The Inter-Integrated Circuit, or I$^2$C (pronounced I squared C) bus and its subset, the System Management Bus (SMBus), are synchronous serial interfaces that are ubiquitous on desktops and embedded devices. Let's find out how the kernel supports I$^2$C/SMBus host adapters and client devices by implementing example drivers to access an I$^2$C EEPROM and an I$^2$C RTC. And before wrapping up this chapter, let's also peek at two other serial interfaces supported by the kernel: the Serial Peripheral Interface or SPI (often pronounced spy) bus and the 1-wire bus.

All these serial interfaces (I$^2$C, SMBus, SPI, and 1-wire) share two common characteristics:

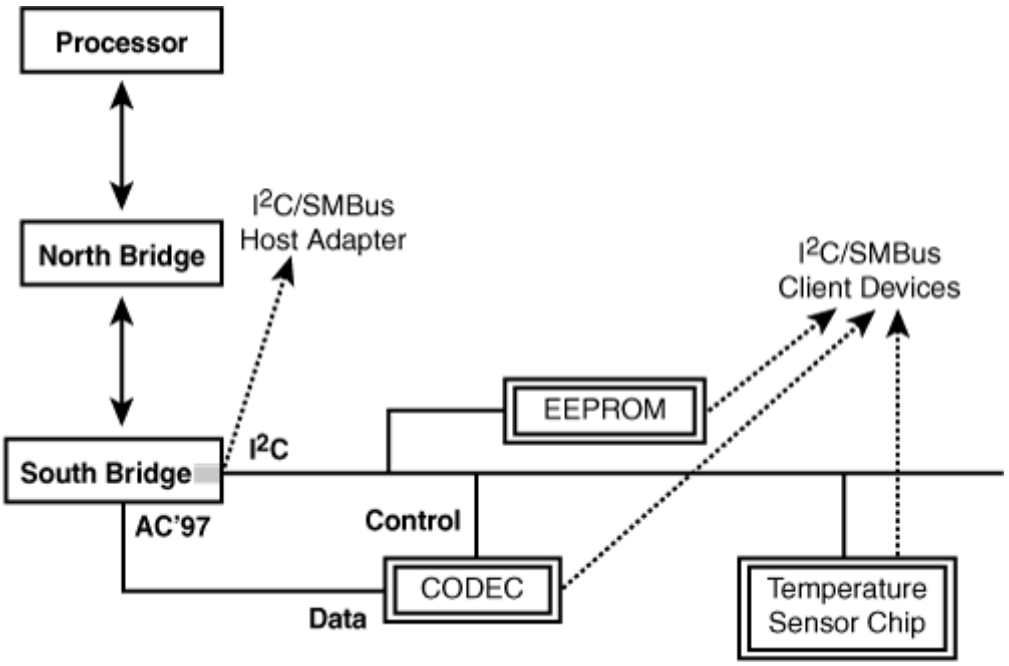
- The amount of data exchanged is small.
- The required data transfer rate is low.

## What's I$^2$C/ SMBus?

I$^2$C is a serial bus that is widely used in desktops and laptops to interface the processor with devices such as EEPROMs, audio codecs, and specialized chips that monitor parameters such as temperature and power-supply voltage. In addition, I$^2$C is widely used in embedded devices to communicate with RTCs, smart battery circuits, multiplexers, port expanders, optical transceivers, and other similar devices. Because I$^2$C is supported by a large number of microcontrollers, there are loads of cheap I$^2$C devices available in the market today.

I$^2$C and SMBus are master-slave protocols where communication takes place between a host adapter (or host controller) and client devices (or slaves). The host adapter is usually part of the South Bridge chipset on desktops and part of the microcontroller on embedded devices. Figure 8.1 shows an example I$^2$C bus on PC-compatible hardware.

**Figure 8.1. I$^2$C/ SMBus on PC-compatible hardware.**




I$^2$C and its subset SMBus are 2-wire interfaces originally developed by Philips and Intel, respectively. The two wires are clock and bidirectional data, and the corresponding lines are called Serial CLock (SCL) and Serial DAta (SDA). Because the I$^2$C bus needs only a pair of wires, it consumes less space on the circuit board. However, the supported bandwidths are also low. I$^2$C allows up to 100Kbps in the standard mode and 400Kbps in a fast mode. (SMBus supports only up to 100Kbps, however.) The bus is thus suitable only for slow peripherals. Even though I$^2$C supports bidirectional exchange, the communication is half duplex because there is only a single data wire.

I$^2$C and SMBus devices own 7-bit addresses. The protocol also supports 10-bit addresses, but many devices respond only to 7-bit addressing, which yields a maximum of 127 devices on the bus. Due to the master-slave nature of the protocol, device addresses are also known as slave addresses.

## I²C Core

The I²C core is a code base consisting of routines and data structures available to host adapter drivers and client drivers. Common code in the core makes the driver developer's job easier. The core also provides a level of indirection that renders client drivers independent of the host adapter, allowing them to work unchanged even if the client device is used on a board that has a different I²C host adapter. This philosophy of a core layer and its attendant benefits is also relevant for many other device driver classes in the kernel, such as PCMCIA, PCI, and USB.

In addition to the core, the kernel I²C infrastructure consists of the following:

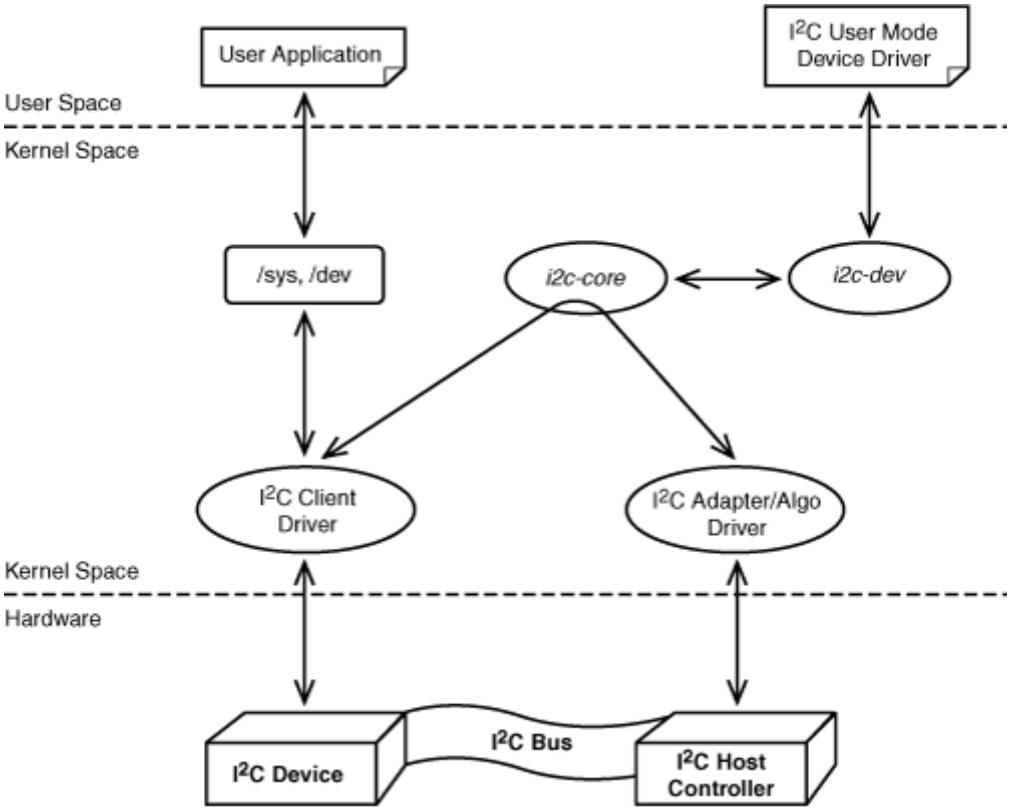
- Device drivers for I²C host adapters. They fall in the realm of bus drivers and usually consist of an adapter driver and an algorithm driver. The former uses the latter to talk to the I²C bus.
- Device drivers for I²C client devices.
- i2c-dev, which allows the implementation of user mode I²C client drivers.

You are more likely to implement client drivers than adapter or algorithm drivers because there are a lot more I²C devices than there are I²C host adapters. So, we will confine ourselves to client drivers in this chapter.

Figure 8.2 illustrates the Linux I²C subsystem. It shows I²C kernel modules talking to a host adapter and a client device on an I²C bus.

**Figure 8.2. The Linux I²C subsystem.**

[View full size image]




Because SMBus is a subset of I²C, using only SMBus commands to talk to your device yields a driver that works with both SMBus and I²C adapters. Table 8.1 lists the SMBus-compatible data transfer routines provided by the I²C core.

**Table 8.1. SMBus-Compatible Data Access Functions Provided by the I²C Core**

| Function | Purpose |
|---|---|
| `i2c_smbus_read_byte()` | Reads a single byte from the device without specifying a location offset. Uses the same offset as the previously issued command. |
| `i2c_smbus_write_byte()` | Sends a single byte to the device at the same memory offset as the previously issued command. |
| `i2c_smbus_write_quick()` | Sends a single bit to the device (in place of the Rd/Wr bit shown in Listing 8.1). |
| `i2c_smbus_read_byte_data()` | Reads a single byte from the device at a specified offset. |
| `i2c_smbus_write_byte_data()` | Sends a single byte to the device at a specified offset. |
| `i2c_smbus_read_word_data()` | Reads 2 bytes from the specified offset. |
| `i2c_smbus_write_word_data()` | Sends 2 bytes to the specified offset. |

| Function | Purpose |
| --- | --- |
| *i2c_smbus_read_block_data()* | Reads a block of data from the specified offset. |
| *i2c_smbus_write_block_data()* | Sends a block of data (<= 32 bytes) to the specified offset. |

## Bus Transactions

Before implementing an example driver, let's get a better understanding of the I $^2$C protocol by peering at the wires through a magnifying glass. Listing 8.1 shows a code snippet that talks to an I $^2$C EEPROM and the corresponding transactions that occur on the bus. The transactions were captured by connecting an I $^2$C bus analyzer while running the code snippet. The code uses user mode I $^2$C functions. (We talk more about user mode I $^2$C programming in Chapter 19, "Drivers in User Space.")

### Listing 8.1. Transactions on the I $^2$C Bus

Code View:

```
/* ... */
/*
 * Connect to the EEPROM. 0x50 is the device address.
 * smbus_fp is a file pointer into the SMBus device.
 */
ioctl(smbus_fp, 0x50, slave);

/* Write a byte (0xAB) at memory offset 0 on the EEPROM */
i2c_smbus_write_byte_data(smbus_fp, 0, 0xAB);

/*
 * This is the corresponding transaction observed
 * on the bus after the write:
 * S 0x50 Wr [A] 0 [A] 0xAB [A] P
 *
 * S is the start bit, 0x50 is the 7-bit slave address (0101000b),
 * Wr is the write command (0b), A is the Accept bit (or
 * acknowledgment) received by the host from the slave, 0 is the
 * address offset on the slave device where the byte is to be
 * written, 0xAB is the data to be written, and P is the stop bit.
 * The data enclosed within [] is sent from the slave to the
 * host, while the rest of the bits are sent by the host to the
 * slave.
 */
/* Read a byte from offset 0 on the EEPROM */
res = i2c_smbus_read_byte_data(smbus_fp, 0);

/*
 * This is the corresponding transaction observed
 * on the bus after the read:
 * S 0x50 Wr [A] 0 [A] S 0x50 Rd [A] [0xAB] NA P
 *
 * The explanation of the bits is the same as before, except that
 * Rd stands for the Read command (1b), 0xAB is the data received
 * from the slave, and NA is the Reverse Accept bit (or the
 * acknowledgment sent by the host to the slave).
 */
```

## Device Example: EEPROM

Our first example client device is an EEPROM sitting on an I²C bus, as shown in Figure 8.1. Almost all laptops and desktops have such an EEPROM for storing BIOS configuration information. The example EEPROM has two memory banks. The driver exports /dev interfaces corresponding to each bank: /dev/eep/0 and /dev/eep/1. Applications operate on these nodes to exchange data with the EEPROM.

Each I²C/SMBus client device is assigned a slave address that functions as the device identifier. The EEPROM in the example answers to two slave addresses, ***SLAVE_ADDR1*** and ***SLAVE_ADDR2***, one per bank.

The example driver uses I²C commands that are compatible with SMBus, so it works with both I²C and SMBus EEPROMs.

### Initializing

As is the case with all driver classes, I²C client drivers also own an ***init()*** entry point. Initialization entails allocating data structures, registering the driver with the I²C core, and hooking up with sysfs and the Linux device model. This is done in Listing 8.2.

**Listing 8.2. Initializing the EEPROM Driver**

Code View:

```
/* Driver entry points */
static struct file_operations eep_fops = {
  .owner   = THIS_MODULE,
  .llseek  = eep_llseek,
  .read    = eep_read,
  .ioctl   = eep_ioctl,
  .open    = eep_open,
  .release = eep_release,
  .write   = eep_write,
};

static dev_t dev_number;          /* Allotted Device Number */
static struct class *eep_class;   /* Device class */

/* Per-device client data structure for each
 * memory bank supported by the driver
 */

struct eep_bank {
  struct i2c_client *client;      /* I²C client for this bank */
  unsigned int addr;              /* Slave address of this bank */
  unsigned short current_pointer; /* File pointer */
  int bank_number;                /* Actual memory bank number */
  /* ... */                       /* Spinlocks, data cache for
                                     slow devices,.. */
};

#define NUM_BANKS 2               /* Two supported banks */
#define BANK_SIZE 2048            /* Size of each bank */

struct ee_bank *ee_bank_list;     /* List of private data
                                     structures, one per bank */
```

```
/*
 * Device Initialization
 */
int __init
eep_init(void)
{

  int err, i;

  /* Allocate the per-device data structure, ee_bank */
  ee_bank_list = kmalloc(sizeof(struct ee_bank)*NUM_BANKS,
                         GFP_KERNEL);
  memset(ee_bank_list, 0, sizeof(struct ee_bank)*NUM_BANKS);
  /* Register and create the /dev interfaces to access the EEPROM
     banks. Refer back to Chapter 5, "Character Drivers" for
     more details */
  if (alloc_chrdev_region(&dev_number, 0,
                          NUM_BANKS, "eep") < 0) {
    printk(KERN_DEBUG "Can't register device\n");
    return -1;
  }

  eep_class = class_create(THIS_MODULE, DEVICE_NAME);
  for (i=0; i < NUM_BANKS;i++) {

    /* Connect the file operations with cdev */
    cdev_init(&ee_bank[i].cdev, &ee_fops);

    /* Connect the major/minor number to the cdev */
    if (cdev_add(&ee_bank[i].cdev, (dev_number + i), 1)) {
      printk("Bad kmalloc\n");
      return 1;
    }
    class_device_create(eep_class, NULL, (dev_number + i),
                          NULL, "eeprom%d", i);
  }

  /* Inform the I²C core about our existence. See the section
     "Probing the Device" for the definition of eep_driver */
  err = i2c_add_driver(&eep_driver);

  if (err) {
    printk("Registering I2C driver failed, errno is %d\n", err);
    return err;
  }

  printk("EEPROM Driver Initialized.\n");
  return 0;
}
```

Listing 8.2 initiates creation of the device nodes, but to complete their production, add the following to an appropriate rule file under /etc/udev/rules.d/:

```
KERNEL:"eeprom[0-1]*", NAME="eep/%n"
```

This creates /dev/eep/0 and /dev/eep/1 in response to reception of the corresponding uevents from the kernel. A user mode program that needs to read from the $n^{th}$ memory bank can then operate on /dev/eep/n.

Listing 8.3 implements the *open()* method for the EEPROM driver. The kernel calls *eep_open()* when an application opens /dev/eep/X. *eep_open()* stores the per-device data structure in a private area so that it's directly accessible from the rest of the driver methods.

**Listing 8.3. Opening the EEPROM Driver**

```
 int
eep_open(struct inode *inode, struct file *file)
{

  /* The EEPROM bank to be opened */
  n = MINOR(file->f_dentry->d_inode->i_rdev);

  file->private_data = (struct ee_bank *)ee_bank_list[n];

  /* Initialize the fields in ee_bank_list[n] such as
     size, slave address, and the current file pointer */
  /* ... */
}
```

## Probing the Device

The $I^2C$ client driver, in partnership with the host controller driver and the $I^2C$ core, attaches itself to a slave device as follows:

1. During initialization, it registers a *probe()* method, which the $I^2C$ core invokes when an associated host controller is detected. In Listing 8.2, *eep_init()* registered *eep_probe()* by invoking *i2c_add_driver()*:

   ```
   static struct i2c_driver eep_driver =
   {
     .driver = {
       .name          =  "EEP",            /* Name */
     },
     .id              = I2C_DRIVERID_EEP,  /* ID */
     .attach_adapter = eep_probe,          /* Probe Method */
     .detach_client   = eep_detach,        /* Detach Method */
   };

   i2c_add_driver(&eep_driver);   `
   ```

   The driver identifier, *I2C_DRIVERID_EEP*, should be unique for the device and should be defined in include/linux/i2c-id.h.

2. When the core calls the driver's *probe()* method signifying the presence of a host adapter, it, in turn, invokes *i2c_probe()* with arguments specifying the addresses of the slave devices that the driver is responsible for and an associated *attach()* routine.

Listing 8.4 implements *eep_probe()*, the *probe()* method of the EEPROM driver. *normal_i2c* specifies the EEPROM bank addresses and is populated as part of the *i2c_client_address_data* structure. Additional fields in this structure can be used to request finer addressing control. You can ask the I $^2$C core to ignore a range of addresses using the *ignore* field. Or you may use the *probe* field to specify (adapter, slave address) pairs if you want to bind a slave address to a particular host adapter. This will be useful, for example, if your processor supports two I $^2$C host adapters, and you have an EEPROM on bus 1 and a temperature sensor on bus 2, both answering to the same slave address.

3. The host controller walks the bus looking for the slave devices specified in Step 2. To do this, it generates a bus transaction such as *S SLAVE_ADDR Wr*, where *S* is the start bit, *SLAVE_ADDR* is the associated 7-bit slave address as specified in the device's datasheet, and *Wr* is the write command, as described in the section "Bus Transactions." If a working slave device exists on the bus, it'll respond by sending an acknowledgment bit (*[A]*).

4. If the host adapter detects a slave in Step 3, the I $^2$C core invokes the *attach()* routine supplied via the third argument to *i2c_probe()* in Step 2. For the EEPROM driver, this routine is *eep_attach()*, which registers a per-device client data structure, as shown in Listing 8.5. If your device expects an initial programming sequence (for example, registers on an I $^2$C Digital Visual Interface transmitter chip have to be initialized before the chip can start functioning), perform those operations in this routine.

### Listing 8.4. Probing the Presence of EEPROM Banks

```
#include <linux/i2c.h>

/* The EEPROM has two memory banks having addresses SLAVE_ADDR1
 * and SLAVE_ADDR2, respectively
 */
static unsigned short normal_i2c[] = {
  SLAVE_ADDR1, SLAVE_ADDR2, I2C_CLIENT_END
};

static struct i2c_client_address_data addr_data = {
  .normal_i2c = normal_i2c,
  .probe      = ignore,
  .ignore     = ignore,
  .forces     = ignore,
};

static int
eep_probe(struct i2c_adapter *adapter)
{
  /* The callback function eep_attach(), is shown
   * in Listing 8.5
   */
   return i2c_probe(adapter, &addr_data, eep_attach);
}
```

### Listing 8.5. Attaching a Client

```
int
eep_attach(struct i2c_adapter *adapter, int address, int kind)
{
  static struct i2c_client *eep_client;

  eep_client = kmalloc(sizeof(*eep_client), GFP_KERNEL);

  eep_client->driver  = &eep_driver; /* Registered in Listing 8.2 */
  eep_client->addr    = address;     /* Detected Address */
  eep_client->adapter = adapter;     /* Host Adapter */
  eep_client->flags   = 0;
  strlcpy(eep_client->name, "eep", I2C_NAME_SIZE);

  /* Populate fields in the associated per-device data structure */
  /* ... */

  /* Attach */
  i2c_attach_client(new_client);
}
```

## Checking Adapter Capabilities

Each host adapter might be limited by a set of constraints. An adapter might not support all the commands that Table 8.1 contains. For example, it might allow the SMBus ***read_word*** command but not the ***read_block*** command. A client driver has to check whether a command is supported by the adapter before using it.

The I$^2$C core provides two functions to do this:

1. ***i2c_check_functionality()*** checks whether a particular function is supported.

2. ***i2c_get_functionality()*** returns a mask containing all supported functions.

See include/linux/i2c.h for the list of possible functionalities.

## Accessing the Device

To read data from the EEPROM, first glean information about its invocation thread from the private data field associated with the device node. Next, use SMBus-compatible data access routines provided by the I $^2$C core (Table 8.1 shows the available functions) to read the data. Finally, send the data to user space and increment the internal file pointer so that the next ***read()***/***write()*** operation starts from where the last one ended. These steps are performed by Listing 8.6. The listing omits sanity and error checks for convenience.

**Listing 8.6. Reading from the EEPROM**

Code View:

```
ssize_t
eep_read(struct file *file, char *buf,
         size_t count, loff_t *ppos)
{
  int i, transferred, ret, my_buf[BANK_SIZE];

  /* Get the private client data structure for this bank */
  struct ee_bank *my_bank =
                  (struct ee_bank *)file->private_data;

  /* Check whether the smbus_read_word() functionality is
     supported */
  if (i2c_check_functionality(my_bank->client,
                              I2C_FUNC_SMBUS_READ_WORD_DATA)) {

    /* Read the data */
    while (transferred < count) {
      ret = i2c_smbus_read_word_data(my_bank->client,
                                     my_bank->current_pointer+i);
      my_buf[i++] = (u8)(ret & 0xFF);
      my_buf[i++] = (u8)(ret >> 8);
      transferred += 2;
    }

    /* Copy data to user space and increment the internal
       file pointer. Sanity checks are omitted for simplicity */
    copy_to_user(buffer, (void *)my_buf, transferred);
    my_bank->current_pointer += transferred;
  }

  return transferred;
}
```

Writing to the device is done similarly, except that an ***i2c_smbus_write_XXX()*** function is used instead.

> Some EEPROM chips have a Radio Frequency Identification (RFID) transmitter to wirelessly transmit stored information. This is used to automate supply-chain processes such as inventory monitoring and asset tracking. Such EEPROMs usually implement safeguards via an access protection bank that controls access permissions to the data banks. In such cases, the driver has to wiggle corresponding bits in the access protection bank before it can operate on the data banks.

To access the EEPROM banks from user space, develop applications that operate on /dev/eep/n. To dump the contents of the EEPROM banks, use od:

```
bash> od -a /dev/eep/0
0000000   S   E   R   #  dc4  ff soh   R   P nul nul nul nul nul nul nul
0000020   @   1   3   R   1   1   5   3   Z   J   1   V   1   L   4   6
0000040   5   1   0   H  sp   1   S   2   8   8   8   7   J   U   9   9
```

```
0000060   H   0   0   6   6 nul nul nul  bs   3   8   L   5   0   0   3
0000100   Z   J   1   N   U   B   4   6   8   6   V   7 nul nul nul nul
0000120 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
*
0000400
```

As an exercise, take a stab at modifying the EEPROM driver to create /sys interfaces to the EEPROM banks rather than the /dev interfaces. You may reuse code from Listing 5.7, "Using Sysfs to Control the Parallel LED Board," in Chapter 5 to help you in this endeavor.

## More Methods

To obtain a fully functional driver, you need to add a few remaining entry points. These are hardly different from those of normal character drivers discussed in Chapter 5, so the code listings are not shown:

- To support the `lseek()` system call that assigns a new value to the internal file pointer, implement the `llseek()` driver method. The internal file pointer stores state information about EEPROM access.
- To verify data integrity, the EEPROM driver can support an `ioctl()` method to adjust and verify checksums of stored data.
- The `poll()` and `fsync()` methods are not relevant for the EEPROM.
- If you choose to compile the driver as a module, you have to supply an `exit()` method to unregister the device and clean up client-specific data structures. Unregistering the driver from the $I^2C$ core is a one-liner:

  ```
  i2c_del_driver(&eep_driver);
  ```