How to instantiate I2C devices

Unlike PCI or USB devices, I2C devices are not enumerated at the hardware level. Instead, the software must know which devices are connected on each I2C bus segment, and what address these devices are using. For this reason, the kernel code must instantiate I2C devices explicitly. There are several ways to achieve this, depending on the context and requirements.

Method 1: Declare the I2C devices statically

This method is appropriate when the I2C bus is a system bus as is the case for many embedded systems. On such systems, each I2C bus has a number which is known in advance. It is thus possible to pre-declare the I2C devices which live on this bus.

This information is provided to the kernel in a different way on different architectures: device tree, ACPI or board files.

When the I2C bus in question is registered, the I2C devices will be instantiated automatically by i2c-core. The devices will be automatically unbound and destroyed when the I2C bus they sit on goes away (if ever).

Declare the I2C devices via devicetree

On platforms using devicetree, the declaration of I2C devices is done in subnodes of the master controller.

Example:

```
i2c1: i2c@400a0000 {
        /* ... master properties skipped ... */
        clock-frequency = <100000>;

        flash@50 {
                compatible = "atmel,24c256";
                reg = <0x50>;
        };

        pca9532: gpio@60 {
                compatible = "nxp,pca9532";
                gpio-controller;
                #gpio-cells = <2>;
                reg = <0x60>;
        };
};
```

Here, two devices are attached to the bus using a speed of 100kHz. For additional properties which might be needed to set up the device, please refer to its devicetree documentation in Documentation/devicetree/bindings/.

Declare the I2C devices via ACPI

ACPI can also describe I2C devices. There is special documentation for this which is currently located at Documentation/firmware-guide/acpi/enumeration.rst.

Declare the I2C devices in board files

In many embedded architectures, devicetree has replaced the old hardware description based on board files, but the latter are still used in old code. Instantiating I2C devices via board files is done with an array of struct i2c_board_info which is registered by calling i2c_register_board_info().

Example (from omap2 h4):

```
static struct i2c_board_info h4_i2c_board_info[] __initdata = {
      {
              I2C_BOARD_INFO("isp1301_omap", 0x2d),
              .irq            = OMAP_GPIO_IRQ(125),
      },
      {       /* EEPROM on mainboard */
              I2C_BOARD_INFO("24c01", 0x52),
              .platform_data  = &m24c01,
      },
      {       /* EEPROM on cpu card */
              I2C_BOARD_INFO("24c01", 0x57),
              .platform_data  = &m24c01,
      },
};

static void __init omap_h4_init(void)
{
      (...)
      i2c_register_board_info(1, h4_i2c_board_info,
                      ARRAY_SIZE(h4_i2c_board_info));
```

```
        (...)
}
```

The above code declares 3 devices on I2C bus 1, including their respective addresses and custom data needed by their drivers.

Method 2: Instantiate the devices explicitly

This method is appropriate when a larger device uses an I2C bus for internal communication. A typical case is TV adapters. These can have a tuner, a video decoder, an audio decoder, etc. usually connected to the main chip by the means of an I2C bus. You won't know the number of the I2C bus in advance, so the method 1 described above can't be used. Instead, you can instantiate your I2C devices explicitly. This is done by filling a struct i2c_board_info and calling i2c_new_client_device().

Example (from the sfe4001 network driver):

```
static struct i2c_board_info sfe4001_hwmon_info = {
        I2C_BOARD_INFO("max6647", 0x4e),
};

int sfe4001_init(struct efx_nic *efx)
{
        (...)
        efx->board_info.hwmon_client =
                i2c_new_client_device(&efx->i2c_adap, &sfe4001_hwmon_info);

        (...)
}
```

The above code instantiates 1 I2C device on the I2C bus which is on the network adapter in question.

A variant of this is when you don't know for sure if an I2C device is present or not (for example for an optional feature which is not present on cheap variants of a board but you have no way to tell them apart), or it may have different addresses from one board to the next (manufacturer changing its design without notice). In this case, you can call i2c_new_scanned_device() instead of i2c_new_client_device().

Example (from the nxp OHCI driver):

```
static const unsigned short normal_i2c[] = { 0x2c, 0x2d, I2C_CLIENT_END };

static int usb_hcd_nxp_probe(struct platform_device *pdev)
{
        (...)
        struct i2c_adapter *i2c_adap;
        struct i2c_board_info i2c_info;

        (...)
        i2c_adap = i2c_get_adapter(2);
        memset(&i2c_info, 0, sizeof(struct i2c_board_info));
        strscpy(i2c_info.type, "isp1301_nxp", sizeof(i2c_info.type));
        isp1301_i2c_client = i2c_new_scanned_device(i2c_adap, &i2c_info,
                                                    normal_i2c, NULL);
        i2c_put_adapter(i2c_adap);
        (...)
}
```

The above code instantiates up to 1 I2C device on the I2C bus which is on the OHCI adapter in question. It first tries at address 0x2c, if nothing is found there it tries address 0x2d, and if still nothing is found, it simply gives up.

The driver which instantiated the I2C device is responsible for destroying it on cleanup. This is done by calling i2c_unregister_device() on the pointer that was earlier returned by i2c_new_client_device() or i2c_new_scanned_device().

Method 3: Probe an I2C bus for certain devices

Sometimes you do not have enough information about an I2C device, not even to call i2c_new_scanned_device(). The typical case is hardware monitoring chips on PC mainboards. There are several dozen models, which can live at 25 different addresses. Given the huge number of mainboards out there, it is next to impossible to build an exhaustive list of the hardware monitoring chips being used. Fortunately, most of these chips have manufacturer and device ID registers, so they can be identified by probing.

In that case, I2C devices are neither declared nor instantiated explicitly. Instead, i2c-core will probe for such devices as soon as their drivers are loaded, and if any is found, an I2C device will be instantiated automatically. In order to prevent any misbehavior of this mechanism, the following restrictions apply:

- The I2C device driver must implement the detect() method, which identifies a supported device by reading from arbitrary registers.
- Only buses which are likely to have a supported device and agree to be probed, will be probed. For example this avoids probing for hardware monitoring chips on a TV adapter.

Example: See lm90_driver and lm90_detect() in drivers/hwmon/lm90.c

I2C devices instantiated as a result of such a successful probe will be destroyed automatically when the driver which detected them is removed, or when the underlying I2C bus is itself destroyed, whichever happens first.

Those of you familiar with the I2C subsystem of 2.4 kernels and early 2.6 kernels will find out that this method 3 is essentially similar to what was done there. Two significant differences are:

- Probing is only one way to instantiate I2C devices now, while it was the only way back then. Where possible, methods 1 and 2 should be preferred. Method 3 should only be used when there is no other way, as it can have undesirable side effects.
- I2C buses must now explicitly say which I2C driver classes can probe them (by the means of the class bitfield), while all I2C buses were probed by default back then. The default is an empty class which means that no probing happens. The purpose of the class bitfield is to limit the aforementioned undesirable side effects.

Once again, method 3 should be avoided wherever possible. Explicit device instantiation (methods 1 and 2) is much preferred for it is safer and faster.

Method 4: Instantiate from user-space

In general, the kernel should know which I2C devices are connected and what addresses they live at. However, in certain cases, it does not, so a sysfs interface was added to let the user provide the information. This interface is made of 2 attribute files which are created in every I2C bus directory: `new_device` and `delete_device`. Both files are write only and you must write the right parameters to them in order to properly instantiate, respectively delete, an I2C device.

File `new_device` takes 2 parameters: the name of the I2C device (a string) and the address of the I2C device (a number, typically expressed in hexadecimal starting with 0x, but can also be expressed in decimal.)

File `delete_device` takes a single parameter: the address of the I2C device. As no two devices can live at the same address on a given I2C segment, the address is sufficient to uniquely identify the device to be deleted.

Example:

```
# echo eeprom 0x50 > /sys/bus/i2c/devices/i2c-3/new_device
```

While this interface should only be used when in-kernel device declaration can't be done, there is a variety of cases where it can be helpful:

- The I2C driver usually detects devices (method 3 above) but the bus segment your device lives on doesn't have the proper class bit set and thus detection doesn't trigger.
- The I2C driver usually detects devices, but your device lives at an unexpected address.
- The I2C driver usually detects devices, but your device is not detected, either because the detection routine is too strict, or because your device is not officially supported yet but you know it is compatible.
- You are developing a driver on a test board, where you soldered the I2C device yourself.

This interface is a replacement for the force_* module parameters some I2C drivers implement. Being implemented in i2c-core rather than in each device driver individually, it is much more efficient, and also has the advantage that you do not have to reload the driver to change a setting. You can also instantiate the device before the driver is loaded or even available, and you don't need to know what driver the device needs.