Linux Mailbox framework overview

Contents

1. Linux Mailbox framework overview 3
2. Device tree 8
3. IPCC internal peripheral 13
4. Linux RPMsg framework overview 21
5. Linux remoteproc framework overview 28
6. Menuconfig or how to configure kernel 37

Stable: 30.01.2020 - 13:51 / Revision: 30.01.2020 - 13:49

Stable: 30.01.2020 - 13:31 / Revision: 30.01.2020 - 13:40

A quality version of this page, approved on *30 January 2020*, was based off this revision.

This article gives information about the Linux® mailbox framework. The mailbox framework is involved in interprocessor communication in heterogeneous multicore systems.

Contents

1 Framework purpose 4
2 System overview 5
2.1 Component description 5
2.2 API description 5
3 Configuration 6
4 Device tree configuration 7
5 How to trace and debug the framework 8
5.1 How to trace 8
6 References 8

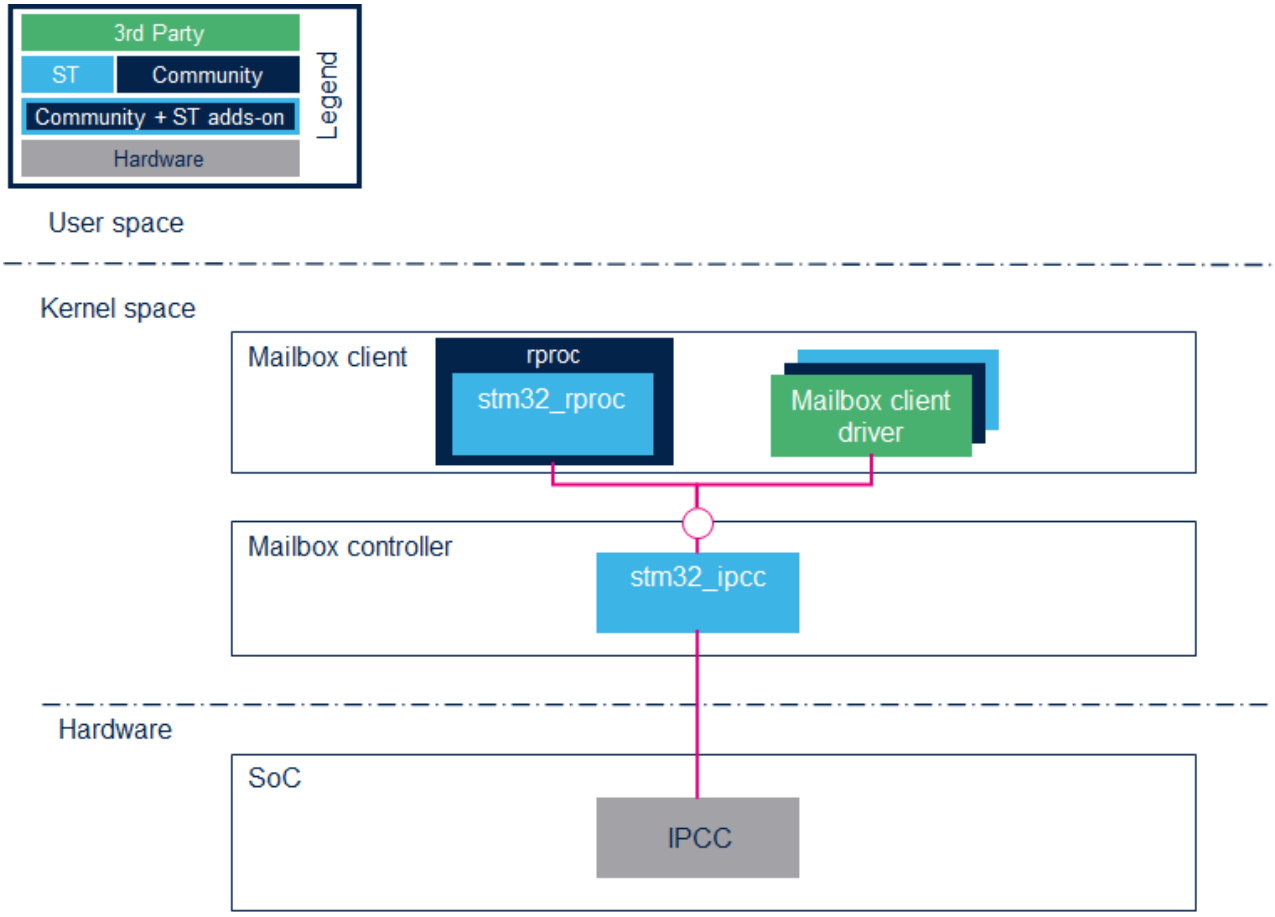
1 Framework purpose

The mailbox is used in interprocessor communication to exchange messages or signals between the host and the coprocessor cores. The mailbox framework is based on:

- A **mailbox controller** that is platform dependent:
  - It is in charge of configuring and handling IRQ from the IPCC peripheral.
  - It provides a generic API to the mailbox client.
- A **mailbox client** that is in charge of the message to send or receive.

A general presentation of the mailbox framework is available in the Linux mailbox documentation [1].

2 System overview



2.1 Component description

- **Mailbox controller**

  The mailbox controller is the **stm32_ipcc**. It configures and controls the IPCC peripheral

- **Mailbox client**

  The user can define his own mailbox client.

  For example, the RPMsg framework uses mailbox for the interprocessor communication.

  In this case the mailbox client is the remoteproc driver that forwards services from/to the RPMsg framework.

2.2 API description

The APIs are described in the Linux documentation:

- Mailbox client API [2]
- Mailbox controller API [3]

3 Configuration

Activate **stm32 IPCC** mailbox in kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel

```
Device drivers  --->
    -*- Mailbox Hardware Support  --->
        <*> STM32 IPCC Mailbox
```

4 Device tree configuration

The mailbox device node must be declared and enabled in the Linux kernel device tree. Here is an extract of the STM32MP1 evaluation board device tree:

```
ipcc: mailbox@4c001000 {
        compatible = "st,stm32-ipcc";
        #mbox-cells = <1>;
        reg = <0x4c001000 0x400>;
        interrupts-extended = <&intc GIC_SPI 100 IRQ_TYPE_NONE>,
                              <&intc GIC_SPI 101 IRQ_TYPE_NONE>,
                              <&exti 62 1>;
        interrupt-names = "rx", "tx", "wakeup";
        clocks = <&rcc_clk IPCC>;
        wakeup-source;

        Status = "Okay";
 };
```

Then client has to reserve channels. Here is an example of channel allocation for the remoteproc node:

```
&m4_rproc {
       memory-region = <&ipc_share>;
       mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
       mbox-names = "vq0", "vq1", "init_shdn";
       status = "okay";
};
```

5 How to trace and debug the framework

5.1 How to trace

Dynamic debug traces can be added using the following commands:

```
echo -n 'file stm32-ipcc.c +p' > /sys/kernel/debug/dynamic_debug/control
echo -n 'file mailbox.c +p' > /sys/kernel/debug/dynamic_debug/control
```

6 References

- Linux Mailbox documentation
- Mailbox client API
- Mailbox controller API

Inter-Processor Communication Controller

Application programming interface

Remote Processor Messaging

Generic Interrupt Controller

Serial Peripheral Interface

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

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

Contents

1 Purpose 9
1.1 Source files 9
1.2 Bindings 9
1.3 Build 9
1.4 Tools 10
2 STM32 11
3 How to go further 12
4 References 13

1 Purpose

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

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

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

1.1 Source files

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

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

1.2 Bindings

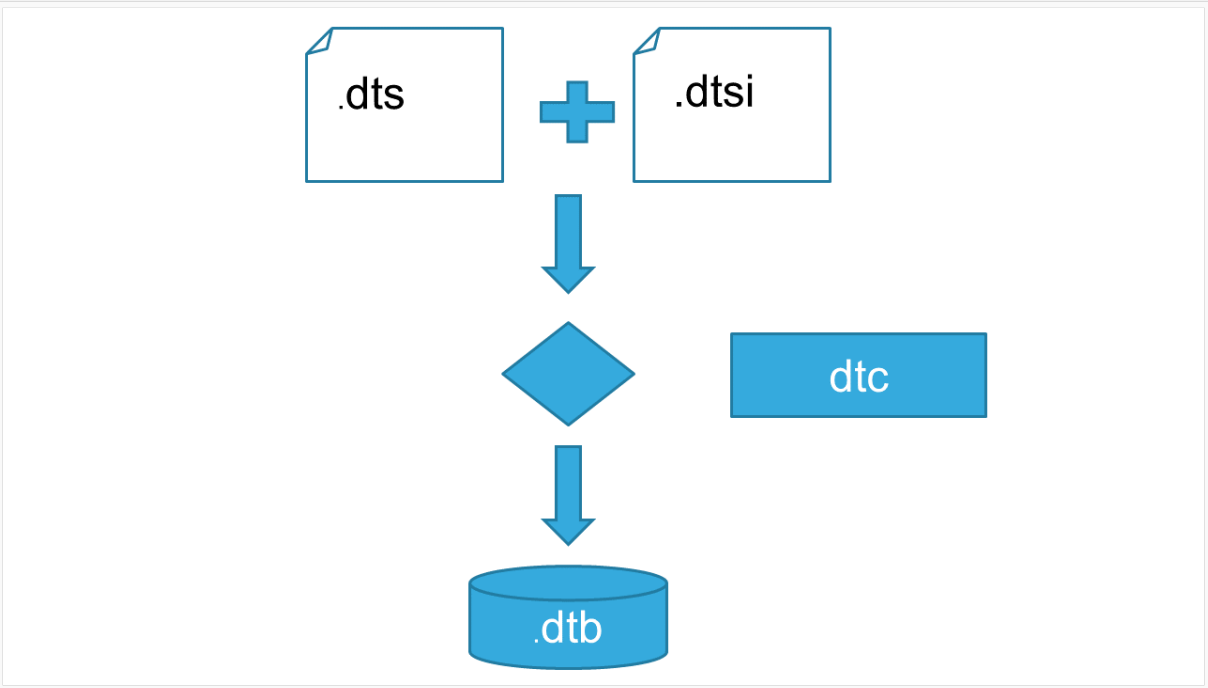
The device tree data structures and properties are named **bindings**. Those bindings are described in:

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

1.3 Build

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

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



Device tree build process

Information

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

1.4 Tools

The device tree compiler offers also some tools:

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

There are several ways to get those tools:

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

2 STM32

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

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

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

3 How to go further

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

4 References

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

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

Linux® is a registered trademark of Linus Torvalds.

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

Trusted Firmware for Arm Cortex-A

Stable: 22.01.2020 - 16:10 / Revision: 22.01.2020 - 10:51

A quality version of this page, approved on *22 January 2020*, was based off this revision.

Contents

1 Article purpose 14
2 Peripheral overview 15
2.1 Features 16
2.2 Security support 16
3 Peripheral usage and associated software 17
3.1 Boot time 17
3.2 Runtime 17
3.2.1 Overview 17
3.2.2 Software frameworks 17
3.2.3 Peripheral configuration 18
3.2.4 Peripheral assignment 19
4 References 21

1 Article purpose

The inter-processor communication controller (IPCC) is used to exchange data between two processors. It provides a non blocking signaling mechanism to post and retrieve information in an atomic way. Note that shared memory buffers are allocated in the MCU SRAM, which is not part of the IPCC block.

2 Peripheral overview

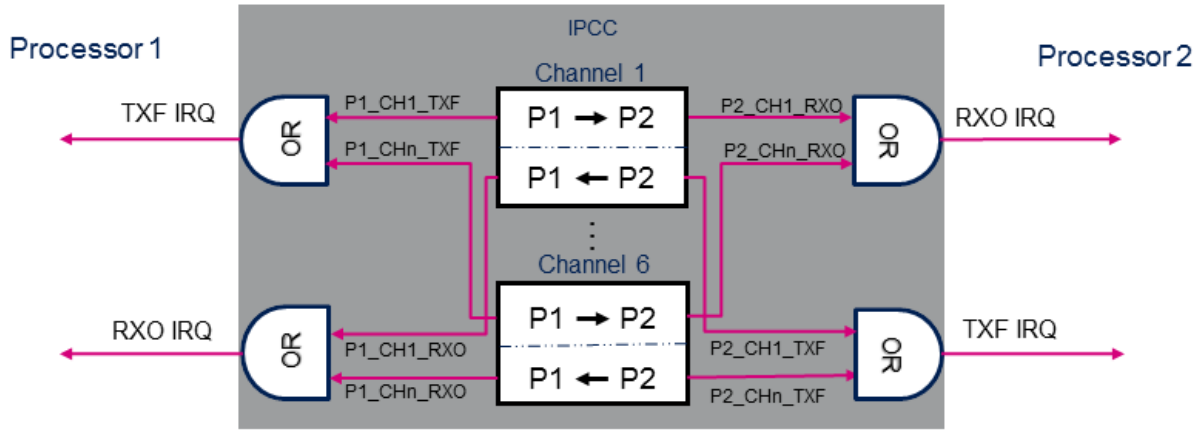
The **IPCC** peripheral provides a hardware support to manage inter-processor communication between two processor instances. Each processor owns specific register bank and interrupts.

The IPCC provides the signaling for **six bidirectional** channels.
Each channel is divided into two subchannels that offer a unidirectional signaling from the "sender" processor to the "receiver" processor:

- P1_TO_P2 subchannel
- P2_TO_P1 subchannel

A subchannel consists in:

- One flag that toggles between occupied and free: the flag is set to occupied by the "sender" processor and cleared by the "receiver" processor.
- Two associated interrupts (shared with the other channels):
  - RXO: RX channel occupied, connected to the "receiver" processor.
  - TXF: TX channel free, connected to the "sender" processor.
- Two associated interrupt masks multiplexing channel IRQs.



The IPCC supports the following channel operating modes:

- **Simplex communication mode**:
  - Only one subchannel is used.
  - Unidirectional messages: once the "sender" processor has posted the communication data in the memory, it sets the channel status flag to occupied. The "receiver" processor clears the flag when the message is treated.
- **Half-duplex communication mode**:
  - Only one subchannel is used.
  - Bidirectional messages: once the "sender" processor has posted the communication data in the memory, it sets the channel status flag to occupied. The "receiver" processor clears the flag when the message is treated and the response is available in shared memory.

- **Full-duplex communication mode**:
  - The subchannels are used in Asynchronous mode.
  - Any processor can post asynchronously a message by setting the subchannel status flag to occupied. The "receiver" processor clears the flag when the message is treated. This mode can be considered as a combination of two simplex modes on a given channel.

2.1 Features

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

2.2 Security support

The IPCC is a **non-secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

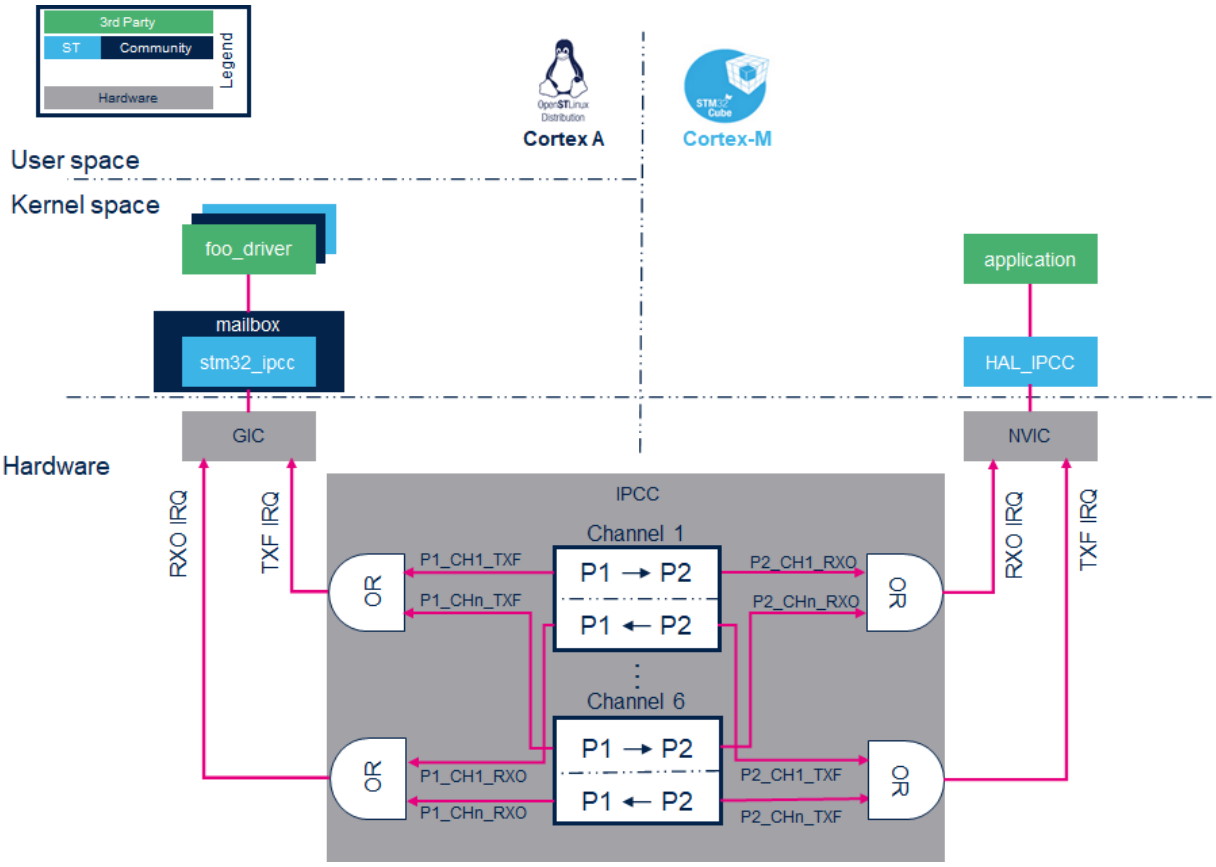
The IPCC is not used at boot time.

3.2 Runtime

3.2.1 Overview

STMicroelectronics distribution uses the IPCC peripheral for inter-processor communication with the following configuration:

- IPCC processor 1 interface is assigned to Arm®Cortex®-A7 non-secure context and handled by Linux mailbox framework.
- IPCC processor 2 interface is assigned to Arm®Cortex®-M4 context and handled by the IPCC HAL driver.



3.2.2 Software frameworks

Domain	Peripheral	Software frameworks	Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)	

Domain	Peripheral	Software frameworks			Comment
Coprocessor	IPCC		Linux mailbox framework	STM32Cube IPCC 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.

The IPCC peripheral is shared between the Arm Cortex-A and Cortex-M contexts. A particular attention must therefore be paid to have a complementary configuration on both contexts. In STMicroelectronics distribution, the IPCC is configured as described below. To ensure the coherency of the system, it is recommended to keep this configuration unchanged in your implementation.

- Processor interface

Processor	Context	
Cortex-A7 non-secure (Linux)	**Cortex-M4** (STM32Cube)	
Processor 1 interface		
Processor 2 interface		

- Channel allocation

Chann	Mode	Usage	Software client frameworks	
Cortex-A7 non-secure (Linux)	**Cortex-M4** (STM32Cube)			
Channel 1	Full-duplex comm	RPMsg transfer from Cortex-M to Cortex-A • The Cortex-M core uses this channel to indicate that a message is available	RPMsg framework	OpenAMP

Channel	Mode	Usage	Software client frameworks	
	unication	• The Cortex-A core uses this channel to indicate that the message is treated		
Channel 2	Full-duplex communication	RPMsg transfer from Cortex-A to Cortex-M • The Cortex-A core uses this channel to indicate that a message is available • The Cortex-M core uses this channel to indicate that the message is treated	RPmsg framework	OpenAMP
Channel 3	Simplex communication	Cortex-M4 shutdown request	RemoteProc framework	CprocSync cube utility
Channel 4		free		
Channel 5		free		
Channel 6		free		

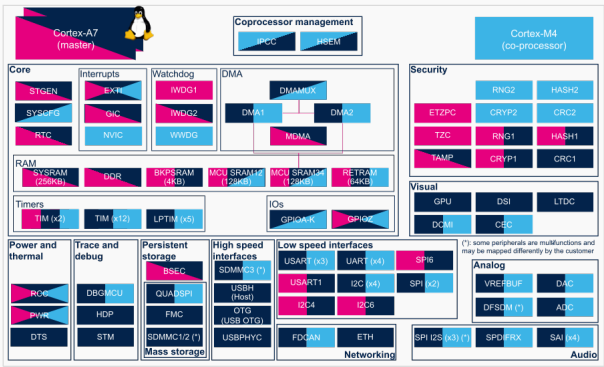
3.2.4 Peripheral assignment

It does not make sense to allocate the IPCC to a single runtime execution context. It is consequently enabled by default for both cores in the STM32CubeMX.

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	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Coprocessor	IPCC	IPCC				Shared (none or both)

4 References

Inter-Processor Communication Controller

Receive

Transmit

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.

Remote Processor Messaging

Stable: 16.02.2021 - 17:39 / Revision: 16.02.2021 - 17:00

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

This article gives information about the Linux® Remote Processor Messaging (RPMsg) framework. The RPMsg framework is a virtio-based messaging bus that allows a local processor to communicate with remote processors available on the system.

Contents

1 Framework purpose 22
2 System overview 23
2.1 Component description 24
2.2 RPMsg definitions 24
2.3 API description 24
3 Configuration 25
3.1 Kernel configuration 25
3.2 Device tree configuration 25
4 How to use the framework 26
5 How to trace and debug the framework 27
5.1 How to trace 27
6 References 28

1 Framework purpose

The Linux®RPMsg framework is a messaging mechanism implemented on top of the virtio framework[1][2] to communicate with a remote processor. It is based on virtio vrings to send/receive messages to/from the remote CPU over shared memory.

The vrings are uni-directional, one vring is dedicated to messages sent to the remote processor, and the other vring is used for messages received from the remote processor. In addition, shared buffers are created in memory space visible to both processors.

The Mailbox framework is then used to notify cores when new messages are waiting in the shared buffers.

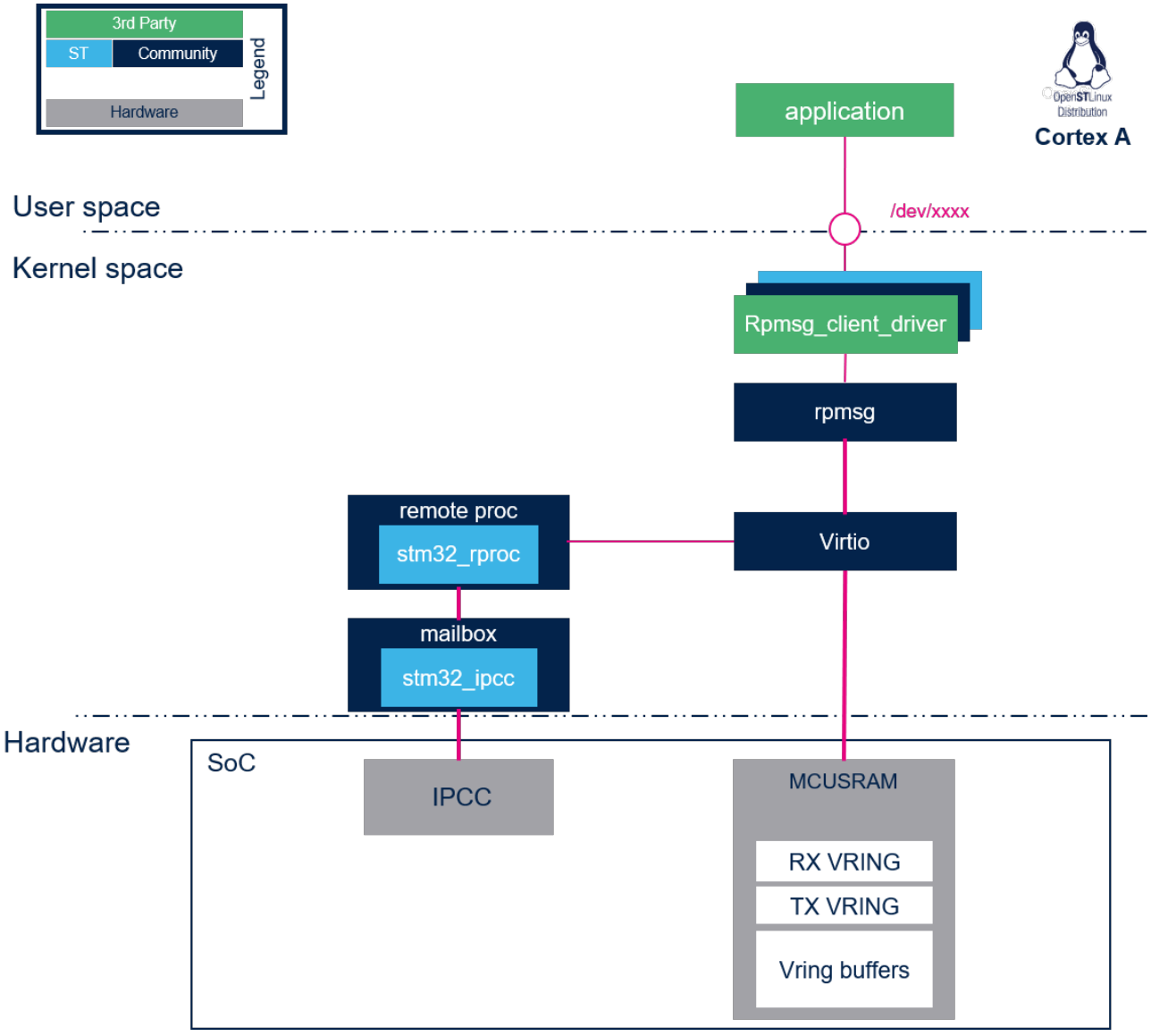
Relying on these frameworks, The RPMsg framework implements a communication based on channels. The channels are identified by a textual name and have a local (“source”) RPMsg address, and a remote (“destination”) RPMsg address”.

On the remote processor side, a RPMSG framework must also be implemented. Several solutions exist, we recommend using OpenAMP

Overviews of the communication mechanisms involved are available at:

- OpenAMP wiki [3][4]

2 System overview



2.1 Component description

- **remoteproc**: The remoteproc framework allows different platforms/architectures to control (power on, load firmware, power off) the remote processors. This framework also adds rpmsg virtio devices for remote processors that support the RPMsg protocol. More details on this framework are available in the remote proc framework page.
- **virtio**: VirtIO framework that supports virtualization. It provides an efficient transport layer based on a shared ring buffer (vring). For more details about this framework please refer to the link below:
  - Virtio: An I/O virtualization framework for Linux [1]
  - virtio introduction - SlideShare [2]
- **rpmsg**: A virtio-based messaging bus that allows kernel drivers to communicate with remote processors available on the system. It provides the messaging infrastructure, facilitating the writing of wire-protocol messages by client drivers. Client drivers can then, in turn, expose appropriate user space interfaces if needed.
- **rpmsg_client_driver** is the client driver that implements a service associated to the remote processor. This driver is probed by the RPMsg framework when an associated service is requested by a remote processor using a "new service announcement" RPMsg message.

2.2 RPMsg definitions

To implement an Rpmsg client, **channel** and **endpoint** concepts need to be understood for a good understanding of the framework.

- RPMsg channel:

An RPMsg client is associated to a communication channel between master and remote processors. This RPMsg client is identified by the textual **service name**, registered in the RPMsg framework. The communication channel is established when a match is found between the local service name registered and the remote service announced.

- RPMsg endpoint:

The RPMsg endpoints provide logical connections through an RPMsg channel. An RPMsg endpoint has a unique source address and associated call back function, allowing the user to bind multiple endpoints on the same channel. When a client driver creates an endpoint with the local address, all the inbound messages with a destination address equal to the endpoint local address are routed to that endpoint. Notice that every channel has a default endpoint, which enables applications to communicate without even creating new endpoints.

2.3 API description

The User API usage is described in Linux kernel RPMsg documentation [5]

3 Configuration

3.1 Kernel configuration

The RPMsg framework is automatically enabled when the remote STM32_RPROC configuration is activated.

3.2 Device tree configuration

No device tree configuration is needed. The Memory region allocated for rpmsg buffers is declared in the remote proc framework device tree.

4 How to use the framework

The Rpmsg framework is used by linux driver client. For details and an example of a simple client, please refer to associated Linux documentation [5]

5 How to trace and debug the framework

5.1 How to trace

RPMsg and virtio dynamic debug traces can be added using the following commands:

```
echo -n 'file virtio_rpmsg_bus.c +p' > /sys/kernel/debug/dynamic_debug/control
echo -n 'file virtio_ring.c +p' > /sys/kernel/debug/dynamic_debug/control
```

6 References

- 1.01.1 An I/O virtualization framework for Linux
- 2.02.1 virtio introduction - SlideShare
- RPMsg Messaging Protocol
- RPMsg Communication Flow
- 5.05.1 Linux kernel rpmsg documentation

Remote Processor Messaging

Central processing unit

Application programming interface

Stable: 07.12.2020 - 10:37 / Revision: 07.12.2020 - 10:35

A quality version of this page, approved on *7 December 2020*, was based off this revision.

This article gives information about the Linux® remoteproc framework.

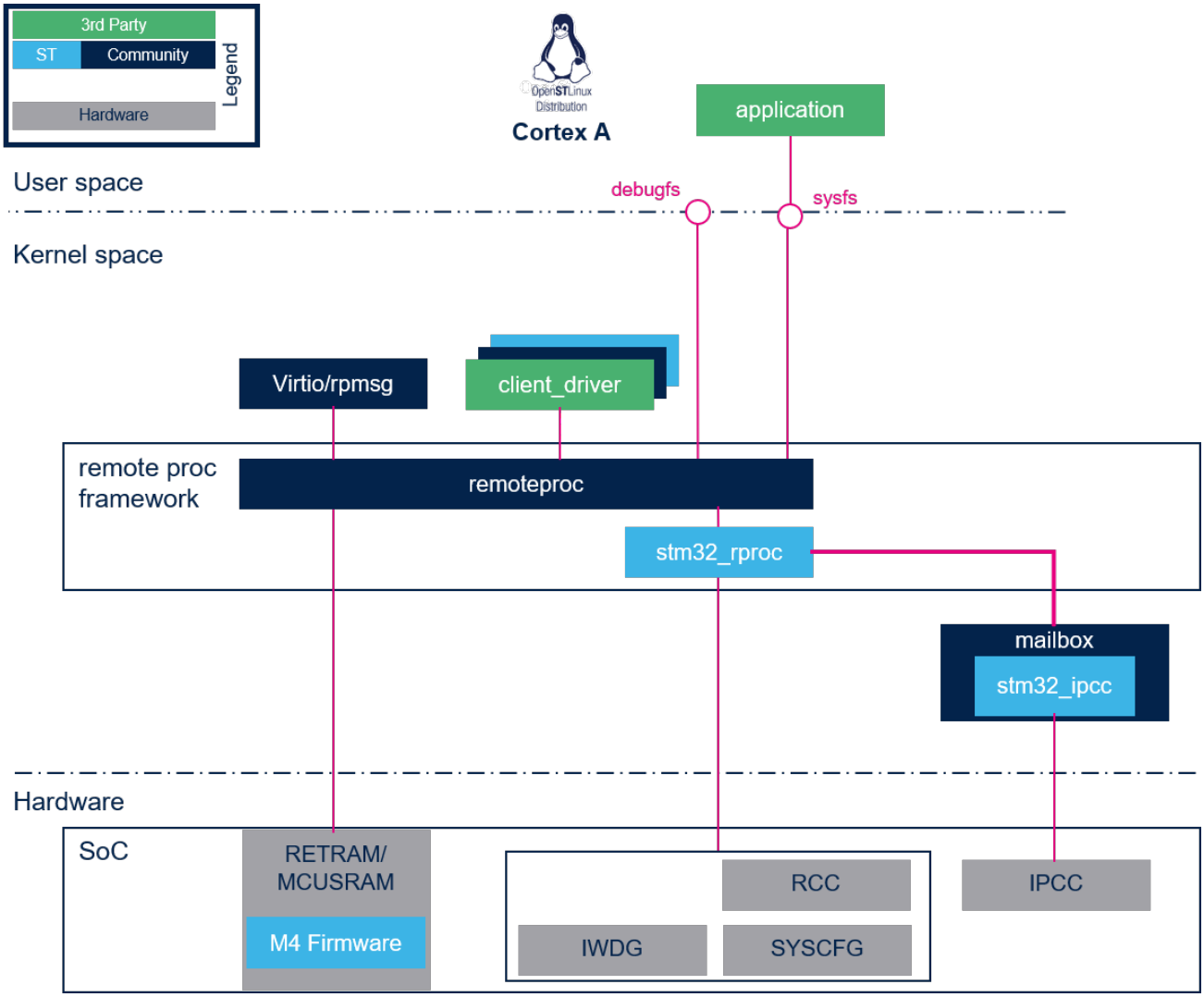
Contents

1 Framework purpose 29
2 System overview 30
2.1 Component description 30
2.2 API description 31
3 Configuration 32
3.1 Kernel configuration 32
3.2 Device tree configuration 32
4 How to use the framework 34
4.1 Remote processor boot 34
4.1.1 Remote processor boot through sysfs 34
4.1.2 Remote processor 'auto' boot (not recommended by STMicroelectronics) 34
4.1.3 Remote processor 'early' boot 35
4.2 Remote processor stop 35
5 How to trace and debug the framework 36
5.1 How to monitor 36
5.2 How to trace 36
6 References 37

1 Framework purpose

The remote processor (RPROC) framework allows the different platforms/architectures to control (power on, load firmware, power off) remote processors while abstracting the hardware differences. In addition it offers services to monitor and debug the remote coprocessor.

2 System overview



2.1 Component description

remoteproc: this is the remote processor framework generic part. Its role is to:

- Load the ELF firmware in the remote processor memory.
- Parse the firmware resource table to set associated resources (such as IPC, memory carveout and traces).
- Control the remote processor execution (start, stop...).
- Provide a service to monitor and debug the remote firmware.

stm32_rproc: this is the remote processor platform driver. Its role is to:

- Register the vendor specific functions (callback) to the RPROC framework.
- Handle the platform resources associated to the remote processor (such as registers, watchdogs, reset, clock and memories).
- Forward notifications (kicks) to the remote processor through the mailbox framework.

2.2 API description

The API usage and remote processor binary firmware structure (resource table, ...) are described in the Linux kernel remoteproc documentation [1].

3 Configuration

Warning

The remoteproc framework needs the mailbox framework to be configured. Refer to mailbox kernel configuration for details.

3.1 Kernel configuration

Activate the remoteproc driver and framework in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel.

```
Device drivers  --->
   Remoteproc drivers  --->
     <*> Support for Remote Processor subsystem
     <*> STM32 remoteproc support
```

3.2 Device tree configuration

The *STM32 remoteproc bindings*[2] documentation deals with all required or optional STM32 remoteproc DT properties.

It also introduces *memory regions* properties that define the RETRAM and MCUSRAM base addresses and sizes in RETRAM and MCUSRAM, from the Arm®Cortex®-A point of view..
Simplified example:

```
/* Memory region declaration, containing vring and rpmsg buffers */
reserved-memory {
        /* RETRAM memory region reserved for firmware code and data */
        retram: retram@0x38000000 {
                reg = <0x38000000 0x10000>;
        };
        /* MCUSRAM memory region reserved for firmware code and data */
        mcusram: mcusram@0x10000000 {
                reg = <0x10000000 0x40000>;
        };
        /* MCUSRAM aliased memory region reserved for firmware code and data */
        mcusram2: mcusram2@0x30000000 {
                reg = <0x30000000 0x40000>;
        };
```

```
 /* stm32 M4 remoteproc device */
 m4_rproc: m4@0 {
        ...
        memory-region = <&retram>, <&mcusram>, <&mcusram2>, <&vdev0vring0>,
                        <&vdev0vring1>, <&vdev0buffer>;
        ...
};
```

Information

The firmware memory mapping must be set according to these values in the STM32Cube firmware linker script.

For additional details, please refer to STM32MP15 Memory mapping.

4 How to use the framework

4.1 Remote processor boot

There are three possibilities to load and start the remote processor firmware:

- Start the firmware through the SysFS interface.
- Automatically start the firmware on remoteproc driver probing (not recommended by STMicroelectronics).
- Early boot the firmware during boot time (before Linux boot).

4.1.1 Remote processor boot through sysfs

- The firmware components are stored in the file system, by default in the **/lib/firmware/** folder. Optionally another location can be set. In this case the remoteproc framework parses this new path in priority.

Below the command for adding a new path for firmware parsing:

```
Board $> echo -n <firmware_path> > /sys/module/firmware_class/parameters/path
```

Warning

This path is common for all firmwares loaded by Linux (Bluetooth, Wifi...)

- If the firmware elf filename differs from the default one (rproc-%s-fw), set the name with the following command: (replace X with remoteproc instance number: 0 by default)

```
Board $> echo -n <firmware_name.elf> > /sys/class/remoteproc/remoteprocX/firmware
```

- To start the firmware, use the following command:

```
Board $> echo start >/sys/class/remoteproc/remoteprocX/state
```

Information

Based on the above commands, a userland service can be implemented to automatically load the firmware during the userland initialization phase.

4.1.2 Remote processor 'auto' boot (not recommended by STMicroelectronics)

The remote processor can be automatically booted during platform boot. To do this, the following conditions must be fulfilled:

- The firmware must be present in **/lib/firmware** before the remoteproc driver is probed.
- The filesystem on Linux (Cortex-A) must be available before the remoteproc driver is probed. However, in normal conditions, the remoteproc driver is probed before the filesystem is mounted, and the firmware is consequently not available during the Linux driver probing phase. Possible solutions could be:
  - to use an initramfs[3]
  - or compile remoteproc as a module and not as kernel built-in driver.

```
*The firmware must be named rproc-%s-fw, where %s corresponds to the name of the
remoteproc node in the device tree. For example, for rproc-m4-fw, the remoteproc device
tree must be defined as follows:
m4 {
        compatible = "st,stm32mp1-rproc";
[...]
        status = "okay";
};
```

- The "auto_boot" property has to be defined in the remoteproc node device tree:

```
m4 {
        compatible = "st,stm32mp1-rproc";
[...]
        auto_boot;
        status = "okay";
};
```

4.1.3 Remote processor 'early' boot

The coprocessor can be started by the second stage bootloader (eg U-Boot). This mode allows to start the coprocessor firmware before the Linux one. For instance, it can be used to execute first actions for projects that have hard constraints on boot time. On Linux boot, the remoteproc framework attaches itself to the firmware by parsing the resource table, based on the information added by the bootloader in the backup registers (Cortex-M4 state and resource table address). Refer to How to start the coprocessor from the bootloader for details on this mode.

4.2 Remote processor stop

It is possible to stop the remote processor firmware through the SysFS interface. On stop request, the stm32_rproc driver:

- informs the remote firmware relying on the "shutdown" channel of the the IPCC mailbox. This mechanism allows the remote processor firmware to shut down properly.
- resets the coprocessor, on "shutdown" message acknowledgement or after a timeout of 500 ms.

Information

The use of the IPCC "shutdown" channel is optional. If the mailbox channel is not declared in the device trree, the remote processor is immediately reset, without informing firmware of the remote processor.

To stop the firmware, use the following command:

```
Board $> echo stop >/sys/class/remoteproc/remoteprocX/state
```

5 How to trace and debug the framework

5.1 How to monitor

- The remoteproc firmware state can be monitored using following command:

```
Board $> cat /sys/class/remoteproc/remoteprocX/state
```

5.2 How to trace

- remoteproc framework and driver debug traces can be added in the kernel logs thanks to the dynamic debug mechanism:

```
Board $> echo -n 'file stm32_rproc.c +p' > /sys/kernel/debug/dynamic_debug/control
Board $> echo -n 'file remoteproc*.c +p' > /sys/kernel/debug/dynamic_debug/control
```

- A log buffer can de defined in the remoteproc firmware and declared in the resource table. If the feature is activated on the remote firmware, log traces can be dumped from the trace buffer using the following command:

```
Board $>cat /sys/kernel/debug/remoteproc/remoteprocX/trace0
```

6 References

- Linux kernel remoteproc documentation
- Documentation/devicetree/bindings/remoteproc/stm32-rproc.txt , Linux Foundation, STM32 remoteproc DT bindings
- ramfs-rootfs-initramfs Linux documentation

Linux® is a registered trademark of Linus Torvalds.

Executable and linkable file

Inter-Processor Communication

Application programming interface

Device Tree

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

Cortex®

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

Initial ramdisk (https://en.wikipedia.org/wiki/Initial_ramdisk)

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

Inter-Processor Communication Controller

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 38
2 Menuconfig and Developer Package 40
3 Menuconfig and Distribution Package 42
4 References 43

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)
```

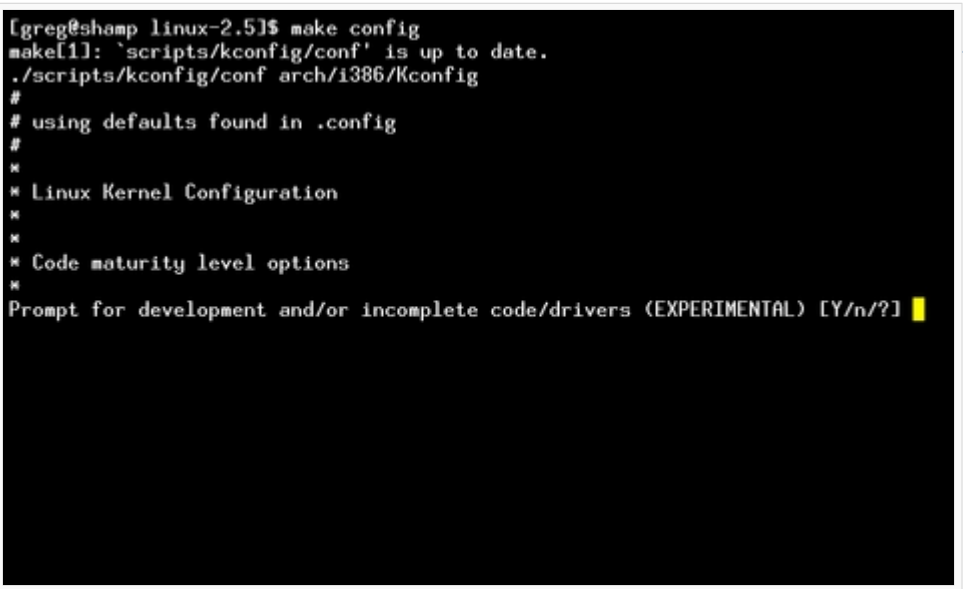


Figure 1. Configuring the kernel with make config

```
   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 :

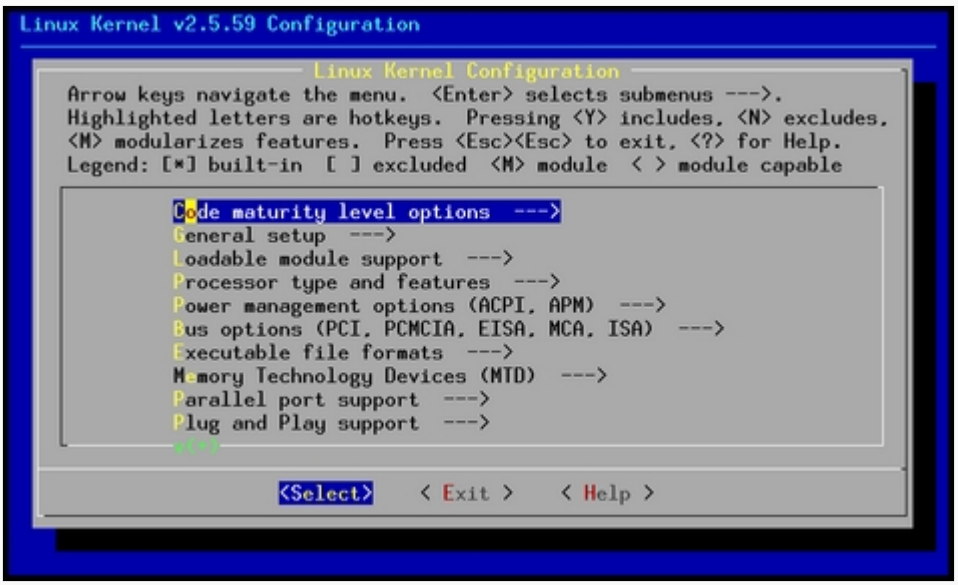


Figure 2. Make menuconfig makes it easier to back up and correct mistakes

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.

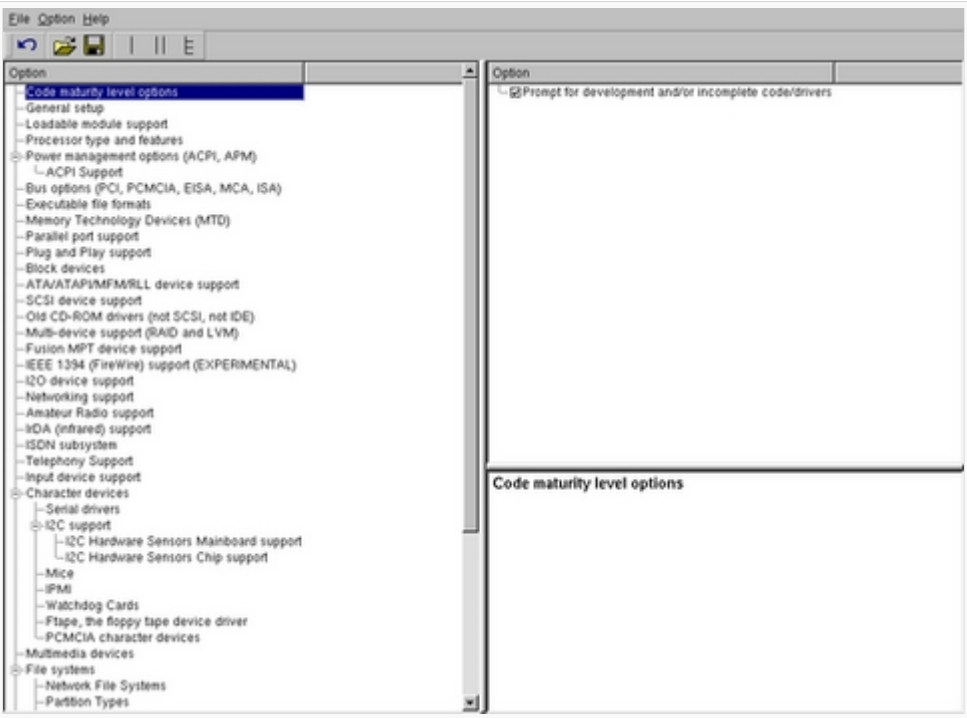


Figure 3. The Qt-Based make xconfig

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.)