I2C overview

A quality version of this page, approved on *30 March 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.

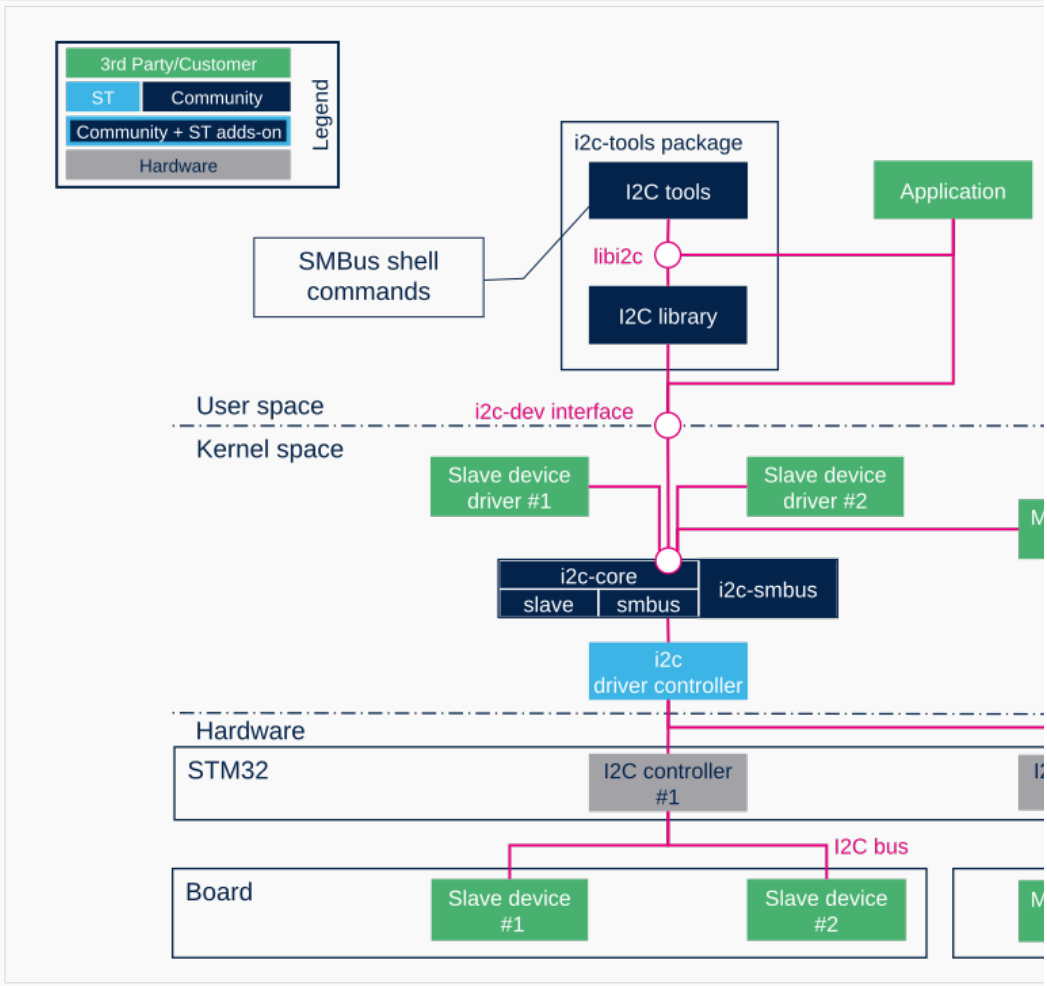
An introduction to I²C[2] is proposed through this external resource.

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

2 System overview

I^{2}C 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 I^{2}C peripheral interface.

Around the microprocessor device, the user can add many I^{2}C 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 I^{2}C devices

- Slave devices X are physical devices (connected to STM32 via I^{2}C bus) that behave as Slave with respect to STM32.

STM32 remains the master on the I^{2}C 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

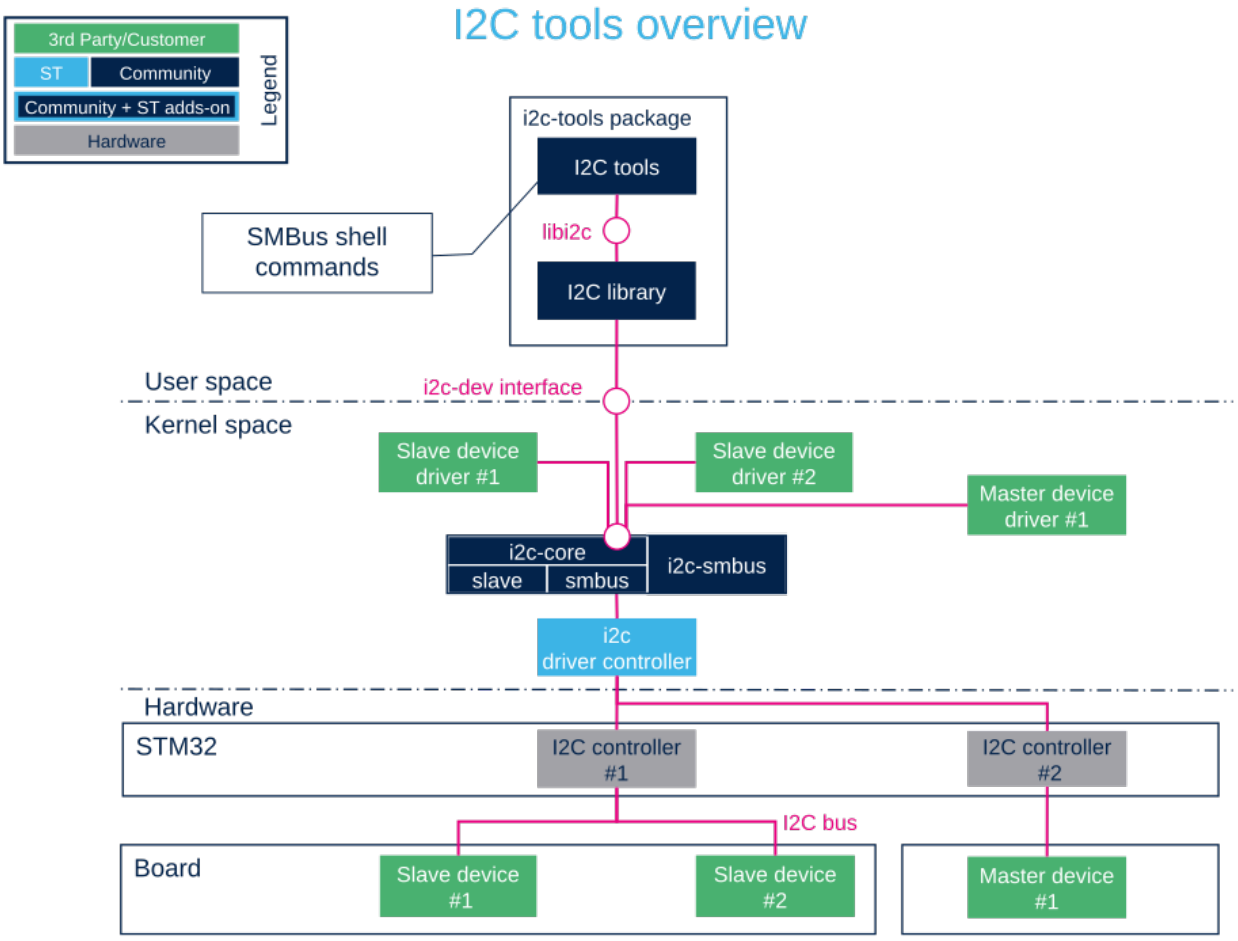
This section describes how to use the framework to access I2C peripherals.

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.

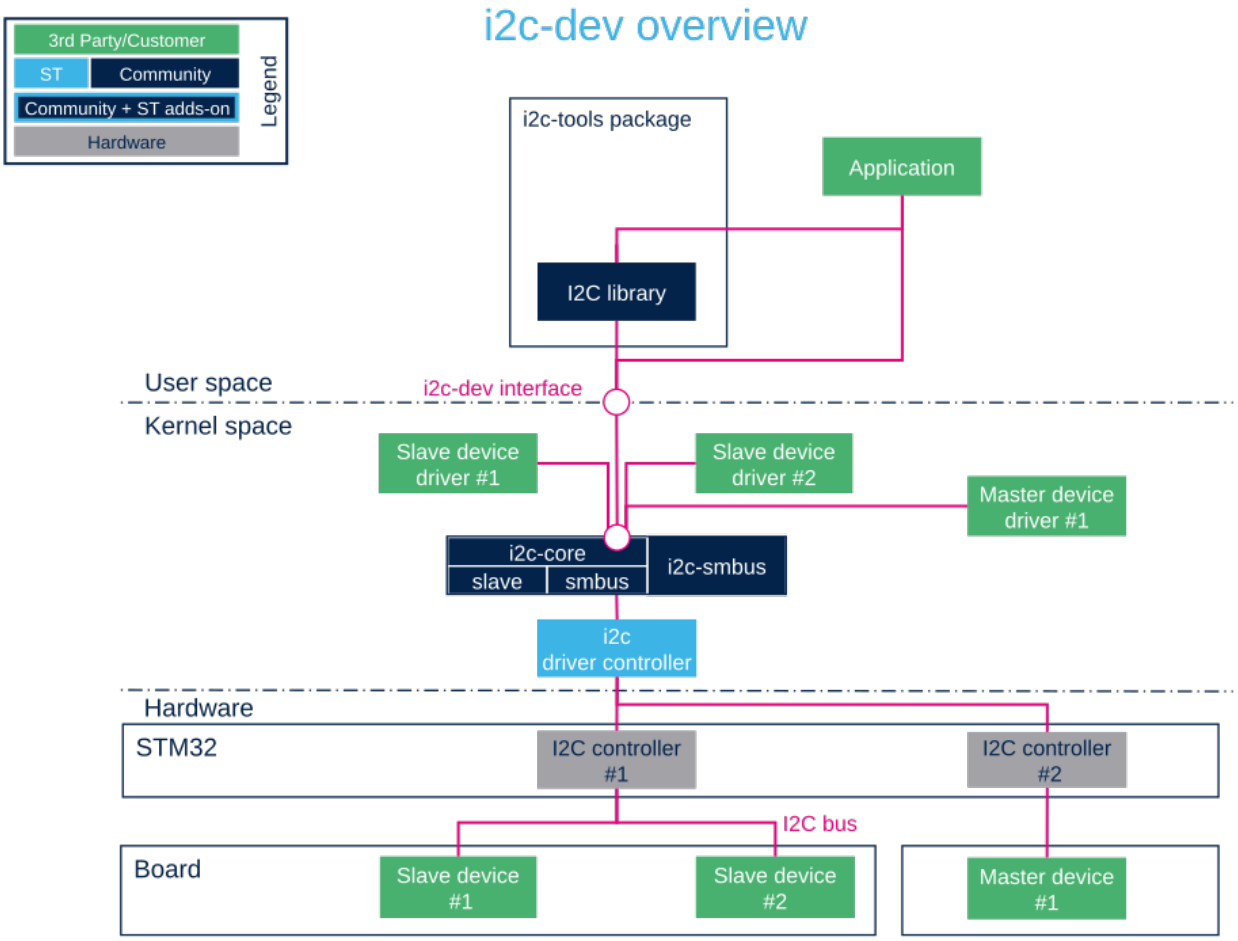
Full explanation is available via this link.



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.

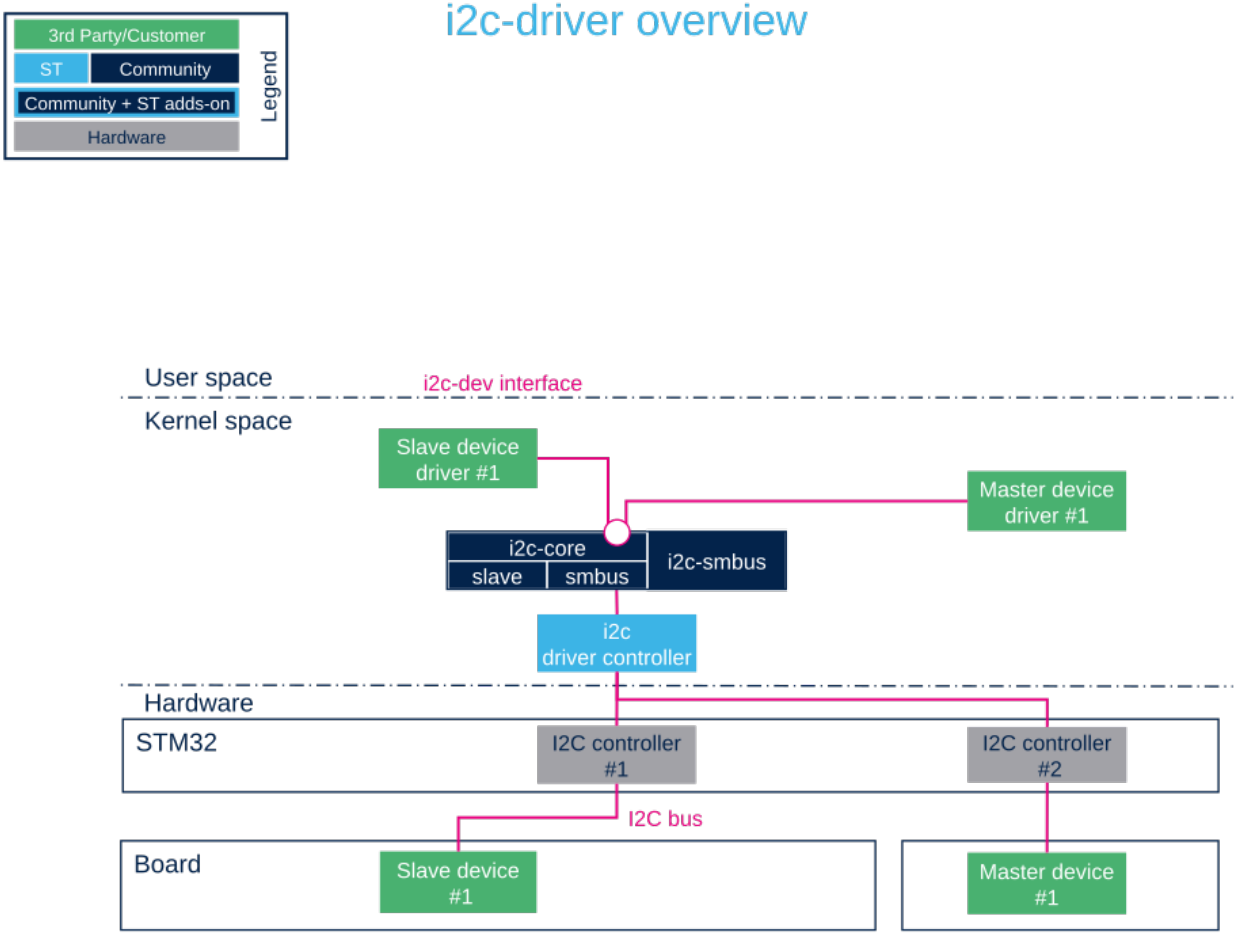


4.3 Kernel space driver

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

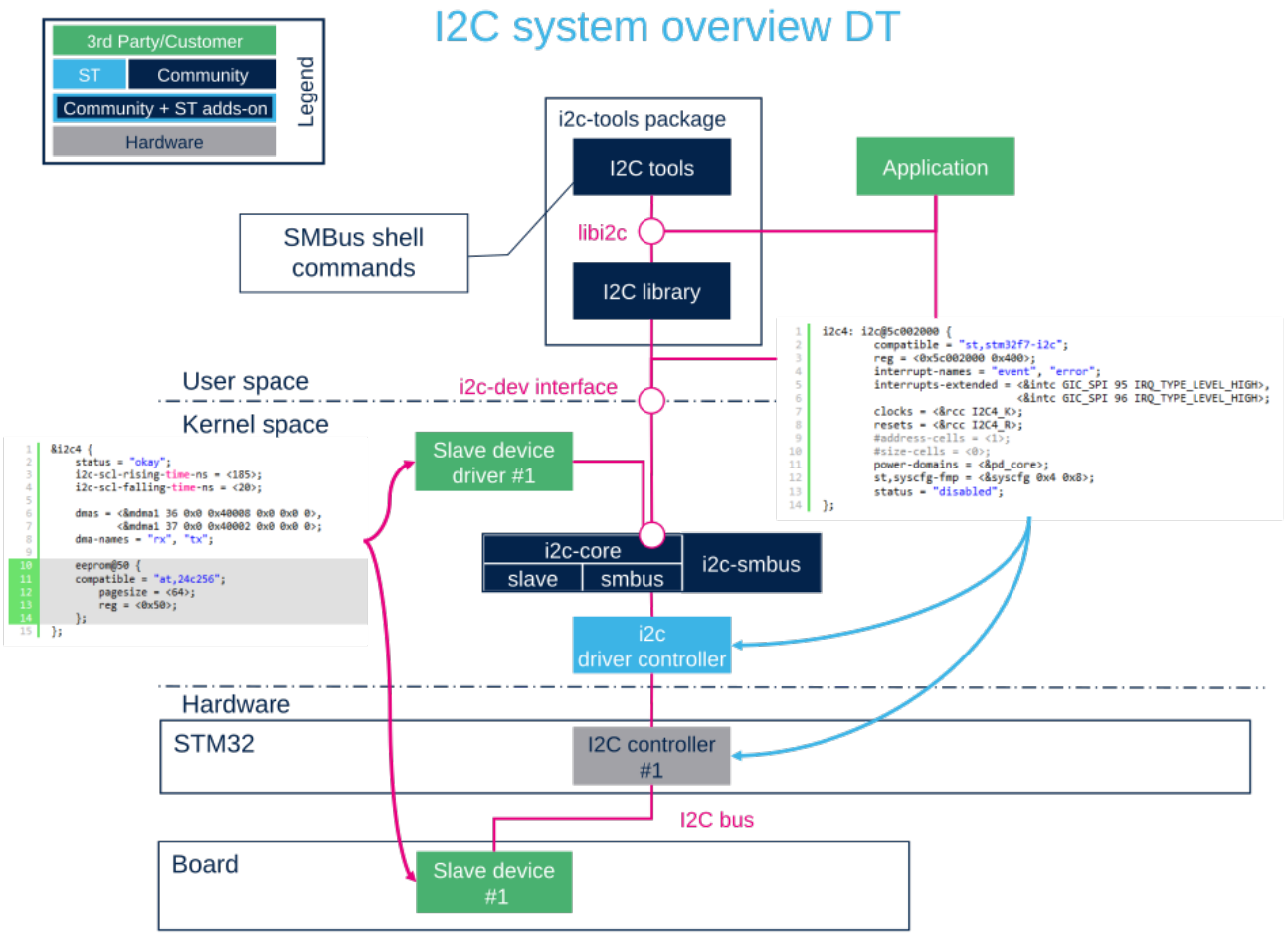
```
 1 &i2c4 {
 2     status = "okay";
 3     i2c-scl-rising-time-ns = <185>;
 4     i2c-scl-falling-time-ns = <20>;
 5
 6     dmas = <&mdma1 36 0x0 0x40008 0x0 0x0 0>,
 7            <&mdma1 37 0x0 0x40002 0x0 0x0 0>;
 8     dma-names = "rx", "tx";
 9
10     eeprom@50 {
```

```
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
```

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