ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Category:I2C

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Contents

1. Category:I2C 3
2. I2C device tree configuration 4
3. I2C i2c-tools 13
4. I2C overview 23

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the Linux® **I2C** software framework.

It is recommended to first read the I2C overview article.

Linux® is a registered trademark of Linus Torvalds.

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Pages in category "I2C"

The following 3 pages are in this category, out of 3 total.

- I2C i2c-tools
- I2C device tree configuration
- I2C overview

Stable: 03.06.2021 - 12:51 / Revision: 03.06.2021 - 08:49

A quality version of this page, approved on *3 June 2021*, was based off this revision.

Contents

1 Article purpose 5
2 DT bindings documentation 6
3 DT configuration 7
3.1 DT configuration (STM32 level) 7
3.2 DT configuration (board level) 7
3.2.1 I²C internal peripheral related properties 8
3.2.2 I²C devices related properties 9
3.2.3 How to measure I2C timings 9
3.3 DT configuration examples 9
3.3.1 Example of an external EEPROM slave device 9
3.3.2 Example of an EEPROM slave device emulator registering on STM32 side 10
3.3.3 Example of a stts751 thermal sensor with SMBus Alert feature enabled 10
4 How to configure the DT using STM32CubeMX 12
5 References 13

1 Article purpose

This article explains how to configure the *I2C internal peripheral*[1] **when the peripheral is assigned to Linux®OS**, and in particular:

- how to configure the STM32 I2C peripheral
- how to configure the STM32 external I2C devices present either on the board or on a hardware extension.

The configuration is performed using the **device tree mechanism**[2].

It is used by the *STM32 I2C Linux® driver* that registers relevant information in the I2C framework.

If the peripheral is assigned to another execution context, refer to How to assign an internal peripheral to a runtime context article for guidelines on peripheral assignment and configuration.

2 DT bindings documentation

The I2C is represented by:

- The *Generic device tree bindings for I2C busses*[3]
- The *STM32 I2C controller device tree bindings*[4]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 DT configuration

This hardware description is a combination of the **STM32 microprocessor** device tree files (*.dtsi* extension) and **board** device tree files (*.dts* extension). See the Device tree for an explanation of the device tree file split.

STM32CubeMX can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.

3.1 DT configuration (STM32 level)

At device level, the I2C controller is declared as follows:

```
i2c2: i2c@40013000 {
        compatible = "st,stm32mp15-i2c";
        reg = <0x5c002000 0x400>;
        interrupt-names = "event", "error";
        interrupts-extended = <&exti 22 IRQ_TYPE_LEVEL_HIGH>,
                                          <&intc GIC_SPI 34 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&rcc I2C2_K>;
        resets = <&rcc I2C2_R>;
        #address-cells = <1>;
        #size-cells = <0>;
        dmas = <&dmamux1 35 0x400 0x80000001>,
                     <&dmamux1 36 0x400 0x80000001>;
        dma-names = "rx", "tx";
        power-domains = <&pd_core>;
        st,syscfg-fmp = <&syscfg 0x4 0x2>;
        wakeup-source;
        status = "disabled";
};
```

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

Refer to the DTS file: stm32mp151.dtsi[5]

3.2 DT configuration (board level)

```
&i2c2 {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&i2c2_pins_a>;
        pinctrl-1 = <&i2c2_pins_sleep_a>;
        i2c-scl-rising-time-ns = <185>;
        i2c-scl-falling-time-ns = <20>;
        st,smbus-alert;
        st,smbus-host-notify;
        status = "okay";
        /delete-property/dmas;
```

```
        /delete-property/dma-names;

        ov5640: camera@3c {
         [...]
        };
};
```

There are two levels of device tree configuration:

3.2.1 I²C internal peripheral related properties

The device tree properties related to the I²C internal peripheral and to the I²C bus which belong to i2cx node

- **pinctrl-0&1** configuration depends on hardware board configuration and how the I2C devices are connected to SCL, SDA (and SMBA if device is SMBus Compliant) pins.
More details about pin configuration are available here: Pinctrl device tree configuration
- **clock-frequency** represents the I2C bus speed : **normal (100KHz)**, **Fast (400KHz)** and **Fast+(up to 1MHz)**. This value is given in Hz.
- **dmas** By default, DMAs are enabled for all I2C instances. This is up to the user to **remove** them if not needed. ***/delete-property/*** is used to remove DMA usage for I2C. Both **/delete-property/dma-names** and **/delete-property/dmas** have to be inserted to get rid of DMAs.
- **i2c-scl-rising/falling-time-ns** are optional values depending on the board hardware characteristics: wires length, resistor and capacitor of the hardware design.
These values must be provided in nanoseconds and can be measured by observing the SCL rising and falling slope on an oscilloscope. See how to measure I2C timings.
The I2C driver uses this information to compute accurate I2C timings according to the requested ***clock-frequency***.
The STM32CubeMX implements an algorithm that follows the I2C standard and takes into account the user inputs.
When those values are not provided, the driver uses its default values.
Providing wrong parameters will produce inaccurate ***clock-frequency***. In case the driver fails to compute timing parameters in line with the user input (SCL raising/falling and clock frequency), the clock frequency will be downgraded to a lower frequency.
Example: if user specifies 400 kHz as clock frequency but the algorithm fails to generate timings for the specified SCL rising and falling time, the clock frequency will be dropped to 100 kHz.

Information

I2C timings are highly recommended for I2C bus frequency higher than 100KHz.

- **st,smbus-alert** optional property allow to enable the driver handling of the SMBus Alert mechanism. When enabled, the slave driver's alert function will be called whenever the slave device generates an SMBus Alert message.
- **st,smbus-host-notify** optional property allow to enable the driver handling of the SMBus Host Notify mechanism. When enabled, an IRQ handler will get called whenever a slave device sends a Host Notify message.

Information

See Linux smbus-protocol documentation [6] for more details about SMBus Alert & Host Notify handling.

3.2.2 I²C devices related properties

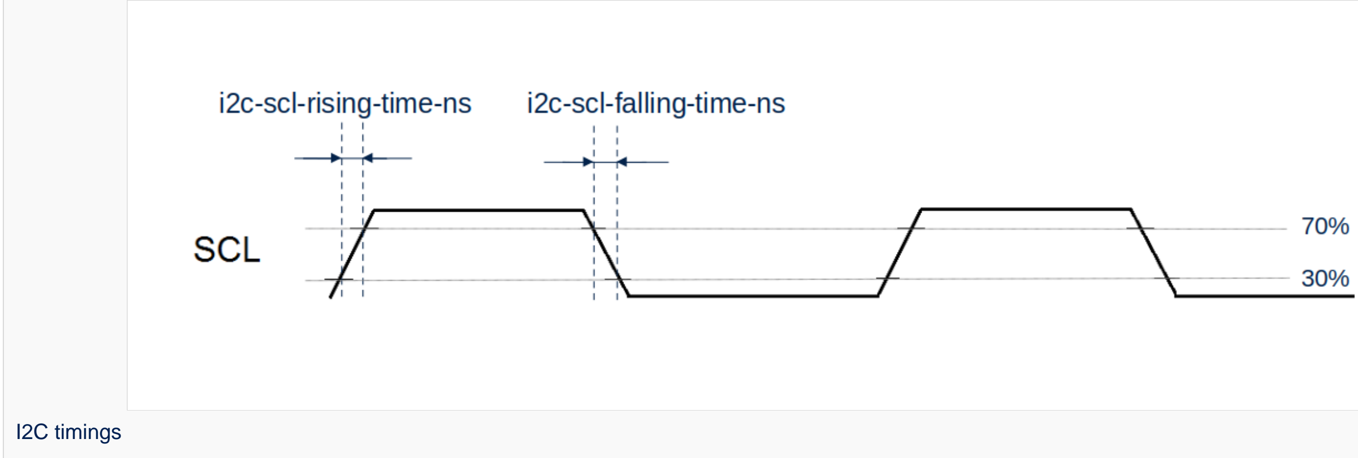
The device tree properties related to I²C devices connected to the specified I²C bus. Each I²C device is represented by a sub-node.

- **reg** represents the I2C peripheral slave address on the bus.

Be aware that some slave address bits can have a special meaning for the framework. For instance, the 31st bit indicates 10-bit device capability.
Refer to i2c.txt[3] for further details

3.2.3 How to measure I2C timings

i2c-scl-rising-time-ns is measured on the SCL rising edge and **i2c-scl-falling-time-ns** on the SCL falling edge. On the oscilloscope, measure the time between the 30% to 70% range of amplitude for rising time and falling time in nanoseconds.



I2C timings

3.3 DT configuration examples

3.3.1 Example of an external EEPROM slave device

```
i2c4: {
    status = "okay";
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;

    eeprom@50 {
        compatible = "at,24c256";
        pagesize = <64>;
        reg = <0x50>;
    };
};
```

The above example registers an EEPROM device on i2c-X bus (X depends on how many adapters are probed at runtime) at address 0x50 and this instance is compatible with the driver registered with the same compatible property.
Please note that the driver is going to use MDMA for data transfer and that SCL rising/falling times have been provided as inputs.

3.3.2 Example of an EEPROM slave device emulator registering on STM32 side

```
i2c4: {
    eeprom@64 {
        status = "okay";
        compatible = "linux,slave-24c02";
        reg = <0x40000064>;
    };
};
```

The above example registers an EEPROM emulator on STM32 side at slave address 0x64.
STM32 acts as an I2C EEPROM that can be accessed from an external master device connected on I2C bus.

3.3.3 Example of a stts751 thermal sensor with SMBus Alert feature enabled

The stts751 thermal sensor [7] is able to send an SMBus Alert when configured threshold are reached.
The device driver can be enabled in the kernel:

```
[x] Device Drivers
    [x] Hardware Monitoring support
        [x] ST Microelectronics STTS751
```

This can be done manually in your kernel:

CONFIG_SENSORS_STTS751=y

Since the SMBus Alert is relying on a dedicated pin to work, the pinctrl of the I2C controller (here i2c2) must be updated to add the corresponding SMBA pin.

For the i2c2 controller:

```
i2c2_pins_a: i2c2-0 {
    pins {
        pinmux = <STM32_PINMUX('H', 4, AF4)>, /* I2C2_SCL */
                 <STM32_PINMUX('H', 5, AF4)>, /* I2C2_SDA */
                 <STM32_PINMUX('H', 6, AF4)>; /* I2C2_SMBA */
        bias-disable;
        drive-open-drain;
        slew-rate = <0>;
    };
};

i2c2_pins_sleep_a: i2c2-1 {
    pins {
        pinmux = <STM32_PINMUX('H', 4, ANALOG)>, /* I2C2_SCL */
                 <STM32_PINMUX('H', 5, ANALOG)>, /* I2C2_SDA */
                 <STM32_PINMUX('H', 6, ANALOG)>; /* I2C2_SMBA */
    };
};
```

Within the device-tree, the st,smbus-alert property must be added, as well as the node to enable the stts751.

```
i2c2: {
    st,smbus-alert;
    stts751@3b {
        status = "okay";
        compatible = "stts751";
        reg = <0x3b>;
    };
};
```

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 How to configure the DT using STM32CubeMX

The STM32CubeMX tool can be used to configure the STM32MPU device and get the corresponding platform configuration device tree files.
The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties and they are preserved from one generation to another. Refer to STM32CubeMX user manual for further information.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

5 References

Please refer to the following links for additional information:

- I2C internal peripheral
- Device tree
- 3.03.1 Documentation/devicetree/bindings/i2c/i2c.txt , Generic device tree bindings for I2C busses
- Documentation/devicetree/bindings/i2c/i2c-stm32.txt
- arch/arm/boot/dts/stm32mp151.dtsi
- Documentation/i2c/smbus-protocol.rst
- https://www.st.com/en/mems-and-sensors/stts751.html

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

Linux® is a registered trademark of Linus Torvalds.

Operating System

Device Tree

Generic Interrupt Controller

Serial Peripheral Interface

Device Tree Source (in software context) or Digital Temperature Sensor (in peripheral context)

Serial clock line

Serial DAta line

System Management Bus

Direct Memory Access

Electrically-erasable programmable read-only memory

Stable: 10.04.2020 - 15:47 / Revision: 10.04.2020 - 15:43

A quality version of this page, approved on *10 April 2020*, was based off this revision.

Contents

1 Article purpose 14
2 Introduction 15
3 Tools list 16
4 Installation on your target 17
5 Getting started 18
5.1 Devices detection 18
5.2 Read register 18
5.3 Write register 19
5.4 Auto-increment devices 19
5.5 I2C transfer 20
5.6 16 bits devices handling 21
6 References 23

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

This article aims to give some first information useful to start with the Linux® tool : I2C tools.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 Introduction

i2c-tools is a complete user-space package that comes on top of I2C subsystem. It offers:

- tools: a set of I2C programs that make it easy to debug I2C peripherals without having to write any code
- libi2c: library to develop applications.

3 Tools list

- i2cdetect[1]
- i2cdump[2]
- i2cget[3]
- i2cset[4]
- i2ctransfer[5]

4 Installation on your target

i2c-tools is embedded by default in OpenSTLinux distribution.

I2C tools are already bundled within OpenEmbedded. No installation is thus required.

Warning

With OpenEmbedded Rocko (2.4.1)**, I2C tools revision is v3.1.2 and doesn't embed** i2ctransfer.

i2ctransfer only comes starting v4.0 included into **OpenEmbedded Thud (2.6.x)**

5 Getting started

5.1 Devices detection

It can be very helpful to see which peripherals are connected to a specific I2C bus.
Check all instantiated I2C adapters:

```
Board $> i2cdetect -l
```

If I2C adapters are instantiated, the following return will be print :

```
i2c-0   i2c             ST I2C(0xAAAAAAA)                       I2C adapter
i2c-1   i2c             ST I2C(0xBBBBBBB)                       I2C adapter
i2c-2   i2c             ST I2C(0xCCCCCCC)                       I2C adapter
i2c-3   i2c             ST I2C(0xDDDDDDD)                       I2C adapter
i2c-4   i2c             ST I2C(0xEEEEEEE)                       I2C adapter
i2c-5   i2c             ST I2C(0xFFFFFFF)                       I2C adapter
```

Get the list of detected peripherals on the specific I2C bus:

```
Board $> i2cdetect -y <i2cbus number>
```

```
Board $> i2cdetect -y 3
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- UU -- -- -- -- --
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

UU -> Probing was skipped, because this address is currently in use by a driver. This strongly suggests that there is a device at this address probed with a driver.

More information about **i2cdetect**[1]

5.2 Read register

Read all the registers from a peripheral:

```
Board $> i2cdump -f -y <i2cbus number> <peripheral address>
```

```
Board $> i2cdump -f -y 0 0x5f
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bc    ...............?
```

```
10: 3f 00 87 33 96 be ec a1 9e b2 fe 00 e8 01 80 82    ?.?3???????.????
20: 00 00 00 00 00 00 00 00 51 f2 ae 00 10 f3 c6 00    ........Q??.???.
30: 41 92 a0 0e 00 c4 ee ff 32 03 bf d3 ff ff d0 02    A???.??.2???..??
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bc    ...............?
90: 3f 00 87 33 96 be ec a1 9e b2 fe 00 e8 01 80 82    ?.?3???????.????
a0: 00 00 00 00 00 00 00 00 51 f2 ae 00 10 f3 c6 00    ........Q??.???.
b0: 41 92 a0 0e 00 c4 ee ff 32 03 bf d3 ff ff d0 02    A???.??.2???..??
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

More information about **i2cdump**[2].

To read directly one register, use **i2cget**[3]:

```
Board $> i2cget -f -y <i2cbus number> <peripheral address>
```

Example: read register 0x0f of the peripheral at address 0x5f on bus 0:

```
Board $> i2cget -f -y 0 0x5f 0x0f
0xbc
```

5.3 Write register

To write directly a register, use **i2cset**[4]:

```
Board $> i2cset -f -y <i2cbus number> <peripheral address> <value>
```

Example: write 0xac in register 0x0f of the peripheral at address 0x5f on the bus 0:

```
Board $> i2cset -f -y 0 0x5f 0x0f 0xac
```

Information

A write can fail if the register is in read only mode.

5.4 Auto-increment devices

Even if not part of the I2C standard, it is common to find an automatic incrementation feature on I2C devices, in particular those dealing with large set of registers (typically I2C RAM or EEPROM).

Such devices automatically increment an internal address pointer at each read or write operation, so when several read commands are issued **at the same address**, the value returned at each read may be different each time.

Here are some examples with WM8994 audio codec device on STM32MP157x-EV1 Evaluation board:

First start an audio playback to power-up audio codec device (refer to How to play audio)

Read WM8994 software reset register at address "0x0000":

```
Board $> i2cget -y 0 0x1b 0x00 w
0x9489
```

0x9489 read as a word, understand 0x89 0x94 which is the device id (WM8994) and is indeed the content of the software reset register.

If the same command is repeated again:

```
Board $> i2cget -y 0 0x1b 0x00 w
0x0000
Board $> i2cget -y 0 0x1b 0x00 w
0x0060
```

different values are returned, that do not correspond to the software reset register anymore "0x0000", but correspond successively to registers "0x0001" and "0x0002".

To reset the internal address counter, just write a value at the targeted register address:

```
Board $> i2cset -y 0 0x1b 0x00 0x00
```

Then subsequent read will restart at this address:

```
Board $> i2cget -y 0 0x1b 0x00 w
0x9489
```

Please note that the auto-increment mode may usually be disabled by writing into a device specific configuration register (refer to the device datasheet for details).

5.5 I2C transfer

With OpenEmbedded Rocko (2.4.1)**, I2C tools revision is v3.1.2 and doesn't embed** i2ctransfer.

i2ctransfer becomes available starting with I2C tools revision v4.0 included into **OpenEmbedded Thud (2.6.x)**

This is a user-space program used to send concatenated I2C messages.

Most devices require a write access to a register before being able to read. **i2ctransfer**[5] offers a way to combine write and read procedures. It also handles multiple bytes write/read in a single command with an additional suffix.

To read a set of bytes:

```
Board $> i2ctransfer -f -y <i2cbus number> r<number of bytes>@<peripheral address>
```

To write a set of bytes:

```
Board $> i2ctransfer -f -y <i2cbus number> w<number of bytes>@<peripheral address> <byte
value 1> <byte value 2> ... <byte value n>
```

To write a set of bytes then read a set of bytes:

```
Board $> i2ctransfer -f -y <i2cbus number> w<number of bytes to write>@<peripheral
address> <byte value 1> <byte value 2> ... <byte value n> r<number of bytes to read>
```

Example (bus 0, read 8 bytes at offset 0x64 from EEPROM at 0x50)

```
Board $> i2ctransfer 0 w1@0x50 0x64 r8
```

"w1" for "write 1 byte" (the 0x64 offset), "r8" for "read 8 bytes"

Example (same EEPROM, at offset 0x42 write 0xff 0xfe ... 0xf0)

```
Board $> i2ctransfer 0 w17@0x50 0x42 0xff-
```

"w17" for "write 17 bytes", first 0x42 byte for the offset, and 0xff- for the 16 subsequent bytes ("-" for auto value decrease starting from 0xff).

See following chapter for 16 bits addressing devices handling.

5.6 16 bits devices handling

The I2C standard protocol supports natively 7 bits of address (or 10 bits of address in extended mode) followed by 8 bits of data.

However some I2C devices embed 16-bit data registers with internal 16-bit address space. Here is how the i2c-tool allows to drive such devices.

- To read a 16 bits value, add "**w**" for "word" at the end of command:

```
Board $> i2cget -f -y <i2cbus number> <peripheral address> <address> w
```

Please note that <address> is 8-bit wide, while the returned data is 16-bit wide. The **interpretation of <address> is device dependent** (One possible interpretation is that <address> drives the 8 MSB bits of the 16-bit address while the 8 LSB bits are set to 0).

- To write a 16 bits value specifying the 16 bits address, send both the address and the data as a set of bytes in a single "I2C block write" by adding "**i**" at the end of **i2cset**[4] command:

```
Board $> i2cset -f -y <i2cbus number> <peripheral address> <MSB address> <LSB address>
<MSB value> <LSB value> i
```

Here are some examples with WM8994 audio codec device on STM32MP157x-EV1 Evaluation board:

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Warning

First start an audio playback to power-up audio codec device (refer to How to play audio)

- Read the device id from register "Software Reset" at address 0x0000:

```
Board $> i2cget -y 0 0x1b 0x0 w
0x9489
```

"w" stands for "word" access. Since the word is read in **little endian** and the device is big endian, we have to reverse the endianness.

The returned word 0x9489 should be interpreted as 0x89 0x94 which is the indeed the (WM8994) device ID.

- Update value of register "AIF1 Control" at address 0x0300:

```
Board $> i2cget -y 0 0x1b 0x3 w
0x5040
```

Current value is 0x4050.

Let's assume the AIF1ADC_TDM pin needs to be put in tristate, this is done by settings bit 13, hence by writing 0x6050:

```
Board $> i2cset -y 0 0x1b 0x03 0x00 0x60 0x50 i
Board $> i2cget -y 0 0x1b 0x3 w
0x5060
```

The "AIF1 Control" register value has been updated to 0x6050 as expected.

Doing the same with I2C transfer is far more simple:

```
Board $> i2ctransfer -f -y 0 w4@0x1b 0x03 0x00 0x60 0x50 -r2
Board $> 0x60 0x50
```

"w4" for "write 4 bytes": the first 2 bytes for address (0x0300), the next 2 bytes for register address (0x6050)

"r2" for "read 2 bytes": the word register value

6 References

All these tools (i2cset, i2cget, i2cdump, i2cdectect and i2ctransfer) are available in this GIT:

```
git clone git://git.kernel.org/pub/scm/utils/i2c-tools/i2c-tools.git or
git clone https://git.kernel.org/pub/scm/utils/i2c-tools/i2c-tools.git
```

Source code browsing I2C Tools Code

- 1.01.1 https://www.mankier.com/8/i2cdetect
- 2.02.1 https://www.mankier.com/8/i2cdump
- 3.03.1 https://www.mankier.com/8/i2cget
- 4.04.14.2 https://www.mankier.com/8/i2cset
- 5.05.1 https://www.mankier.com/8/i2ctransfer

Linux® is a registered trademark of Linus Torvalds.

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Electrically-erasable programmable read-only memory

Stable: 22.02.2021 - 10:09 / Revision: 22.02.2021 - 08:23

A quality version of this page, approved on *22 February 2021*, was based off this revision.

This article provides basic information on the Linux®I2C system and how I2C STM32 drivers is plugged upon.

Contents

1 Framework purpose 25
2 System overview 26
2.1 Component description 26
2.1.1 Board external I²C devices 26
2.1.2 STM32 I2C internal peripheral controller 27
2.1.3 i2c-stm32 27
2.1.4 i2c-core 27
2.1.5 Board peripheral drivers 27
2.1.6 i2c-dev 27
2.1.7 i2c-tools 27
2.1.8 Application 27
2.2 API description 28
2.2.1 libi2c 28
2.2.2 User space application 28
2.2.3 Kernel space peripheral driver 28

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

3 Configuration 29
3.1 Kernel configuration 29
3.2 Device tree configuration 29
4 How to use the framework 30
4.1 i2c-tools package 30
4.2 User space application 30
4.3 Kernel space driver 31
4.4 Board description 32
4.4.1 Device tree 32
4.4.2 sysfs 33
4.4.3 Application code 34
5 How to trace and debug the framework 35
5.1 How to trace 35
5.1.1 Dynamic trace 35
5.1.2 Bus snooping 35
5.2 How to debug 36
5.2.1 Detect I2C configuration 36
5.2.1.1 sysfs 36
5.2.2 devfs 37
5.2.3 i2c-tools 37
6 Source code location 38
7 To go further 39
8 References 40

1 Framework purpose

This article aims to explain how to use I2C and more accurately:

- how to activate I2C interface on a Linux® BSP
- how to access I2C from kernel space
- how to access I2C from user space.

This article describes Linux® I^{2}C[1] interface in **master** and **slave** modes.

An introduction to I^{2}C[2] is proposed through this external resource.

For a **slave** interface description, see **slave-interface**[3].

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 System overview

I²C is an acronym for the “Inter-IC” bus, a simple bus protocol which is widely used where low data rate communications suffice. I2C is the acronym for the microprocessor I²C peripheral interface.

Around the microprocessor device, the user can add many I²C external devices to create a custom board. Each external device can be accessed through the I2C from the user space or the kernel space.

Implementation architecture

2.1 Component description

2.1.1 Board external I²C devices

- Slave devices X are physical devices (connected to STM32 via I²C bus) that behave as Slave with respect to STM32.

STM32 remains the master on the I²C bus.

- Master devices X are physical devices (connected to STM32 via I^2C bus) that behave as Master with respect to STM32. STM32 behaves as a Slave in this case on the I^2C bus.

2.1.2 STM32 I2C internal peripheral controller

It corresponds to STM32 I2C adapter that handles communications with any external device connected on the same bus.
It manages Slave devices (if any) and behaves as Slave if an external Master is connected.

STM32 microprocessor devices usually embed several instances of the I2C internal peripheral allowing to manage multiple I2C buses. A driver is provided that pilots the hardware.

2.1.3 i2c-stm32

The internal STM32 I2C controller driver offers ST I2C internal peripheral controller abstraction layer to i2c-core-base.
It defines all I2C transfer method algorithms to be used by I2C Core base, this includes I2C and SMBus[4] transfers API, Register/Unregister slave API and functionality check.
Even if I2C Core can emulate SMBus protocol throughout standard I2C messages, all SMBus functions are implemented within the driver.

2.1.4 i2c-core

This is the brain of the communication: it instantiates and manages all buses and peripherals.

- **i2c-core** as stated by its name, this is the I2C core engine but it is also in charge of parsing device tree entries for both adapter and/or devices
- **i2c-core-smbus** deals with all SMBus related API.
- **i2c-core-slave** manages I2C devices acting as slaves running in STM32.
- **i2c-smbus** handles specific protocol SMBus Alert. (SMBus host notification handled by I2C core base)

2.1.5 Board peripheral drivers

This layer represents all drivers associated to physical peripherals.
A peripheral driver can be compiled as a kernel module or directly into the kernel (aka built-in).

2.1.6 i2c-dev

i2c-dev is the interface between the user and the peripheral. It is a kernel driver offering I2C bus access to user space application using this dev-interface API. See examples API Description.

2.1.7 i2c-tools

I2C Tools package provides:

- shell commands to access I2C with SMBus protocol via i2c-dev
- library to use SMBus functions into a user space application

All those functions are described in this smbus-protocol API.
Note : some peripherals can work without SMBus protocol.

2.1.8 Application

An application can control all peripherals using different methods offered by I2C Tools, libI2C (I2C Tools), i2c-dev.

2.2 API description

2.2.1 libi2c

I2C tools[5] package offers a set of shell commands using mostly SMBus protocols to access I2C and an API to develop an application (libi2c).
All tools and libi2c rely on SMBus API but i2ctransfer does not since it relies on standard I2C protocol.
Tools and libi2c access SMBus and I2C API through out **devfs** read/write/ioctl call.

The SMBus protocols constitute a subset of the data transfer formats defined in the I^2C specification.
I2C peripherals that do not comply to these protocols cannot be accessed by standard methods as defined in the SMBus specification.
See external references for further details on **I^2C**[6] and **SMBus protocol**[7].

libi2c API mimics *SMBus protocol*[7] but at user space level.
Same API can be found in this library as in *SMBus protocol*[7]. All SMBus API are duplicated here with the exception of specific SMBus protocol API like SMBus Host Notify and SMBus Alert.

2.2.2 User space application

User space application is using a kernel driver (i2c-dev) which offers I2C access through devfs.
Supported system calls : open(), close(), read(), write(), ioctl(), llseek(), release().

Constant	Description
I2C_SLAVE/I2C_SLAVE_FORCE	Sets slave address for read/write operations
I2C_FUNCS	Gets bus functionalities
I2C_TENBIT	10bits address support
I2C_RDWR	Combined R/W transfer (one STOP only)
I2C_SMBUS	Perform an SMBus transfer instead of standard I^2C

Supported ioctls commands

The above commands are the main ones (more are defined in the framework): see **dev-interface API**[8] for complete list.

2.2.3 Kernel space peripheral driver

Kernel space peripheral driver accesses both I^2C and SMBus devices and uses following **I2C core API**[9]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 Configuration

3.1 Kernel configuration

Activate I2C in kernel configuration with Linux Menuconfig tool: Menuconfig or how to configure kernel.

```
[x] Device Drivers
    [x] I2C support
        [x] I2C device interface
        [ ] I2C Hardware Bus support
            [x] STMicroelectronics STM32F7 I2C support
```

This can be done manually in your kernel:

```
CONFIG_I2C=y
CONFIG_I2C_CHARDEV=y
CONFIG_I2C_STM32F7=y
```

If software needs SMBus specific protocols like SMBus Alert protocol and the SMBus Host Notify protocol, then add:

```
[x] Device Drivers
    [x] I2C support
        [x] I2C device interface
        [ ] Autoselect pertinent helper modules
        [x] SMBus-specific protocols
        [ ] I2C Hardware Bus support
            [x] STMicroelectronics STM32F7 I2C support
```

This can be done manually in your kernel:

```
CONFIG_I2C_SMBUS=y
```

3.2 Device tree configuration

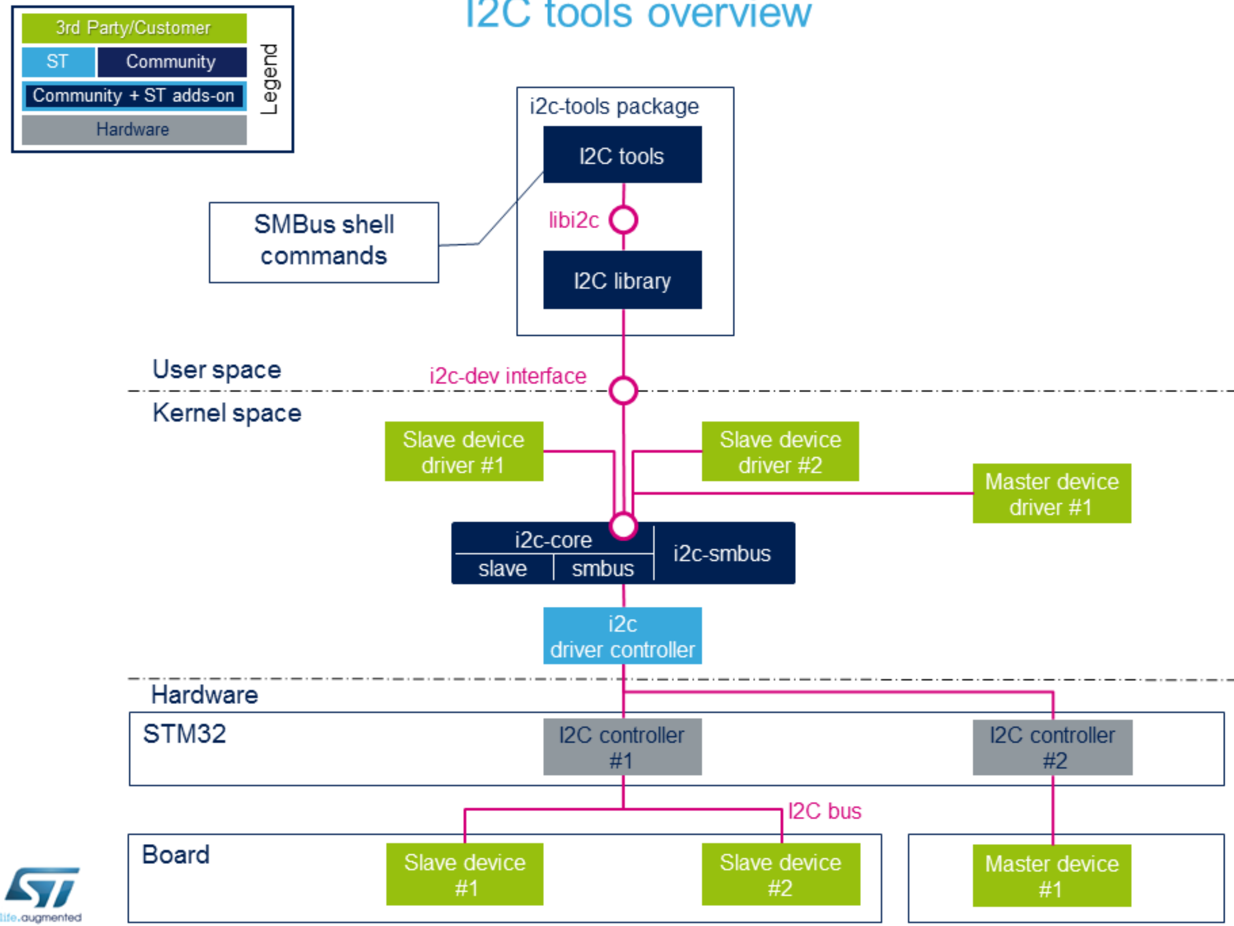
Please refer to I2C device tree configuration.

4 How to use the framework

This section describes how to use the framework to access I2C peripherals.

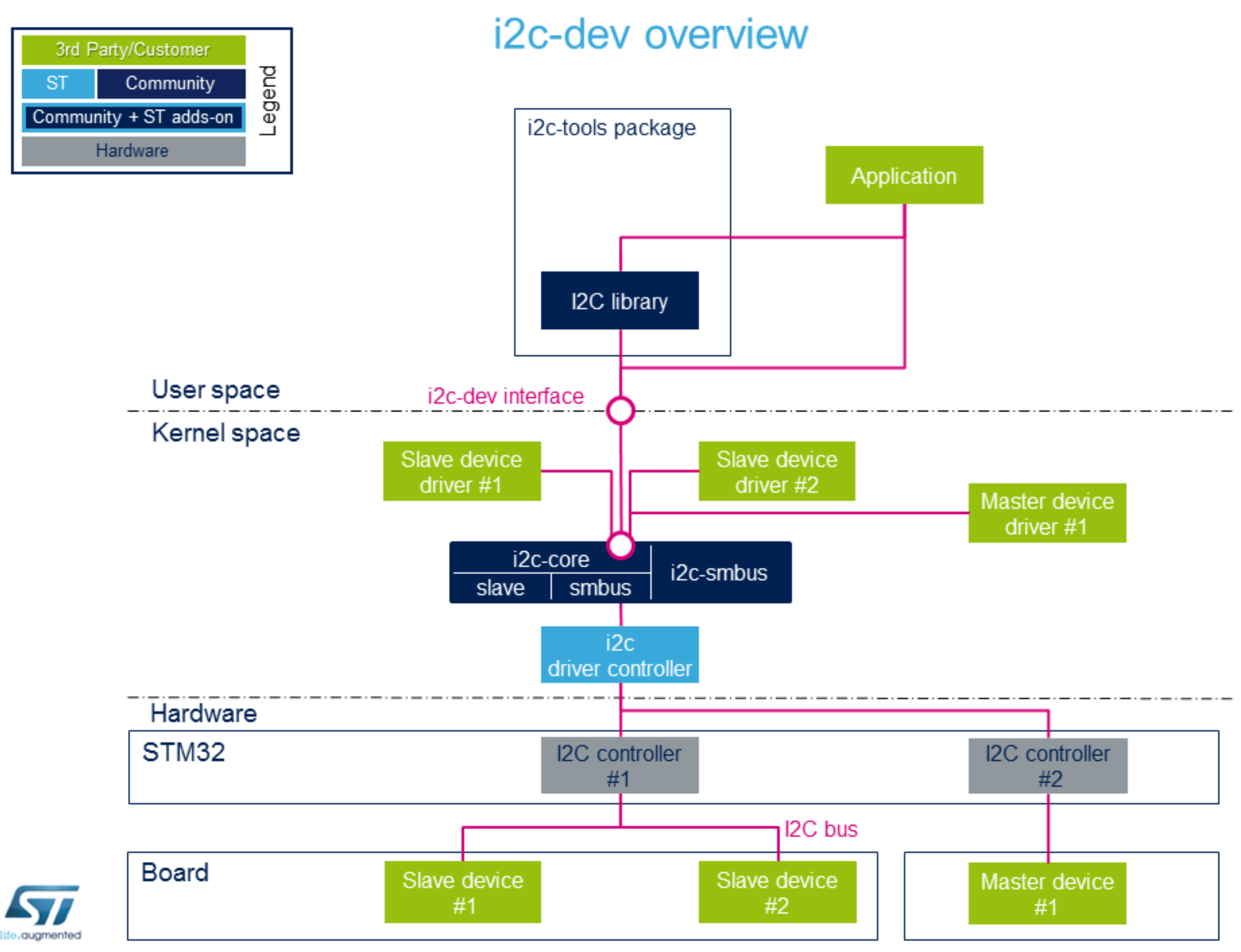
4.1 i2c-tools package

Using I2C Tools in user space with shell commands based on the **SMBus API protocol**[7] makes it easy to access I2C quickly without the need to write any code.
Use case : a lot of shell commands allow detection of I2C bus and access to I2C peripherals by SMBus protocol. The package includes a library in order to use SMBus protocol into a C program.
Full explanation is available via this link.



4.2 User space application

Allows to develop an application using the i2c-dev kernel driver in user space with this **device interface**[8].
Use case : by loading i2c-dev module, user can access I2C through the **/dev** interface. Access to I2C can be done very easily with functions open(), ioctl(), read(), write() and close(). If the peripheral is compatible, SMBus protocol access is also possible using the I2C Tools library.

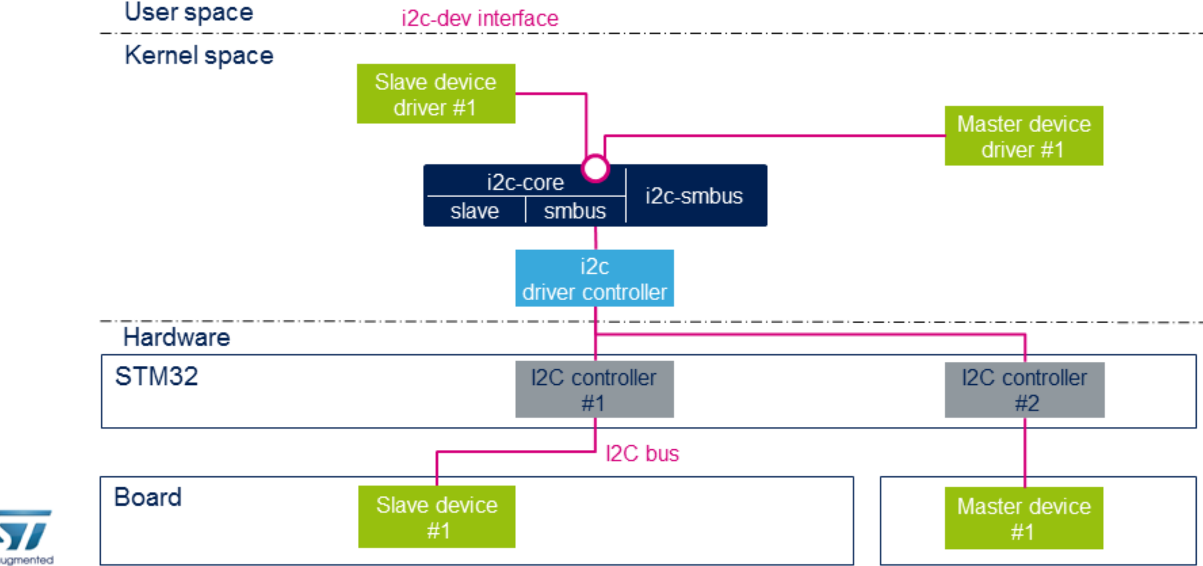


4.3 Kernel space driver

Allows to develop a driver compiled into the kernel or inserted as a module using this **I2C core API**[9]

The Linux kernel provides example about how to write an I2C client driver.[10]

Use case : control I2C peripheral with a specific driver inside the kernel space. The driver initializes all parameters while system is booting and creates an access to the peripheral data through sysfs for example.



4.4 Board description

To instantiate a peripheral, several methods exist: see **instantiating devices**[11] for more details.

The below information focuses on **device tree**, **sysfs** and **Application Code**.

4.4.1 Device tree

The device tree is a description of the hardware that is used by the kernel to know which devices are connected. In order to add a slave device on an I2C bus, complete the device tree with the information related to the new device.

Example : with an EEPROM

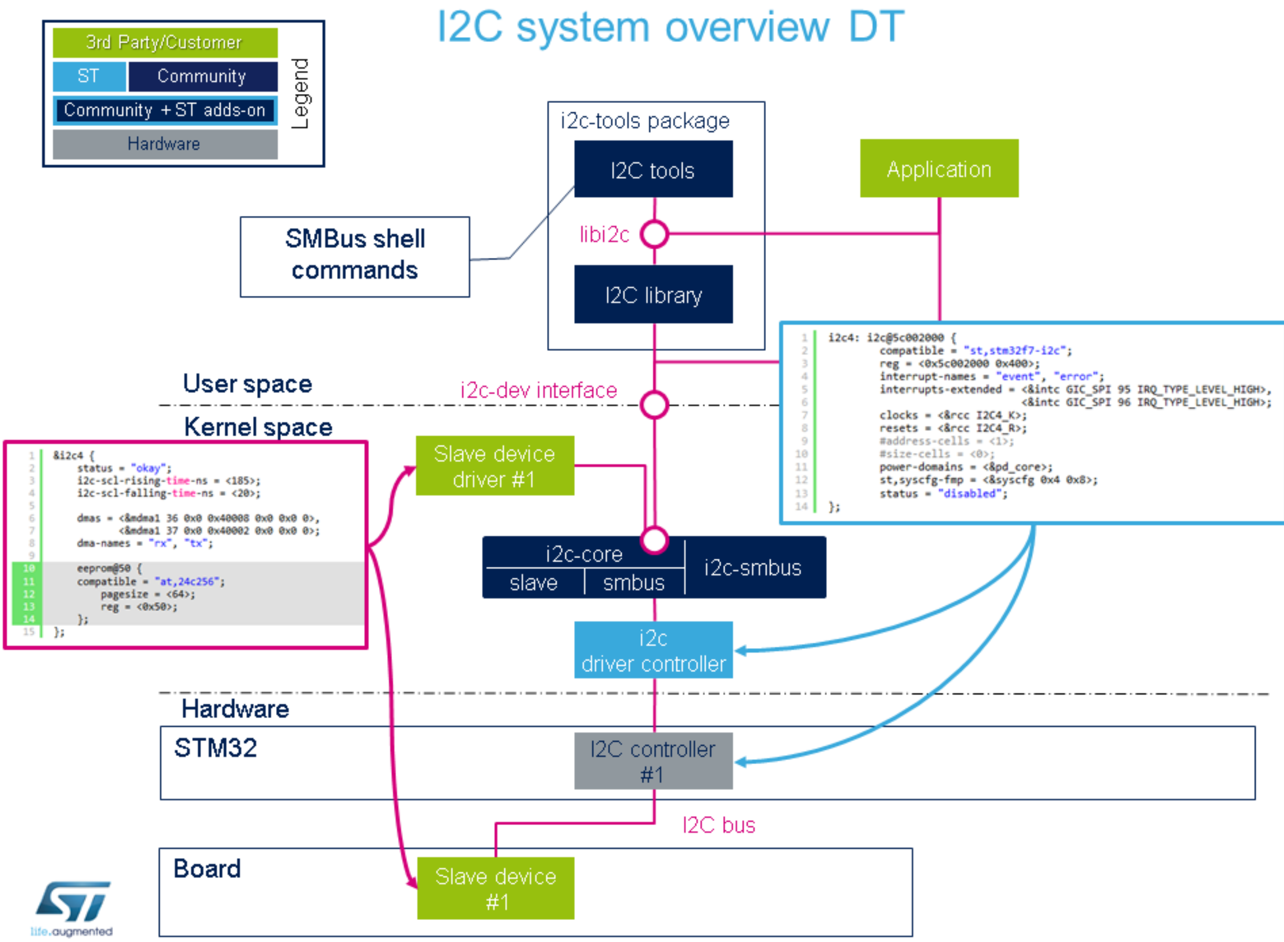
```
 1 &i2c4 {
 2     status = "okay";
 3     i2c-scl-rising-time-ns = <185>;
 4     i2c-scl-falling-time-ns = <20>;
 5
 6     dmas = <&mdma1 36 0x0 0x40008 0x0 0x0 0>,
 7            <&mdma1 37 0x0 0x40002 0x0 0x0 0>;
 8     dma-names = "rx", "tx";
 9
10     eeprom@50 {
```

```
11      compatible = "at,24c256";
12          pagesize = <64>;
13          reg = <0x50>;
14      };
15 };
```

The EEPROM is now instantiated on the bus i2c-X (X depends on how many adapters are probed at runtime) at address 0x50 and it is compatible with the driver registered with the same property.
Please note the driver specifies a SCL rising/falling time as input.

Please refer to I2C device tree configuration for proper configuration and explanation.

Be aware the I2C specification reserves a range of addresses for special purposes, see **slave addressing**[12].
The below figure shows the relation between the device tree and how it is used :



4.4.2 sysfs

Through sysfs, i2c-core offers the possibility to instantiate and remove a peripheral:

Add a peripheral "myPeripheralName" attached to the bus x at the address 0xAA
Note that the field "myPeripheralName" should have the same name as the compatible driver string so that they match one another.

```
echo myPeripheralName 0xAA > i2c-x/new_device
```

Remove a peripheral attached to the bus x at the address 0xAA

```
echo 0xAA > i2c-x/delete_device
```

Into each driver directory (**/sys/bus/i2c/drivers/at24/** for the EEPROM peripheral example), it is possible to:
bind a peripheral with a driver

```
echo 3-0050 > bind
```

unbind a peripheral with a driver

```
echo 3-0050 > unbind
```

4.4.3 Application code

Here is a minimalist code to register a new slave device onto I2C adapter without Device Tree usage.

```
 1 #include <linux/i2c.h>
 2 
 3 /* Create a device with slave address <0x42> */
 4 static struct i2c_board_info stm32_i2c_test_board_info = {
 5     I2C_BOARD_INFO("i2c_test07", 0x42);
 6 };
 7 
 8 /*
 9    Module define creation skipped
10 */
11 
12 static int __init i2c_test_probe(void)
13 {
14     struct i2c_adapter *adap;
15     struct i2c_client *client;
16 
17     /* Get I2C controller */
18     adap = i2c_get_adapter(i);
19     /* Build new devices */
20     client = i2c_new_device(adap,&stm32_i2c_test_board_info);
21 }
```

5 How to trace and debug the framework

In Linux® kernel, there are standard ways to debug and monitor I2C. The debug can take place at different levels: hardware and software.

5.1 How to trace

5.1.1 Dynamic trace

Detailed dynamic trace is available here How to use the kernel dynamic debug

```
Board $> echo  "file i2c-* +p" > /sys/kernel/debug/dynamic_debug/control
```

This command enables all traces related to I2C core and drivers at runtime.

Nonetheless at Linux® Kernel menu configuration level, it provides the granularity for debugging: Core and/or Bus.

```
Device Drivers ->
    [*] I2C support ->
        [*] I2C Core debugging messages
        [*] I2C Bus debugging messages
```

- I2C Core debugging messages (CONFIG_I2C_DEBUG_CORE)

Compile I2C engine with DEBUG flag.

- I2C Bus debugging messages (CONFIG_I2C_DEBUG_BUS)

Compile I2C drivers with DEBUG flag.

Having both **I2C Core** and **I2C Bus** debugging messages is equivalent to using the above dynamic debug command: the dmesg output will be the same.

5.1.2 Bus snooping

Bus snooping is really convenient for viewing I2C protocol and see what has been exchanged between the STM32 and the devices.
As this debug feature uses Ftrace, please refer to the Ftrace article for enabling it.

In order to access to events for I2C bus snooping, the following kernel configuration is necessary:

```
Kernel hacking ->
    [*] Tracers ->
        [*] Trace process context switches and events
```

Depending on the protocol being used, it is necessary to enable i2c and/or smbus tracers as follow:

```
echo 1 > /sys/kernel/debug/tracing/events/i2c/enable
echo 1 > /sys/kernel/debug/tracing/events/smbus/enable
```

Then tracing is enabled using the following command:

```
echo 1 > /sys/kernel/debug/tracing/tracing_on
```

After a transaction, trace can be read by looking at the trace file:

```
cat /sys/kernel/debug/tracing/trace
```

Here is part of the output, and how it looks like when using *i2cdetect* command on the i2c-0 bus:

```
... smbus_write: i2c-0 a=003 f=0000 c=0 QUICK l=0 []
... smbus_result: i2c-0 a=003 f=0000 c=0 QUICK wr res=-6
... smbus_write: i2c-0 a=004 f=0000 c=0 QUICK l=0 []
... smbus_result: i2c-0 a=004 f=0000 c=0 QUICK wr res=-6
```

Information

Notice that i2cdetect, i2cget/i2cput, i2cdump are doing smbus protocol based transactions.

On the contrary, below output shows the result of a transaction done in I2C protocol mode:

```
... i2c_write: i2c-1 #0 a=042 f=0000 l=1 [45]
... i2c_result: i2c-1 n=1 ret=1
... i2c_write: i2c-2 #0 a=020 f=0000 l=1 [45]
... i2c_result: i2c-2 n=1 ret=1
```

The utilization of traces of I2C bus is well described here **I2C bus snooping**[13].

5.2 How to debug

5.2.1 Detect I2C configuration

5.2.1.1 sysfs

When a peripheral is instantiated, i2c-core and the kernel export different files through sysfs :
/sys/class/i2c-adapter/i2c-x shows all instantiated I2C buses with 'x' being the I2C bus number.
/sys/bus/i2c/devices lists all instantiated peripherals. For example, there is a directory named **3-0050** that corresponds to the EEPROM peripheral at address 0x50 on bus number 3.
/sys/bus/i2c/drivers lists all instantiated drivers. Directory named **at24/** is the driver of EEPROM.

```
/sys/bus/i2c/devices/3-0050/
            /       /
            /       /i2c-3/3-0050/
            /
            /drivers/at24/3-0050/
```

```
/sys/class/i2c-adapter/i2c-0/
                      /i2c-1/
                      /i2c-2/
                      /i2c-3/3-0050/
                      /i2c-4/
                      /i2c-5/
```

5.2.2 devfs

If i2c-dev driver is compiled into the kernel, the directory **dev** contains all I2C bus names numbered i2c-0 to i2c-n.

```
/dev/i2c-0
    /i2c-1
    /i2c-2
    /i2c-3
    /i2c-4
    /i2c-n
```

5.2.3 i2c-tools

Check all I2C instantiated adapters:

```
Board $>i2cdetect -l
```

See i2c-tools for full description.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

6 Source code location

- I2C Framework driver is in drivers/i2c drivers/i2c
- I2C STM32 Driver is in drivers/i2c/busses/i2c-stm32f7.c
- User API for I2C bus is in include/uapi/linux/i2c.h and I2C dev is include/uapi/linux/i2c-dev.h .

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

7 To go further

Bootlin has written a nice walkthrough article: *Building a Linux system for the STM32MP1: connecting an I2C sensor*[14]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

8 References

- http://www.i2c-bus.org/
- https://bootlin.com/doc/training/linux-kernel/
- Documentation/i2c/slave-interface.rst slave interface description
- https://www.i2c-bus.org/smbus/
- https://i2c.wiki.kernel.org/index.php/I2C_Tools
- Documentation/i2c/summary.rst I2C and SMBus summary
- 7.07.17.27.3 Documentation/i2c/smbus-protocol.rst SMBus protocol summary
- 8.08.1 Documentation/i2c/dev-interface.rst dev-interface API
- 9.09.1 I2C and SMBus Subsystem
- Implementing I2C device drivers
- Documentation/i2c/instantiating-devices.rst How to instantiate I2C devices
- http://www.totalphase.com/support/articles/200349176-7-bit-8-bit-and-10-bit-I2C-Slave-Addressing Slave addressing
- https://linuxtv.org/wiki/index.php/Bus_snooping/sniffing#i2c I2C Bus Snooping
- https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-connecting-an-i2c-sensor/

Linux® is a registered trademark of Linus Torvalds.

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

Board support package

System Management Bus

Application programming interface

also known as

Device File System (See https://en.wikipedia.org/wiki/Device_file#DEVFS for more details)

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Electrically-erasable programmable read-only memory

Serial clock line