.

The i2c_client structure which is handed to the **detect** callback is not a real i2c_client. It is initialized just enough so that you can call i2c_smbus_read_byte_data and friends on it. Don't do anything else with it. In particular, calling dev_dbg and friends on it is not allowed.

## struct i2c_client

represent an I2C slave device

**Definition**:

```
struct i2c_client {
    unsigned short flags;
#define I2C_CLIENT_PEC          0x04    ;
#define I2C_CLIENT_TEN          0x10    ;
#define I2C_CLIENT_SLAVE        0x20    ;
#define I2C_CLIENT_HOST_NOTIFY  0x40    ;
#define I2C_CLIENT_WAKE         0x80    ;
#define I2C_CLIENT_SCCB         0x9000  ;
    unsigned short addr;
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter;
    struct device dev;
    int init_irq;
    int irq;
    struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE);
    i2c_slave_cb_t slave_cb;
#endif;
    void *devres_group_id;
    struct dentry *debugfs;
};
```

**Members**

`flags`
: see I2C_CLIENT_* for possible flags

`addr`
: Address used on the I2C bus connected to the parent adapter.

`name`
: Indicates the type of the device, usually a chip name that's generic enough to hide second-sourcing and compatible revisions.

`adapter`
: manages the bus segment hosting this I2C device

`dev`
: Driver model device node for the slave.

`init_irq`
: IRQ that was set at initialization

`irq`
: indicates the IRQ generated by this device (if any)

`detected`
: member of an i2c_driver.clients list or i2c-core's userspace_devices list

`slave_cb`
: Callback when I2C slave mode of an adapter is used. The adapter calls it to pass on slave events to the slave driver.

`devres_group_id`
: id of the devres group that will be created for resources acquired when probing this device.

`debugfs`
: pointer to the debugfs subdirectory which the I2C core created for this client.

**Description**

An i2c_client identifies a single device (i.e. chip) connected to an i2c bus. The behaviour exposed to Linux is defined by the driver managing the device.

## struct i2c_board_info

template for device creation

**Definition**:

```
struct i2c_board_info {
    char type[I2C_NAME_SIZE];
    unsigned short  flags;
```

```
        unsigned short  addr;
        const char      *dev_name;
        void *platform_data;
        struct device_node *of_node;
        struct fwnode_handle *fwnode;
        const struct software_node *swnode;
        const struct resource *resources;
        unsigned int    num_resources;
        int irq;
    };
```

**Members**

`type`
: chip type, to initialize i2c_client.name

`flags`
: to initialize i2c_client.flags

`addr`
: stored in i2c_client.addr

`dev_name`
: Overrides the default <busnr>-<addr> dev_name if set

`platform_data`
: stored in i2c_client.dev.platform_data

`of_node`
: pointer to OpenFirmware device node

`fwnode`
: device node supplied by the platform firmware

`swnode`
: software node for the device

`resources`
: resources associated with the device

`num_resources`
: number of resources in the **resources** array

`irq`
: stored in i2c_client.irq

**Description**

I2C doesn’t actually support hardware probing, although controllers and devices may be able to use I2C_SMBUS_QUICK to tell whether or not there’s a device at a given address. Drivers commonly need more information than that, such as chip type, configuration, associated IRQ, and so on.

i2c_board_info is used to build tables of information listing I2C devices that are present. This information is used to grow the driver model tree. For mainboards this is done statically using **`i2c_register_board_info()`**; bus numbers identify adapters that aren't yet available. For add-on boards, **`i2c_new_client_device()`** does this dynamically with the adapter already known.

## I2C_BOARD_INFO

`I2C_BOARD_INFO (dev_type, dev_addr)`

macro used to list an i2c device and its address

**Parameters**

`dev_type`
identifies the device type

`dev_addr`
the device's address on the bus.

**Description**

This macro initializes essential fields of a **`struct i2c_board_info`**, declaring what has been provided on a particular board. Optional fields (such as associated irq, or device-specific platform_data) are provided using conventional syntax.

## struct i2c_algorithm

represent I2C transfer methods

**Definition**:

```
struct i2c_algorithm {
    union {
        int (*xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs
        int (*master_xfer)(struct i2c_adapter *adap, struct i2c_ms
    };
    union {
        int (*xfer_atomic)(struct i2c_adapter *adap, struct i2c_ms
        int (*master_xfer_atomic)(struct i2c_adapter *adap, struct
    };
    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,unsigned
    int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,un
    u32 (*functionality)(struct i2c_adapter *adap);
#if IS_ENABLED(CONFIG_I2C_SLAVE);
    union {
        int (*reg_target)(struct i2c_client *client);
        int (*reg_slave)(struct i2c_client *client);
    };
    union {
        int (*unreg_target)(struct i2c_client *client);
        int (*unreg_slave)(struct i2c_client *client);
    };
#endif;
};
```

**Members**

`{unnamed_union}`
: anonymous

`xfer`
: Transfer a given number of messages defined by the msgs array via the specified adapter.

`master_xfer`
: deprecated, use **xfer**

`{unnamed_union}`
: anonymous

`xfer_atomic`
: Same as **xfer**. Yet, only using atomic context so e.g. PMICs can be accessed very late before shutdown. Optional.

`master_xfer_atomic`
: deprecated, use **xfer_atomic**

`smbus_xfer`
: Issue SMBus transactions to the given I2C adapter. If this is not present, then the bus layer will try and convert the SMBus calls into I2C transfers instead.

`smbus_xfer_atomic`
: Same as **smbus_xfer**. Yet, only using atomic context so e.g. PMICs can be accessed very late before shutdown. Optional.

`functionality`
: Return the flags that this algorithm/adapter pair supports from the `I2C_FUNC_*` flags.

`{unnamed_union}`
: anonymous

`reg_target`
: Register given client to local target mode of this adapter

`reg_slave`
: deprecated, use **reg_target**

`{unnamed_union}`
: anonymous

`unreg_target`
: Unregister given client from local target mode of this adapter

`unreg_slave`
deprecated, use **unreg_target**

**Description**

i2c_algorithm is the interface to a class of hardware solutions which can be addressed using the same bus algorithms - i.e. bit-banging or the PCF8584 to name two of the most common.

The return codes from the `xfer{_atomic}` fields should indicate the type of error code that occurred during the transfer, as documented in the Kernel Documentation file I2C/SMBUS Fault Codes. Otherwise, the number of messages executed should be returned.

## struct i2c_lock_operations

represent I2C locking operations

**Definition**:

```
struct i2c_lock_operations {
    void (*lock_bus)(struct i2c_adapter *adapter, unsigned int fla
    int (*trylock_bus)(struct i2c_adapter *adapter, unsigned int f
    void (*unlock_bus)(struct i2c_adapter *adapter, unsigned int f
};
```

**Members**

`lock_bus`
Get exclusive access to an I2C bus segment

`trylock_bus`
Try to get exclusive access to an I2C bus segment

`unlock_bus`
Release exclusive access to an I2C bus segment

**Description**

The main operations are wrapped by i2c_lock_bus and i2c_unlock_bus.

## struct i2c_timings

I2C timing information

**Definition**:

```
struct i2c_timings {
    u32 bus_freq_hz;
    u32 scl_rise_ns;
    u32 scl_fall_ns;
    u32 scl_int_delay_ns;
    u32 sda_fall_ns;
```

```
        u32 sda_hold_ns;
        u32 digital_filter_width_ns;
        u32 analog_filter_cutoff_freq_hz;
    };
```

**Members**

`bus_freq_hz`
: the bus frequency in Hz

`scl_rise_ns`
: time SCL signal takes to rise in ns; t(r) in the I2C specification

`scl_fall_ns`
: time SCL signal takes to fall in ns; t(f) in the I2C specification

`scl_int_delay_ns`
: time IP core additionally needs to setup SCL in ns

`sda_fall_ns`
: time SDA signal takes to fall in ns; t(f) in the I2C specification

`sda_hold_ns`
: time IP core additionally needs to hold SDA in ns

`digital_filter_width_ns`
: width in ns of spikes on i2c lines that the IP core digital filter can filter out

`analog_filter_cutoff_freq_hz`
: threshold frequency for the low pass IP core analog filter

## struct **i2c_bus_recovery_info**

I2C bus recovery information

**Definition**:

```
struct i2c_bus_recovery_info {
    int (*recover_bus)(struct i2c_adapter *adap);
    int (*get_scl)(struct i2c_adapter *adap);
    void (*set_scl)(struct i2c_adapter *adap, int val);
    int (*get_sda)(struct i2c_adapter *adap);
    void (*set_sda)(struct i2c_adapter *adap, int val);
    int (*get_bus_free)(struct i2c_adapter *adap);
    void (*prepare_recovery)(struct i2c_adapter *adap);
    void (*unprepare_recovery)(struct i2c_adapter *adap);
    struct gpio_desc *scl_gpiod;
    struct gpio_desc *sda_gpiod;
    struct pinctrl *pinctrl;
    struct pinctrl_state *pins_default;
    struct pinctrl_state *pins_gpio;
};
```

**Members**

`recover_bus`
: Recover routine. Either pass driver's recover_bus() routine, or i2c_generic_scl_recovery().

`get_scl`
: This gets current value of SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

`set_scl`
: This sets/clears the SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

`get_sda`
: This gets current value of SDA line. This or set_sda() is mandatory for generic SCL recovery. Populated internally, if sda_gpio is a valid GPIO, for generic GPIO recovery.

`set_sda`
: This sets/clears the SDA line. This or get_sda() is mandatory for generic SCL recovery. Populated internally, if sda_gpio is a valid GPIO, for generic GPIO recovery.

`get_bus_free`
: Returns the bus free state as seen from the IP core in case it has a more complex internal logic than just reading SDA. Optional.

`prepare_recovery`
: This will be called before starting recovery. Platform may configure padmux here for SDA/SCL line or something else they want.

`unprepare_recovery`
: This will be called after completing recovery. Platform may configure padmux here for SDA/SCL line or something else they want.

`scl_gpiod`
: gpiod of the SCL line. Only required for GPIO recovery.

`sda_gpiod`
: gpiod of the SDA line. Only required for GPIO recovery.

`pinctrl`
: pinctrl used by GPIO recovery to change the state of the I2C pins. Optional.

`pins_default`
: default pinctrl state of SCL/SDA lines, when they are assigned to the I2C bus. Optional. Populated internally for GPIO recovery, if state with the name PINCTRL_STATE_DEFAULT is found and pinctrl is valid.

`pins_gpio`
: recovery pinctrl state of SCL/SDA lines, when they are used as GPIOs. Optional. Populated internally for GPIO recovery, if this state is called "gpio" or "recovery" and

pinctrl is valid.

## struct i2c_adapter_quirks

describe flaws of an i2c adapter

**Definition**:

```
struct i2c_adapter_quirks {
    u64 flags;
    int max_num_msgs;
    u16 max_write_len;
    u16 max_read_len;
    u16 max_comb_1st_msg_len;
    u16 max_comb_2nd_msg_len;
};
```

**Members**

`flags`
: see I2C_AQ_* for possible flags and read below

`max_num_msgs`
: maximum number of messages per transfer

`max_write_len`
: maximum length of a write message

`max_read_len`
: maximum length of a read message

`max_comb_1st_msg_len`
: maximum length of the first msg in a combined message

`max_comb_2nd_msg_len`
: maximum length of the second msg in a combined message

**Description**

Note about combined messages: Some I2C controllers can only send one message per transfer, plus something called combined message or write-then-read. This is (usually) a small write message followed by a read message and barely enough to access register based devices like EEPROMs. There is a flag to support this mode. It implies max_num_msg = 2 and does the length checks with max_comb_*_len because combined message mode usually has its own limitations. Because of HW implementations, some controllers can actually do write-then-anything or other variants. To support that, write-then-read has been broken out into smaller bits like write-first and read-second which can be combined as needed.

## void i2c_lock_bus(struct i2c_adapter *adapter, unsigned int flags)

Get exclusive access to an I2C bus segment

**Parameters**

`struct i2c_adapter *adapter`
Target I2C bus segment

`unsigned int flags`
I2C_LOCK_ROOT_ADAPTER locks the root i2c adapter, I2C_LOCK_SEGMENT locks only this branch in the adapter tree

### `int i2c_trylock_bus(struct i2c_adapter *adapter, unsigned int flags)`

Try to get exclusive access to an I2C bus segment

**Parameters**

`struct i2c_adapter *adapter`
Target I2C bus segment

`unsigned int flags`
I2C_LOCK_ROOT_ADAPTER tries to locks the root i2c adapter, I2C_LOCK_SEGMENT tries to lock only this branch in the adapter tree

**Return**

true if the I2C bus segment is locked, false otherwise

### `void i2c_unlock_bus(struct i2c_adapter *adapter, unsigned int flags)`

Release exclusive access to an I2C bus segment

**Parameters**

`struct i2c_adapter *adapter`
Target I2C bus segment

`unsigned int flags`
I2C_LOCK_ROOT_ADAPTER unlocks the root i2c adapter, I2C_LOCK_SEGMENT unlocks only this branch in the adapter tree

### `void i2c_mark_adapter_suspended(struct i2c_adapter *adap)`

Report suspended state of the adapter to the core

**Parameters**

`struct i2c_adapter *adap`
Adapter to mark as suspended

**Description**

When using this helper to mark an adapter as suspended, the core will reject further transfers to this adapter. The usage of this helper is optional but recommended for devices having distinct handlers for system suspend and runtime suspend. More complex devices are free to implement custom solutions to reject transfers when suspended.

### void **i2c_mark_adapter_resumed**(struct i2c_adapter *adap)

Report resumed state of the adapter to the core

**Parameters**

`struct i2c_adapter *adap`
: Adapter to mark as resumed

**Description**

When using this helper to mark an adapter as resumed, the core will allow further transfers to this adapter. See also further notes to **`i2c_mark_adapter_suspended()`**.

### bool **i2c_check_quirks**(struct i2c_adapter *adap, u64 quirks)

Function for checking the quirk flags in an i2c adapter

**Parameters**

`struct i2c_adapter *adap`
: i2c adapter

`u64 quirks`
: quirk flags

**Return**

true if the adapter has all the specified quirk flags, false if not

## module_i2c_driver

`module_i2c_driver (__i2c_driver)`

Helper macro for registering a modular I2C driver

**Parameters**

`__i2c_driver`
: i2c_driver struct

**Description**

Helper macro for I2C drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces **`module_init()`** and **`module_exit()`**

## `builtin_i2c_driver`

`builtin_i2c_driver (__i2c_driver)`

Helper macro for registering a builtin I2C driver

**Parameters**

`__i2c_driver`
: i2c_driver struct

**Description**

Helper macro for I2C drivers which do not do anything special in their init. This eliminates a lot of boilerplate. Each driver may only use this macro once, and calling it replaces device_initcall().

## `int i2c_register_board_info(int busnum, struct i2c_board_info const *info, unsigned len)`

statically declare I2C devices

**Parameters**

`int busnum`
: identifies the bus to which these devices belong

`struct i2c_board_info const *info`
: vector of i2c device descriptors

`unsigned len`
: how many descriptors in the vector; may be zero to reserve the specified bus number.

**Description**

Systems using the Linux I2C driver stack can declare tables of board info while they initialize. This should be done in board-specific init code near arch_initcall() time, or equivalent, before any I2C adapter driver is registered. For example, mainboard init code could define several devices, as could the init code for each daughtercard in a board stack.

The I2C devices will be created later, after the adapter for the relevant bus has been registered. After that moment, standard driver model tools are used to bind “new style” I2C drivers to the devices. The bus number for any device declared using this routine is not available for dynamic allocation.

The board info passed can safely be __initdata, but be careful of embedded pointers (for platform_data, functions, etc) since that won’t be copied.

### `struct i2c_client *i2c_verify_client(struct device *dev)`

return parameter as i2c_client, or NULL

**Parameters**

`struct device *dev`
: device, probably from some driver model iterator

**Description**

When traversing the driver model tree, perhaps using driver model iterators like **`device_for_each_child()`**, you can't assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an i2c_client.

### `struct i2c_client *i2c_new_client_device(struct i2c_adapter *adap, struct i2c_board_info const *info)`

instantiate an i2c device

**Parameters**

`struct i2c_adapter *adap`
: the adapter managing the device

`struct i2c_board_info const *info`
: describes one I2C device; bus_num is ignored

**Context**

can sleep

**Description**

Create an i2c device. Binding is handled through driver model probe()/remove() methods. A driver may be bound to this device when we return from this function, or any later moment (e.g. maybe hotplugging will load the driver module). This call is not appropriate for use by mainboard initialization logic, which usually runs during an arch_initcall() long before any i2c_adapter could exist.

This returns the new i2c client, which may be saved for later use with **`i2c_unregister_device()`**; or an ERR_PTR to describe the error.

### `void i2c_unregister_device(struct i2c_client *client)`

reverse effect of i2c_new_*_device()

**Parameters**

`struct i2c_client *client`

value returned from i2c_new_*_device()

**Context**

can sleep

## `struct i2c_client *i2c_find_device_by_fwnode(struct fwnode_handle *fwnode)`

find an i2c_client for the fwnode

**Parameters**

`struct fwnode_handle *fwnode`
: **`struct fwnode_handle`** corresponding to the **`struct i2c_client`**

**Description**

Look up and return the **`struct i2c_client`** corresponding to the **fwnode**. If no client can be found, or **fwnode** is NULL, this returns NULL.

The user must call put_device(**`client->dev`**) once done with the i2c client.

## `struct i2c_client *i2c_new_dummy_device(struct i2c_adapter *adapter, u16 address)`

return a new i2c device bound to a dummy driver

**Parameters**

`struct i2c_adapter *adapter`
: the adapter managing the device

`u16 address`
: seven bit address to be used

**Context**

can sleep

**Description**

This returns an I2C client bound to the "dummy" driver, intended for use with devices that consume multiple addresses. Examples of such chips include various EEPROMS (like 24c04 and 24c08 models).

These dummy devices have two main uses. First, most I2C and SMBus calls except **`i2c_transfer()`** need a client handle; the dummy will be that handle. And second, this prevents the specified address from being bound to a different driver.

This returns the new i2c client, which should be saved for later use with **`i2c_unregister_device()`**; or an ERR_PTR to describe the error.

```
struct i2c_client *devm_i2c_new_dummy_device(struct
                  device *dev, struct i2c_adapter *adapter,
                  u16 address)
```

return a new i2c device bound to a dummy driver

**Parameters**

`struct device *dev`
: device the managed resource is bound to

`struct i2c_adapter *adapter`
: the adapter managing the device

`u16 address`
: seven bit address to be used

**Context**

can sleep

**Description**

This is the device-managed version of **i2c_new_dummy_device**. It returns the new i2c client or an ERR_PTR in case of an error.

```
struct i2c_client *i2c_new_ancillary_device(struct
                  i2c_client *client, const char *name, u16
                  default_addr)
```

Helper to get the instantiated secondary address and create the associated device

**Parameters**

`struct i2c_client *client`
: Handle to the primary client

`const char *name`
: Handle to specify which secondary address to get

`u16 default_addr`
: Used as a fallback if no secondary address was specified

**Context**

can sleep

**Description**

I2C clients can be composed of multiple I2C slaves bound together in a single component. The I2C client driver then binds to the master I2C slave and needs to create I2C dummy clients to communicate with all the other slaves.

This function creates and returns an I2C dummy client whose I2C address is retrieved from the platform firmware based on the given slave name. If no address is specified by the firmware default_addr is used.

On DT-based platforms the address is retrieved from the “reg” property entry cell whose “reg-names” value matches the slave name.

This returns the new i2c client, which should be saved for later use with **`i2c_unregister_device()`**; or an ERR_PTR to describe the error.

### `struct i2c_adapter *i2c_verify_adapter(struct device *dev)`

return parameter as i2c_adapter or NULL

**Parameters**

`struct device *dev`
: device, probably from some driver model iterator

**Description**

When traversing the driver model tree, perhaps using driver model iterators like **`device_for_each_child()`**, you can’t assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an i2c_adapter.

### `int i2c_handle_smbus_host_notify(struct i2c_adapter *adap, unsigned short addr)`

Forward a Host Notify event to the correct I2C client.

**Parameters**

`struct i2c_adapter *adap`
: the adapter

`unsigned short addr`
: the I2C address of the notifying device

**Context**

can’t sleep

**Description**

Helper function to be called from an I2C bus driver's interrupt handler. It will schedule the Host Notify IRQ.

### `int i2c_add_adapter(struct i2c_adapter *adapter)`

declare i2c adapter, use dynamic bus number

**Parameters**

`struct i2c_adapter *adapter`
the adapter to add

**Context**

can sleep

**Description**

This routine is used to declare an I2C adapter when its bus number doesn't matter or when its bus number is specified by an dt alias. Examples of bases when the bus number doesn't matter: I2C adapters dynamically added by USB links or PCI plugin cards.

When this returns zero, a new bus number was allocated and stored in adap->nr, and the specified adapter became available for clients. Otherwise, a negative errno value is returned.

### `int i2c_add_numbered_adapter(struct i2c_adapter *adap)`

declare i2c adapter, use static bus number

**Parameters**

`struct i2c_adapter *adap`
the adapter to register (with adap->nr initialized)

**Context**

can sleep

**Description**

This routine is used to declare an I2C adapter when its bus number matters. For example, use it for I2C adapters from system-on-chip CPUs, or otherwise built in to the system's mainboard, and where i2c_board_info is used to properly configure I2C devices.

If the requested bus number is set to -1, then this function will behave identically to i2c_add_adapter, and will dynamically assign a bus number.

If no devices have pre-been declared for this bus, then be sure to register the adapter before any dynamically allocated ones. Otherwise the required bus ID may not be available.

When this returns zero, the specified adapter became available for clients using the bus number provided in adap->nr. Also, the table of I2C devices pre-declared using

**`i2c_register_board_info()`** is scanned, and the appropriate driver model device nodes are created. Otherwise, a negative errno value is returned.

### `void i2c_del_adapter(struct i2c_adapter *adap)`

unregister I2C adapter

**Parameters**

`struct i2c_adapter *adap`
: the adapter being unregistered

**Context**

can sleep

**Description**

This unregisters an I2C adapter which was previously registered by **i2c_add_adapter** or **i2c_add_numbered_adapter**.

### `int devm_i2c_add_adapter(struct device *dev, struct i2c_adapter *adapter)`

device-managed variant of **`i2c_add_adapter()`**

**Parameters**

`struct device *dev`
: managing device for adding this I2C adapter

`struct i2c_adapter *adapter`
: the adapter to add

**Context**

can sleep

**Description**

Add adapter with dynamic bus number, same with **`i2c_add_adapter()`** but the adapter will be auto deleted on driver detach.

### `struct i2c_adapter *i2c_find_adapter_by_fwnode(struct fwnode_handle *fwnode)`

find an i2c_adapter for the fwnode

**Parameters**

`struct fwnode_handle *fwnode`

**`struct fwnode_handle`** corresponding to the **`struct i2c_adapter`**

**Description**

Look up and return the **`struct i2c_adapter`** corresponding to the **fwnode**. If no adapter can be found, or **fwnode** is NULL, this returns NULL.

The user must call put_device(**`adapter->dev`**) once done with the i2c adapter.

### `struct i2c_adapter *i2c_get_adapter_by_fwnode(struct fwnode_handle *fwnode)`

find an i2c_adapter for the fwnode

**Parameters**

`struct fwnode_handle *fwnode`
: **`struct fwnode_handle`** corresponding to the **`struct i2c_adapter`**

**Description**

Look up and return the **`struct i2c_adapter`** corresponding to the **fwnode**, and increment the adapter module's use count. If no adapter can be found, or **fwnode** is NULL, this returns NULL.

The user must call i2c_put_adapter(adapter) once done with the i2c adapter. Note that this is different from i2c_find_adapter_by_node().

### `void i2c_parse_fw_timings(struct device *dev, struct i2c_timings *t, bool use_defaults)`

get I2C related timing parameters from firmware

**Parameters**

`struct device *dev`
: The device to scan for I2C timing properties

`struct i2c_timings *t`
: the i2c_timings struct to be filled with values

`bool use_defaults`
: bool to use sane defaults derived from the I2C specification when properties are not found, otherwise don't update

**Description**

Scan the device for the generic I2C properties describing timing parameters for the signal and fill the given struct with the results. If a property was not found and use_defaults was true, then maximum timings are assumed which are derived from the I2C specification. If use_defaults is not used, the results will be as before, so drivers can apply their own defaults before calling this helper. The latter is mainly intended for avoiding regressions of existing drivers which want to switch to this function. New drivers almost always should use the defaults.

### `void i2c_del_driver(struct i2c_driver *driver)`

unregister I2C driver

**Parameters**

`struct i2c_driver *driver`
: the driver being unregistered

**Context**

can sleep

### `int __i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)`

unlocked flavor of i2c_transfer

**Parameters**

`struct i2c_adapter *adap`
: Handle to I2C bus

`struct i2c_msg *msgs`
: One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

`int num`
: Number of messages to be executed.

**Description**

Returns negative errno, else the number of messages executed.

Adapter lock must be held when calling this function. No debug logging takes place.

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

execute a single or combined I2C message

**Parameters**

`struct i2c_adapter *adap`
: Handle to I2C bus

`struct i2c_msg *msgs`

One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

`int num`
: Number of messages to be executed.

**Description**

Returns negative errno, else the number of messages executed.

Note that there is no requirement that each message be sent to the same slave address, although that is the most common model.

## `int i2c_transfer_buffer_flags(const struct i2c_client *client, char *buf, int count, u16 flags)`

issue a single I2C message transferring data to/from a buffer

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`char *buf`
: Where the data is stored

`int count`
: How many bytes to transfer, must be less than 64k since msg.len is u16

`u16 flags`
: The flags to be used for the message, e.g. I2C_M_RD for reads

**Description**

Returns negative errno, or else the number of bytes transferred.

## `int i2c_get_device_id(const struct i2c_client *client, struct i2c_device_identity *id)`

get manufacturer, part id and die revision of a device

**Parameters**

`const struct i2c_client *client`
: The device to query

`struct i2c_device_identity *id`
: The queried information

**Description**

Returns negative errno on error, zero on success.

```
const struct i2c_device_id
                  *i2c_client_get_device_id(const struct
                  i2c_client *client)
```

get the driver match table entry of a device

**Parameters**

`const struct i2c_client *client`
: the device to query. The device must be bound to a driver

**Description**

Returns a pointer to the matching entry if found, NULL otherwise.

```
u8 *i2c_get_dma_safe_msg_buf(struct i2c_msg *msg,
                  unsigned int threshold)
```

get a DMA safe buffer for the given i2c_msg

**Parameters**

`struct i2c_msg *msg`
: the message to be checked

`unsigned int threshold`
: the minimum number of bytes for which using DMA makes sense. Should at least be 1.

**Return**

NULL if a DMA safe buffer was not obtained. Use msg->buf with PIO.
: Or a valid pointer to be used with DMA. After use, release it by calling **`i2c_put_dma_safe_msg_buf()`**.

**Description**

This function must only be called from process context!

```
void i2c_put_dma_safe_msg_buf(u8 *buf, struct i2c_msg
                  *msg, bool xferred)
```

release DMA safe buffer and sync with i2c_msg

**Parameters**

`u8 *buf`
: the buffer obtained from **`i2c_get_dma_safe_msg_buf()`**. May be NULL.

`struct i2c_msg *msg`
: the message which the buffer corresponds to

`bool xferred`

bool saying if the message was transferred

### u8 **i2c_smbus_pec**(u8 crc, u8 *p, size_t count)

Incremental CRC8 over the given input data array

**Parameters**

`u8 crc`
: previous return crc8 value

`u8 *p`
: pointer to data buffer.

`size_t count`
: number of bytes in data buffer.

**Description**

Incremental CRC8 over count bytes in the array pointed to by p

### s32 **i2c_smbus_read_byte**(const struct i2c_client *client)

SMBus “receive byte” protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

**Description**

This executes the SMBus “receive byte” protocol, returning negative errno else the byte received from the device.

### s32 **i2c_smbus_write_byte**(const struct i2c_client *client, u8 value)

SMBus “send byte” protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 value`
: Byte to be sent

**Description**

This executes the SMBus “send byte” protocol, returning negative errno else zero on success.

### s32 **i2c_smbus_read_byte_data**(const struct i2c_client *client, u8 command)

SMBus "read byte" protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

**Description**

This executes the SMBus "read byte" protocol, returning negative errno else a data byte received from the device.

### s32 **i2c_smbus_write_byte_data**(const struct i2c_client *client, u8 command, u8 value)

SMBus "write byte" protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

`u8 value`
: Byte being written

**Description**

This executes the SMBus "write byte" protocol, returning negative errno else zero on success.

### s32 **i2c_smbus_read_word_data**(const struct i2c_client *client, u8 command)

SMBus "read word" protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

**Description**

This executes the SMBus “read word” protocol, returning negative errno else a 16-bit unsigned “word” received from the device.

### s32 **i2c_smbus_write_word_data**(const struct i2c_client *client, u8 command, u16 value)

SMBus “write word” protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

`u16 value`
: 16-bit “word” being written

**Description**

This executes the SMBus “write word” protocol, returning negative errno else zero on success.

### s32 **i2c_smbus_read_block_data**(const struct i2c_client *client, u8 command, u8 *values)

SMBus “block read” protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

`u8 *values`
: Byte array into which data will be read; big enough to hold the data returned by the slave. SMBus allows at most 32 bytes.

**Description**

This executes the SMBus “block read” protocol, returning negative errno else the number of data bytes in the slave’s response.

Note that using this function requires that the client’s adapter support the I2C_FUNC_SMBUS_READ_BLOCK_DATA functionality. Not all adapter drivers support this; its emulation through I2C messaging relies on a specific mechanism (I2C_M_RECV_LEN) which may not be implemented.

### `s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values)`

SMBus "block write" protocol

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

`u8 length`
: Size of data block; SMBus allows at most 32 bytes

`const u8 *values`
: Byte array which will be written.

**Description**

This executes the SMBus "block write" protocol, returning negative errno else zero on success.

### `s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr, unsigned short flags, char read_write, u8 command, int protocol, union i2c_smbus_data *data)`

execute SMBus protocol operations

**Parameters**

`struct i2c_adapter *adapter`
: Handle to I2C bus

`u16 addr`
: Address of SMBus slave on that bus

`unsigned short flags`
: I2C_CLIENT_* flags (usually zero or I2C_CLIENT_PEC)

`char read_write`
: I2C_SMBUS_READ or I2C_SMBUS_WRITE

`u8 command`
: Byte interpreted by slave, for protocols which use such bytes

`int protocol`
: SMBus protocol operation to execute, such as I2C_SMBUS_PROC_CALL

`union i2c_smbus_data *data`
: Data to be read or written

**Description**

This executes an SMBus protocol operation, and returns a negative errno code else zero on success.

## `s32 i2c_smbus_read_i2c_block_data_or_emulated(const struct i2c_client *client, u8 command, u8 length, u8 *values)`

read block or emulate

**Parameters**

`const struct i2c_client *client`
: Handle to slave device

`u8 command`
: Byte interpreted by slave

`u8 length`
: Size of data block; SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes

`u8 *values`
: Byte array into which data will be read; big enough to hold the data returned by the slave. SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes.

**Description**

This executes the SMBus “block read” protocol if supported by the adapter. If block read is not supported, it emulates it using either word or byte read protocols depending on availability.

The addresses of the I2C slave device that are accessed with this function must be mapped to a linear region, so that a block read will have the same effect as a byte read. Before using this function you must double-check if the I2C slave does support exchanging a block transfer with a byte transfer.

## `struct i2c_client *i2c_new_smbus_alert_device(struct i2c_adapter *adapter, struct i2c_smbus_alert_setup *setup)`

get ara client for SMBus alert support

**Parameters**

`struct i2c_adapter *adapter`
: the target adapter

`struct i2c_smbus_alert_setup *setup`
: setup data for the SMBus alert handler

**Context**

can sleep

**Description**

Setup handling of the SMBus alert protocol on a given I2C bus segment.

Handling can be done either through our IRQ handler, or by the adapter (from its handler, periodic polling, or whatever).

This returns the ara client, which should be saved for later use with i2c_handle_smbus_alert() and ultimately **`i2c_unregister_device()`**; or an ERRPTR to indicate an error.