



Serial TTY overview



A quality version of this page, accepted 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	3
2 System overview	5
2.1 Components description	6
2.2 APIs description	6
3 Configuration	7
3.1 Kernel Configuration	7
3.2 Device tree configuration	7
4 How to use TTY	8
5 How to trace and debug the framework	9
5.1 How to monitor	9
5.2 How to trace	9
5.2.1 Kernel boot log	9
5.2.2 dmesg output information	9
5.2.3 Dynamic trace	9
5.3 How to debug	10
5.3.1 devfs	10
5.3.2 sysfs	10
5.3.3 procfs	10
6 How to go further	13
7 References	14



1 Framework purpose

The TTY subsystem 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 or ACPI)
- "drivers" are not kernel drivers but user space daemons
- the associated resources (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).

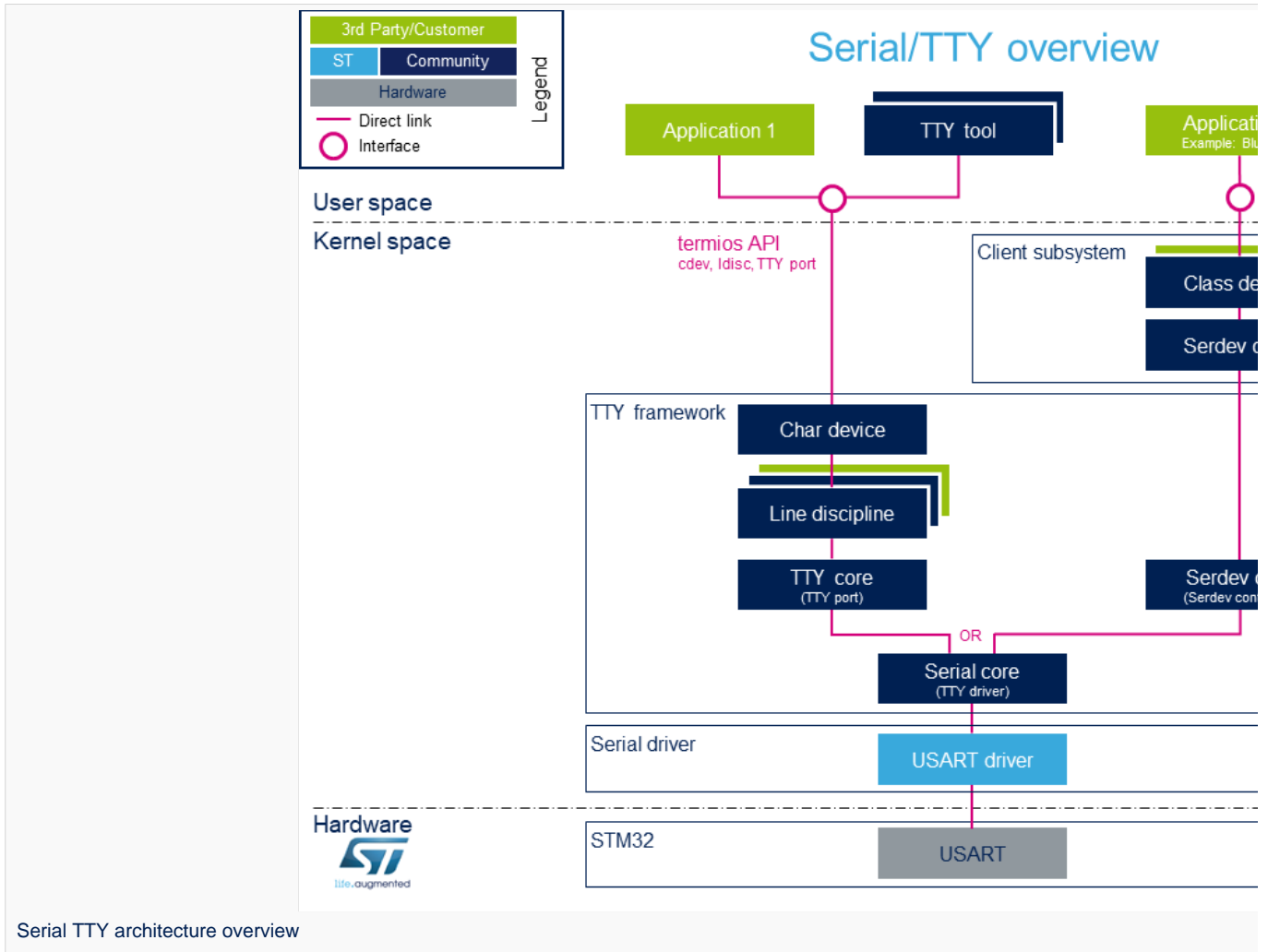
The Synchronous mode of USART is not supported by Linux[®] low-level driver .

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



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 TTY 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.



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

Linux[®] is a registered trademark of Linus Torvalds.

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