# Bluetooth overview

# Contents



Stable: 15.04.2020 - 08:31 / Revision: 15.04.2020 - 08:29

Stable: 15.04.2020 - 08:31 / Revision: 15.04.2020 - 08:31

A quality version of this page, approved on *15 April 2020*, was based off this revision.

This article explains how a Linux® Bluetooth framework is composed, how to configure it, and how to use it.

Contents

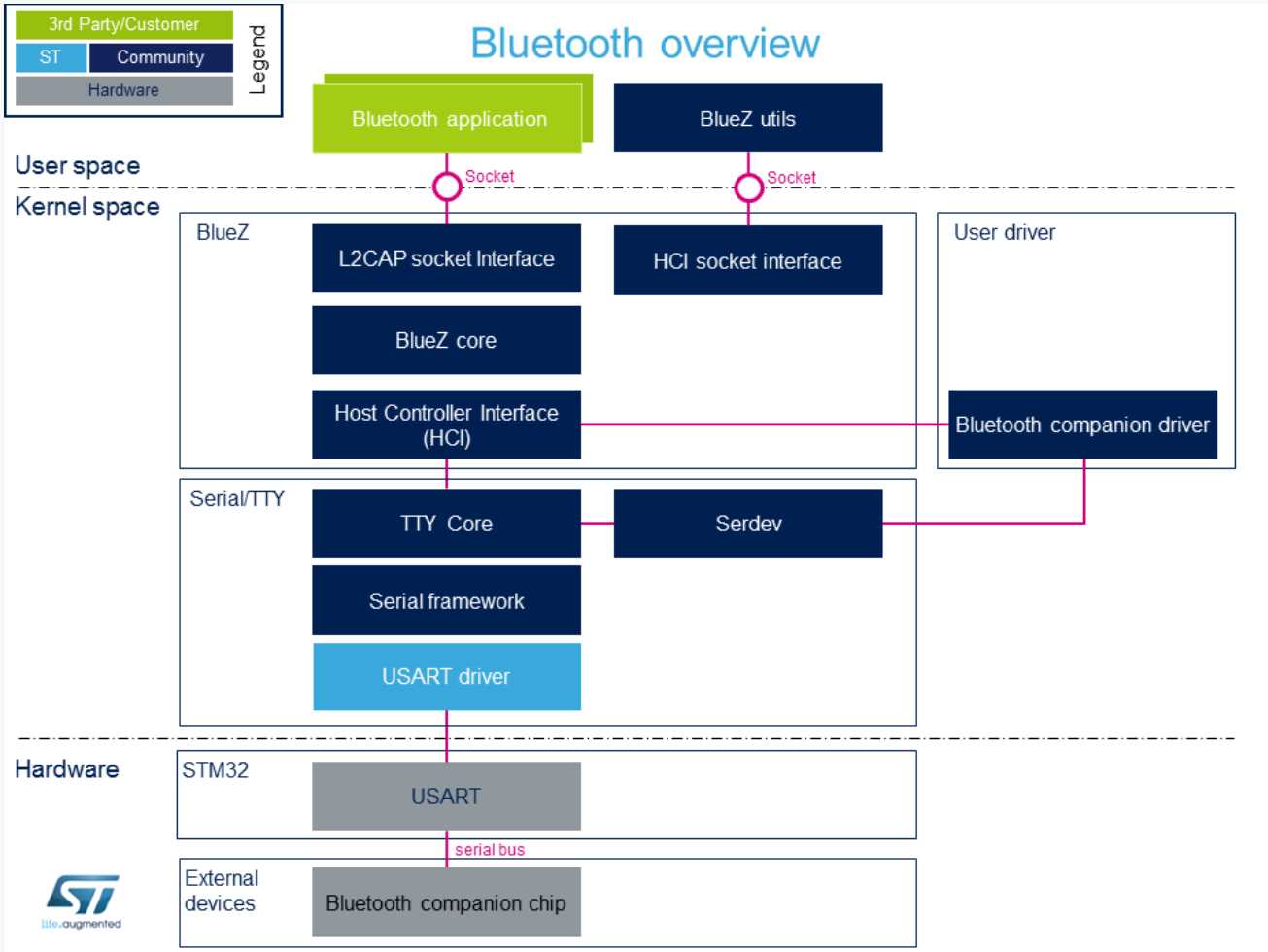
# 1 Framework purpose

**Bluetooth** is a protocol for wireless communication over short distances. It was developed in the 1990s to reduce the need for cable interconnects. Devices such as mobile phones, laptops, PCs, printers, digital cameras and video game consoles can connect to each other and exchange information using radio waves. This can be done securely. Bluetooth is only used for relatively short distances, typically of a few meters.
The Linux kernel has a popular Bluetooth stack: BlueZ. This stack is included in most Linux kernels, and runs in both the user space and kernel space of the Bluetooth protocol.
Bluetooth Low Energy is completely supported at the kernel level in Linux.

Bluetooth can be used in many different use cases, as mentioned in the How to use Bluetooth section:

- how to set up a Bluetooth connection Setup Bluetooth
- how to scan Bluetooth devices Scan Bluetooth devices
- how to scan BLE devices Scan BLE devices

# 2 System overview




Bluetooth Overview

## 2.1 Component descriptions

*From User space to hardware*

- **Bluetooth Applications** (User space)

Lots of applications use bluetooth:

bluetoothd [1]: bluetoothd daemon, which manages all the Bluetooth devices

...

- **BlueZ Utils** (User space)

Development and debugging utilities for the bluetooth protocol stack

There is a set of utilities to manage Bluetooth devices:

bluetoothctl [2]: Pairing a device from the shell is one of the simplest and most reliable options

To see all other utilities: https://www.archlinux.org/packages/extra/x86_64/bluez-utils/

- **BlueZ** (Kernel space)

BlueZ [3] is the official Linux Bluetooth stack. It provides, in a modular way, support for the core Bluetooth layers and protocols. Currently BlueZ consists of many separate modules:
- bluetooth kernel subsystem core
- a "controller stack" containing the timing critical radio interface like HCI [4]
- a "host stack" dealing with high level data like L2CAP [5]

- **Bluetooth companion driver** (Kernel space)

Bluetooth companion driver registers and controls the Bluetooth device

- **Serial/TTY** (Kernel space)

See Serial TTY overview

- **SoC: USART** (Hardware)

See Serial TTY overview

## 2.2 APIs description

The BlueZ API [6] is documented in the Linux Kernel:
BlueZ exposes a socket API that is similar to network socket programming; the is socket created, used to communicate, PF_BLUETOOTH protocol family [7]

# 3 Configuration

## 3.1 Kernel configuration

Bluetooth must be enabled in the kernel configuration, as shown below. On top of this, the user has to activate STM32 support and STM32 USART support. The user can use the Linux Menuconfig tool: Menuconfig or how to configure kernel and select:

```
[*] Networking support  --->
    [*]   Bluetooth subsystem support
        [*]   Bluetooth Classic (BR/EDR)
features
        [*]   Bluetooth High Speed (HS)
features
        [*]   Bluetooth Low Energy (LE) features
        [*]   Bluetooth device drivers   --->
                [*]   HCI UART driver
[*] Device Drivers  --->
    [*]   Character devices  --->
        [*]   Serial device bus
[*] Security options  --->
    [*]   Enable access key retention support
```

For example if the companion chip is the Murata product 1DX[8]

```
[*] Networking support  --->
    [*]   Bluetooth subsystem support  --->
        [*]   Bluetooth device drivers   --->
                [*]   Broadcom protocol support
```

## 3.2 Device tree

DT bindings documentation deals with all of the required and optional device tree properties.

Detailed DT configuration for STM32 internal peripherals: Bluetooth device tree configuration.

# 4 How to use Bluetooth

## 4.1 How to use the Bluetooth user space interface

Please see the examples based on the following use cases:

- how to set up a Bluetooth connection Setup Bluetooth
- how to scan Bluetooth devices Scan Bluetooth devices
- how to scan BLE devices Scan BLE devices

# 5 How to trace and debug the framework

This part is an example based on the Murata companion chip

## 5.1 How to verify than Bluetooth driver is correctly probed

- In dmesg log, check "usart" logs :

```
[    0.485894] STM32 USART driver initialized
[    0.487163] 4000e000.serial: ttySTM1 at MMIO 0x4000e000 (irq = 21, base_baud =
4000000) is a stm32-usart
[    0.487514] stm32-usart 4000e000.serial: interrupt mode used for rx (no dma)
[    0.487531] stm32-usart 4000e000.serial: interrupt mode used for tx (no dma)
```

And, if the companion chip is the Murata 1DX :

```
[   13.755069] Bluetooth: HCI device and connection manager initialized
[   13.800349] Bluetooth: HCI socket layer initialized
[   13.837861] Bluetooth: L2CAP socket layer initialized
[   13.843218] Bluetooth: SCO socket layer initialized
[   14.279668] Bluetooth: HCI UART driver ver 2.3
[   14.282780] Bluetooth: HCI UART protocol H4 registered
[   14.288198] Bluetooth: HCI UART protocol Broadcom registered
[   14.289402] hci_uart_bcm serial0-0: No reset resource, using default baud rate
[   14.465008] Bluetooth: hci0: BCM: chip id 94
[   14.469843] Bluetooth: hci0: BCM: features 0x2e
[   14.497113] Bluetooth: hci0: BCM43430A1
[   14.499593] Bluetooth: hci0: BCM43430A1 (001.002.009) build 0000
```

# 6 References


- [1], bluetoothd
- [2], bluetoothctl
- [3], BlueZ
- [4], HCI
- [5], L2CAP
- [6], BlueZ API
- [7], Socket
- [8], 1DX




Bluetooth Low Energy. Bluetooth LE, marketed as Bluetooth Smart is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group aimed at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries.
Compared to Classic Bluetooth, Bluetooth Smart is intended to provide considerably reduced power consumption and cost while maintaining a similar communication range. (sourcehttps://en.wikipedia.org/wiki/Bluetooth_Low_Energy)

TeleTYpewriter

Universal Synchronous/Asynchronous Receiver/Transmitter

Application programming interface

High Speed (MIPI® Alliance DSI standard)

Universal Asynchronous Receiver/Transmitter

Device Tree

Stable: 11.06.2020 - 09:33 / Revision: 11.06.2020 - 06:41

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

Contents

# 1 Article purpose

This article explains how to configure *Bluetooth* [1] **when the peripheral *(or peripheral associated to the framework)* is assigned to the Linux®OS**.

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

The Bluetooth companion chip chosen on our platform is a Cypress chip [3]

# 2 Bluetooth DT bindings documentation

The *Bluetooth*[4] **tree bindings are composed of:**

- STM32 USART device tree bindings [5]
- The Cypress device, used as child node [6] of the host USART device to which the slave device is attached.

# 3 Bluetooth 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.

## 3.1 Bluetooth DT configuration (STM32 level)

The USART peripheral node is located in *stm32mp151.dtsi*

- This is a set of properties that may not vary for given STM32 device, such as: registers address, clock, reset...

The USART DT configuration is explained in Serial TTY device tree configuration

## 3.2 Bluetooth DT configuration (board level)

```
&usart2 {
        ...
        uart-has-rtscts;                                        /* enable hardware flow
control */
        ...
        bluetooth {                                               /* node of Bluetooth
companion chip */
                shutdown-gpios = <&gpioz 6 GPIO_ACTIVE_HIGH>;  /* GPIO specifier, used to
enable the BT module */
                compatible = "brcm,bcm43438-bt";
                max-speed = <3000000>;
        };
};
```

Specific properties for USART:

- uart-has-rtscts; bool flag to enable hardware flow control

# 4 How to configure Bluetooth using CubeMX

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

- Bluetooth
- Device tree
- MURATA CYW4343W datasheet
- WLAN_and_Bluetooth_hardware_component
- Serial TTY device tree configuration
- Documentation/devicetree/bindings/net/broadcom-bluetooth.txt



Operating System

Device Tree

Universal Synchronous/Asynchronous Receiver/Transmitter

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.)

BlueTooth

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

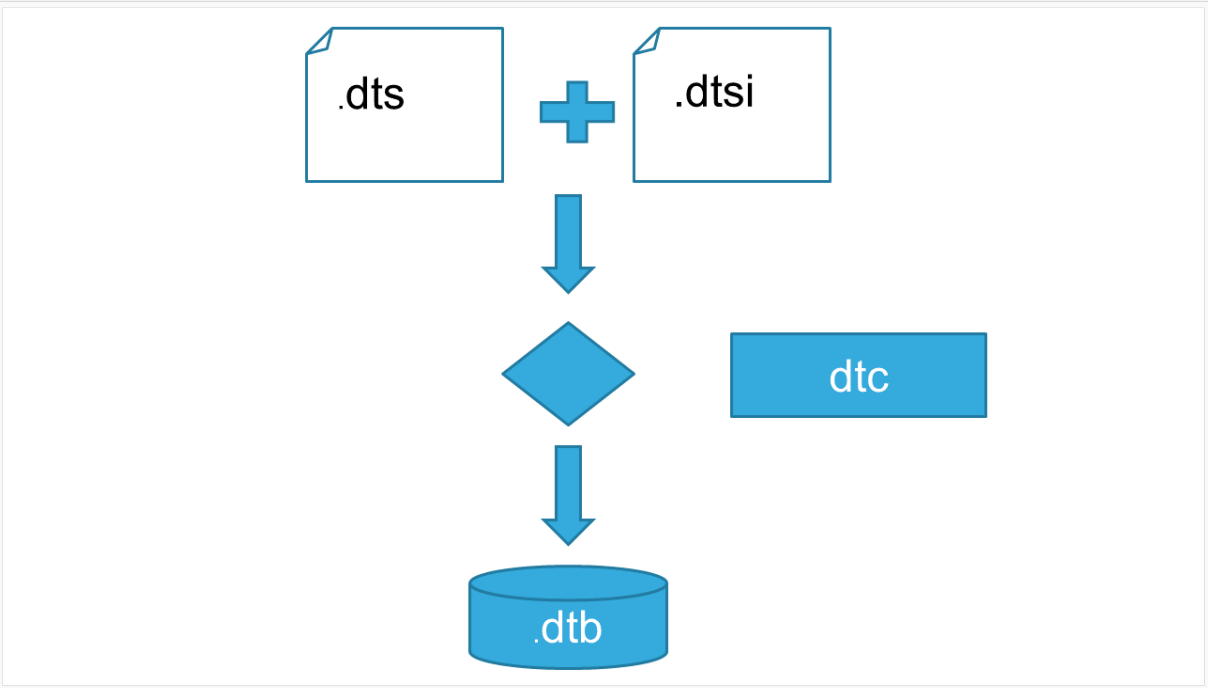
The device tree data structures and properties are named **bindings**. Those bindings are described in:

- 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: 03.02.2020 - 08:42 / Revision: 03.02.2020 - 08:29

A quality version of this page, approved on *3 February 2020*, was based off this revision.

This page lists the different operations needed to scan, connect and display BLE device information. BLE stands for **B**luetooth **L**ow **E**nergy

Contents

# 1 Packages needed

- bluez5

# 2 Configuration

Init file modification for automatic Bluetooth start
/etc/udev/rules.d/10-local.rules:

```
# Set bluetooth power up
ACTION=="add", KERNEL=="hci0", RUN+="/usr/bin/hciconfig hci0 up"
```

# 3 Existing tool selection

Bluez provides some tools, by default, to analyze Bluetooth networks.
**hciconfig** to configure hci connections
**hcitool** to scan, find a device, connect to a device, manage a device list.. deviceS may be normal or low energy
**gatttool** for BLE device management

## 3.1 BLE device connection step-by-step

Command to scan all low-energy Bluetooth hardware:

```
Board $> hciconfig hci0 up
Board $> hcitool lescan
```

To scan available BLE devices:

```
Board $> hcitool lewladd <BLE_MAC_ADDRRESS>
```

To add BLE device in the white list (optional):

```
Board $> hcitool lecc <BLE_MAC_ADDRESS>
```

To connect a BLE device:

Once a BLE device is identified, its characteristics (attibutes) can be discovered, read and modified using GATTTOOL. GATTTOOL offers two working modes: interractive and non-interractive

### 3.1.1 Gatttool Interractive mode

```
Board $> gatttool -b <MAC Address> --interactive
```

In interractive mode, a new prompt is available to perform BLE commands.
**connect**: to connect to a specified device
**primary**: to disable all primary attributes
**char-read-hnd <handle>** to read specified handle/attribute values
**char-write-cmd <handle> <value>** to modify handle values

### 3.1.2 Gatttool Non-interractive mode:

In non-interractive mode, commands are issued one by one. At each command, GATTTOOL performs device connection, action and disconnection.
Few a few examples are given below:

```
Board $> gatttool -b <Mac Address> --primary
Board $> gatttool -b <MAC Address> --characteristics
Board $> gatttool -b <MAC Address> --char-read
Board $> gatttool -b <MAC Address> --char-desc
```

Bluetooth Low Energy. Bluetooth LE, marketed as Bluetooth Smart is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group aimed at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries.
Compared to Classic Bluetooth, Bluetooth Smart is intended to provide considerably reduced power consumption and cost while maintaining a similar communication range. (sourcehttps://en.wikipedia.org/wiki/Bluetooth_Low_Energy)

Stable: 03.02.2020 - 08:42 / Revision: 03.02.2020 - 08:29

A quality version of this page, approved on *3 February 2020*, was based off this revision.

## 1 Scan for available Bluetooth devices

```
Board $> hcitool scan
Scanning ...
       A0:AF:BD:3B:26:61       lmecxl0923
       B0:55:08:40:33:84       HUAWEI P8 lite 2017
```

## 2 Setting the device to Visible

To control the visiblity of our Bluetooth device to other Bluetooth hardware.

- to enable visibility:

```
Board $> hciconfig hciX piscan
```

(hciX corresponds to the available Bluetooth hardware: hci0)

- To disable visibility:

```
Board $> hciconfig hciX noscan
```

(hciX corresponds to the available Bluetooth hardware: hci0)

## 3 How to scan via bluetoothd/systemd

Systemd provides a tool for Bluetooth management: bluetoothctl.
Example session with bluetoothclt for scanning, pairing, connecting:

```
Board $> bluetoothctl
[NEW] Controller 43:43:A1:12:1F:AC stm32mp1 [default]
Agent registered
[bluetooth]# power on
Changing power on succeeded
[CHG] Controller 43:43:A1:12:1F:AC Powered: yes
[bluetooth]# agent on
```

```
Agent is already registered
[bluetooth]# default-agent
Default agent request successful
[bluetooth]# scan on
Discovery started
[CHG] Controller 43:43:A1:12:1F:AC Discovering: yes
[NEW] Device D8:DB:36:D1:39:88 STM
...
[bluetooth]# scan off
[CHG] Controller 43:43:A1:12:1F:AC Discovering: no
Discovery stopped
[bluetooth]#pair D8:DB:36:D1:39:88
Pairing successful
[bluetooth]# connect D8:DB:36:D1:39:88
Connection successful
[STM]# quit
Agent unregistered
[DEL] Controller 43:43:A1:12:1F:AC stm32mp1 [default]
```

GPIO alternate function

System Trace Module

Stable: 03.02.2020 - 08:42 / Revision: 03.02.2020 - 08:29

A quality version of this page, approved on *3 February 2020*, was based off this revision.

# 1 How to setup Bluetooth

## 1.1 Setup

- Check that the driver has been probed correctly in the kernel log message

```
Board $>[    0.923697] STM32 USART driver initialized
Board $>[    0.928711] 4000e000.serial: ttyS1 at MMIO 0x4000e000 (irq = 42, base_baud =
6046875) is a stm32-usart
```

- Check the Bluetooth interface

```
Board $> hciconfig -a
hci0:   Type: Primary  Bus: UART
       BD Address: 43:43:A1:12:1F:AC  ACL MTU: 1021:8  SCO MTU: 64:1
       DOWN
       RX bytes:619 acl:0 sco:0 events:31 errors:0
       TX bytes:410 acl:0 sco:0 commands:31 errors:0
       Features: 0xbf 0xfe 0xcf 0xfe 0xdb 0xff 0x7b 0x87
       Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
       Link policy: RSWITCH SNIFF
       Link mode: SLAVE ACCEPT
```

## 1.2 Initialize the Bluetooth interface

```
Board $> hciconfig hci0 up
Board $> hciconfig -a
hci0:   Type: Primary  Bus: UART
       BD Address: AA:AA:AA:AA:AA:AA  ACL MTU: 1021:8  SCO MTU: 64:1
       UP RUNNING
       RX bytes:1378 acl:0 sco:0 events:74 errors:0
       TX bytes:1186 acl:0 sco:0 commands:74 errors:0
       Features: 0xbf 0xfe 0xcf 0xfe 0xdb 0xff 0x7b 0x87
       Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
       Link policy: RSWITCH SNIFF
       Link mode: SLAVE ACCEPT
       Name: 'BlueZ 5.46'
       Class: 0x000000
       Service Classes: Unspecified
       Device Class: Miscellaneous,
       HCI Version: 4.1 (0x7)  Revision: 0x0
       LMP Version: 4.1 (0x7)  Subversion: 0x2209
       Manufacturer: Broadcom Corporation (15)
```

Universal Synchronous/Asynchronous Receiver/Transmitter

Universal Asynchronous Receiver/Transmitter

Automatic current limit (LCD power improvement solution)

Receive

Transmit

uniprocessor

Stable: 11.02.2021 - 11:10 / Revision: 19.01.2021 - 10:34

A quality version of this page, approved on *11 February 2021*, was based off this revision.

Contents

# 1 Linux configuration genericity

The process of building a kernel has two parts: configuring the kernel options and building the source with those options.

The Linux® kernel configuration is found in the generated file: .config.

.config is the result of configuring task which is processing platform defconfig and fragment files if any.

For OpenSTLinux distribution the defconfig is located into the kernel source code and fragments into stm32mp BSP layer :

- arch/arm/configs/**multi_v7_defconfig**

Every new kernel version brings a bunch of new options, we do not want to back port them into a specific defconfig file each time the kernel releases, so we use the same defconfig file based on ARM SoC v7 architecture.
STM32MP1 specificities are managed with fragments config files.

- meta-st/meta-st-stm32mp/recipes-kernel/linux/linux-stm32mp/<kernel version>/**fragment-*.config**

.config result is located in the build folder:

- build-openstlinuxweston-stm32mp1/tmp-glibc/work/stm32mp1-ostl-linux-gnueabi/linux-stm32mp/4.14-48/linux-stm32mp1-standard-build/**.config**

**To modify the kernel options, it is not recommended to edit this file directly.**

- A user runs either a text-mode :

```
   PC $> make config
starts a character based question and answer session (Figure 1)
```

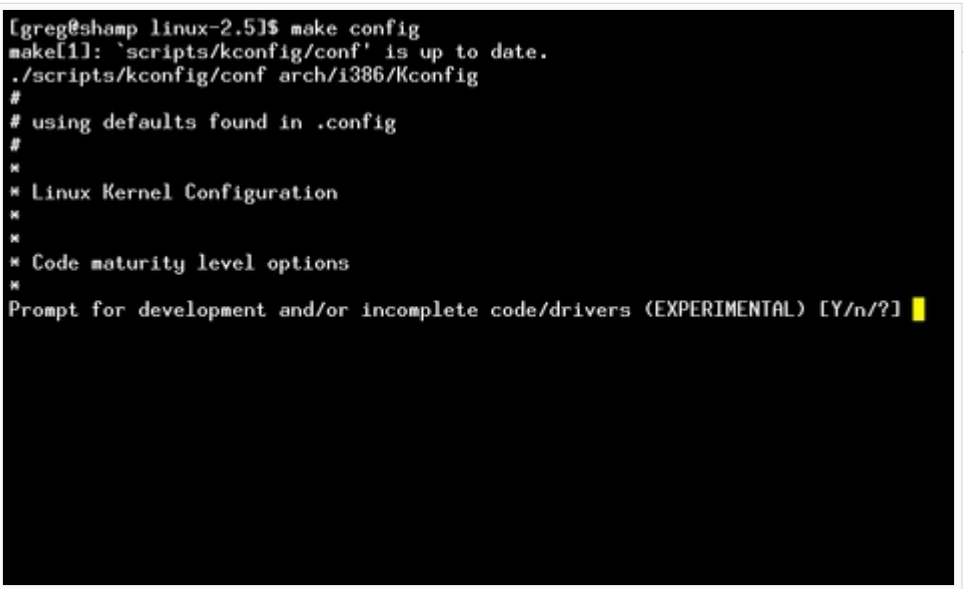



Figure 1. Configuring the kernel with make config

```
   PC $> make
menuconfig
starts a terminal-
oriented
configuration tool
(using ncurses)
(Figure 2)
The ncurses text
version is more
popular and is run
with the make
menuconfig option.
Wikipedia Menuconfig[1
] also explains how
to "navigate" within
the configuration
menu, and highlights
main key strokes.
```

- or a graphical kernel

configurator :

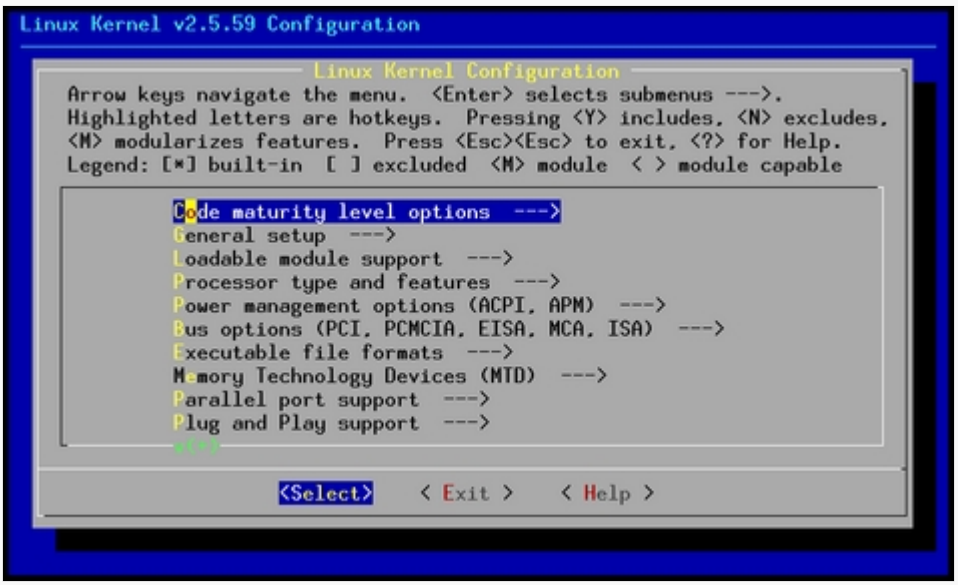



Figure 2. Make menuconfig makes it easier to back up and correct mistakes

**PC $>** make xconfig starts a X based configuration tool (Figure 3)

Ultimately these configuration tools edit the .config file.

An option indicates either some driver is built into the kernel ("=y") or will be built as a module ("=m") or is not selected.

The unselected state can either be indicated by a line starting with "#" (e.g. "# CONFIG_SCSI is not set") or by the absence of the relevant line from the .config file.

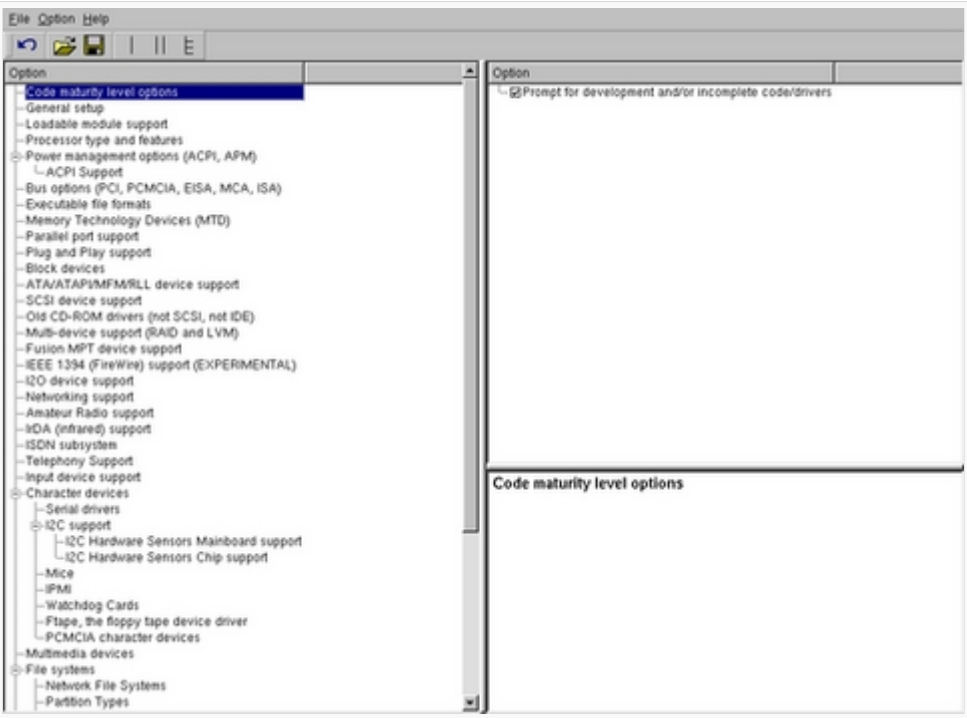



Figure 3. The Qt-Based make xconfig

The 3 states of the main selection option for the SCSI subsystem (which actually selects the SCSI mid level driver) follow. Only one of these should appear in an actual .config file:

```
CONFIG_SCSI=y
CONFIG_SCSI=m
# CONFIG_SCSI is not
set
```

# 2 Menuconfig and Developer Package

For this use case, the prerequesite is that OpenSTLinux SDK has been installed and configured.

To verify if your cross-compilation environment has been put in place correctly, run the following command:

```
PC $> set | grep CROSS
CROSS_COMPILE=arm-ostl-linux-gnueabi-
```

For more details, refer to *<Linux kernel installation directory>/README.HOW_TO.txt* helper file (the latest version of this helper file is also available in GitHub: README.HOW_TO.txt ).

- Go to the <Linux kernel build directory>

```
PC $> cd <Linux kernel build directory>
```

- Save initial configuration (to identify later configuration updates)

```
PC $> make arch=ARM savedefconfig
Result is stored in defconfig file
PC $> cp defconfig defconfig.old
```

- Start the Linux kernel configuration menu

```
PC $> make arch=ARM menuconfig
```

- Navigate forwards or backwards directly between feature
  - un/select, modify feature(s) you want
  - When the configuration is OK : exit and save the new configuration

```
useful keys to know:
enter: enter in config subdirectory
space: hit several times to either select [*], select in module [m] or unselect [ ]
/: to search for a keyword, this is usefull to navigate in tree
?: to have more information on selected line
```

- Compare the old and new config files after operating modifications with menuconfig

```
PC $> make arch=ARM savedefconfig
```

Retrieve configuration updates by comparing the new defconfig and the old one

```
PC $> meld defconfig defconfig.old
```

- Cross-compile the Linux kernel (please check the load address in the *README.HOW_TO.txt* helper file)

```
PC $> make arch=ARM uImage LOADADDR=<loadaddr of kernel>
PC $> cp arch/arm/boot/uImage install_artifact/boot/
```

- Update the Linux kernel image on board

```
PC $> scp install_artifact/boot/uImage root@<board ip address>:/boot/
```

**Information**

If the */boot* mounting point doesn't exist yet, please see how to create a mounting point

- Reboot the board

```
Board $> cd /boot; sync; systemctl reboot
```

Note that this use case modifies the configuration file in the Linux kernel build directory, not in the Linux kernel source directory: this is a temporary modification useful for a prototyping.

- To make this temporary modification permanent, the delta between defconfig and defconfig.old must be saved in a configuration fragment file (fragment-*.config) based on fragment.cfg file, and the Linux kernel configuration/compilation steps must be re-executed (as explained in the README.HOW_TO.txt helper file).

# 3 Menuconfig and Distribution Package

- Start the Linux kernel configuration menu

```
PC $> bitbake virtual/kernel -c menuconfig
```

- Navigate forwards or backwards directly between feature
  - un/select, modify feature(s) you want
  - When the configuration is OK : exit and save the new configuration

```
useful keys to know:
enter: enter in config subdirectory
space: hit several times to either select [*], select in module [m] or unselect [ ]
/: to search for a keyword, this is usefull to navigate in tree
?: to have more information on selected line
```

- Cross-compile the Linux kernel

```
PC $> bitbake virtual/kernel
```

- Update the Linux kernel image on board

```
PC $> scp <build dir>/tmp-glibc/deploy/images/<machine name>/uImage root@<board ip
address>:/boot
```

**Information**

If the */boot* mounting point does not exist yet, please see how to create a mounting point

- Reboot the board

```
Board $> cd /boot; sync; systemctl reboot
```

Note that this use case modifies the configuration file in the Linux kernel build directory, not in the Linux kernel source directory: this is a temporary modification useful for a prototyping.

- To make this temporary modification permanent, it must be saved in a configuration fragment file (fragment-*.config) based on **fragment.cfg** file, and the Linux kernel configuration/compilation steps must be re-executed: **bitbake <name of kernel recipe>**.

# 4 References

- Wikipedia Menuconfig

Linux® is a registered trademark of Linus Torvalds.

Board support package

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

Stable: 17.03.2020 - 14:44 / Revision: 25.02.2020 - 14:49

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

This article gives information about the Linux®TTY framework. It explains how to activate the **UART** interface, and how to access it from user and kernel spaces.

Contents

# 1 Framework purpose

The **TTY** subsytem controls the communication between UART devices and the programs using these devices.
The TTY subsystem is responsible for:

- controlling the physical flow of data on asynchronous lines (including the transmission speed, character size, and line availability).
- interpreting the data by recognizing special characters and adapting to national languages.
- controlling jobs and terminal access by using the concept of controlling terminal.

**The synchronous mode of the STM32 USART peripheral is not supported by the TTY subsystem**.
A controlling terminal manages the input and output operations of a group of processes. The TTY special file (ttyX filesystem entry) supports the controlling terminal interface.
To perform its tasks, the TTY subsystem is composed of modules, also called disciplines. A module is a set of processing rules that govern the interface for communication between the computer and an asynchronous device. Modules can be added and removed dynamically for each TTY.

The TTY subsystem supports three main types of modules:

- TTY drivers: TTY drivers, or hardware disciplines, directly control the hardware (TTY devices) or pseudo-hardware (PTY devices). They perform the actual input and output to the adapter by providing services to the modules above it. The services are flow control and special semantics when a port is being opened.
- Line disciplines: the line disciplines provide editing, job control, and special character interpretation. They perform all the transformations that occur on the inbound and outbound data streams. The line disciplines also perform most of the error handling and status monitoring for the TTY drivers.
- Converter modules: the converter modules, or mapping disciplines, translate, or map, input and output characters.

Since kernel 4.12 version, the serial device bus (also called Serdev) has been introduced in the TTY framework to improve the interface offered to devices attached to a serial port (ex: Bluetooth, NFC, FM Radio and GPS devices), as the line disciplines "drivers" have some known limitations:

- the devices are encoded in the user space rather than in the firmware (Device Tree of ACPI)
- "drivers" are not kernel drivers but user space daemons
- the associated ressources (GPIOs and interrupts, regulators, clocks, audio interface) are not described in the kernel space, which impacts power management
- "drivers" are registered when a port is opened

The Serdev allows a device to be attached on UART without known the line disciplines limitations:

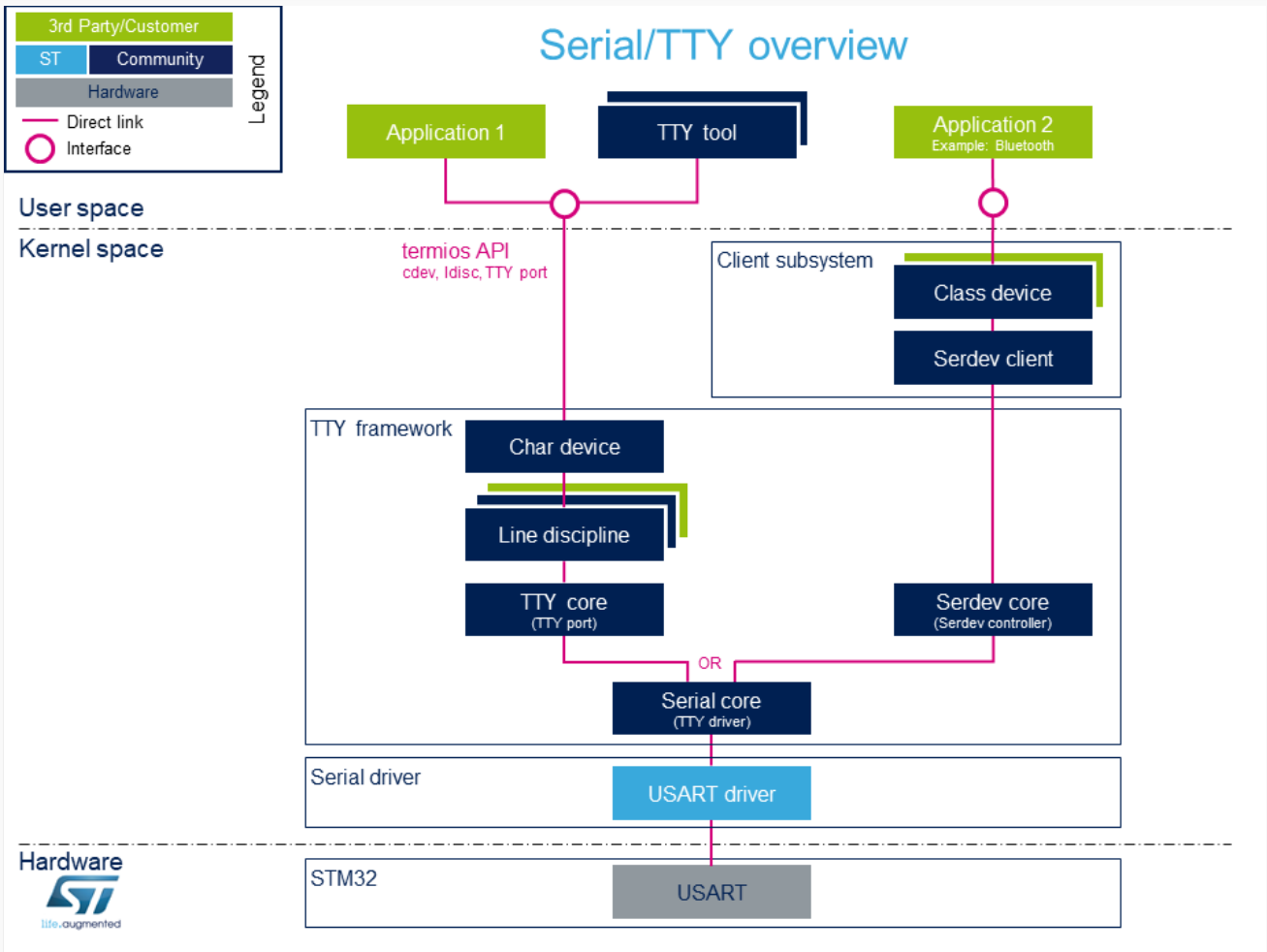
- New bus type: serial
- Serdev controllers
- Serdev devices (clients or slaves)
- Serdev TTY-port controller
  - Only in-kernel controller implementation
  - Registered by TTY driver when client is defined
  - clients are described by firmware (Device Tree or ACPI)

The USART low-level driver provided by STMicroelectronics, (**drivers/tty/serial/stm32-usart.c**) supports RS-232 standard (for serial communication transmission of data), and RS-485 standard (for modbus protocol applications as example).
The Synchronous mode of USART is not supported by Linux® low-level driver .

The TTY framework is used to access the serial device in the following use cases:

- **tty virtual console** during Linux boot sequence
- **pts pseudo-terminal** to access over a terminal
- **user space application**

# 2 System overview




Serial TTY architecture overview

Note: during boot, while a serial device is probed, the serial framework instantiates an associated tty terminal as a virtual device. Then the system sees this tty virtual device as a child of the associated serial device.

## 2.1 Components description

*From client application to hardware*

- Application: customer application to read/write data from the peripheral connected on the serial port.
- TTY tools: tools provided by Linux community, such as **stty**, **ldattach**, **inputattach**, **tty**, **ttys**, **agetty**, **mingetty**, **kermit** and **minicom**.
- Termios: API which offers an interface to develop an application using serial drivers.
- Client subsystem: kernel subsystem client of **serdev** core (Example: Bluetooth_overview ).
- TTY framework: high-level TTY structures management, including **tty character device driver** , **TTY core functions** , **line discipline** and **Serdev core functions** management.
- Serial framework: low-level serial driver management, including the **serial core** functions .
- USART driver: stm32-usart **low-level serial driver** for all STM32 family devices.
- STM32 USART: **STM32 frontend IP** connected to the external devices through a serial port.

## 2.2 APIs description

The TTY provides only **character device interface** (named /dev/ttyX) to the user space. The main API for user space TTY client applications is provided by the portable POSIX terminal interface termios, which relies on /dev/ttyX interface for TTY link configuration.

The **termios API** [1] is a user land API, and its functions describe a general terminal interface that is provided to control asynchronous communications ports.

The POSIX termios API abstracts the low-level details of the hardware, and provides a simple, yet complete, programming interface that can be used for advanced projects. It is a wrapper on **character device API** [2] ioctl operations.

Note: If a serial interface is needed at kernel level (to control an external device through U(S)ART by a kernel driver for example), the customer can use a **line discipline** or a **Serdev** client.

- The **line discipline** will be responsible for:
  - creating this new kernel API
  - routing data flow between the serial core and the new kernel API
- The **Serdev** provides an interface to kernel drivers.
  - This interface resembles line-discipline operations: open and close, terminal settings, write, modem control, read (callback), and write wakeup (callback)

# 3 Configuration

This section describes how to configure a device on a serial port.

## 3.1 Kernel Configuration

The serial driver, serial framework, and TTY framework are activated by default in ST deliveries. Nevertheless, if a specific configuration is needed, this section indicates how IIO can be activated/deactivated in the kernel.

Activate the device TTY in kernel configuration with Linux Menuconfig tool.

For TTY, select:

```
Device Drivers  --->
   Character devices  --->
     [*]   Enable TTY

Allows to remove the TTY support which can save space, and blocks features that require TT
Y from inclusion in the kernel.
The TTY is required for any text terminals or serial port communication. Most users
should leave this enabled.
```

For the STM32 serial driver, select:

```
Device Drivers  --->
   Character devices  --->
      Serial drivers  --->
     <*> STMicroelectronics STM32 serial port support
     [*]   Support for console on STM32

This driver is for the on-chip serial controller on STMicroelectronics STM32 MCUs.
The USART supports Rx and Tx functionality. It supports all industry standard baud rates.
```

## 3.2 Device tree configuration

The UART configuration thanks to the device tree is described in the dedicated article Serial TTY device tree configuration.

# 4 How to use TTY

This section describes how to use TTY from the user land (from a terminal or an application) and from the kernel space, based on the two following use cases:

- How to configure the serial port by using the termios structure
- How to send/receive data

The termios structure allows to configure communication ports with many settings, such as :

- Baud rate
- Character size mask
- Parity bit enabling
- Parity and framing errors detection settings
- Start/stop input and output control
- RTS/CTS (hardware) flow control
- ...

As the USART internal peripheral supports 7, 8 and 9 word length data, the following termios character size and parity bit configurations are supported:

- CS6 with parity bit
- CS7 with or without parity bit
- CS8 with or without parity bit

Tips to use TTY:

- **How to use TTY from user terminal**: TTY usage from a user terminal is described in a dedicated article, How to use TTY from a user terminal
- **How to use TTY from an application**: TTY usage from an application is described in a dedicated article, How to use TTY from an application
- **TTY line discipline**: TTY line discipline is described in a dedicated article, Serial TTY line discipline

# 5 How to trace and debug the framework

## 5.1 How to monitor

As Debugfs does not propose any information about serial or TTY frameworks, the way to monitor Serial and TTY frameworks is to use the linux kernel log method (based on printk) described in Dmesg_and_Linux_kernel_log article.

## 5.2 How to trace

### 5.2.1 Kernel boot log

The following extract of **kernel boot log** shows a serial driver properly probed:

```
[    0.793340] STM32 USART driver initialized
[    0.798779] 4000f000.serial: ttySTM1 at MMIO 0x4000f000 (irq = 25, base_baud =
4000000) is a stm32-usart
[    0.808875] stm32-usart 4000f000.serial: interrupt mode used for rx (no dma)
[    0.816106] stm32-usart 4000f000.serial: interrupt mode used for tx (no dma)
[    0.824253] 40010000.serial: ttySTM0 at MMIO 0x40010000 (irq = 27, base_baud =
4000000) is a stm32-usart
[    0.833796] console [ttySTM0] enabled
[    0.833796] console [ttySTM0] enabled
[    0.840862] bootconsole [earlycon0] disabled
[    0.840862] bootconsole [earlycon0] disabled
[    0.850132] stm32-usart 40010000.serial: interrupt mode used for rx (no dma)
[    0.855755] stm32-usart 40010000.serial: interrupt mode used for tx (no dma)
```

### 5.2.2 dmesg output information

The system log shows the UART devices and associated TTY terminals registered during the probe.

```
Board $> dmesg | grep ttySTM*
[    0.000000] Kernel command line: root=/dev/mmcblk0p5 rootwait rw earlyprintk
console=ttySTM1,115200
# ttySTM1 terminal is associated with usart3 (4000f000.serial) #
[    0.798779] 4000f000.serial: ttySTM1 at MMIO 0x4000f000 (irq = 25, base_baud =
4000000) is a stm32-usart
# ttySTM0 terminal is associated with uart4 (40010000.serial) for console#
[    0.824253] 40010000.serial: ttySTM0 at MMIO 0x40010000 (irq = 27, base_baud =
4000000) is a stm32-usart
# ttySTM0 terminal is activated by default for console #
[    0.833796] console [ttySTM0] enabled
```

### 5.2.3 Dynamic trace

A detailed dynamic trace is available in How to use the kernel dynamic debug

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

This command enables all the traces related to the TTY core and drivers at runtime.
A finer selection can be made by choosing only the files to trace.

**Information**

Reminder: *loglevel* needs to be increased to 8 by using either boot arguments or the *dmesg -n 8* command through the console

## 5.3 How to debug

While a TTY serial port is instantiated, the TTY core exports different files through devfs, sysfs and procfs.

### 5.3.1 devfs

- The repository **/dev** contains all the probed TTY serial devices.

```
Board $> ls /dev/ttySTM*
# ttySTM1 and ttySTM0 terminals are probed #
/dev/ttySTM1  /dev/ttySTM0
```

### 5.3.2 sysfs

- **/sys/class/tty/** lists all TTY devices which ttySx correspond to serial port devices.

```
Board $> ls /sys/class/tty/*/device/driver
/sys/class/tty/ttySTM1/device/driver -> ../../../../bus/platform/drivers/stm32-usart
/sys/class/tty/ttySTM0/device/driver -> ../../../../bus/platform/drivers/stm32-usart
```

- **/sys/devices/platform/soc/** lists all the usart devices probed

```
Board $> ls -d /sys/devices/platform/soc/*.serial
# Serial devices 4000f000.serial (usart3) and 40010000.serial (uart4) are probed #
/sys/devices/platform/soc/4000f000.serial  /sys/devices/platform/soc/40010000.serial
```

- **/sys/devices/platform/soc/*device*.serial/tty** lists the TTY terminal associated to a serial device

```
Board $> ls /sys/devices/platform/soc/4000f000.serial/tty/
# ttySTM1 is associated to serial device 4000f000.serial (usart3)  #
ttySTM1
```

### 5.3.3 procfs

- The repository **/proc/device-tree** lists all the usart devices declared in the device-tree, including the disabled ones.

```
Board $> ls -d /proc/device-tree/soc/serial@*
/proc/device-tree/soc/serial@4000e000  /proc/device-tree/soc/serial@40010000  /proc
/device-
tree/soc/serial@40018000  /proc/device-tree/soc/serial@44003000
/proc/device-tree/soc/serial@4000f000  /proc/device-tree/soc/serial@40011000  /proc
/device-
tree/soc/serial@40019000  /proc/device-tree/soc/serial@5c000000
```

Then for each device listed, device-tree properties are available.

```
Board $> ls /proc/device-tree/soc/serial@40010000/
clock-names  compatible       interrupts-extended  name     pinctrl-0  pinctrl-names
reg     wakeup-source
clocks       interrupt-names  linux,phandle        phandle  pinctrl-1  power-domains
status
```

As an example, the status entry provides the status of the device in the device tree node.

```
Board $> cat /proc/device-tree/soc/serial@40010000/status
# status of device serial@40010000 (uart4) is set to "okay" in the device tree #
okay
```

- The file **/proc/interrupts** lists the interrupts for active serial ports.

```
Board $> cat /proc/interrupts | grep serial
26:          0          0     GIC-0  71 Level     4000f000.serial
27:          0          0  stm32-exti-h  28 Edge      4000f000.serial
28:      13509          0     GIC-0  84 Level     40010000.serial
29:          0          0  stm32-exti-h  30 Edge      40010000.serial
```

- The file **/proc/tty/driver/stm32-usart** lists serial core counters and modem information for each serial instance.

  Driver information:
  - serial driver name
  - serial device start address
  - irq number

  Counters:
  - tx: Number of bytes sent
  - rx: Number of bytes received
  - fe: Number of framing errors received
  - pe: Number of parity errors received
  - brk: Number of break signals received
  - oe: Number of overrun errors received
  - bo: Number of framework buffer overrun errors received

  Modem information:
  - RTS: Request To Send
  - CTS: Clear To Send
  - DTR: Data Terminal Ready
  - DSR: Data Set Ready
  - CD: Carrier Detect
  - RI: Ring Indicator

```
Board $> cat /proc/tty/driver/stm32-usart
serinfo:1.0 driver revision:
0: uart:stm32-usart mmio:0x40010000 irq:29 tx:22722 rx:2276 RTS|CTS|DTR|DSR|CD
1: uart:stm32-usart mmio:0x4000F000 irq:27 tx:0 rx:1149 fe:121 oe:2 pe:296 brk:3 RTS|CTS|D
TR|DSR|CD
3: uart:stm32-usart mmio:0x4000E000 irq:25 tx:0 rx:0 CTS|DSR|CD
```

# 6 How to go further

The Linux community provides many detailed documentation about Linux serial/TTY. Please find below a selection of the most relevant ones:

- Linux Serial-HOWTO [3] describes how to set up serial ports from both hardware and software perspectives.
- Serial Programming Guide for POSIX Compliant Operating Systems [4] , by Michael Sweet.

More information can be found in the following web articles in order to get a good understanding of the Linux TTY framework:

- TTY Subsystem [5], by IBM
- The TTY demystified [6], by Linus Akesson
- Serial drivers training [7], by Bootlin
- Linux Serial drivers [8], by Alessandro Rubini
- Serial Device Bus [9], by Johan Hovold

# 7 References


- termios API, Linux Programmer's Manual termios API Documentation (user land API with serial devices)
- Character device API overview, *Accessing hardware from userspace* training, Bootlin documentation
- Linux Serial-HOWTO, tdlp.org training document, describes how to set up serial ports from both hardware and software perspectives
- Serial Programming Guide for POSIX Compliant Operating Systems, by Michael Sweet, training document
- TTY Subsystem, by IBM
- The TTY demystified TTY subsystem presentation article, by Linus Akesson
- Linux serial drivers training Linux Serial Drivers training, by Bootlin
- Linux Serial Drivers Serial drivers article describing data flows, by Alessandro Rubini
- The Serial Device Bus Serdev framework presentation, by Johan Hovold




TeleTYpewriter

Universal Asynchronous Receiver/Transmitter

Universal Synchronous/Asynchronous Receiver/Transmitter

Serial device bus

Near Field Communication ( is a short-range wireless standard for communication over distances up to around 10cm that will enable enhanced services for users of NFC-enabled smart phones. These could include receiving coupons from retailers upon entering a store, or sharing contacts or photos, in addition to making mobile payments and collecting data from medical monitors, smart meters or other equipment containing ST’s dual-interface EEPROM. We produce a wireless memory that can transmit and receive information from the heart of an application to a smart phone containing NFC technology. NFC is expected to become a widely used system for making payments by smart phone in the United States and it is estimated that by 2015, 30.5% (iSupply) of all handsets shipped will contain NFC technology.)

Application programming interface

Portable Operating System Interface based on uniX (https://en.wikipedia.org/wiki/POSIX_terminal_interface for more details)

terminal input output structure

Android Runtime (see https://source.android.com/devices/tech/dalvik)

Industrial I/O Linux subsystem

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

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)

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

Generic Interrupt Controller