Serial TTY overview

Contents

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

Stable: 17.03.2020 - 14:44 / Revision: 23.02.2020 - 14:46

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:

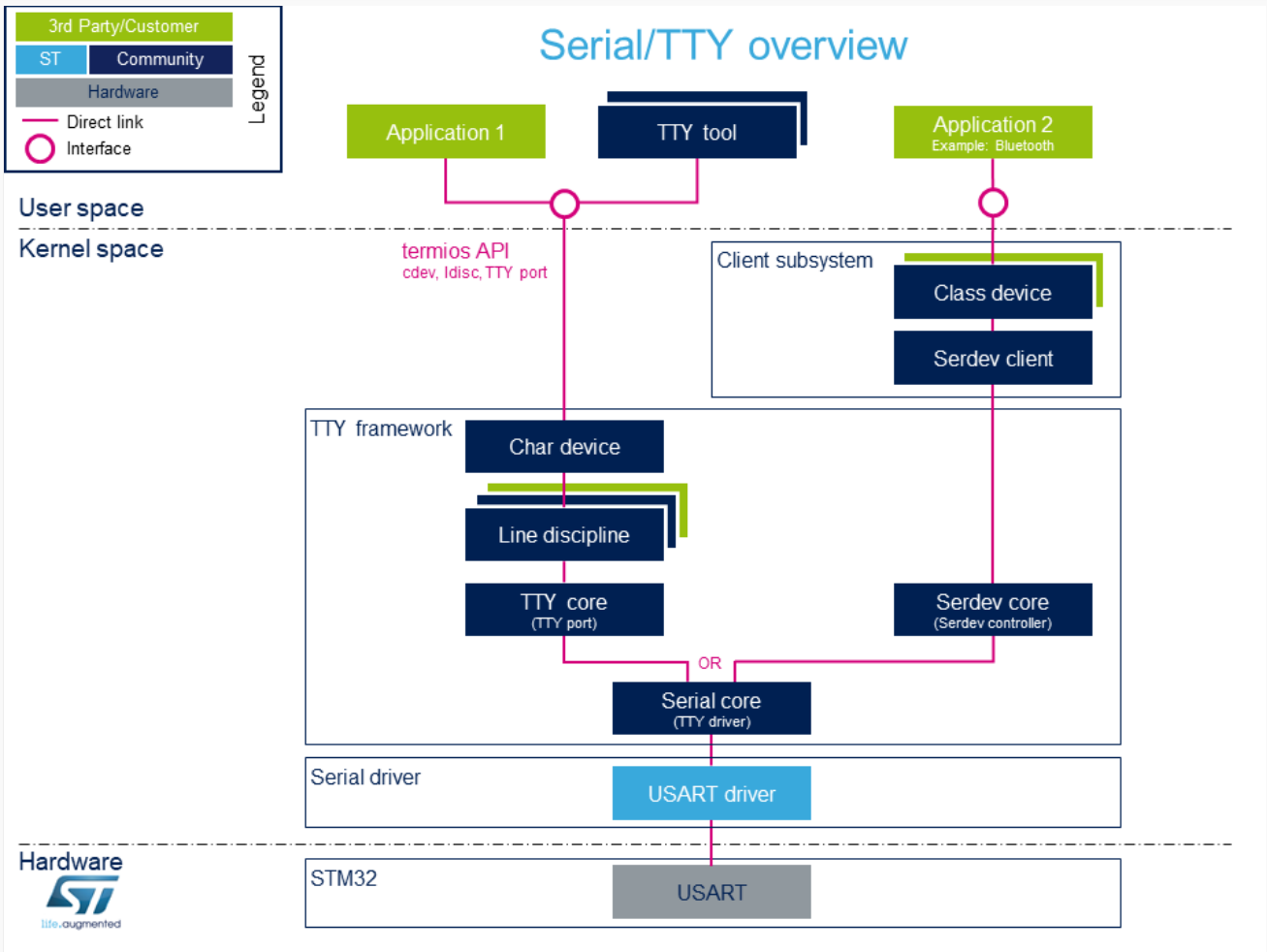
- New bus type: serial
- Serdev controllers
- Serdev devices (clients or slaves)
- Serdev TTY-port controller
  - Only in-kernel controller implementation
  - Registered by TTY driver when client is defined
  - clients are described by firmware (Device Tree or ACPI)

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

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

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

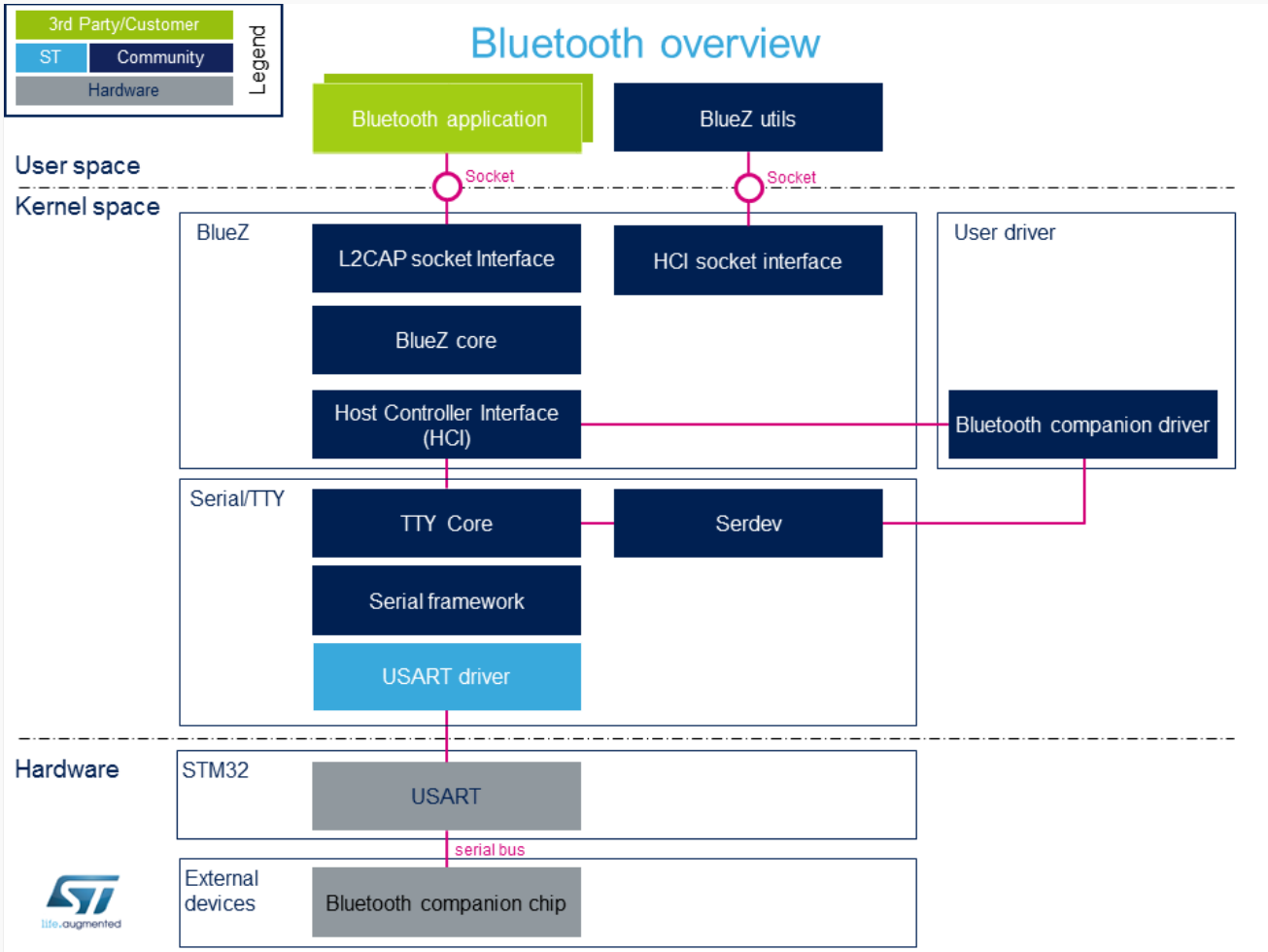
1 Framework purpose

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

Linux® is a registered trademark of Linus Torvalds.

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: 11.03.2021 - 15:14 / Revision: 11.03.2021 - 15:13

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

Contents

1 Article purpose

The purpose of this article is to provide information about the Linux® kernel log including configuration, and to detail usage of dmesg command.

2 Introduction

Linux kernel is able to print log and trace messages, which are by default stored in a ring buffer.

The same messages can also be displayed, applying filter, on uart/console using serial port. This is defined in the kernel command-line, with the "console" parameter. See [1] for detail.

dmesg is a shell command on the kernel console, which also displays the content of the ring buffer, with filter or not (default).

3 printk function

The simplest way to get some debug information from the kernel code is by printing out various information with the kernel's equivalent of printf - the printk function and its derivatives.

```
printk("My Debugger is Printk\n");
```

See elinux.org[2] for reference. This information will be sent to the console, and also stored in a ring buffer.

You can also check to the printk-format.txt[3] document provided in the Linux kernel package to get detail about syntax and formatting.

4 Linux kernel ring buffer

The Linux kernel also manages a ring buffer to store log and trace messages.

The size of the buffer cannot be modified in runtime, and its default size value is 2^CONFIG_LOG_BUF_SHIFT bytes.

To change it, there are 3 possibles ways:

- Modify CONFIG_LOG_BUF_SHIFT value in defconfig file or use the config fragment file:

```
In example for 64K : CONFIG_LOG_BUF_SHIFT=16
```

- or use the Linux kernel menuconfig update

```
 Location:
   -> General setup
     -> Kernel log buffer size (16 => 64KB, 17 => 128KB)
```

- Or modify kernel arguments[4] in kernel command-line (via bootargs value in device tree, or directly in extlinux uboot config file)

```
bootargs = "root=/dev/mmcblk0p5 rootwait rw console=ttySTM0,115200 log_buf_len=65536";
```

This ring buffer can be displayed using *dmesg* command (see dmesg).

5 Loglevels

As reference, please see elinux.org[5].

The log level is used by the kernel to determine the importance of a message and to decide whether it should be presented to the user immediately, by printing it to the current console.

For this, the kernel compares the log level of the message to the console_loglevel (a kernel variable) and if the priority is higher (i.e. a lower value) than the console_loglevel, the message will be printed to the current console. As example, if console_loglevel=5, all messages with log level 0 to 4 will be displayed.

Please note that all messages with loglevel lower or equal to KERN_INFO level are stored in the ring buffer.

5.1 loglevels values

Name	String	Meaning	alias functions	dev alias function
KERN_EMERG	"0"	Emergency messages, system is about to crash or is unstable	pr_emerg	dev_emerg
KERN_ALERT	"1"	Something bad happened and action must be taken immediately	pr_alert	dev_alert
KERN_CRIT	"2"	A critical condition occurred like a serious hardware/software failure	pr_crit	dev_crit
KERN_ERR	"3"	An error condition, often used by drivers to indicate difficulties with the hardware	pr_err	dev_err
KERN_WARNING	"4"	A warning, meaning nothing serious by itself but might indicate problems	pr_warning	dev_warn
KERN_NOTICE	"5"	Nothing serious, but notably nevertheless. Often used to report security events	pr_notice	dev_notice
KERN_INFO	"6"	Informational message e.g. startup information at driver initialization	pr_info	dev_info
KERN_DEBUG	"7"	Debug messages	pr_debug, pr_devel if DEBUG is defined	dev_dbg

"Loglevels table"

Important: please note that Higher priority message is loglevel 0

5.2 Set loglevel filter value for console

5.2.1 Default values

To determine your current console_loglevel on the target you can verifiy with the following command:

```
Board $> cat /proc/sys/kernel/printk
7       4           1       7
current default_msg minimum default_console
```

The first integer shows you the current console_loglevel; the second the default log level, see Use loglevel in the kernel source for log and trace.

This is defined at compilation:

- Current console loglevel via CONFIG_CONSOLE_LOGLEVEL_DEFAULT=7 (defined in file *lib/Kconfig.debug*)
- Default message loglevel via CONFIG_MESSAGE_LOGLEVEL_DEFAULT=4 (defined in file *lib/Kconfig.debug*)
- Minimum console loglevel via #define CONSOLE_LOGLEVEL_MIN 1 (defined in file *include/linux/printk.h*)
- **Default console** loglevel is **equal to CONFIG_CONSOLE_LOGLEVEL_DEFAULT**

5.2.2 Using kernel command-line

The console loglevel can be also set via a kernel command-line parameter if you want to use a different value than one specify by CONFIG_CONSOLE_LOGLEVEL_DEFAULT.
For example:

```
root=/dev/mmcblk0p5 rootwait rw console=ttySTM0,115200 loglevel=4
```

In that case only messages with a higher priority than KERN_WARNING (means < 4, KERN_EMERG to KERN_ERR) will be displayed on the console.

2 ways to add this command-line parameter which is set in the extlinux.conf file of boot partition:

- If using SD card, this is possible to edit the file on host PC:

```
Insert SD card on host PC
Check for mounting boot partition (i.e. /media/$USER/bootfs)
Check for your HW config (i.e. booting on mmc0 (SD Card) with ev1 board)
PC $> cd /media/$USER/bootfs/mmc0_stm32mp157c-ev1_extlinux/
PC $> gedit extlinux.conf
Add loglevel=8 at the end of APPEND line
Save and insert SD card on the board
```

- If using SD Card or eMMC, this is possible to edit the file directly on the board side:

```
When software is boot
Mount boot partition
Board $> mount /dev/mmcblk0p4 /boot (if not already done)
Update the kernel command line
Board $> cd /boot
Board $> cd mmc0_stm32mp157c-ev1_extlinux (case SD card on ev1 board)
Modify extlinux.conf to add loglevel=8 at the end of APPEND line by using ‘vi’ editor
Save and reboot the board
```

5.2.3 Using sysfs in runtime

To change your current console_loglevel simply write to this file:

```
Board $> echo <loglevel> > /proc/sys/kernel/printk
```

or using dmesg command.

As example:

```
Board $> echo 8 > /proc/sys/kernel/printk          # Temporary increase loglevel to
display messages up to loglevel 8
```

In that case, every kernel messages will appear on your console, as **all priority higher than 8 (lower loglevel values)** will be displayed.

Please note that after reboot, this configuration is reset.

5.2.4 Using menuconfig before compilation

As values are defined first at compilation step, this is also possible to set them (**CONFIG_CONSOLE_LOGLEVEL_DEFAULT** and **CONFIG_MESSAGE_LOGLEVEL_DEFAULT**) using the Linux kernel Menuconfig tool (Menuconfig or how to configure kernel):

```
Symbol: CONSOLE_LOGLEVEL_DEFAULT [=7]
Location:
  Kernel hacking --->
    printk and dmesg options --->
      (7) Default console loglevel (1-15)

Symbol: MESSAGE_LOGLEVEL_DEFAULT [=4]
Location:
  Kernel hacking --->
    printk and dmesg options --->
      (4) Default message log level (1-7)
```

5.3 Use loglevel in kernel source for log and trace

5.3.1 Using printk

A loglevel information can be added in the printk function call, with the following syntax.

```
printk(KERN_ERR "something went wrong, return code: %d\n",ret);
```

When not present, default loglevel value is given by CONFIG_MESSAGE_LOGLEVEL_DEFAULT (usually "4" =KERN_WARNING)

5.3.2 Using dedicated functions

In the loglevels table above, there are some alias functions *pr_* and *dev_*.

These functions are defined to replace *printk + loglevel info inside*, in order to simplify syntax.

```
pr_err("something went wrong, return code: %d\n",ret);
```

*dev_* functions are taken one more parameter to provide more information about current device or driver where message is coming from.

- Example for *pr_info*

```
pr_info("%s%s%s at %s (irq = %d, base_baud = %d) is a %s\n",
        port->dev ? dev_name(port->dev) : "",
        port->dev ? ": " : "",
        port->name,
        address, port->irq, port->uartclk / 16, uart_type(port));
```

will display information below:

```
[ 0.919488] 40010000.serial: ttySTM0 at MMIO 0x40010000 (irq = 41, base_baud = 6046875)
is a stm32-usart
```

- Example for *dev_info*

```
dev_info(&pdev->dev, "interrupt mode used for rx (no dma)\n");
```

will display information below, including device reference automatically:

```
[    1.046700] stm32-usart 40010000.serial: interrupt mode used for rx (no dma)
```

6 earlyprintk

earlyprintk is a Linux kernel debug feature useful to get traces for kernel issues which happen before the normal console is initialized.

6.1 Linux kernel configuration

In order to enable earlyprintk feature, the Linux kernel configuration must activate **CONFIG_DEBUG_LL**, **CONFIG_STM32MP1_DEBUG_UART** and **CONFIG_EARLY_PRINTK** using the Linux kernel Menuconfig tool (Menuconfig or how to configure kernel):

```
Symbol: DEBUG_LL
Location:
  Kernel hacking --->
    [*] Kernel low-level debugging functions

Symbol: STM32MP1_DEBUG_UART
Location:
  Kernel hacking --->
    [*] Kernel low-level debugging functions
      [*] Kernel low-level debugging port
      (X) Use STM32MP1 UART for low-level debug

Symbol: EARLY_PRINTK
Location:
  Kernel hacking --->
    [*] Early printk
```

6.2 Serial port configuration

When enabling the Linux kernel configuration **CONFIG_STM32MP1_DEBUG_UART**, it configures the addresses of the UART registers to be used.

By default, on STM32MP1 boards, UART4 is used for console for Linux kernel and by extension at all boot stages.

In case the UART port is different on a new board, you must apply the following changes:

- Update value for **CONFIG_DEBUG_UART_PHYS**, to select the UART port for the debug console

```
Symbol: DEBUG_UART_PHYS [=0x40010000]
Location:
  Kernel hacking --->
    [*] Kernel low-level debugging port (Use STM32MP1 UART for low-level debug)
    (0x40010000)  Physical base address of debug UART
```

- Update value for **CONFIG_DEBUG_UART_VIRT**, to define the associated virtual address to be used

```
Symbol: DEBUG_UART_VIRT [=0xFE010000]
Location:
  Kernel hacking --->
    [*] Kernel low-level debugging port (Use STM32MP1 UART for low-level debug)
    (0xFE010000)  Virtual base address of debug UART
```

Following rules to be respected for defining the virtual address:

- The 20 low weight bits (21 in case LPAE is enabled) must be kept in order to align region size of 1MB (2MB in LPAE is enabled).
- It must be mapped at the upper address of the vmalloc area, in order to not be overwritten by kernel which is stating from lower addresses: i.e here we select 0xFE0xxxxx

```
CONFIG_DEBUG_UART_PHYS: 0x40010000 /* UART4 */
CONFIG_DEBUG_UART_VIRT: 0xFE010000
```

- Please find below table for USART/UART of STMP32MP1:

Name	Physical base address	Virtual base address
USART 1	5c000000	**FE0**00000
USART 2	4000e000	**FE0**0e000
USART 3	4000f000	**FE0**0f000
UART4	40010000	**FE0**10000
UART5	40011000	**FE0**11000
USART 6	44003000	**FE0**03000
UART7	40018000	**FE0**18000
UART8	40019000	**FE0**19000

Warning

Note that the UART port used for console must be aligned for all components of the boot chain: FSBL(TF-A), SSBL-U-Boot) and Linux kernel

Especially because the Linux kernel do not configure all the setting registers for the UART port, as this is done by SSBL (U-Boot - How to debug)
See also TF-A - How to debug for FSBL changes)

6.3 Linux kernel boot command configuration

The Linux kernel boot command must contain the command-line parameter **earlyprintk**.

For instance, the kernel *bootargs* can be modified in the following ways:

- Mount a boot partition from the Linux kernel console, and then update the extlinux.conf file using the vi editor (see man page [6], or introduction page[7]). For example:

```
Board $> mount /dev/mmcblk0p4 /boot
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

or

- Edit the extlinux.conf file by using UMS (USB Mass Storage): see How to use USB mass storage in U-Boot for details.

To mount partitions (mmc 0:microSD card / mmc 1: eMMC):

- Press any key to stop at U-Boot execution when booting the board.

```
Board $> ...
Board $> Hit any key to stop autoboot:  0
Board $> STM32MP>
```

- Then

```
STM32MP> ums 0 mmc 0
```

- Check for the boot partition mounted on your host PC (/media/*$USER*/bootfs)
- Edit the extlinux file corresponding to your setup (/media/*$USER*/bootfs/mmc0_stm32mp157c-ev2_extlinux/extlinux.conf

- Update the kernel command line by adding the **earlyprintk** parameter:

```
root=/dev/mmcblk0p6 rootwait rw earlyprintk console=ttySTM0,115200
```

Save and quit file update, and then reboot the board.

6.4 Get trace

Earlyprintk traces are pushed automatically to the serial console defined as seen previously, and also added to the kernel ring log buffer.

7 dmesg command

As reference, please see man page[8].

The Kernel ring buffer can be displayed using *dmesg* command. It will display on the console all the content of the ring buffer.

- It is possible to filter messages following the loglevels:

```
Board $> dmesg -n <loglevel>
```

In that case, only messages with a value lower (**not lower equal**) than the console_loglevel will be printed.

Here, *<loglevel>* can be a numeric value, but also a string:

```
Supported log levels (priorities):
  emerg (0)
  alert (1)
   crit (2)
    err (3)
   warn (4)
 notice (5)
   info (6)
  debug (7)
```

As example:

```
Board $> dmesg -n 8          # Temporary change loglevel to display messages up to debug
level
or
Board $> dmesg -n debug
```

In that case, every kernel messages will appear on your console, as **all priority higher than 8 (lower loglevel values)** will be displayed.

- It is possible to clear the dmesg buffer

```
Board $> dmesg -c         # Display the full content of dmesg ring buffer, and then clear
it
Board $> dmesg -C         # Clear the dmesg ring buffer
```

8 /var/log/messages file system entry

An other way to display the content of the Linux kernel log is to look at the content of the file /var/log/messages.

It contains general system activity messages from the start-up. It also provides useful information about origin of the message, and log level.

9 Dynamic debug message

These messages are using the loglevel 7 (KERN_DEBUG).

Please see How to use the kernel dynamic debug article.

10 References

- Linux Serial Console
- https://elinux.org/Debugging_by_printing
- https://www.kernel.org/doc/Documentation/printk-formats.txt
- The kernel's command-line parameters
- https://elinux.org/Debugging_by_printing#Log_Levels
- http://ex-vi.sourceforge.net/vi.html
- http://ex-vi.sourceforge.net/viin/paper.html
- http://man7.org/linux/man-pages/man1/dmesg.1.html

Linux® is a registered trademark of Linus Torvalds.

SD memory card (https://www.sdcard.org)

former spelling for e•MMC ('e' in italic)

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

Low layer of STM32Cube

Universal Asynchronous Receiver/Transmitter

Universal Synchronous/Asynchronous Receiver/Transmitter

First Stage Boot Loader

Trusted Firmware for Arm Cortex-A

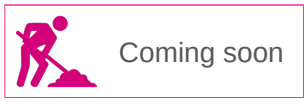
Second Stage Boot Loader

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

User-space Mode Setting

Stable: 04.02.2020 - 08:03 / Revision: 04.02.2020 - 07:55

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



Stable: 17.03.2020 - 14:36 / Revision: 25.02.2020 - 15:13

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

Contents

1 Purpose

This article describes how to use TTY with a user terminal. The TTY overview is described in Serial TTY overview article.

The use case of the following examples is a data transfer between a STM32 MPU board and PC, over a USB to a RS232 adapter cable.
The setup of this use case is described in details in the How to get Terminal article.

For the following examples:

- uart4 is activated by default (for the Linux console)
- usart3 is enabled by device tree
- The usart3 pins are connected to a RS232 card
- The RS232 card is connected to the PC over the USB to RS232 adapter cable.

Note: Some TTY tools are used in this article. A list of TTY tools is defined a dedicated article [TTY Tools].

2 Print the file name of the terminal connected to standard input (with tty tool)

```
Board $> tty
# The console is connected to uart4 (aka ttySTM0) #
/dev/ttySTM0
```

3 Change serial port configuration (with stty tool)

Many serial port properties can be displayed and changed with the stty tool. The full feature list is available in stty user manual pages[1].

```
Board $> stty --help
```

- Display the current configuration:

  The termios default configuration is specific to each Linux distribution. Before starting a serial communication between two devices, it is recommended to check that termios configurations are compatible on both devices. The termios configurations need to be aligned first.

```
Board $> stty -a -F /dev/ttySTM1
# Display the configuration of uart3 (aka ttySTM1) #
speed 115200 baud; rows 45; columns 169; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>;
swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -
ixany -imaxbel -iutf8
opost -olcuc ocrnl -onlcr -onocr onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig -icanon iexten -echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl
echoke -flusho -extproc
```

- Display only the current baud rate:

```
Board $> stty -F /dev/ttySTM1 speed
# uart3 (aka ttySTM1) baud rate is set to 115200 bps #
115200
```

- Change the baud rate:

  stty -F /dev/ttySTMx *EXPECTED_BAUDRATE*

  Example: change the baud rate to 19200

```
# Change uart3 (aka ttySTM1) baud rate to 19200 bps #
Board $> stty -F /dev/ttySTM1 19200
```

The stty tool proposes many arguments allowing many operations on a tty terminal, such as:

- special settings (various arguments such as speed, line discipline, minimum number of characters for a completed read, size, timeout, etc...)
- control settings
- input settings
- output settings
- local settings

- combination settings

Note: If you want to go further, an interesting tutorial describes termios and stty [2].

4 Send / Receive data (with stty, minicom, echo and cat tools)

Serial counters can be very useful to debug the following use cases.

4.1 Default configuration (8 data bits frame, no parity errors detection, no framing errors detection)

Canonical mode, input modes and output modes termios settings have a major influence on data processing. The following settings can be deactivated for testing.
In case of unexpected behavior, all canonical mode, input modes and output modes settings must be checked. mkssoftware proposes an enriched version of termios manual [3], where the following definitions are provided.

- echo: Enable echo. If ECHO is set input characters are echoed back to the terminal.
- icanon: Canonical input (erase and kill processing). If ICANON is set canonical processing is enabled. In canonical mode input processing is processed in units of lines. A line is delimited by a '\n' character or and end-of-file (EOF) character. A read request does not return until an entire line is read from the port or a signal is received.
- onlcr: Map NL to CR-NL on output. If ONLCR is set the NL character is transmitted as the CR-NL character pair.

Sending data can be simply done by opening the device as a file and writing data to it.

- Configure a port on ttySTM1 (aka usart3). echo, icanon and onlcr properties are deactivated to handle raw data.

```
Board $> stty -F /dev/ttySTM1 115200 -echo -icanon -onlcr
```

- Display the current configuration on ttySTM1 (usart3):

```
# display the configuration of uart3 (aka ttySTM1) #
Board $> stty -a -F /dev/ttySTM1
speed 115200 baud; rows 45; columns 169; line = 0;
```

- Open a port on ttySTM1 (usart3) to receive data

```
Board $> cat /dev/ttySTM1 &
```

- On the remote PC, identify the tty terminal associated to the RS232 card connected on STM32MPU USART3 pins

```
# Command to execute from host terminal #
PC $> ls /dev/ttyUSB*
/dev/ttyUSB0
```

- Open a minicom in a second terminal on the remote device connected on USART3 pins

```
PC $> minicom -D /dev/ttyUSB0
```

- Display the current configuration on ttyUSB0 (remote device):

```
# Display the configuration of host uart (aka ttyUSB0) #
PC $> stty -a -F /dev/ttyUSB0
speed 115200 baud; rows 45; columns 169; line = 0;
```

- Send data from remote PC to STM32MPU over USART3 with default termios configuration (8 frames length, no parity)

```
# Execute this command from host terminal #
PC $> echo "HELLO" > /dev/ttyUSB0
```

- Send data from STM32MPU to remote PC over USART3 with default termios configuration (8 frames length, no parity)

```
# Execute this command from STM32 terminal #
Board $> echo "HELLO" > /dev/ttySTM1
```

4.2 Parity errors detection

Some additional termios functions allow to enable parity errors detection:

- parenb: Parity enable
- parodd: Odd parity else even
- inpck: Enable input parity or framing check
- ignpar: Ignore characters with parity or framing errors

Exemples:

- Configure a port on ttySTM1 (usart3) with even parity enabling

```
# STM32 parity enabling #
Board $> stty -F /dev/ttySTM1 115200 -echo -icanon -onlcr parenb -parodd inpck ignpar
```

- Open a port on ttySTM1 (usart3) to receive data

```
Board $> cat /dev/ttySTM1 &
```

```
Open a minicom in a second terminal on the remote device connected on USART3 pins
PC $> minicom -D /dev/ttyUSB0
```

- Configure a port on ttyUSB0 (remote device) with even parity enabling:

```
# Remote device parity enabling #
PC $> stty -F /dev/ttyUSB0 115200 -echo -icanon -onlcr parenb -parodd inpck ignpar
```

- Send data from remote PC to STM32MPU over USART3

```
# Execute this command from host terminal #
PC $> echo "HELLO" > /dev/ttyUSB0
```

- Send data from STM32MPU to remote PC over USART3

```
#  Execute this command from STM32 terminal #
Board $> echo "HELLO" > /dev/ttySTM1
```

4.3 Framing errors detection

Some additional termios functions allow to enable framing errors detection:

- csize: Number of bits per byte (character size and parity bit configurations)
- inpck: Enable input framing check
- ignpar: Ignore characters with parity or framing errors

Exemples:

- Configure a port on ttySTM1 (usart3) with framing check enabling and 7 data bits length frames

```
# STM32 framing enabling #
Board $> stty -F /dev/ttySTM1 115200 -echo -icanon -onlcr cs7 inpck ignpar
```

- Open a port on ttySTM1 (usart3) to receive data

```
Board $> cat /dev/ttySTM1 &
```

```
Open a minicom in a second terminal on the remote device connected on USART3 pins
PC $> minicom -D /dev/ttyUSB0
```

- Configure a port on ttyUSB0 (remote device) with framing check enabling and 7 data bits length frames

```
# Remote device parity enabling #
PC $> stty -a -F /dev/ttyUSB0 115200 -echo -icanon -onlcr cs7 inpck ignpar
speed 115200 baud; rows 45; columns 169; line = 0;
```

- Send data from remote PC to STM32MPU over USART3

```
# Execute this command from host terminal #
PC $> echo "HELLO" > /dev/ttyUSB0
```

- Send data from STM32MPU to remote PC over USART3

```
# Execute this command from STM32 terminal #
Board $> echo "HELLO" > /dev/ttySTM1
```

5 Identify processes using a tty serial device (with fuser tool)

```
Board $> fuser /dev/ttySTM0
# The process numbered 395, 691 and 3872 are using a tty serial device #
395 691 3872
```

6 Link a tty serial device with a line discipline (with ldattach tool)

Attach ttySTM1 with line discipline number *n* :

```
Board $> ldattach n /dev/ttySTM1
```

7 File transfer over serial console

Please see the dedicated article How to transfer a file over serial console.

8 References

- stty manual page
- A Brief Introduction to termios: termios(3) and stty stty tutorial
- struct_termios man page

TeleTYpewriter

Microprocessor Unit

Linux® is a registered trademark of Linus Torvalds.

also known as

terminal input output structure

Stable: 02.11.2020 - 10:48 / Revision: 19.10.2020 - 12:09

A quality version of this page, approved on *2 November 2020*, was based off this revision.

Contents

1 Introduction

As prerequisite to reading this article, please refer to the Dmesg and Linux kernel log page.

"Dynamic debug is designed to allow you to dynamically enable/disable kernel code to obtain additional kernel information. Currently, if **CONFIG_DYNAMIC_DEBUG** is set, all **pr_debug()/dev_dbg()** calls can be dynamically enabled per-callsite." extracted from the Linux kernel documentation[1].

The related debugfs entry is usually:

```
/sys/kernel/debug/dynamic_debug/control
```

Note that the verbose **dev_vdbg()** calls cannot be dynamically activated.

When the dynamic debug traces are activated, the trace results are printed in **dmesg (or /proc/kmsg)**, and in the console if console loglevel is set to 8.

2 More technical information

The dynamic debug trace configuration is done through a **control** file in the **debugfs** filesystem: *<debugfs>/dynamic_debug /control*

The command includes keywords and flag elements (for details see the Linux kernel documentation[1]).

- Keywords

Possible keywords are:

```
func : function name
file : source filename
module : module name
format : format string
line : line number (including ranges of line numbers)
```

The colored keywords above are illustrated by examples in the next chapter.

- Flags

The flag specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

```
- : remove the given flags
+ : add the given flags
= : set the flags to the given flags
```

Possible flags are:

```
f : Include the function name in the printed message
l : Include line number in the printed message
m : Include module name in the printed message
p : Causes a printk() message to be emitted to dmesg
t : Include thread ID in messages not generated from interrupt context
```

3 Examples

- Track all dev_*dbg/pr_debug() in a **file** (you can add several files if necessary):

```
Board $> mount -t debugfs none /sys/kernel/debug
Board $> echo "file stm32-adc.c +p" > /sys/kernel/debug/dynamic_debug/control
```

Note that just the file name or full file path can be given, here *stm32-adc.c* or *drivers/iio/adc/stm32-adc.c*

- Track only one **line** with dev_dbg() in a **file** (you can add several files and several lines if necessary, please use the last line number of the function call):

```
Board $> echo "file stm32-adc.c line 1438 +p" > /sys/kernel/debug/dynamic_debug/control
```

- For an entire "**module** (module means ~.ko, so not applicable for a statically linked driver)":

```
Board $> echo "module cfg80211 +p" > /sys/kernel/debug/dynamic_debug/control
```

- If you want to list all available traces *(warning: it is a long file so you may need to use "tee" or another solution to save it)*:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | tee /tmp/dynamic_log.log
```

- For instance, if you are looking for a particular **file** to find a particular **line**:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | grep adc
drivers/iio/adc/stm32-adc.c:1515 [stm32_adc]stm32_adc_conf_scan_seq =p "%s chan %d to %s%
d\012"
drivers/iio/adc/stm32-adc.c:1438 [stm32_adc]stm32_adc_awd_set =p "%s chan%d htr:%d ltr:%
d\012"
drivers/iio/adc/stm32-adc.c:2182 [stm32_adc]stm32_adc_dma_start =p "%s size=%d watermark=%
d\012"
drivers/iio/adc/stm32-adc.c:2304 [stm32_adc]stm32_adc_trigger_handler =p "%s bufi=%d\012"
drivers/iio/adc/stm32-adc.c:2443 [stm32_adc]stm32_adc_chan_of_init =p "Configured to use
injected\012"
drivers/iio/adc/stm32-adc.c:2364 [stm32_adc]stm32_adc_of_get_resolution =p "Using %u bits
resolution\012"
```

- Multiple commands can be written together, separated by ';' or '\n'.

```
Board $> echo "file stm32-adc.c +p; file stm32-adc-core.c +p" > /sys/kernel/debug
/dynamic_debug/control
```

- A another method is to use a wildcard. The match rule supports * (matches zero or more characters) and ? (matches exactly one character). For example, you can match all USB drivers:

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

4 Synchronous tracing on the console

In the case of a crash, or impossibility to call dmesg, it is sometimes useful to have traces synchronously emitted on the console.
Only error, warning and informational traces are emitted synchronously on the console (that is, loglevel=5), so if you need to see the lower level traces too, you need to change the console loglevel to "8".

```
<enable the conditional traces>
Board $> echo 8 > /proc/sys/kernel/printk
or
Board $> dmesg -n 8
or
Board $> dmesg -n debug
```

Please follow this article to get a serial console for the target: How to get Terminal

Warning

As all traces are now synchronously emitted, real-time is affected

If you want to return to the default console log level, you have to get this default value from the procfs entry */proc/sys /kernel/printk*:

```
Board $> cat /proc/sys/kernel/printk
8       4       1       7
Board $> dmesg -n 7
Board $> cat /proc/sys/kernel/printk
7       4       1       7
```

5 Debug messages during boot process

In order to activate debug messages during the boot process, even before userspace and debugfs exist, use the kernel's command-line parameter: **dyndbg**

For instance, the kernel *bootargs* can be modified in the following ways:

- Mount a boot partition from the Linux kernel console, and then update the extlinux.conf file using the vi editor (see man page [2], or introduction page[3]). For example:

```
Board $> mount /dev/mmcblk0p4 /boot
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

or

- Edit the extlinux.conf file by using UMS (USB Mass Storage): see How to use USB mass storage in U-Boot for details.

To mount partitions (mmc 0:microSD card / mmc 1: eMMC):

- Press any key to stop at U-Boot execution when booting the board.

```
Board $> ...
Board $> Hit any key to stop autoboot:  0
Board $> STM32MP>
```

- Then

```
STM32MP> ums 0 mmc 0
```

- Check for the boot partition mounted on your host PC (/media/*$USER*/bootfs)
- Edit the extlinux file corresponding to your setup (/media/*$USER*/bootfs/mmc0_extlinux/stm32mp157f-dk2_extlinux.conf

- Update the kernel command line, adding the dyndbg parameter:

```
root=PARTUUID=e91c4e10-16e6-4c0e-bd0e-77becf4a3582 rootwait rw console=ttySTM0,115200 dynd
bg="file drivers/usb/core/hub.c +p"
```

Save and quit file update, and then reboot the board.

Note: to display these debug messages in the console, in addition to the dmesg, add **loglevel=8** in the kernel command line.

- Reboot the board and check for a kernel command-line, and that debug messages are present in the dmesg output

6 References

- 1.01.1 Documentation/admin-guide/dynamic-debug-howto.rst
- http://ex-vi.sourceforge.net/vi.html
- http://ex-vi.sourceforge.net/viin/paper.html

- Useful external links

Document link	Document Type	Description
The dynamic debugging interface (lwn.net)	User guide	http://lwn.net
Documentation/dynamic-debug-howto.txt (lwn.txt)	User guide	http://lwn.net
Dynamic debug howto (kernel.org)	Standard	http://www.kernel.org

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

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

User-space Mode Setting

former spelling for e•MMC ('e' in italic)

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

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)
```

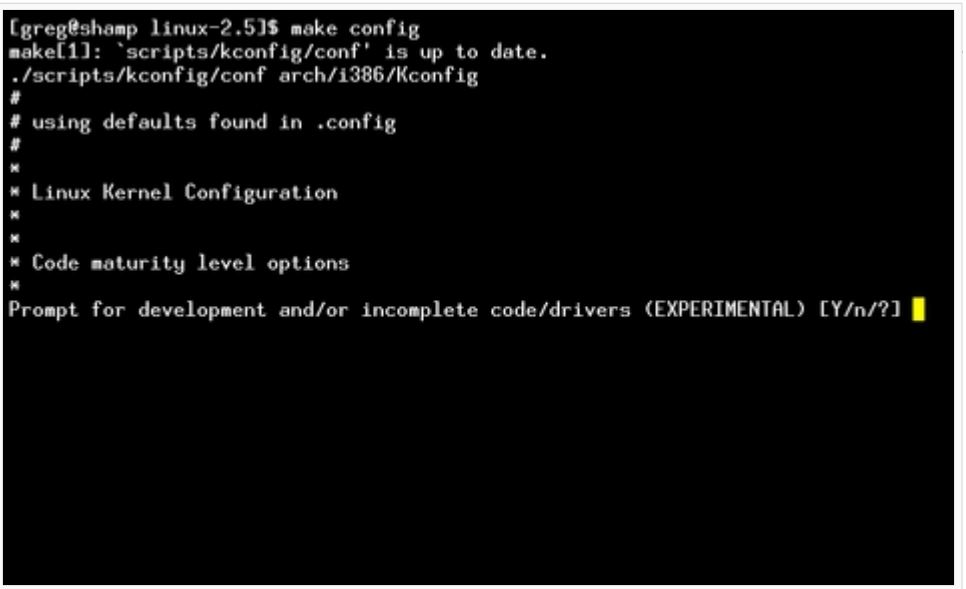


Figure 1. Configuring the kernel with make config

```
   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 :

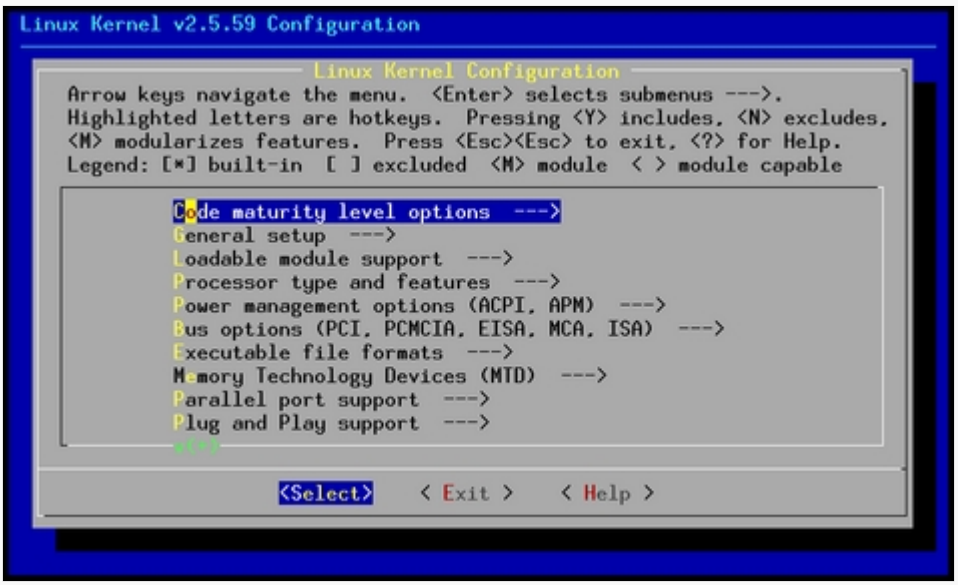


Figure 2. Make menuconfig makes it easier to back up and correct mistakes

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.

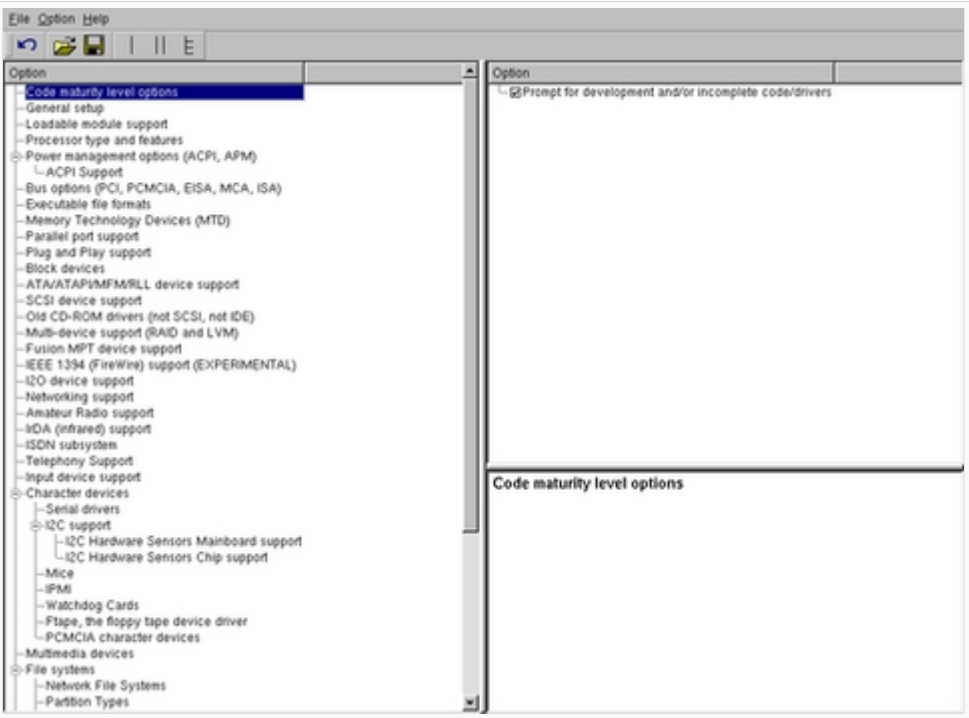


Figure 3. The Qt-Based make xconfig

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: 02.11.2020 - 15:54 / Revision: 03.09.2020 - 13:39

A quality version of this page, approved on *2 November 2020*, was based off this revision.

Contents

1 Article Purpose

This article explains how to configure the USART when it is assigned to the Linux®OS. In that case, it is controlled by the Serial and TTY frameworks.

The configuration is performed using the device tree mechanism that provides a hardware description of the USART peripheral, used by the stm32-usart Linux driver.

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 USART is a multifunction device.

Each function is represented by a separate binding document:

- Generic UART bindings[1] used by UART framework.
- STM32 USART driver bindings[2] used by stm32-usart driver. This bindings documentation explains how to write device tree files for STM32 USARTs.

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)

All STM32 USART nodes (excepted USART1, secure instance under ETZPC control) are described in microprocessor device tree (ex: stm32mp151.dtsi [3]) with default parameters and disabled status.
The required and optional properties are fully described in the bindings files.

Warning

This device tree configuration related to the STM32 should be kept as is, without being modified by the customer.

3.2 DT configuration (Board level)

Part of the device tree is used to describe the USART hardware used on a given board:

- Which USART instances are enabled (by setting status to "okay")
- Which features are used (such as DMA transfer or direct transfer, transfer speed or parity)
- Which pins are configured via pinctrl.
- Which serial aliases are linked to UART instances. Please check the alias already used in other device tree files to avoid alias conflicts. The alias defines the index of the ttySTMx instance linked the UART.

Note:

- As the pin configuration can be different for each board, several DT configurations can be defined for each UART instance.
- The pin configuration is described in board datasheet. Each new pin configuration described in boards datasheet needs to be defined in device tree.

Three device tree configurations can be defined for each pin muxing configuration:

- Default: for standard usage (mandatory)
- "sleep": for Sleep mode, when the UART instance is not a wake up source (mandatory)
- "idle": for Sleep mode, when the UART instance is a wake up source (optional)

3.3 DT configuration examples

3.3.1 Activation of a USART or UART instance

Information

Some UART pins are available on GPIO expansion and Arduino connectors (depending on the connectors available on the board).

- STM32MP157x-EV1 Evaluation board GPIO expansion connector
- STM32MP157x-DKx Discovery kit GPIO expansion connector

To communicate with a UART instance, an RS232 card must be plugged on the UART pins.

The example below shows how to configure and enable a UART instance at board level, based on STM32MP157C-EV1 board USART3 example.
Note: For STM32 boards, the configuration is already defined in the device tree. Only the device activation is needed.

To activate a UART instance, please follow steps below:

- Define the instance pin configuration (ex: stm32mp15-pinctrl.dtsi [4]).

```
usart3_pins_a: usart3-0 {
        pins1 {
                /* USART3 TX and RTS pins activation for default mode */
                pinmux = <STM32_PINMUX('B', 10, AF7)>, /* USART3_TX */
                         <STM32_PINMUX('G', 8, AF8)>; /* USART3_RTS */
                bias-disable;
                drive-push-pull;
                slew-rate = <0>;
        };
        pins2 {
                /* USART3 RX and CTS_NSS pins activation for default mode */
                pinmux = <STM32_PINMUX('B', 12, AF8)>, /* USART3_RX */
                         <STM32_PINMUX('I', 10, AF8)>; /* USART3_CTS_NSS */
                bias-disable;
        };
};

usart3_idle_pins_a: usart3-idle-0 {
        pins1 {
                /* USART3 TX, RTS, and CTS_NSS pins deactivation for sleep mode */
                pinmux = <STM32_PINMUX('B', 10, ANALOG)>, /* USART3_TX */
                         <STM32_PINMUX('G', 8, ANALOG)>, /* USART3_RTS */
                         <STM32_PINMUX('I', 10, ANALOG)>; /* USART3_CTS_NSS */
        };
        pins2 {
                /* USART3_RX pin still active for wake up */
                pinmux = <STM32_PINMUX('B', 12, AF8)>; /* USART3_RX */
                bias-disable;
        };
};

usart3_sleep_pins_a: usart3-sleep-0 {
        pins {
                /* USART3_TX, RTS, CTS_NSS, and RX pins deactivation for sleep mode */
                pinmux = <STM32_PINMUX('B', 10, ANALOG)>, /* USART3_TX */
                         <STM32_PINMUX('G', 8, ANALOG)>, /* USART3_RTS */
                         <STM32_PINMUX('I', 10, ANALOG)>, /* USART3_CTS_NSS */
                         <STM32_PINMUX('B', 12, ANALOG)>; /* USART3_RX */
        };
};
```

- Define the serial alias for this instance at board level (ex: stm32mp157c-ev1.dts [5]).

```
aliases {
        /* Serial1 alias (ie ttySTM1) assigned to usart3 */
        serial1 = &usart3;
        ethernet0 = &ethernet0;
};
```

- Configure and activate the instance at board level (ex: stm32mp157c-ev1.dts [5]).

```
&usart3 {
        pinctrl-names = "default", "sleep", "idle";        /* pin configurations
definition */
        pinctrl-0 = <&usart3_pins_a>;                        /* default pin configuration
selection */
        pinctrl-1 = <&usart3_sleep_pins_a>;                /* sleep pin configuration
selection */
        pinctrl-2 = <&usart3_idle_pins_a>;                /* idle pin configuration
selection */
        status = "okay";                                        /* device activation */
};
```

Note: The pin configuration selected has to be aligned with the pin configuration described in the board datasheet.

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

- Documentation/devicetree/bindings/serial/serial.txt , UART generic device tree bindings
- Documentation/devicetree/bindings/serial/st,stm32-usart.txt , STM32 USART device tree bindings
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file
- arch/arm/boot/dts/stm32mp15-pinctrl.dtsi , STM32MP15 pinctrl device tree file
- 5.05.1 arch/arm/boot/dts/stm32mp157c-ev1.dts , STM32MP157c ev1 board device tree file

Linux® is a registered trademark of Linus Torvalds.

Operating System

Universal Synchronous/Asynchronous Receiver/Transmitter

Device Tree

Universal Asynchronous Receiver/Transmitter

Direct Memory Access

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

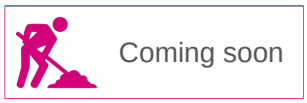
Receive

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

Stable: 19.02.2019 - 13:32 / Revision: 29.01.2019 - 11:08

A quality version of this page, approved on *19 February 2019*, was based off this revision.

Template:ArticleMainWriter Template:ArticleApprovedVersion



Stable: 21.02.2020 - 15:07 / Revision: 21.02.2020 - 10:43

A quality version of this page, approved on *21 February 2020*, was based off this revision.

Contents

1 Article Purpose

This article aims at giving some first information useful to start with the Linux®TTY tools. These tools are useful for interacting with TTY terminals.

2 Introduction

These tools use TTY sysfs and character device directly (See TTY user space interface for further details).

3 Tools list

Please find below a list of useful TTY tools provided by Linux® community:

- **fuser** [1] - to identify processes using files or sockets.
- **inputattach** [2] (based on termios API) - to attach a serial line to an input-layer device.

Inputattach attaches a serial line to an input-layer device via a line discipline. Exactly one of the available modes must be specified on the command line.

- **kermit** [3] - transport and platform independent interactive and scriptable communications software.

C-Kermit is a modem program, a Telnet client, an Rlogin client, an FTP client, an HTTP client, and on selected platforms, also an X.25 client. It can make its own secure internet connections using IETF-approved security methods including Kerberos IV, Kerberos V, SSL/TLS, and SRP and it can also make SSH connections through an external SSH client application. It can be the far-end file-transfer or client/server partner of a desktop Kermit client. It can also accept incoming dialed and network connections. It can even be installed as an internet service on its own standard TCP socket, 1649 [RFC2839, RFC2840].

- **ldattach** [4] (based on termios API) - to attach a line discipline to a serial line.

The ldattach daemon opens the specified device file (which should refer to a serial device) and attaches the line discipline ldisc to it for processing of the sent and/or received data. It then goes into the background keeping the device open so that the line discipline stays loaded. The line discipline ldisc may be specified either by name or by number. In order to detach the line discipline, kill the ldattach process. With no arguments, ldattach prints usage information.

- **minicom** [5] - friendly serial communication program.

Minicom is a communication program which somewhat resembles the shareware program TELIX but is free with source code and runs under most Unices. Features include dialing directory with auto-redial, support for UUCP-style lock files on serial devices, a separate script language interpreter, capture to file, multiple users with individual configurations, and more.

- **setserial** [6] - To get/set Linux serial port information.

Setserial is a program designed to set and/or report the configuration information associated with a serial port. This information includes the I/O port and the IRQ used by a particular serial port, and whether or not the break key should be interpreted as the Secure Attention Key, and so on.

- **stty** [7] (based on termios API) - to change and print terminal line settings.
- **tty** [8] - to print the file name of the terminal connected to standard input

Information

The descriptions above are provided by the manual pages of the tools.

4 Getting started

Examples of TTY tools usage are handled in the following articles :

- How to use TTY with User Terminal
- How to transfer a file over serial console

5 Installation on your target

Some of the TTY tools aren't embedded by default in OpenSTLinux distribution. They can be compiled independently and then installed on the target (see Adding Linux user space applications).

6 References

- fuser man page
- inputattach man page
- kermit man page
- ldattach man page
- minicom man page
- setserial man page
- stty man page
- tty man page

TeleTYpewriter

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

terminal input output structure

Application programming interface

Stable: 03.03.2021 - 14:35 / Revision: 03.03.2021 - 10:24

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

Contents

1 Article purpose

The purpose of this article is to:

- briefly introduce the USART 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 needed, how to configure the USART peripheral.

2 Peripheral overview

The **USART** peripheral is used to interconnect STM32 MPU devices with other systems, typically via RS232 or RS485 protocols. In addition, the USART supports the **Synchronous** mode that can be used for smartcard interfacing or SPI master /slave operation.

The **UART** peripheral is similar to the USART but does not support the Synchronous mode.

High-speed data communications can be achieved by using the DMA internal peripheral for multibuffer configuration.

2.1 Features

Refer to 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

USART1 is a **secure** instance (under ETZPC control).
The other UARTs and USARTs are **non-secure** instances.

3 Peripheral usage and associated software

3.1 Boot time

All USART (except USART1) and UART instances are boot devices that support serial boot for Flash programming with STM32CubeProgrammer.

3.2 Runtime

3.2.1 Overview

The STM32 MPU devices feature four USART instances (supporting both Asynchronous and Synchronous modes), and four UART instances (supporting only Asynchronous mode).

USART1 can be allocated to:

- the Arm®Cortex®-A7 secure core to be used under OP-TEE with the USART OP-TEE driver, typically to communicate with a smartcard.

All USART and UART instances can be allocated to:

- the Arm®Cortex®-A7 non-secure core to be used under Linux® with the tty framework. However, the Linux® kernel supports only the UART Asynchronous mode (Synchronous mode not supported).

or

- the Arm®Cortex®-M4 to be used with STM32Cube MPU Package with USART HAL driver. Both USART Synchronous and Asynchronous modes are supported by the STM32Cube MPU Package.

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	USART	USART OP-TEE driver	Linux serial /tty framework	STM32Cube USART driver	

3.2.3 Peripheral configuration

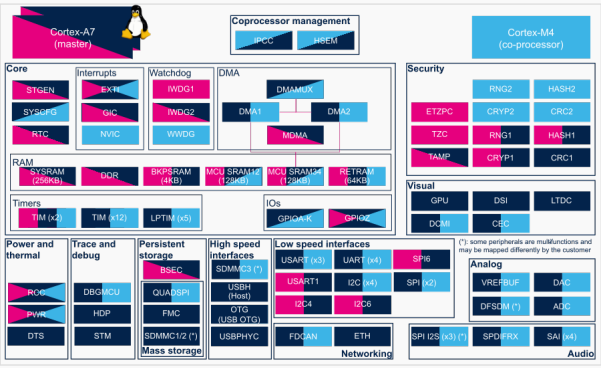
The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals) according to the information given in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

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	USART	USART1				Assignment (single choice)
		USART2				Assignment (single choice)
		USART3				Assignment (single choice)
		UART4				Assignment (single choice). Used for Linux® serial console on ST boards.
		UART5				Assignment (single choice)
		USART6				Assignment (single choice)
		UART7				Assignment (single choice)
		UART8				Assignment (single choice)

4 How to go further

Additional documentation on USART peripheral is available on st.com:

- STM32 USART training [1] presents the STM32 Universal Synchronous/Asynchronous Receiver/Transmitter interface.
- STM32 USART automatic baud rate detection [2] presents STM32 USART automatic baud rate detection.

5 References

- Please refer to **stm32f7_peripheral_usart** document on st.com
- STM32 USART automatic baud rate detection application note (AN4908)

Universal Synchronous/Asynchronous Receiver/Transmitter

Microprocessor Unit

Serial Peripheral Interface

Universal Asynchronous Receiver/Transmitter

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

Cortex®

Open Portable Trusted Execution Environment

Linux® is a registered trademark of Linus Torvalds.