I2C device tree configuration

Contents

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

Stable: 03.06.2021 - 12:01 | Revision: 03.06.2021 - 08:40

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

Contents

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]

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.

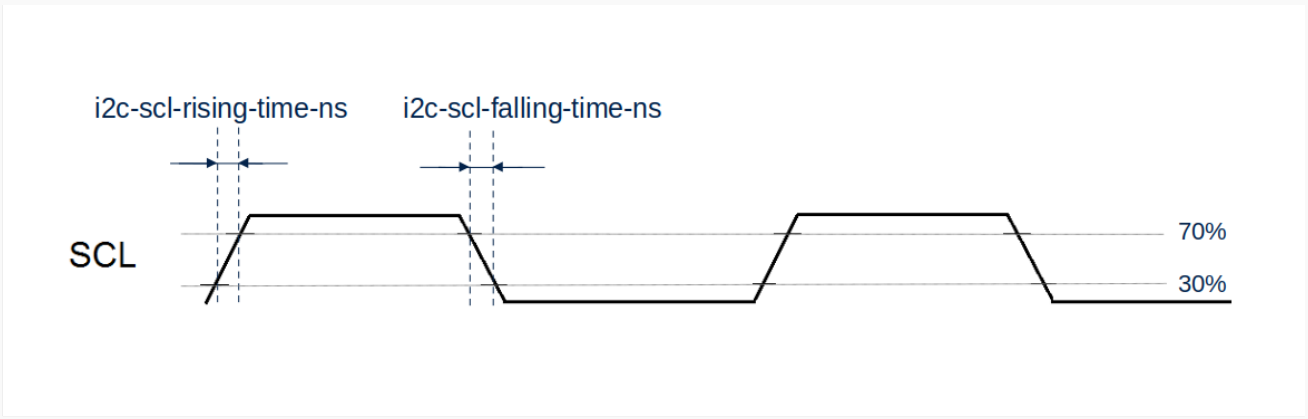
- reg represents the I2C peripheral slave address on the bus.

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>;
    };
};
```

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.

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: 04.02.2020 - 07:47 / Revision: 04.02.2020 - 07:34

A quality version of this page, approved on *4 February 2020*, was based off this revision.

Contents

1 Purpose

The objective of this chapter is to give general information about the device tree.
An extract of the **device tree specification**[1] explains it as follows:

"A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent. ... Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time."

In other words, a device tree describes the hardware that can not be located by probing. For more information, please refer to the device tree specification[1]

1.1 Source files

- **.dts:** The device tree source (DTS). This format is a textual representation of a device tree in a form that can be processed by DTC (Device Tree Compiler) into a binary device tree in the form expected by software components: Linux® Kernel, U-Boot and TF-A.

- **.dtsi:** Source files that can be included from a DTS file.

1.2 Bindings

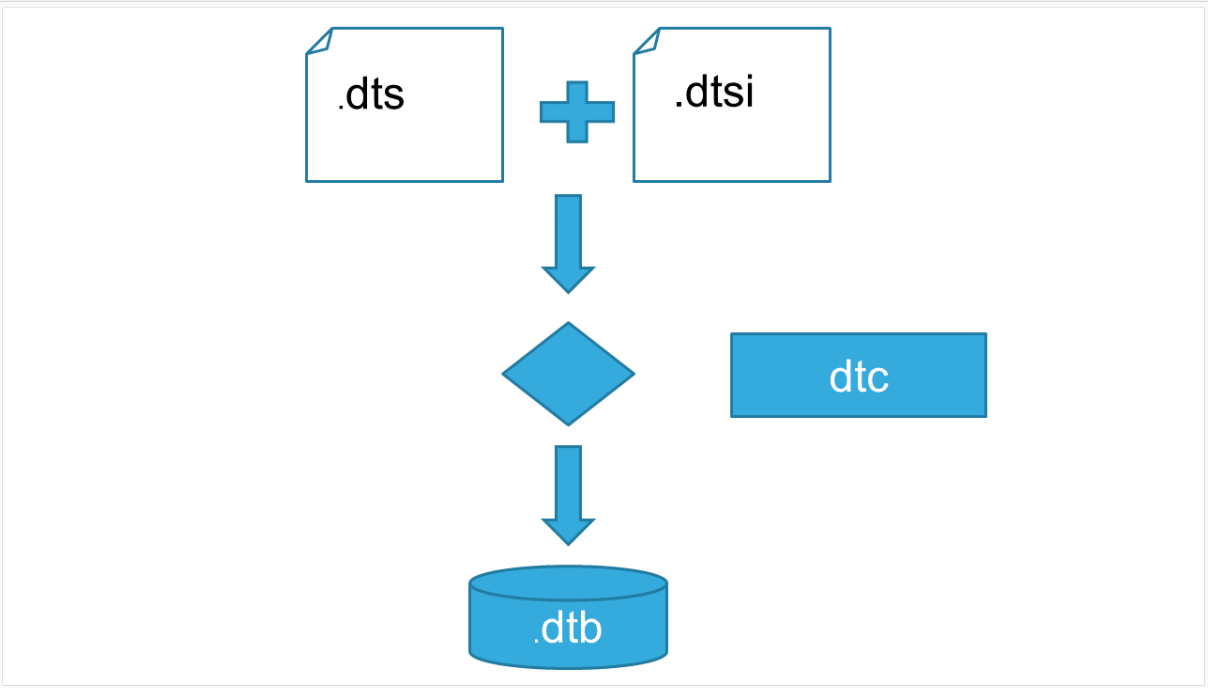
The device tree data structures and properties are named **bindings**. Those bindings are described in:

- The Device tree specification[1] for generic bindings.
- The software component documentations:
  - Linux® Kernel: Linux kernel device tree bindings
  - U-Boot: U-Boot device tree bindings
  - TF-A: TF-A device tree bindings

1.3 Build

- A tool named DTC (Device Tree Compiler) allows compiling the DTS sources into a binary.
- input file: the *.dts* file described in section above.
- output file: the *.dtb* file described in section above.
- More information are available in DTC manual[2].

- DTC source code is located here[3]. DTC tool is also available directly in particular software components: **Linux Kernel, U-Boot, TF-A ...**. For those components, the device tree building is directly integrated in the component build process.



Device tree build process

Information

If dts files use some defines, dts files should be preprocessed before being compiled by DTC.

1.4 Tools

The device tree compiler offers also some tools:

- **fdtdump:** Print a readable version of a flattened device tree file (dtb)
- **fdtget:** Read properties from a device tree
- **fdtput:** Write properties to a device tree
- ...

There are several ways to get those tools:

- In the device tree compiler project source code[3]
- Directly in software components: **Kernel, u-boot, tf-a ...**
- Available in Debian package[4]

2 STM32

For STM32MP1, the device tree is used by three software components: Linux® kernel, U-Boot and TF-A.

The device tree is part of the OpenSTLinux distribution. It can also be generated by STM32CubeMX tool.

To have more information about the device tree usage on STM32MP1 (how the device tree source files are split, how to find the device tree source files per software components, how is STM32CubeMX generating the device tree ...) see STM32MP15 device tree page.

3 How to go further

- Device Tree for Dummies[5] - Free Electrons
- Device Tree Reference[6] - eLinux.org
- Device Tree usage[7] - eLinux.org

4 References

- 1.01.11.2 https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.2(latest) ,Device tree specification
- https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/manual.txt(master) ,DTC manual
- 3.03.1 https://git.kernel.org/pub/scm/utils/dtc/dtc.git(master) ,DTC source code
- https://packages.debian.org/search?keywords=device-tree-compiler(master) ,DTC debian package
- Device Tree for Dummies, Free Electrons
- Device Tree Reference, eLinux.org
- Device Tree Usage, eLinux.org

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

Linux® is a registered trademark of Linus Torvalds.

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Trusted Firmware for Arm Cortex-A

Stable: 16.02.2021 - 17:29 / Revision: 16.02.2021 - 17:11

A quality version of this page, approved on *16 February 2021*, was based off this revision.

Contents

1 Article purpose

This article explains how to configure the software that assigns a peripheral to a runtime context.

2 Introduction

A peripheral can be **assigned** to a runtime context via the configuration defined in the device tree. The device tree can be either generated by the STM32CubeMX tool or edited manually.

On STM32MP15 line devices, the assignement can be strengthened by a hardware mechanism: the ETZPC internal peripheral, which is configured by the TF-A boot loader. The ETZPC internal peripheral isolates the peripherals for the Cortex-A7 secure or the Cortex-M4 context. The peripherals assigned to the Cortex-A7 non-secure context are visible from any context, without any isolation.

The components running on the platform after TF-A execution (such as U-Boot, Linux, STM32Cube and OP-TEE) must have a **configuration** that is consistent with the assignment and the isolation configurations.

The following sections describe how to configure TF-A, U-Boot, Linux and STM32Cube accordingly.

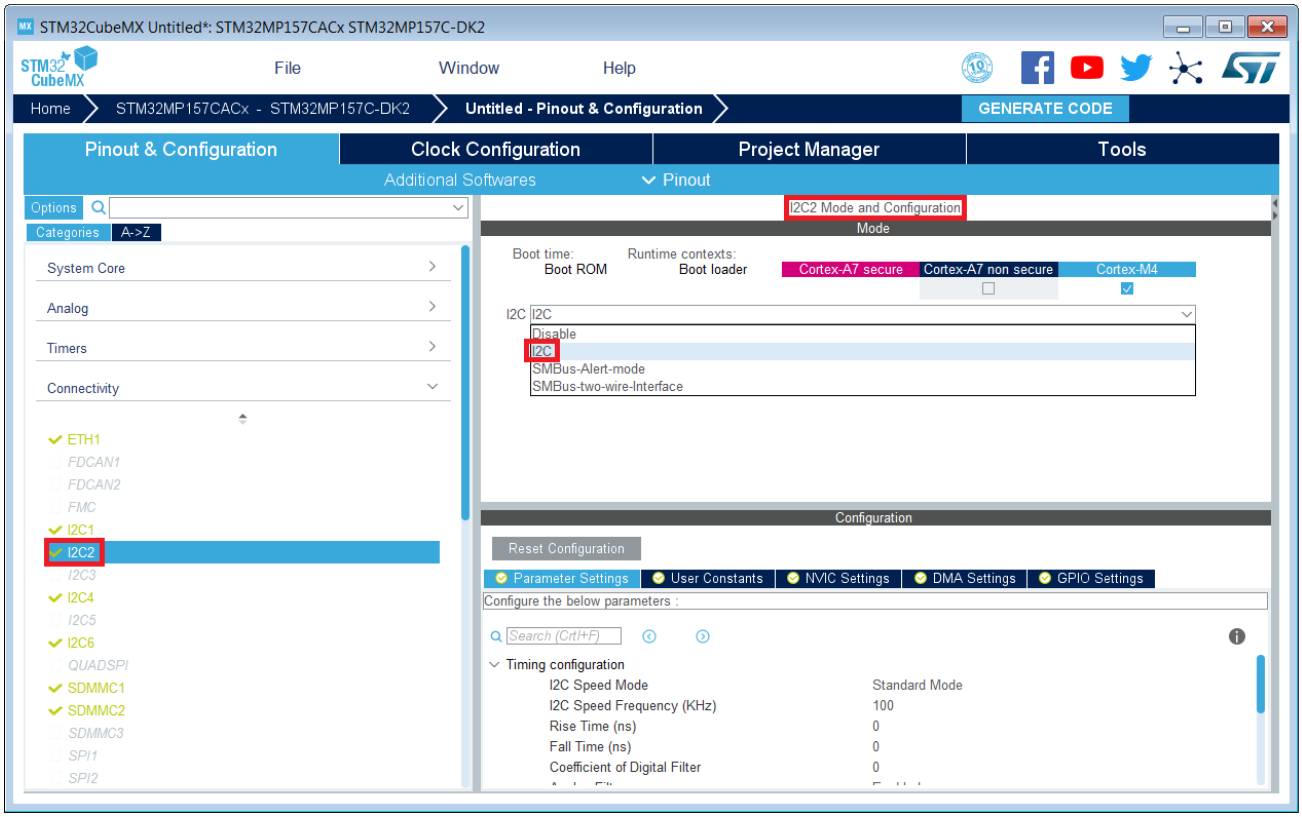
Information

Beyond the peripherals assignment, explained in this article, it is also important to understand How to configure system resources (i.e clocks, regulator, gpio,...), shared between the Cortex-A7 and Cortex-M4 contexts

3 STM32CubeMX generated assignment

The screenshot below shows the STM32CubeMX user interface:

- **I2C2** peripheral is selected, on the left
- **I2C2 Mode and Configuration** panel, on the right, shows that this I2C instance can be assigned to the Cortex-A7 non-secure or the Cortex-M4 (that is selected) runtime context
- **I2C** mode is enabled in the drop down menu



Information

The context assignment table is displayed inside each peripheral **Mode and Configuration** panel but it is possible to display it for all the peripherals in the **Options** menu via the **Show contexts** option

The **GENERATE CODE** button, on the top right, produces the following:

- The **TF-A device tree** with the ETZPC configuration that isolates the I2C2 instance (in the example) for the Cortex-M4 context. This same device tree can be used by **OP-TEE**, when enabled
- The **U-Boot device tree** widely inherited from the Linux one, just below
- The **Linux kernel device tree** with the I2C node disabled for Linux and enabled for the coprocessor
- The **STM32Cube project** with I2C2 HAL initialization code

The Manual assignment section, just below, illustrates what STM32CubeMX is generating as it follows the same example.

Information

In addition of this generation, the user may have to manually complete the system resources configuration in the user sections embedded in the STM32CubeMX generated device tree. Refer to How to configure system resources for details.

4 Manual assignment

This section gives step by step instructions, per software components, to manually perform the peripherals assignments.
It takes the same I2C2 example as the previous section, that showed how to use STM32CubeMX, in order to make the move from one approach to the other easier.

Information

The assignments combinations described in the STM32MP15 peripherals overview article are naturally supported by STM32MPU Embedded Software distribution. Note that the STM32MP15 reference manual may describe more options that would require embedded software adaptations

4.1 TF-A

The assignment follows the ETZPC device tree configuration, with below possible values:

- **DECPROT_S_RW** for the Cortex-A7 secure (Secure OS like OP-TEE)
- **DECPROT_NS_RW** for the Cortex-A7 non-secure (Linux)
  - As stated earlier in this article, there is no hardware isolation for the Cortex-A7 non-secure so this value allows accesses from any context
- **DECPROT_MCU_ISOLATION** for the Cortex-M4 (STM32Cube)

Example:

```
@etzpc: etzpc@5C007000 {
        st,decprot = <
            DECPROT(STM32MP1_ETZPC_I2C2_ID, DECPROT_MCU_ISOLATION, DECPROT_UNLOCK)
        >;
};
```

Information

The value **DECPROT_NS_RW** can be used with **DECPROT_LOCK** as last parameter. In Cortex-M4 context, this specific configuration allows the generation of an error in the resource manager utility while trying to use on Cortex-M4 side a peripheral that is assigned to the Cortex-A7 non-secure context. If **DECPROT_UNLOCK** is used, then the utility allows the Cortex-M4 to use a peripheral that is assigned to the Cortex-A7 non-secure context.

4.2 U-boot

No specific configuration is needed in U-Boot to configure the access to the peripheral.

Information

U-Boot does not perform any check with regards to ETZPC configuration before accessing to a peripheral. In case of inconsistency an illegal access is generated.

Information

U-Boot checks the consistency between ETZPC isolation configuration and Linux kernel device tree configuration to guarantee that Linux kernel do not access an unauthorized device. In order to avoid the access to an unauthorized device, the U-boot fixes up the Linux kernel device tree to disable the peripheral nodes which are not assigned to the Cortex-A7 non-secure context.

4.3 Linux kernel

Each assignable peripheral is declared twice in the Linux kernel device tree:

- Once in the **soc** node from arch/arm/boot/dts/stm32mp151.dtsi , corresponding to Linux assigned peripherals
  - Example: i2c2
- Once in the **m4_rproc** node from arch/arm/boot/dts/stm32mp157-m4-srm.dtsi , corresponding to the Cortex-M4 context.

Those nodes are disabled, by default.

  - Example: m4_i2c2

In the board device tree file (*.dts), each assignable peripheral has to be enabled only for the context to which it is assigned, in line with TF-A configuration.
As a consequence, a peripheral assigned to the Cortex-A7 secure has both nodes disabled in the Linux device tree.

Example:

```
&i2c2 {
        status = "disabled";
};
...
&m4_i2c2 {
        status = "okay";
};
```

Information

In addition of this assignment, the user may have to complete the system resources configuration in the device tree nodes. Refer to How to configure system resources for details.

4.4 STM32Cube

There is no configuration to do on STM32Cube side regarding the assignment and isolation. Nevertheless, the resource manager utility, relying on ETZPC configuration, can be used to check that the corresponding peripheral is well assigned to the Cortex-M4 before using it.

Example:

```
int main(void)
{
...
 /* Initialize I2C2-------------------------------------------------------- */
 /* Ask the resource manager for the I2C2 resource */
 ResMgr_Init(NULL, NULL);
 if (ResMgr_Request(RESMGR_ID_I2C2, RESMGR_FLAGS_ACCESS_NORMAL | \
```

```
                                        RESMGR_FLAGS_CPU1, 0, NULL) != RESMGR_OK)
 {
   Error_Handler();
 }
...
 if (HAL_I2C_Init(&I2C2) != HAL_OK)
 {
   Error_Handler();
 }
}
```

4.5 OP-TEE

The OP-TEE OS may use STM32MP1 resources. OP-TEE STM32MP1 drivers register the device driver they intend to used in a secure context. This information is used to consolidate system configuration including secure hardening of configurable peripherals.

In most case, the OP-TEE driver probe relies on OP-TEE device tree porperty *secure-status = "okay"*.

Cortex®

Trusted Firmware for Arm Cortex-A

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Linux® is a registered trademark of Linus Torvalds.

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

Open Portable Trusted Execution Environment

Hardware Abstraction Layer

Operating System

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained, system-on-chip (SoC) designs.)

Extended TrustZone Protection Controller

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

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]

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.

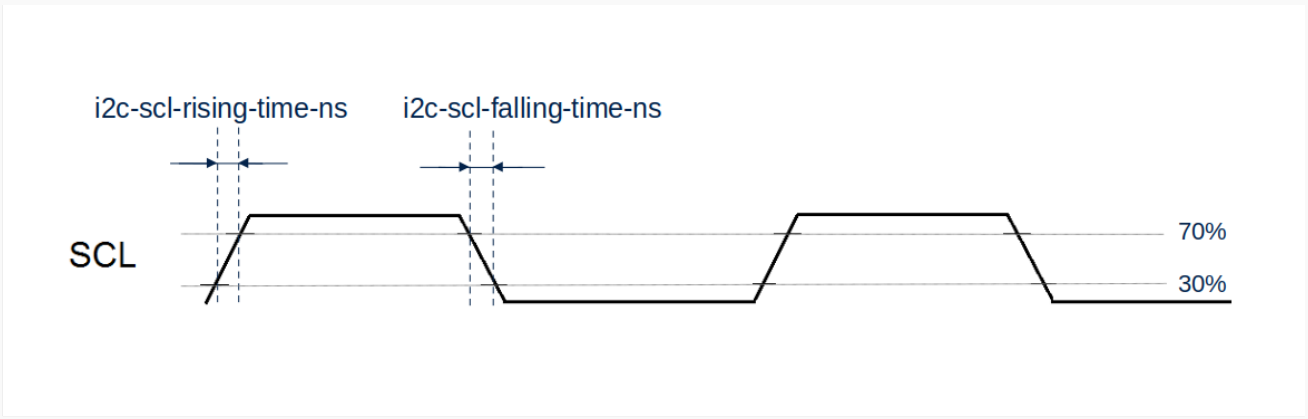
- reg represents the I2C peripheral slave address on the bus.

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>;
    };
};
```

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.

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: 25.09.2020 - 09:43 / Revision: 25.09.2020 - 09:38

A quality version of this page, approved on *25 September 2020*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to:

- briefly introduce the I2C peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the I2C peripheral.

2 Peripheral overview

The I2C bus interface serves as an interface between the microcontroller and the serial I2C bus.
It provides multi-master capability, and controls all I2C bus-specific sequencing, protocol, arbitration and timing.
The I2C controller allows to be a slave as well if need be.
It is also SMBus 2.0 compatible.
For more information about I2C please refer to this link: **I2C wikipedia**[1] or **i2c-bus.org**[2]
For more information about SMBus please refer to this link: **SMBus wikipedia**[3] or **i2c-bus.org**[4]

2.1 Features

Here are the main features:

- Multi-master
- Standard (100 KHz) and fast speed modes (400 KHz and Plus 1 MHz)
- I2C 10-bit address
- I2C slave capabilities (programmable I2C address)
- DMA capabilities
- SMBus 2.0 compatible
  - Standard bus protocol (quick command; byte, word, block read/write)
  - Host notification
  - Alert

Refer to the STM32MP15 reference manuals for the complete list of features, and to the software components, introduced below, to see which features are implemented.

2.2 Security support

- There are six I2C instances.
  - I2C instances 1, 2, 3 and 5 are **non-secure**.
  - I2C instances 4 and 6 can be **secure** (under ETZPC control).

3 Peripheral usage and associated software

3.1 Boot time

The I2C peripheral is usually not used at boot time. But it may be used by the SSBL and/or FSBL (see Boot chain overview), for example, to configure a PMIC (see PMIC hardware components), or to access data stored in an external EEPROM.

3.2 Runtime

3.2.1 Overview

I2C4&6 instances can be allocated to:

- the Arm®Cortex®-A7 secure core to be controlled in OP-TEE by the OP-TEE I2C driver

All I2C instances can be allocated to:

- the Arm®Cortex®-A7 non-secure core to be controlled in U-Boot or Linux® by the I2C framework

All but I2C4&6 instances can be allocated to:

- the Arm®Cortex®-M4 to be controlled in STM32Cube MPU Package by STM32Cube I2C driver

Chapter Peripheral assignment describes which peripheral instance can be assigned to which context.

3.2.2 Software frameworks

Domain	Peripheral	Software frameworks			Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Low speed interface	I2C	OP-TEE I2C driver	I2C Engine framework	STM32Cube I2C driver	

3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals), according to the information given in the corresponding software framework article.

For Linux® kernel configuration, please refer to I2C configuration.

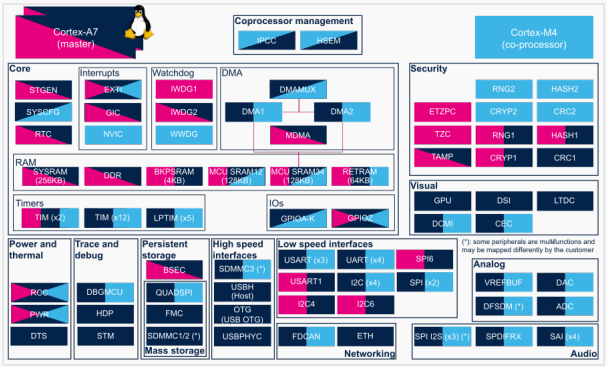
Please refer to I2C device tree configuration for detailed information on how to configure I2C peripherals.

3.2.4 Peripheral assignment

Check boxes illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

- means that the peripheral can be assigned () to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to How to assign an internal peripheral to a runtime context for more information on how to assign peripherals manually or via STM32CubeMX.
The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possiblities might be described in STM32MP15 reference manuals .



Internal peripherals

Domain	Periphera l	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Low speed interface	I2C	I2C1				Assignment (single choice)
		I2C2				Assignment (single choice)
		I2C3				Assignment (single choice)
		I2C4				Assignment (single choice). Used for PMIC control on ST boards .
		I2C5				Assignment (single choice)
		I2C6				Assignment (single choice)

4 References

- http://en.wikipedia.org/wiki/I2C
- https://www.i2c-bus.org/specification/
- https://en.wikipedia.org/wiki/System_Management_Bus
- https://www.i2c-bus.org/smbus/

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

System Management Bus

Direct Memory Access

Second Stage Boot Loader

First Stage Boot Loader

Power Management Integrated Circuit

Electrically-erasable programmable read-only memory

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cortex®

Open Portable Trusted Execution Environment

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Linux® is a registered trademark of Linus Torvalds.

Microprocessor Unit

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

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® I²C[1] interface in **master** and **slave** modes.

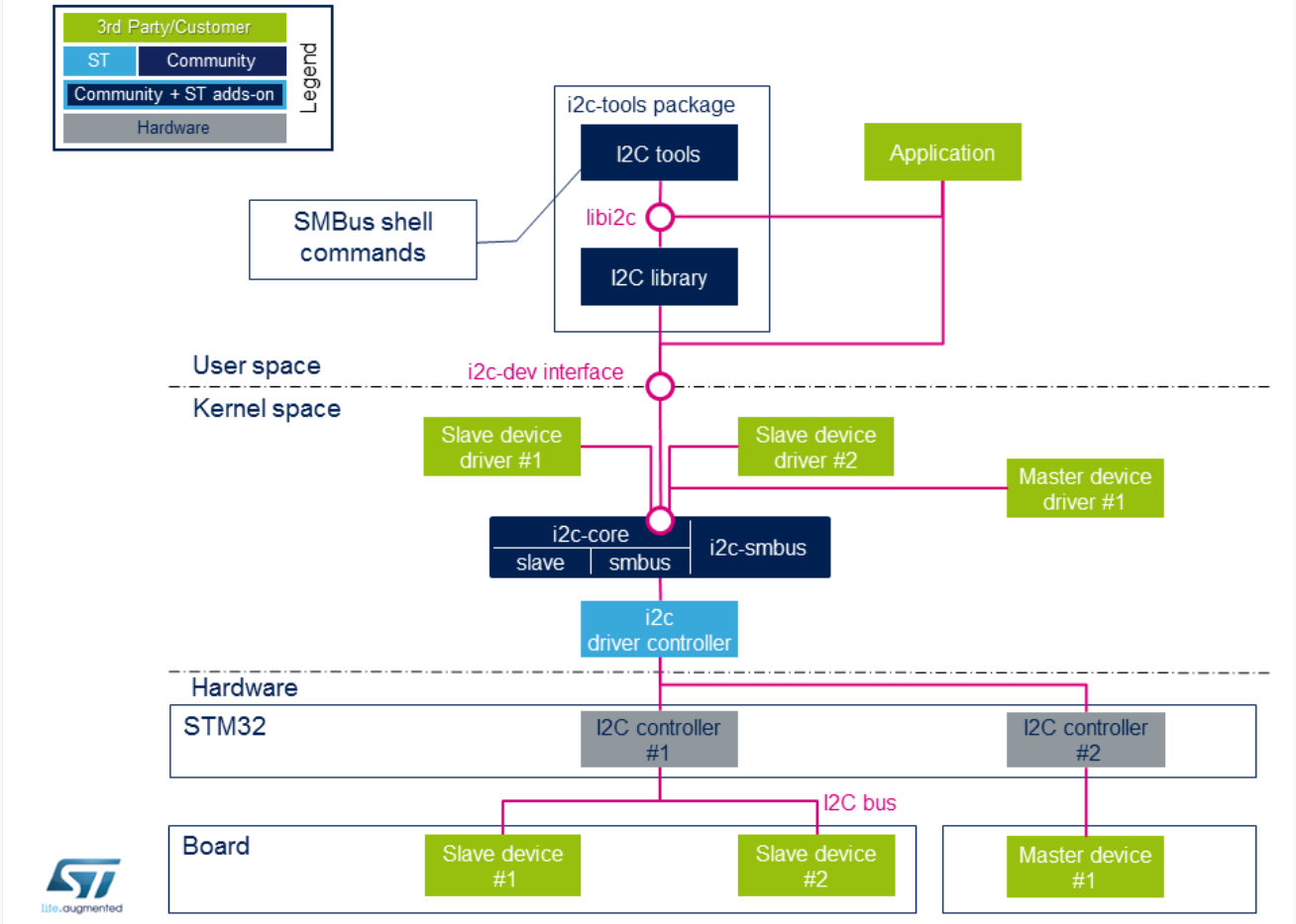
An introduction to I²C[2] is proposed through this external resource.

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

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^{2}C bus) that behave as Master with respect to STM32.

STM32 behaves as a Slave in this case on the I^{2}C 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]

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

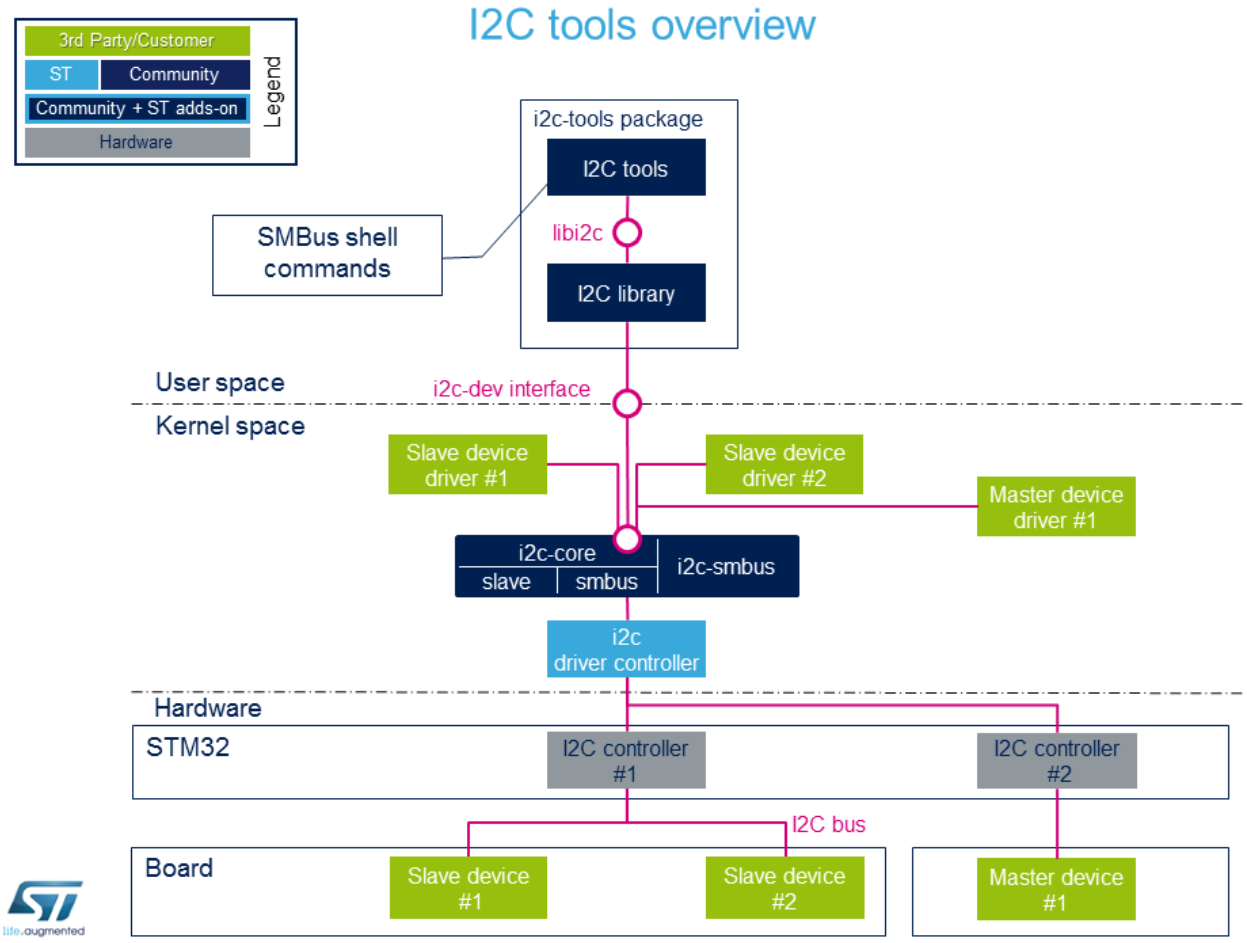
This section describes how to use the framework to access I2C peripherals.

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.

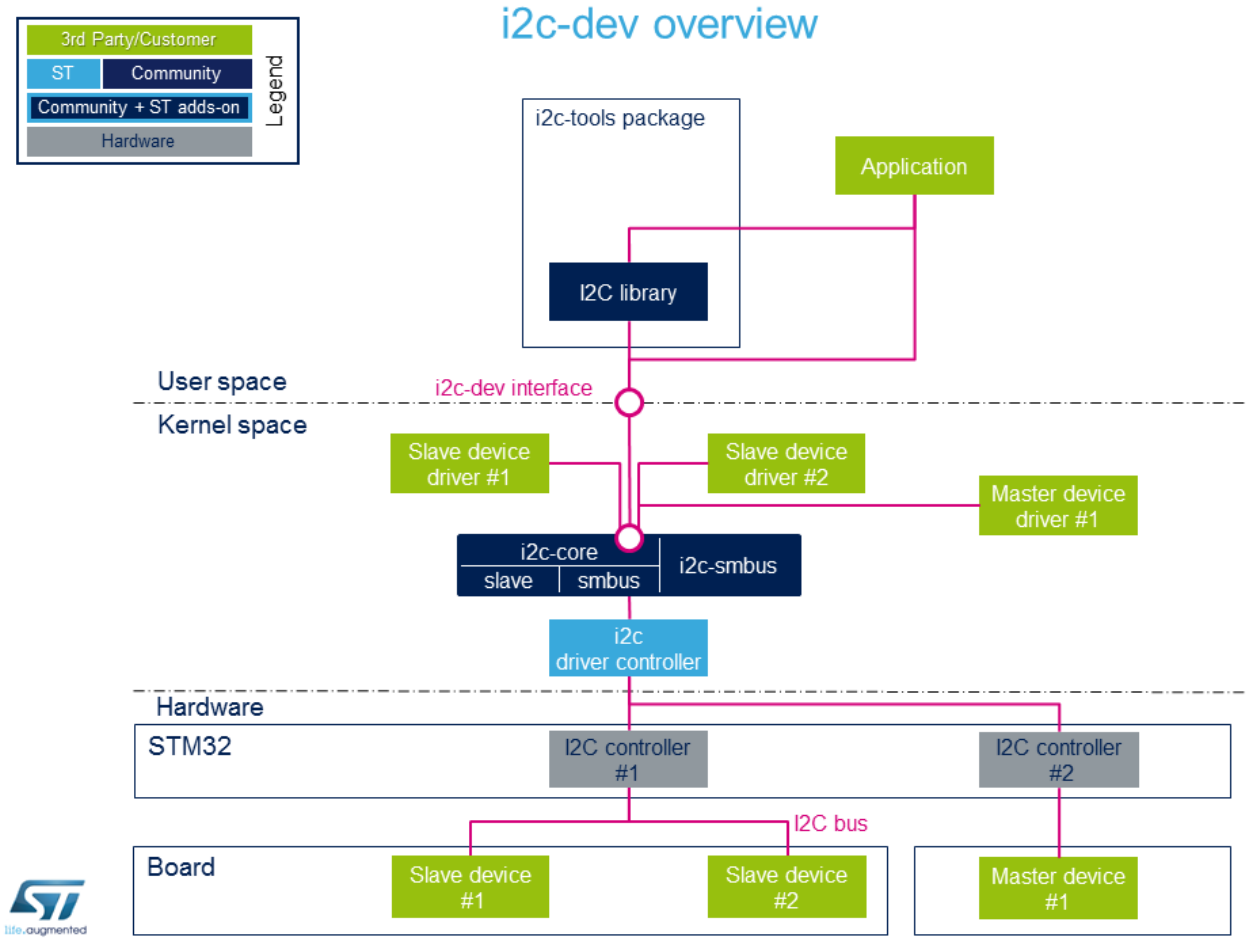
Full explanation is available via this link.



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.

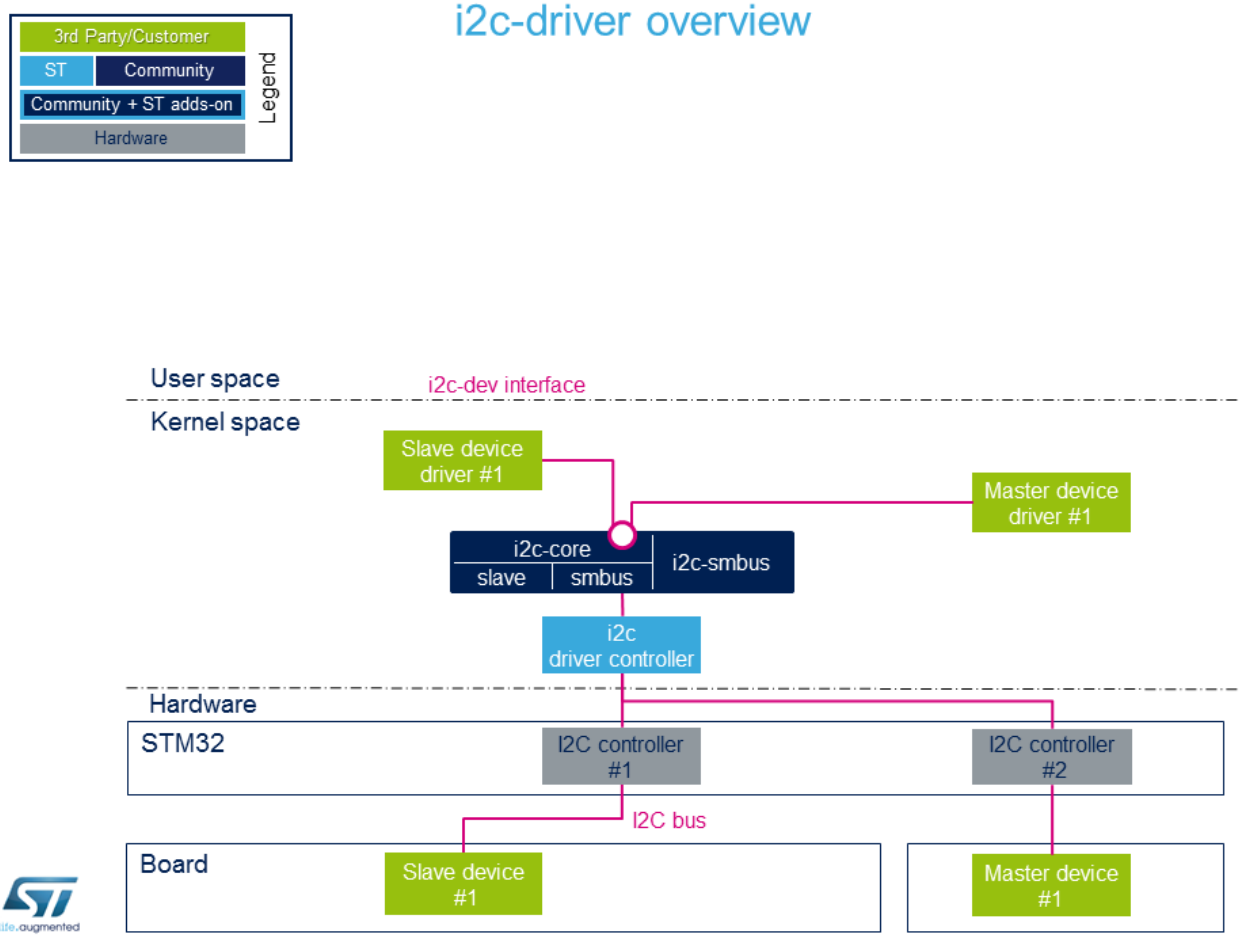


4.3 Kernel space driver

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

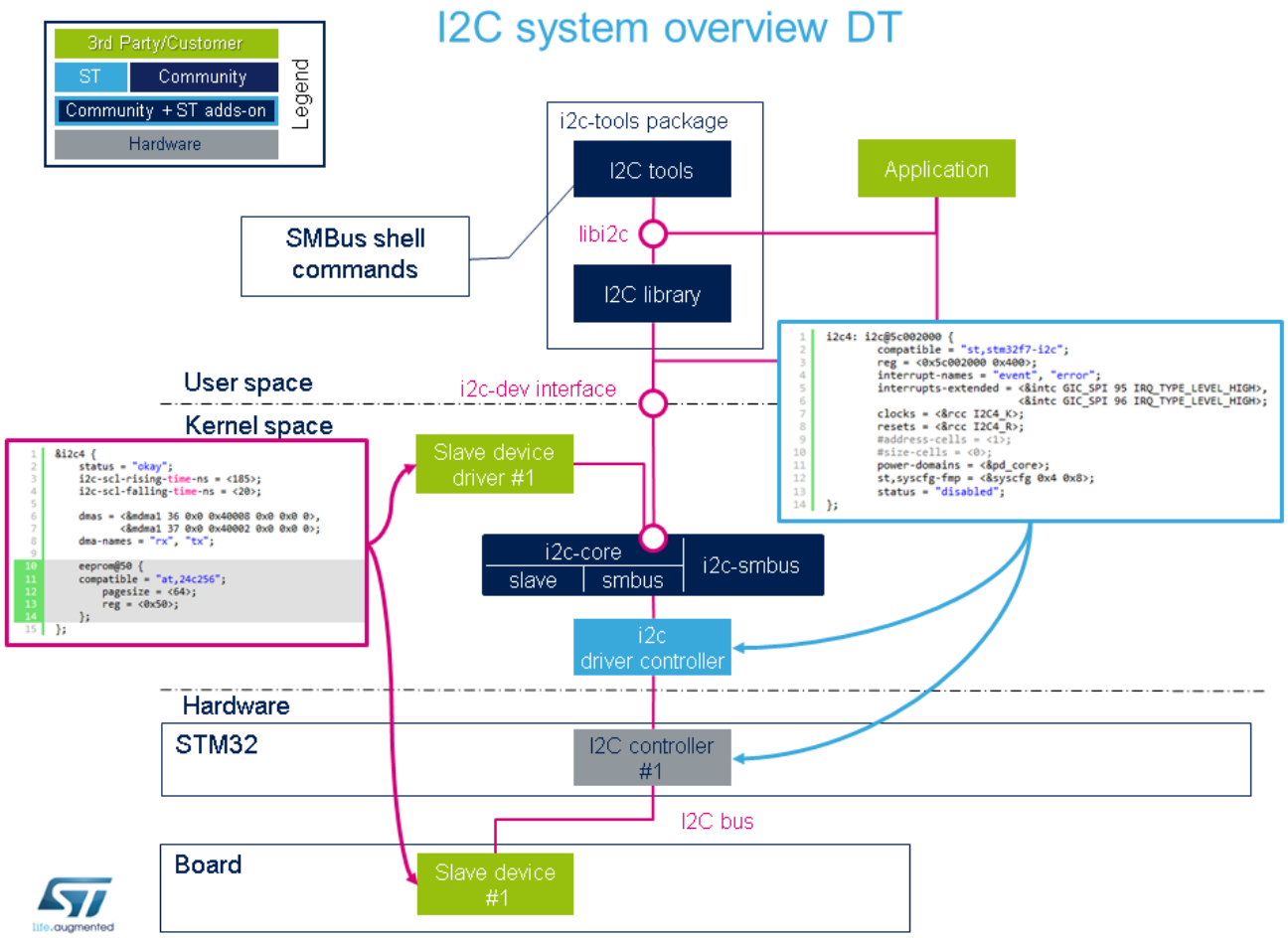
```
1 &i2c4 {
2     status = "okay";
3     i2c-scl-rising-time-ns = <185>;
4     i2c-scl-falling-time-ns = <20>;
5
6     dmas = <&mdma1 36 0x0 0x40008 0x0 0x0 0>,
7            <&mdma1 37 0x0 0x40002 0x0 0x0 0>;
8     dma-names = "rx", "tx";
9
10    eeprom@50 {
```

```
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.

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 .

7 To go further

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

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/

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

Stable: 11.06.2020 - 09:00 / Revision: 10.06.2020 - 15:35

A quality version of this page, approved on *11 June 2020*, was based off this revision.

Contents

1 Purpose

The purpose of this article is to explain how to configure the GPIO internal peripheral through **the pin controller (pinctrl) framework**, **when this peripheral is assigned to Linux®OS (Cortex-A)**. The configuration is performed using the Device tree.

To better understand I/O management, it is recommended to read the Overview of GPIO pins article.

This article also provides an example explaining how to add a new pin in the device tree.

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 Pinctrl device tree bindings are composed of:

- generic DT bindings[1] used by the pinctrl framework.
- vendor pinctrl DT bindings[2] used by the stm32-pinctrl driver: this binding document explains how to write device tree files for pinctrl.

3 DT configuration

3.1 DT configuration (STM32 level)

The pin controller node is located in the pinctrl dtsi file *stm32mp15-pinctrl.dtsi*[3]. See Device tree for more explanations about device tree file split. The pin controller node is composed of three parts:

3.1.1 STM32 pin controller information

```
pinctrl: pin-controller@50002000 {                  Comments
         #address-cells = <1>;
         #size-cells = <1>;
         ranges = <0 0x50002000 0xa400>;     -->Provides IP start address and memory map
device size
         interrupt-parent = <&exti>;         -->Provides interrupt parent controller (used
when the GPIO is configured as an external interrupt)
         st,syscfg = <&exti 0x60 0xff>;      -->Provides phandle for IRQ mux selection
         pins-are-numbered;

        ...
};
```

Warning

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

3.1.2 GPIO bank information

```
pinctrl: pin-controller@50002000 {                  Comments
         ...

        gpioa: gpio@50002000 {
                gpio-controller;
                #gpio-cells = <2>;
                interrupt-controller;       -->Indicates that this GPIO bank can be used
as interrupt controller
                #interrupt-cells = <2>;
                reg = <0x0 0x400>;          -->Provides offset in pinctrl addres map for
the GPIO bank
                clocks = <&rcc GPIOA>;      -->phandle on GPIO bank clock
                st,bank-name = "GPIOA";
                status = "disabled";
        };

        gpiob: gpio@50003000 {
                gpio-controller;
                #gpio-cells = <2>;
                interrupt-controller;
                #interrupt-cells = <2>;
                reg = <0x1000 0x400>;
                clocks = <&rcc GPIOB>;
                st,bank-name = "GPIOB";
```

```
                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.

3.1.3 Pin state definition

- Extract of *stm32mp15-pinctrl.dtsi*[3] file:

```
&pinctrl {
        ...
         usart3_pins_a: usart3@0 {                                                  Comme
nts
                pins1 {
                        pinmux = <STM32_PINMUX('B', 10, AF7)>, /* USART3_TX */      --
>Pin muxing information: AF7 (alternate function 7) selected on PB10 pin
                                 <STM32_PINMUX('G', 8, AF8)>; /* USART3_RTS */      --
>Pin muxing information: AF8 (alternate function 8) selected on PG8 pin
                        bias-disable;                                               --
>Generic bindings corresponding to "no pull-up" and "no pull-down"
                        drive-push-pull;                                            --
>Generic bindings to select pin driving information
                        slew-rate = <0>;                                            --
>Generic bindings to select pin speed
                };
                pins2 {
                        pinmux = <STM32_PINMUX('B', 12, AF8)>, /* USART3_RX */
                                 <STM32_PINMUX('I', 10, AF8)>; /* USART3_CTS_NSS */
                        bias-disable;
                };
       };
       ...
};
```

- Refer to GPIO internal peripheral for more details on hardware pin configuration.

3.2 DT configuration (board level)

As seen in Pin controller configuration (pin state definition part), all pin states are defined inside the pin controller node.

Each device that requires pins has to select the desired pin state phandle inside the board device tree file (see Device tree for more explanations about device tree file split).

The STM32MP1 devices feature a lot of possible pin combinations for a given internal peripheral. From one board to another, different sets of pins can consequently be used for an internal peripheral. Note that "_a", "_b" suffixes are used to identify pin muxing combinations in the device tree pinctrl file. The right suffixed combination must then be used in the device tree board file.

- Example:

```
&usart3 {
        ...
        pinctrl-names = "default","sleep";
        pinctrl-0 = <&usart3_pins_a>;
        pinctrl-1 = <&usart3_sleep_pins_a>;
        ...
};
```

3.3 DT configuration examples

3.3.1 How to add new pin states

To add new pin states and affect them to a foo_device, proceed as follows:

1. Find the pins you need:

   In the example below, the foo_device needs to configure PC13, PG8 and PI2.

   AF2 is selected as alternate function on PC13, and AF5 on PG8 and PI2.

   Each pin requires an internal pull-up.

2. Write your pin state phandle in *stm32mp15-pinctrl.dtsi*.

```
 &pinctrl {
         ...
          foo_pins_a: foo@0 {
                 pins {
                         pinmux = <STM32_PINMUX('C', 13, AF2)>,
                                  <STM32_PINMUX('G', 8, AF5)>,
                                   <STM32_PINMUX('I', 2, AF5)>;
                         bias-pull-up;
                 };
        };
        ...
 };
```

   All the possible settings are described in GPIO internal peripheral.

3. Select the pin state phandle required for your device in the board file.

```
&foo {
        ...
        pinctrl-names = "default";
        pinctrl-0 = <&foo_pins_a>;
        ...
};
```

4 How to configure GPIOs 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.

5 References

Please refer to the following links for additional information:

- Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt , Generic pinctrl device tree bindings
- Documentation/devicetree/bindings/pinctrl/st,stm32-pinctrl.yaml , STM32 pinctrl device tree bindings
- 3.03.1 stm32mp15-pinctrl.dtsi STM32MP15 Pinctrl device tree file

Linux® is a registered trademark of Linus Torvalds.

Operating System

Cortex®

Device Tree

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Transmit

Receive

Compatibility Test Suite (Android specific) or Clear to send (in UART context)

Stable: 23.09.2020 - 13:22 / Revision: 12.06.2020 - 13:25

A quality version of this page, approved on *23 September 2020*, was based off this revision.

1 STM32CubeMX overview

This article describes STM32CubeMX, an official STMicroelectronics graphical software configuration tool.

The STM32CubeMX application helps developers to use the STM32 by means of a user interface, and guides the user through to the initial configuration of a firmware project.

It provides the means to:

- configure pin assignments, the clock tree, or internal peripherals
- simulate the power consumption of the resulting project
- configure and tune DDR parameters
- generate HAL initialization code for Cortex-M4
- generate the Device Tree for a Linux kernel, TF-A and U-Boot firmware for Cortex-A7

It uses a rich library of data from the STM32 microcontroller portfolio.

The application is intended to ease the initial development phase by helping developers to select the best product in terms of features and power.

2 STM32CubeMX main features

- Peripheral and middleware parameters

  Presents options specific to each supported software component

- Peripheral assignment to processors

  Allows assignment of each peripheral to Cortex-A Secure, Cortex-A Non-Secure, or Cortex-M processors

- Power consumption calculator

  Uses a database of typical values to estimate power consumption, DMIPS, and battery life

- Code generation

  Makes code regeneration possible, while keeping user code intact

- Pinout configuration

  Enables peripherals to be chosen for use, and assigns GPIO and alternate functions to pins

- Clock tree initialization

  Chooses the oscillator and sets the PLL and clock dividers

- DDR tunning tool

  Ensures the configuration, testing, and tuning of the MPU DDR parameters. Using U-Boot-SPL Embedded Software.

3 How to get STM32CubeMX

Please, refer to the following link STM32CubeMX to find STM32CubeMX, the Release Note, the User Manual and the product specification.

Doubledata rate (memory domain)

Hardware Abstraction Layer

Cortex®

Linux® is a registered trademark of Linus Torvalds.

Trusted Firmware for Arm Cortex-A

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Microprocessor Unit