Category:IIO

Contents

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 07:55

Stable: 17.06.2020 - 10:20 | Revision: 16.01.2020 - 07:38

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

This category groups together all articles related to the Linux®**IIO** software framework.

It is recommended to first read the IIO overview article.

Linux® is a registered trademark of Linus Torvalds.

Industrial I/O Linux subsystem

Pages in category "IIO"

The following 14 pages are in this category, out of 14 total.

- IIO overview
- IIO libiio
- ADC device tree configuration
- ADC Linux driver
- DAC device tree configuration
- DAC Linux driver
- DFSDM device tree configuration
- DFSDM Linux driver
- How to use the IIO user space interface
- IIO Linux kernel tools
- LPTIM device tree configuration
- LPTIM Linux driver
- TIM device tree configuration
- TIM Linux driver

Stable: 16.01.2020 - 15:02 / Revision: 16.01.2020 - 14:59

A quality version of this page, approved on *16 January 2020*, was based off this revision.

Contents

1 Article purpose

This article introduces the Linux® driver for the ADC[1] internal peripheral:

- Which ADC features are supported by the driver
- How to configure, use and debug the driver
- What is the driver structure, and where the source code can be found.

2 Short description

The ADC Linux® driver (kernel space) is based on the IIO framework. It supports two modes:

1. **IIO direct mode**: single capture on a channel (using interrupts)
2. **IIO triggered buffer mode**: capture on one or more channels (preferably using DMA).

It uses the hardware triggers available in IIO. See TIM Linux driver and LPTIM Linux driver.

3 Configuration

3.1 Kernel configuration

Activate the ADC[1]Linux® driver in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel (enable both CONFIG_STM32_ADC_CORE and CONFIG_STM32_ADC).

```
Device Drivers  --->
   <*> Industrial I/O support  --->
      Analog to digital converters  --->
         <*> STMicroelectronics STM32 adc core
         <*>   STMicroelectronics STM32 adc
```

3.2 Device tree

Refer to the ADC device tree configuration article when configuring the ADC Linux kernel driver.

4 How to use

In "**IIO direct mode**", the conversion result can be read directly from **sysfs** (refer to How to do a simple ADC conversion using the sysfs interface).

In "**IIO triggered buffer mode**", the configuration must be performed using **sysfs** first. Then, **character device** (/dev/iio:deviceX) is used to read data (refer to Convert one or more channels using triggered buffer mode).

5 How to trace and debug

Refer to How to trace with dynamic debug for how to **enable the debug logs** in the driver and in the framework.

Refer to How to debug with debugfs for how to **access the ADC registers**.

The ADC has system wide dependencies towards other key resources:

- **runtime power management** can be disabled, for example it may be forced **on** via *power/control* sysfs entry:

```
Board $> cd /sys/devices/platform/soc/48003000.adc/48003000.adc:adc@0
Board $> cat power/autosuspend_delay_ms
2000
Board $> cat power/control
auto                                    # kernel is allowed to automatically suspend the
ADC device after autosuspend_delay_ms
Board $> echo on > power/control        # force the kernel to resume the ADC device (e.
g. keep clocks and regulators enabled)
```

Information

It might be useful to disable runtime power management, in order to dump registers by any means or to check clock and regulator usage (see example below).

- **clock**[2] usage can be verified by reading *clk_summary*:

```
Board $> cat /sys/kernel/debug/clk/clk_summary | grep adc
         adc12_k                     1        1        0    24000000          0 0
                  adc12              1        1        0   196607910          0 0
```

- **regulator**[3] tree and usage can be verified (e.g. use count, open count or regulator reference voltage) as follows:

```
Board $> cat /sys/kernel/debug/regulator/regulator_summary
 regulator                      use open bypass voltage current     min     max
---------------------------------------------------------------------------------
    v3v3                          4    5      0  3300mV     0mA  3300mV  3300mV
       vdda                       1    2      0  2900mV     0mA  2900mV  2900mV
          40017000.dac                                              0mV     0mV
          48003000.adc                                              0mV     0mV
```

- **pinctrl**[4] usage can be verified by reading *pinmux-pins*:

```
Board $> cd /sys/kernel/debug/pinctrl/soc\:pin-controller@50002000/
Board $> cat pinmux-pins | grep adc
pin 92 (PF12): device 48003000.adc function analog group PF12   # check pin is assigned
to ADC and is configured as "analog"
```

- **interrupts** can be verified by reading "interrupts":

```
Board $> cat /proc/interrupts
           CPU0       CPU1
 56:          2          0     dummy   0 Edge      48003000.adc:adc@0
```

6 Source code location

The ADC source code is composed of:

- stm32-adc-core driver to handle common resources such as clock (selection, prescaler), regulator used as reference voltage, interrupt and common registers.
- stm32-adc driver to handle the resources available for each ADC such as channel configuration and buffer handling.

7 References

- 1.01.1 ADC internal peripheral
- Clock overview
- Regulator overview
- Pinctrl overview

Linux® is a registered trademark of Linus Torvalds.

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Industrial I/O Linux subsystem

Direct Memory Access

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

Stable: 04.11.2020 - 13:06 Revision: 04.11.2020 - 13:03

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

Contents

1 Article purpose

The purpose of this article is to explain how to configure the analog-to-digital converter (*ADC*)[1] **when the peripheral is assigned to Linux®OS**, and in particular:

- how to configure the ADC **peripheral**
- how to configure the **board**, e.g. the ADC voltage reference regulator, channels, pins and sampling time.

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

It is used by the ADC Linux driver that registers relevant information in IIO framework, such as IIO devices, channels and voltage scale for each ADC.

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.

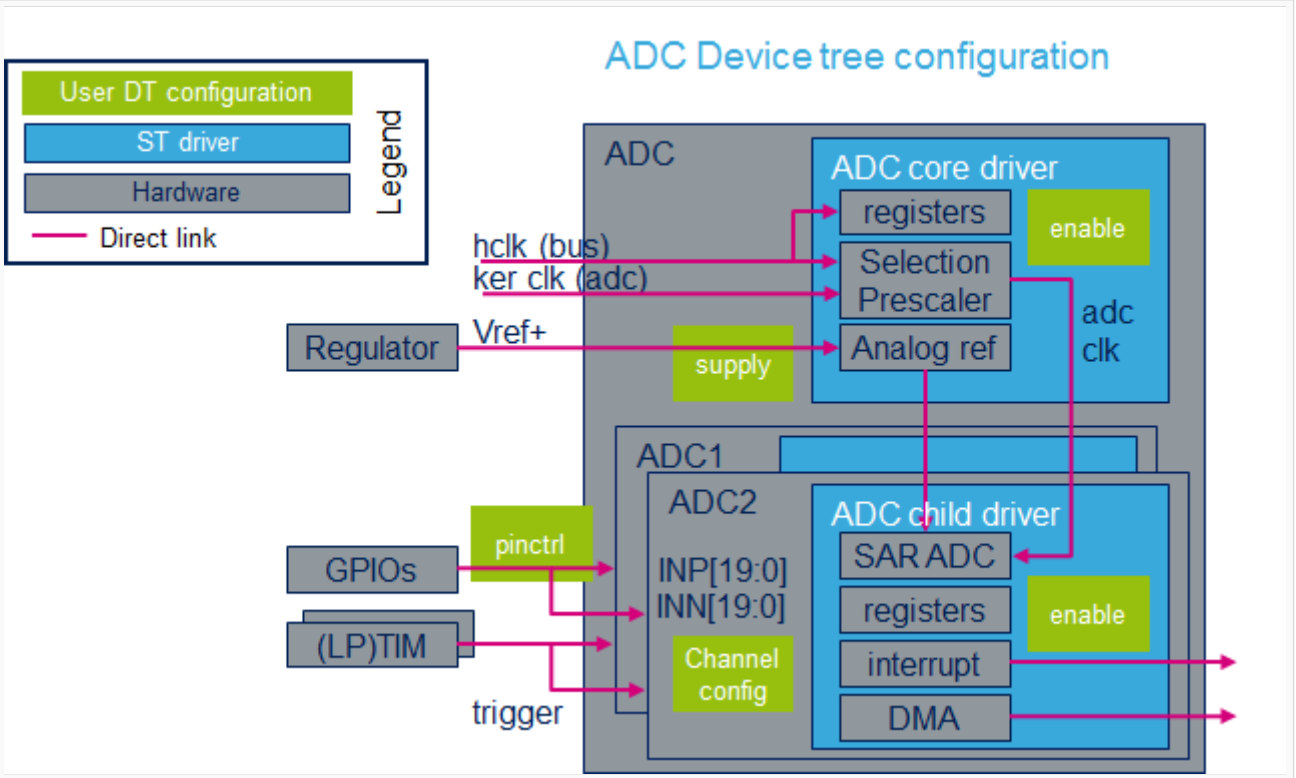
2 DT bindings documentation

STM32 ADC device tree bindings[3] describe all the required and optional functions.

3 DT configuration

This hardware description is a combination of STM32 microprocessor and board device tree files. See Device tree for more explanations on device tree file split.

The **STM32CubeMX** can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.



ADC DT configuration

3.1 DT configuration (STM32 level)

The ADC nodes are declared in stm32mp151.dtsi[4].

- **DT root node** ('**adc**') describes the ADC hardware block parameters such as register areas, clocks and interrupts.
- **DT child nodes** ('**adc1**' and '**adc2**') describe ADC1 and ADC2 independently.

```
adc: adc@address {
        compatible = "st,stm32mp1-adc-core";
        ...                                          /* common resources in 'adc' root node.
*/
        adc1: adc@0 {
                compatible = "st,stm32mp1-adc";
                ...                                  /* private resources in 'adc1' child
node. */
        };
```

```
        adc2: adc@100 {
                compatible = "st,stm32mp1-adc";
                ...                                     /* private resources in 'adc2' child
node. */
        };
};
```

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

Follow the sequences described in the below chapters to configure and enable the ADC on your board.

3.2.1 Common resources for all ADCs

The **DT root node** ('**adc**') must be filled in:

- Enable the ADC block by setting **status = "okay".**
- Configure the pins in use via pinctrl, through **pinctrl-0** and **pinctrl-names**.
- Configure the analog supply voltage regulator[5] by setting **vdda-supply = <&your_vdda_regulator>**.
- Configure the analog reference voltage regulator[5] by setting **vref-supply = <&your_vref_regulator>**.

Information

The ADC can use the internal VREFBUF[6] or any other external regulator[5] wired to VREF+ pin.

3.2.2 Resources dedicated to ADC1 and ADC2

The **DT child nodes** ('**adc1**' and/or '**adc2**') must be filled in:

- Enable 'adc1' and/or 'adc2' by setting **status = "okay".**
- Enable single-ended channel(s) (*<vinp...>*) by setting **st,adc-channels = <0 1 2...>.**
- Enable differential channel(s) pairs (*<vinp vinn>, ...*) by setting **st,adc-diff-channels = <1 0>, <2 6>, ...**
- Set the minimum sampling time [7] for each or all channels by setting **st,min-sample-time-nsecs = <10000>** (optional).
- Set the resolution by setting **assigned-resolution-bits = <12>** (optional).

3.3 DT configuration example

The example below shows how to configure ADC1:

- Input pin: use Pinctrl device tree configuration to configure PF12 as analog input.
- Analog supply: it is provided by one of the PMIC LDO regulators.
- Voltage reference: it is provided by the VREFBUF internal regulator.
- Input channel: configure ADC1_IN6 (e.g on PF12).
- Sampling time: the minimum sampling time is 10 µs.

```
# part of pin-controller dt node
adc1_in6_pins_a: adc1-in6 {
        pins {
                pinmux = <STM32_PINMUX('F', 12, ANALOG)>; /* configure 'PF12' as ANALOG */
        };
};
```

```
&adc {
        /* ADC1 & ADC2 common resources */
        pinctrl-names = "default";
        pinctrl-0 = <&adc1_in6_pins_a>;                 /* Use PF12 pin as ANALOG */
        vdda-supply = <&vdda>;                          /* Example to supply vdda pin by
using a PMIC regulator
        vref-supply = <&vrefbuf>;                       /* Example to use VREFBUF (It needs
to be enabled as well) */
        status = "okay";                                /* Enable ADC12 block */
        adc1: adc@0 {
                /* private resources for ADC1 */
                st,adc-channels = <6>;                  /* ADC1 in6 channel is used */
                st,min-sample-time-nsecs = <10000>;     /* 10µs sampling time */
                status = "okay";                        /* Enable ADC1 */
        };
        adc2: adc@100 {
                /* private resources for ADC2 */
                ...
        };
};
```

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

For additional information, refer to the following links:

- ADC internal peripheral
- Device tree
- Documentation/devicetree/bindings/iio/adc/st,stm32-adc.txt , STM32 ADC device tree bindings
- STM32MP151 device tree file
- 5.05.15.2 Regulator overview
- VREFBUF internal peripheral
- How to get the best ADC accuracy in STM32, by STMicroelectronics

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Linux® is a registered trademark of Linus Torvalds.

Operating System

Industrial I/O Linux subsystem

Device Tree

voltage reference buffer (STM32 specific)

Low-dropout regulator

Power Management Integrated Circuit

Stable: 16.01.2020 - 15:01 / Revision: 16.01.2020 - 14:56

A quality version of this page, approved on *16 January 2020*, was based off this revision.

Contents

1 Article purpose

This article introduces the Linux® driver for the DAC[1] internal peripheral:

- Which DAC features are supported by the driver
- How to configure, use and debug the driver
- What is the driver structure, and where the source code can be found.

2 Short Description

The DAC Linux® driver (kernel space) is based on the IIO framework.

It implements the **IIO direct mode**, to perform single conversions independently on each channel.

3 Configuration

3.1 Kernel configuration

Activate the DAC[1]Linux® driver in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel (enable CONFIG_STM32_DAC).

```
Device Drivers  --->
   <*> Industrial I/O support  --->
      Digital to analog converters  --->
         <*> STMicroelectronics STM32 DAC
```

3.2 Device tree

Refer to the DAC device tree configuration article when configuring the DAC Linux kernel driver.

4 How to use

In "IIO direct mode", conversions can be done directly via sysfs. See How to do a simple DAC conversion using the sysfs interface.

5 How to trace and debug

Refer to How to trace with dynamic debug for how to enable debug logs in the driver and in the Framework.

Refer to How to debug with debugfs for how to access the DAC registers.

The DAC has system wide dependencies towards other key resources:

- **runtime power management** can be disabled, for example it may be forced **on** via *power/control* sysfs entry:

```
Board $> cd /sys/devices/platform/soc/40017000.dac/40017000.dac\:dac@1/
Board $> cat power/autosuspend_delay_ms
2000
Board $> cat power/control
auto                                    # kernel is allowed to automatically suspend the
ADC device after autosuspend_delay_ms
Board $> echo on > power/control        # force the kernel to resume the DAC device (e.
g. keep clocks and regulators enabled)
```

Information

It might be useful to disable runtime power management, in order to dump registers by any means or to check clock and regulator usage (see example below).

- **clock**[2] usage can be verified by reading *clk_summary*:

```
Board $> cat /sys/kernel/debug/clk/clk_summary | grep dac
       dac12_k                         0        0        0       32000          0 0
                    dac12              1        2        0    98303955          0 0
```

- **regulator**[3] tree and usage usage can be verified (e.g. use count, open count and regulator reference voltage) as follows:

```
Board $> cat /sys/kernel/debug/regulator/regulator_summary
 regulator                      use open bypass voltage current     min     max
---------------------------------------------------------------------------------
    v3v3                          4    5      0  3300mV     0mA  3300mV  3300mV
       vdda                       1    2      0  2900mV     0mA  2900mV  2900mV
          40017000.dac                                             0mV     0mV
          48003000.adc                                             0mV     0mV
```

- **pinctrl**[4] usage can be verified by reading *pinmux-pins*:

```
Board $> cd /sys/kernel/debug/pinctrl/soc\:pin-controller@50002000/
Board $> cat pinmux-pins | grep dac
pin 4 (PA4): device 40017000.dac function analog group PA4
pin 5 (PA5): device 40017000.dac function analog group PA5   # check pins are assigned to
DAC and configured as "analog"
```

6 Source code location

The DAC source code is composed of:

- stm32-dac-core driver to handle common resources such as clock or regulator used as reference voltage and common registers.
- stm32-dac driver to handle the resources available for each DAC such as channel configuration or output buffer handling (power-down mode).

7 References

- 1.01.1 DAC internal peripheral
- Clock overview
- Regulator overview
- Pinctrl overview

Linux® is a registered trademark of Linus Torvalds.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Industrial I/O Linux subsystem

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

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Stable: 04.11.2020 - 13:17 / Revision: 04.11.2020 - 13:14

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

Contents

1 Purpose

The purpose of this article is to explain how to configure the digital-to-analog converter (*DAC*)[1] **when the peripheral is assigned to Linux®OS**, and in particular:

- how to configure and enable the DAC **peripheral**
- how to configure the **board**, the DAC channels, reference voltage regulator and pins.

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

It is used by the DAC Linux driver that registers relevant information in IIO framework, such as IIO devices, channels and voltage scale.

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

STM32 DAC device tree bindings[3] deal with all the required or optional properties.

3 DT configuration

This hardware description is a combination of STM32 and board device tree files. See Device tree for explanations on device tree file split.

The **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)

The DAC nodes are declared in stm32mp151.dtsi[4]:

- **DT root node** ('**dac**') describes the DAC hardware block parameters such as registers area and clocks.
- **DT child nodes** ('**dac1**' and '**dac2**') describe DAC channels independently.

```
dac: dac@address {
        compatible = "st,stm32h7-dac-core";
        ...                                                                     /*
common resources in 'dac' root node. */
        dac1: dac@1 {
                compatible = "st,stm32-dac";
                reg = <1>;                                                      /*
DAC identifier (e.g. 1 for DAC1) */
                ...                                                             /*
private resources in 'dac1' child node. */
        };
        dac2: dac@2 {
                compatible = "st,stm32-dac";
                reg = <2>;                                                      /*
DAC identifier (e.g. 2 for DAC2) */
                ...                                                             /*
private resources in 'dac2' child node. */
        };
};
```

Warning

This device tree part is related to STM32 microprocessors. It should be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

Follow the below sequence to configure and enable the DAC on your board:

- Enable **DT root node** named '**dac**' by setting **status = "okay"**.
- Configure pins in use via pinctrl through **pinctrl-0** and **pinctrl-names**.
- Configure analog reference voltage regulator[5] by setting **vref-supply = <&your_regulator>**.
- Enable **DT child node(s)** for '**dac1**' and/or '**dac2**' channels(s) in use by setting **status = "okay"**.

Information

The DAC can use the internal VREFBUF[6] or any other external regulator[5] wired to the VREF+ pin

.

3.3 DT configuration example

The example below shows how to configure DAC1 and DAC2 channels:

- PA4 and PA5 pins both configured as analog pins (see Pinctrl device tree configuration for more details)
- VREFBUF[6] used as reference voltage

```
dac_ch1_pins_a: dac-ch1 {
        pins {
                pinmux = <STM32_PINMUX('A', 4, ANALOG)>;                /* configure
'PA4' as ANALOG */
        };
};
dac_ch2_pins_a: dac-ch2 {
        pins {
                pinmux = <STM32_PINMUX('A', 5, ANALOG)>;                /* configure
'PA5' as ANALOG */
        };
};
```

```
&dac {
        pinctrl-names = "default";
        pinctrl-0 = <&dac_ch1_pins_a &dac_ch2_pins_a>;                  /* Use PA4 and
PA5 pin as ANALOG */
        vref-supply = <&vrefbuf>;                                       /* Example to
use VREFBUF (It needs to be enabled as well) */
        status = "okay";                                                /* Enable the DAC
 block */
        dac1: dac@1 {
                status = "okay";                                        /* Enable DAC1 */
        };
        dac2: dac@2 {
                status = "okay";                                        /* Enable DAC2 */
        };
};
```

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

For additional information, refer to the following links:

- DAC internal peripheral
- Device tree
- Documentation/devicetree/bindings/iio/dac/st,stm32-dac.txt , STM32 DAC device tree bindings
- STM32MP151 device tree file
- 5.05.1 Regulator overview
- 6.06.1 VREFBUF internal peripheral

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Linux® is a registered trademark of Linus Torvalds.

Operating System

Industrial I/O Linux subsystem

Device Tree

voltage reference buffer (STM32 specific)

Stable: 16.01.2020 - 15:11 / Revision: 16.01.2020 - 15:09

A quality version of this page, approved on *16 January 2020*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to introduce the Linux® driver for the DFSDM internal peripheral:

- Which DFSDM features are supported by the driver
- How to configure, use and debug it
- What is the driver structure, where to the find source code.

2 Short Description

The DFSDM Linux® driver (kernel space) is based on the IIO and ALSA frameworks. It offers various modes:

1. **IIO direct mode**: single capture on a channel
2. **IIO software buffer**: capture one or more channels
3. **IIO triggered buffer mode**: capture on one or more channels using triggers

It uses hardware triggers available in IIO. See TIM Linux driver and LPTIM Linux driver.

It offers an extended API[1] for audio usage which allows **PDM microphones capture** through a SPI interface.

3 Configuration

3.1 Kernel configuration

Activate the DFSDM Linux® driver in the kernel configuration by using the menuconfig tool.

3.1.1 IIO driver

Configuration flag: CONFIG_STM32_DFSDM_ADC

```
Device Drivers  --->
   <*> Industrial I/O support  --->
      Analog to digital converters  --->
         <*> STMicroelectronics STM32 dfsdm adc
         <*>   STMicroelectronics STM32 adc                    # DFSDM ADC IIO driver
```

When using the DFSDM ADC IIO driver (only), the user must also enable the driver for the external analog front-end (e.g. sigma delta modulator). The generic sigma delta modulators driver may be used for instance (CONFIG_SD_ADC _MODULATOR):

```
Device Drivers  --->
   <*> Industrial I/O support  --->
      Analog to digital converters  --->
         <*> Generic sigma delta modulator                    # sigma delta
modulator driver
```

3.1.2 Audio driver

Configuration flag: CONFIG_SND_SOC_STM32_DFSDM (Note: This configuration depends on CONFIG_STM32_DFSDM_ADC)

```
Device Drivers  --->
   <*> Sound card support  --->
      <*>   Advanced Linux Sound Architecture  --->
         <*>   ALSA for SoC audio support  --->
            STMicroelectronics STM32 SOC audio support  --->
               <*> SoC Audio support for STM32 DFSDM              # DFSDM Audio driver
for digital microphone capture
```

3.2 Device tree

Refer to the DFSDM device tree configuration article when configuring the DFSDM Linux kernel driver.

4 How to use

4.1 IIO driver

In "**IIO direct mode**", the conversion result can be read directly from **sysfs**, see: How to do a simple ADC conversion using the sysfs interface.

In "**IIO triggered buffer mode**", the configuration must be performed by using **sysfs** first. Then, **character device** (/dev/iio: deviceX) is used to read data, see Convert one or more channels using triggered buffer mode.

For information on the standard IIO consumer interface, please refer to How to use IIO kernel API which gives an example of the IIO consumer kernel API.

4.2 Audio driver

The DFSDM Linux driver can be accessed from userland through an ALSA device. Refer to ALSA overview for information on how to list and use ALSA devices.

5 How to trace and debug

5.1 How to monitor

The DFSDM driver uses resources such as clocks and GPIOs.

- Refer to Pinctrl_overview#How_to_monitor to check DFSDM GPIOs.
- Refer to Clock_overview#How_to_monitor_with_debugfs to check DFSDM clocks.

5.1.1 How to monitor with debugfs

The DFSDM registers are accessed using REGMAP by *DFSDM Linux® driver*.

It comes with debugfs entries to dump registers:

```
$ cd /sys/kernel/debug/regmap/4400d000.dfsdm/
$ cat registers
000: 00000000
004: 00000000
008: 00000000
```

5.1.2 Other ways to monitor

- Man can check the DFSDM **interrupts** and/or the DFSDM **DMA** interrupts:

```
$ cat /proc/interrupts
           CPU0       CPU1
...
 99:          0          0     GIC-0 142 Level     4400d000.dfsdm:filter@0
100:          0          0     GIC-0 143 Level     4400d000.dfsdm:filter@1
101:          0          0     GIC-0 144 Level     4400d000.dfsdm:filter@2
...
```

5.2 How to trace

5.2.1 IIO driver

Refer to How to trace with dynamic debug for how to enable the debug logs in the driver and in the framework.

```
Board $> dmesg -n8
Board $> echo "file drivers/iio/adc/stm32-dfsdm* +p" > /sys/kernel/debug/dynamic_debug
/control
```

To enable dynamic debug at boot time, append the following arguments on the kernel command line:

```
loglevel=8 dyndbg="file drivers/iio/adc/stm32-dfsdm* +p"
```

5.2.2 Audio driver

Refer to ALSA_overview#How_to_trace for details on trace tools.

5.3 How to debug

5.3.1 Audio driver

Refer to ALSA_overview#How_to_debug for details on debugging tools.

6 Source code location

It is composed of:

- stm32-dfsdm-core.c , core part of DFSDM Linux driver to handle common resources: registers, clock
- stm32-dfsdm-adc.c , ADC part of DFSDM Linux driver to handle **ADC** operations
- stm32_adfsdm.c , ASoC DAI part of DFSDM Linux driver to handle **audio** operations

See also sigma delta modulator driver:

- sd_adc_modulator.c , sigma delta modulator Linux driver to handle analog front-end

7 References

- include/linux/iio/adc/stm32-dfsdm-adc.h , DFSDM IIO custom API

Linux® is a registered trademark of Linus Torvalds.

Digital Filter for Sigma-Delta Modulator

Industrial I/O Linux subsystem

Application programming interface

Serial Peripheral Interface

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Secure digital

Advanced Linux sound architecture

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

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

Register map (Linux registers map abstraction API)

Direct Memory Access

Generic Interrupt Controller

ALSA System on Chip

Digital Audio Interface

Stable: 04.11.2020 - 16:04 / Revision: 04.11.2020 - 15:54

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

Contents

1 Article purpose

The purpose of this article is to explain how to configure the DFSDM internal peripheral when the peripheral is assigned to Linux®OS**, and in particular:**

- How to configure the DFSDM peripheral to enable filters and associated channels
- How to configure the board, e.g. serial interface input/output pins

The configuration is performed using the device tree mechanism.

It is used by the DFSDM Linux driver which registers the relevant information in IIO and ALSA frameworks.

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 DFSDM maybe used as various functions: ADC and DMIC (for audio).

Each one is represented by a separate **compatible string**, documented here:

- *STM32 DFSDM ADC device tree bindings*[1]
- *Audio DFSDM device tree bindings*[2]

The external analog frontend (e.g. sigma-delta modulator) is documented here:

- *Device-Tree bindings for sigma delta modulator*[3]

3 DT configuration

This hardware description is a combination of STM32 microprocessor and board device tree files. See Device tree for more explanations on device tree file split.

The **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)

DFSDM nodes are declared in stm32mp151.dtsi[4].

- DT root node ('dfsdm') describes the ADC hardware block parameters such as registers area, clocks.
- DT child nodes ('dfsdm0', 'dfsdm1', ...) describe each filter independently: compatible string, interrupts, DMAs.

```
dfsdm: dfsdm@4400d000 {
        compatible = "st,stm32mp1-dfsdm";
        ...                                                                    /*
common resources in 'dfsdm' root node. */
        dfsdm0: filter@0 {
                compatible = "st,stm32-dfsdm-adc";                             /*
can either be st,stm32-dfsdm-(adc or dmic) */
                ...                                                            /*
private resources in 'dfsdm0' child node. */
        }
        dfsdm1: filter@1 {
                ...
        }
```

Warning

This device tree part is related to STM32 microprocessors. It should be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

Follow the sequences described in the below chapters to configure and enable the DFSDM on your board.

3.2.1 Common resources for all DFSDM filters

Configure the 'dfsdm' **DT root node**:

- Enable the DT root node for the DFSDM, by setting **status = "okay"**.
- Configure the pins in use via pinctrl, by setting **pinctrl-0**, **pinctrl-1** and **pinctrl-names**.
- Configure the SPI clock output frequency, by setting **spi-max-frequency** (optional: only for SPI master mode).
- Configure the audio clock to be used, by setting **clocks** and **clock-names** (optional: to use more accurate clock for audio).

3.2.2 Private resources for each DFSDM filter

Configure the filter(s) **DT child node(s)**:

- Enable the DT child node(s) for the DFSDM filter(s) in use, by setting **status = "okay"**.
- Override the **compatible** string by setting **"st,stm32-dfsdm-dmic"** (optional: only for audio digital microphone).
- Enable the channel(s), by setting **st,adc-channels = <0 1 2...>**.
- Configure the channel(s) by setting **st,adc-channel-names**, **st,adc-channel-types** (e.g. SPI or manchester) and **st,adc-channel-clk-src** (e.g. external or internal).
- Configure the filter order, by setting **st,filter-order**.

3.2.3 Additional configuration for DFSDM ADC

The DFSDM ADC device has an external analog front-end, the *sigma delta modulator*.

Configure the external sigma delta modulator for each channel (optional, not needed for audio digital microphone):

- Add **your_sd_modulator** DT node in the board dts file (see the generic sd-modulator[3] example here after).
- Add **io-channels = <&your_sd_modulator>** to the DFSDM filter child node in order to assign it to the filter channel(s).

3.2.4 Additional configuration for DFSDM audio

Additional child nodes must be added for audio soundcard configuration.

3.3 DT configuration examples

The example below shows how to configure the DFSDM ADC channel 1, assigned to DFSDM filter 0:

- Declare pins used in pinctrl DT node (see Pinctrl device tree configuration):
  - Configure PB13 as DFSDM CLKOUT alternate function (AF3 by default, ANALOG for low-power mode).
  - Configure PC3 as DFSDM DATA1 alternate function (AF3 by default, ANALOG for low-power mode).

```
dfsdm_clkout_pins_a: dfsdm-clkout-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('B', 13, AF3)>;     /* DFSDM_CKOUT */
                bias-disable;
                drive-push-pull;
                slew-rate = <1>;
        };
};

dfsdm_clkout_sleep_pins_a: dfsdm-clkout-sleep-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('B', 13, ANALOG)>;  /* DFSDM_CKOUT */
        };
};
```

```
dfsdm_data1_pins_a: dfsdm-data1-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('C', 3, AF3)>;      /* DFSDM_DATA1 */
        };
};

dfsdm_data1_sleep_pins_a: dfsdm-data1-sleep-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('C', 3, ANALOG)>;   /* DFSDM_DATA1 */
        };
};
```

- Add sd-modulator[3] in the board dts file.

```
sd_adc1: adc-1 {
        compatible = "sd-modulator";
        #io-channel-cells = <0>;
};
```

- Configure and enable DFSDM, configure channel 1 to use SPI (rising edge), associate it to filter0.

```
&dfsdm {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&dfsdm_clkout_pins_a &dfsdm_data1_pins_a>;                 /*
default pins */
        pinctrl-1 = <&dfsdm_clkout_sleep_pins_a &dfsdm_data1_sleep_pins_a>;     /*
sleep pins for low-power mode */
        spi-max-frequency = <2048000>;                                          /*
desired maximum clock rate */
        status = "okay";
        dfsdm0: filter@0 {
                st,adc-channels = <1>;                                          /*
Assign channel 1 to this filter */
                st,adc-channel-names = "in1";                                   /*
Give it a name */
                st,adc-channel-types = "SPI_R";                                 /*
SPI data on rising edge */
                st,adc-channel-clk-src = "CLKOUT_F";                            /*
internal clock source used for conversion */
                io-channels = <&sd_adc1>;                                       /*
phandle to the external sd-modulator */
                st,filter-order = <1>;
                status = "okay";
        };
};
```

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

Please refer to the following links for additional information:

- Documentation/devicetree/bindings/iio/adc/st,stm32-dfsdm-adc.txt , STM32 DFSDM ADC device tree bindings
- Documentation/devicetree/bindings/sound/st,stm32-adfsdm.txt , STM32 audio DFSDM device tree bindings
- 3.03.13.2 Documentation/devicetree/bindings/iio/adc/sigma-delta-modulator.txt , Generic Device-Tree bindings for sigma delta modulator
- arch/arm/boot/dts/stm32mp151.dtsi , STM32MP151 device tree file

Operating System

Digital Filter for Sigma-Delta Modulator

Device Tree

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital microphone

Serial Peripheral Interface

Stable: 30.03.2021 - 14:51 / Revision: 30.03.2021 - 14:49

A quality version of this page, approved on *30 March 2021*, was based off this revision.

How to use the IIO user space interface with a user terminal.

Contents

1 Purpose

This article describes how to use the IIO with a user terminal.

The use cases of the following examples are:

- **analog to digital**
- **digital to analog**
- **quadrature encoder**[1] **monitoring**

Conversions between an STM32 board and an external device:

- Basic reads from ADC (for example by polling) or writes to a DAC are performed using **sysfs**
- More advanced use cases (with timer triggers and buffers) are performed using **sysfs configuration** and **character devices** either directly or with tools
- Simulation of a quadrature encoder device using GPIOs

Information

Some **IIO tools** are used in this article (e.g. lsiio). A list of IIO tools is defined in dedicated articles: IIO Linux kernel tools and libiio tools

2 How to do a simple conversion using the sysfs interface

The IIO sysfs interface can be used to configure devices and do **simple conversions at low rates**.

This is usually referred to as **IIO direct mode** in IIO device drivers.

Documentation/ABI/testing/sysfs-bus-iio[2] is the Linux® kernel documentation that fully describes the IIO standard ABI.

Note: To convert a raw value to standard units, the IIO defines this formula: **Scaled value = (raw + offset) * scale**

2.1 How to do a simple ADC conversion using the sysfs interface

This example shows **how to read** a single data from the ADC, using sysfs.

Information

The ADC is enabled by thedevice tree: ADC DT configuration example

First, look for the IIO device matching the ADC peripheral:

```
$ grep -H "" /sys/bus/iio/devices/*/name | grep adc                    # or use
'lsiio | grep adc'
/sys/bus/iio/devices/iio:device0/name:48003000.adc:adc@0               # Going to
use iio:device0 sysfs, that matches ADC1
/sys/bus/iio/devices/iio:device1/name:48003000.adc:adc@100
```

Then, perform a single conversion on an ADC, and also read the ADC scale and offset:

```
$ cd /sys/bus/iio/devices/iio:device0/
$ cat in_voltage6_raw                                                  # Convert
ADC1 channel 0 (analog-to-digital): get raw value
40603
$ cat in_voltage_scale                                                 # Read
scale
0.044250488
$ cat in_voltage_offset                                                # Read
offset
0
$ awk "BEGIN{printf (\"%d\n\", (40603 + 0) * 0.044250488)}"            # Scaled
value = (raw + offset) * scale
1796                                                                   # Result:
1796 mV
```

2.2 How to do a simple DAC conversion using the sysfs interface

This example shows **how to write** single data to the DAC, using sysfs.

Information

The DAC is enabled by the device tree: DAC DT configuration example

First, look for the IIO device matching the DAC peripheral:

```
$ lsiio | grep dac
Device 003: 40017000.dac:dac@1                                          # Going to
use iio:device3 sysfs, that matches DAC1
Device 004: 40017000.dac:dac@2
```

Then, check the DAC *scale* to compute the *raw* value:

```
$ cd /sys/bus/iio/devices/iio:device3/
$ cat out_voltage1_scale                                                # Read
scale
0.708007812
$ awk "BEGIN{printf (\"%d\n\", 2000 / 0.708007812)}"                    # Example
to convert convert 2000 mV (millivolts)
2824
$ echo 2824 > out_voltage1_raw                                          # Write
raw value to DAC1
$ echo 0 > out_voltage1_powerdown                                       # Enable
DAC1 (out of power-down mode): DAC now converts from digital to analog
                                                                        # User can

now convert new value with 'echo xxxx > out_voltage1_raw'
```

3 Convert one or more channels using triggered buffer mode

Building upon on what is described in the article User space interface, the user should:

- configure and enable the **IIO trigger via sysfs** (*/sys/bus/iio/devices/triggerX*)
- configure and enable the **IIO device via sysfs** (*/sys/bus/iio/devices/iio:deviceX*)
- access configured events and **data from character device** (*/dev/iio:deviceX*)

This is typically the case when using one of the **IIO buffer modes**.

See The Linux driver implementer's API guide - Industrial I/O Buffers for further details.

The STM32 provides several hardware triggers, among which TIM and LPTIM can be used in IIO.

3.1 How to set up a TIM or LPTIM trigger using the sysfs interface

This example shows **how to set up** a TIM or an LPTIM **trigger**, using sysfs.

Information

TIM and/or LPTIM are enabled by device tree: See TIM configured in PWM mode and trigger source example and/or LPTIM DT configuration as PWM and trigger source example

Runtime configuration is performed using the sysfs interface:

```
$ lsiio | grep tim                                                    # Look for
IIO device that matches TIM and/or LPTIM peripheral
Device 010: 44000000.timer:trigger@0
Trigger 000: tim6_trgo
Trigger 001: tim1_trgo
Trigger 002: tim1_trgo2
Trigger 003: tim1_ch1
Trigger 004: tim1_ch2
Trigger 005: tim1_ch3
Trigger 006: tim1_ch4
```

Either the TRGO or the PWM output can be configured, and used as the trigger source for analog conversions.

- To configure the timX_trgo trigger, the **"sampling_frequency"** (Hz) can be set directly:

```
$ cd /sys/bus/iio/devices/trigger0/
$ cat name
tim6_trgo
$ echo 10 > sampling_frequency                                        # Set up
10Hz sampling frequency on tim6_trgo
```

- When using the timX_chY or the **lptimX_outY** trigger, the frequency must be set using the PWM framework. See How to use PWM with sysfs interface.

```
$ cd /sys/bus/platform/devices/44000000.timer:pwm/pwm/pwmchip0
$ echo 0 > export                                                    # Export ti
m1_ch1 PWM
$ echo 100000000 > pwm0/period
$ echo 50000000 > pwm0/duty_cycle
$ echo 1 > pwm0/enable                                               # Enable ti
m1_ch1 with 10Hz frequency and 50% duty cycle
```

3.2 How to perform multiple ADC conversions in triggered buffer mode

This example shows **how to read** multiple data from an ADC, to **scan one or more channels**.

Information

The ADC is enabled by the device tree: ADC DT configuration example

Conversions are triggered by the **TIM or LPTIM hardware trigger**, See How to set up a TIM or LPTIM trigger using the sysfs interface.

As an example, ADC *in0* and *in1* can be converted in sequence.

- **sysfs** interface overview:

```
$ cd /sys/bus/iio/devices/iio\:device0
$ cat name
48003000.adc:adc@0
$ ls scan_elements
in_voltage0_en  in_voltage0_index  in_voltage0_type  in_voltage1_en  in_voltage1_index
in_voltage1_type
$ ls trigger
current_trigger
$ ls buffer
enable  length  watermark
```

- Example to enable ADC channel 0 and channel 1, and use the tim6_trgo trigger source :

```
$ echo 1 > scan_elements/in_voltage0_en                              # Enable channel 0
$ echo 1 > scan_elements/in_voltage1_en                              # Enable channel 1
$ echo "tim6_trgo" > trigger/current_trigger                         # Assign tim6_trgo
trigger to ADC
$ cat trigger/current_trigger
tim6_trgo
$ echo 1 > buffer/enable                                             # Start ADC in buffer mode
```

- **character device** data out:

```
$ hexdump -e '"iio0 :" 8/2 "%04x " "\n"' /dev/iio:device0 &          # Read data from /dev/iio:
device0, display by group of 8, 2 bytes.
iio0 :9f15 0000 9e9f 0000 9f18 0000 9ee4 0000                        # Result: raw data out in
the form of: in0 data | in1 data | in0 data...
...
```

3.3 How to perform multiple ADC conversions in triggered buffer mode using libiio

Prerequisite: please see the similar example: How to perform multiple ADC conversions in triggered buffer mode.

That example uses **iio_readdev**[3] provided by libiio tools.

The example below requests 8 data samples on the ADC configured with:

- channel 0 and channel 1, also referred to as **voltage0** and **voltage1**, enabled
- tim6_trgo, also referred to as **trigger0** to trigger conversions, see How to set up a TIM or LPTIM trigger using the sysfs interface

```
$ iio_readdev -t trigger0 -s 8 -b 8 iio:device0 voltage0 voltage1 | hexdump
0000000 9efe 0000 9ed9 0034 9eff 0000 9ee5 0000
0000010 9edb 0011 9ecc 000b 9eb0 0000 9ed4 0001
```

4 How to use the quadrature encoder with the sysfs interface

Warning

Take care this section is no more dedicated to IIO but is related to the new Linux **counter** framework coming with STM32 MPU ecosystem-v2

This example shows **how to monitor the position** (count) of a **linear** (or rotary) **encoder**.

It uses quadrature the encoder[1] interface available on the TIM and LPTIM internal peripherals.

4.1 How to set up the TIM quadrature encoder with the sysfs interface

Information

The TIM quadrature encoder is enabled by the device tree: TIM configured as quadrature encoder interface

Runtime configuration is performed using the sysfs interface[4]:

```
Board $> grep -H "" /sys/bus/counter/devices/*/name   # Look for TIM counter devices
/sys/bus/counter/devices/counter0/name:44000000.timer:counter
Board $> cd /sys/bus/counter/devices/counter0
Board $> cat count0/function_available                # List available modes:
quadrature x2 a
quadrature x2 b
quadrature x4
Board $> echo "quadrature x4" > count0/function       # set quadrature mode
Board $> echo 65535 > count0/ceiling                  # set ceiling value (upper limit
for the counter)
Board $> echo 0 > count0/count                        # reset the counter
Board $> echo 1 > count0/enable                       # enable the counter
```

Once started, the encoder value and direction are available using:

```
Board $> cat count0/count
0
Board $> cat count0/direction
forward
```

4.2 How to set up the LPTIM quadrature encoder with the sysfs interface

Information

The LPTIM quadrature encoder is enabled by the device tree: LPTIM configured as quadrature encoder interface

Runtime configuration is performed using the sysfs interface[4]:

```
Board $> grep -H "" /sys/bus/counter/devices/*/name   # Look for TIM counter devices
/sys/bus/counter/devices/counter0/name:40009000.timer:counter
Board $> cd /sys/bus/counter/devices/counter0
Board $> cat count0/function_available                # List available modes:
increase
quadrature x4
Board $> echo "quadrature x4" > count0/function       # set quadrature mode
Board $> echo 65535 > count0/ceiling                  # set ceiling value (upper limit
for the counter)
Board $> echo 1 > count0/enable                       # enable the counter
```

Once started, the encoder value is available using:

```
Board $> cat count0/count
0
```

4.3 How to use the TIM or LPTIM quadrature encoder with the sysfs interface

This example shows how to monitor the TIM quadrature encoder interface via sysfs (the LPTIM case is very similar):

- In this example, two GPIO lines (PD1, PG3) are externally connected to the TIM (or LPTIM)
- Then libgpiod[5] is used to set and clear the encoder input pins, to '**emulate**' an external quadrature encoder device.

Information

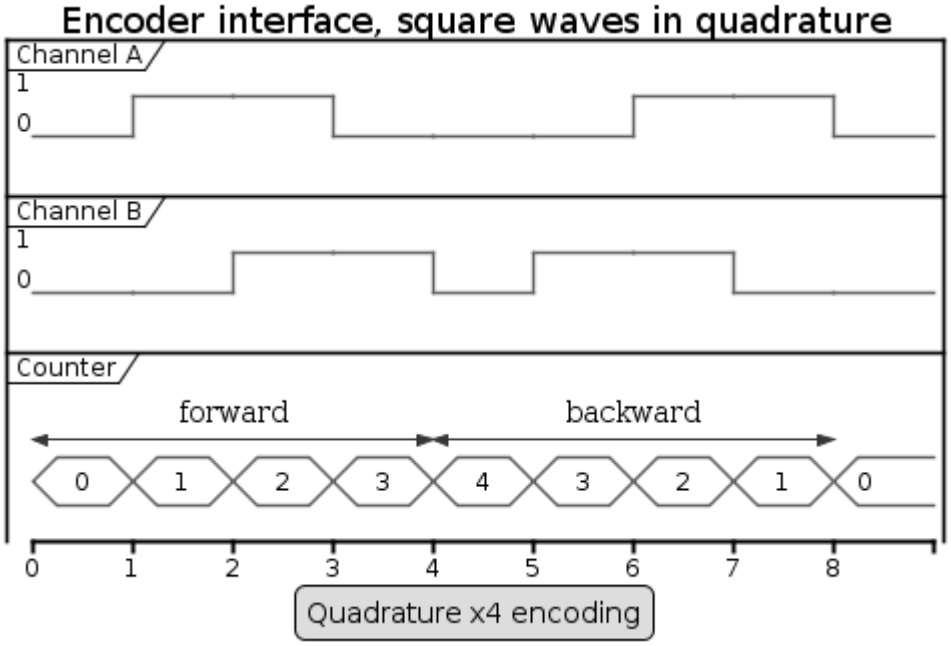
On the STM32MP157X-DKX discovery board, PD1, PG3, TIM1_CH1 and TIM1_CH2 signals are accessible via respectively the D7, D8, D6 and D10 pins of the Arduino Uno connector.

Step-by-step example:

- **Externally connect** and initialise **GPIO pins to TIM** or LPTIM encoder input pins, to '**emulate**' an external quadrature encoder

```
Board $> gpiodetect
...
gpiochip6 [GPIOG] (16 lines)
...
gpiochip3 [GPIOD] (16 lines)
...
Board $> gpioset gpiochip3 1=0     # initialize PD1 to 0 as GPIO, connect it to TIM or LPT
IM channel A input
Board $> gpioset gpiochip6 3=0     # initialize PG3 to 0 as GPIO, connect it to TIM or LPT
IM channel B input
```

- Set up the TIM or LPTIM quadrature encoder with the sysfs interface, see How to set up the TIM quadrature encoder with the sysfs interface or How to set up the LPTIM quadrature encoder with the sysfs interface
- GPIO pins are then set or cleared as follows:




```
Board $> cd /sys/bus/counter/devices/counter0/
Board $> cat count0/count                          # [channel A, channel B] = [0, 0]
0
Board $> gpioset gpiochip3 1=1                     # [channel A, channel B] = [1, 0]
Board $> cat count0/count
1
Board $> gpioset gpiochip6 3=1                     # [channel A, channel B] = [1, 1]
Board $> cat count0/count
2
Board $> gpioset gpiochip3 1=0                     # [channel A, channel B] = [0, 1]
Board $> cat count0/count
3
Board $> gpioset gpiochip6 3=0                     # [channel A, channel B] = [0, 0]
Board $> cat count0/count
4
Board $> cat count0/direction
forward
Board $> gpioset gpiochip6 3=1                     # [channel A, channel B] = [0, 1]
Board $> cat count0/count
3
Board $> cat count0/direction                      # Direction has changed, down-
counting now
backward
...
```

5 References

- 1.01.1 Quadrature encoder, Incremental encoder overview
- Documentation/ABI/testing/sysfs-bus-iio , Linux standard sysfs IIO interface
- https://wiki.analog.com/resources/tools-software/linux-software/libiio/iio_readdev, iio_readdev
- 4.04.1 Documentation/ABI/testing/sysfs-bus-counter , Counter sysfs ABI
- Control GPIO through libgpiod

Industrial I/O Linux subsystem

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

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

Application binary interface. (In computer software, an application binary interface (ABI) describes the low-level interface between a computer program and the operating system or another program.)

Linux® is a registered trademark of Linus Torvalds.

low-power timer (STM32 specific)

Pulse Width Modulation

Microprocessor Unit

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.)

Stable: 16.01.2020 - 14:07 / Revision: 16.01.2020 - 14:03

A quality version of this page, approved on *16 January 2020*, was based off this revision.

Linux® kernel provides some user space tools that can be used for testing the IIO subsystem.

Contents

1 Article purpose

The purpose of this article is to:

- briefly introduce the IIO user space tools that comes with the Linux® kernel
- provide a few examples using these tools

2 Introduction

These tools use IIO sysfs and character device directly without libiio (See IIO user space interface for further details).

3 Tools

The Linux® kernel provides the following IIO user space tools:

- **lsiio**: example application that provides a list of IIO devices and triggers.

```
root@stm32mp1:~# lsiio
Device 001: 48003000.adc:adc@100
Device 000: 48003000.adc:adc@0
Trigger 018: tim3_ch4
Trigger 007: tim1_ch3
...
```

- **iio_event_monitor**: example application that reads events from an IIO device and prints them.

See How to get ADC analog watchdog events.

```
root@stm32mp1:~# iio_event_monitor /dev/iio:device0 &
Event: time: 1529352199639112110, type: voltage, channel: 0, evtype: thresh, direction:
either
Event: time: ...
```

- **iio_generic_buffer**: example application that reads data from buffer.
- **iio_utils**: set of routines built-in above IIO tools, typically used to access sysfs files.

4 Source code

The Linux® kernel IIO tools source code can be found under tools/iio[1] directory:

- tools/iio/lsiio.c
- tools/iio/iio_event_monitor.c
- tools/iio/iio_generic_buffer.c
- tools/iio/iio_utils.c

5 Installation on your target

The Linux® kernel IIO tools aren't embedded by default in OpenSTLinux distribution. They can be compiled independently and then installed on the target. See How to build Linux kernel user space tools.

6 References

- tools/iio , Linux®IIO tools source code

Linux® is a registered trademark of Linus Torvalds.

Industrial I/O Linux subsystem

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

Stable: 16.01.2020 - 14:01 / Revision: 16.01.2020 - 14:00

A quality version of this page, approved on *16 January 2020*, was based off this revision.

Libiio is a complete library which offers tools and an interface to develop an application using IIO subsystem.

Contents

1 Article purpose

The purpose of this article is to:

- briefly introduce the *libiio* main features and API
- provide few examples, using *libiio* tools

2 Introduction

- *Libiio* is a user space library that provides an **interface** for user space applications. It is basically a wrapper that resides above the following interfaces:

1. **/sys/bus/iio/devices** sysfs interface (for configuration/setting)
2. **/dev/iio/deviceX** device interface (for data)

- *Libiio* also provides **tools** that can be used for testing
- *Libiio* design goals:

1. Interface with the kernel, to access IIO[1] devices
2. Provide proper data structures and functions to the user application
3. Support for local and remote backends allowing applications to access the devices when running on a local or a remote machine

The full description of the IIO library is provided by the author of the library, see below references:

- What is libiio[2].
- About libiio[3].

3 API description

The API description can be found here: https://analogdevicesinc.github.io/libiio

4 Tools

Libiio offers tools such as:

- *iiod* server daemon
- *iio_info* to dump attributes

```
root@stm32mp1:~# iio_info
Library version: 0.8 (git tag: v0.8)
IIO context created with local backend.
Backend version: 0.8 (git tag: v0.8)
Backend description string: Linux stm32mp1 4.14.0-00004-gafe4a31 #778 SMP PREEMPT Tue Aug
28 14:02:25 CEST 2018 armv7l
IIO context has 3 devices:
        trigger1: tim6_trgo
                0 channels found:
                3 device-specific attributes found:
                                attr 0: sampling_frequency value: 100
                                attr 1: master_mode value: reset
                                attr 2: master_mode_available value: reset enable update
compare_pulse OC1REF OC2REF OC3REF OC4REF
        iio:device0: 48003000.adc:adc@0 (buffer capable)
                2 channels found:
                        voltage0:  (input, index: 0, format: le:U16/16>>0)
                        3 channel-specific attributes found:
                                attr 0: raw value: 72
                                attr 1: offset value: 0
                                attr 2: scale value: 0.044250488
                        voltage1:  (input, index: 1, format: le:U16/16>>0)
                        3 channel-specific attributes found:
                                attr 0: raw value: 1746
                                attr 1: offset value: 0
                                attr 2: scale value: 0.044250488
...
```

- *iio_readdev*[4] (to read or scan from a device)

```
STM32AP [rc=0]# iio_readdev -t trigger1 -s 8 -b 8 iio:device0 voltage0 voltage1 | hexdump
0000000 0068 055a 0058 0520 00b4 03df 0070 055f
0000010 0096 03d6 0089 038f 0077 05c8 0096 03b3
```

See also: How to use the IIO user space interface

5 Source code

Libiio can be downloaded on a public github[5]. It can be cloned using git command:

```
git clone https://github.com/analogdevicesinc/libiio.git
```

Tools source code can be found under libiio "tests" directory.

6 Installation on your target

Libiio and the tools it provides are embedded by default in OpenSTLinux distribution.

7 References

- IIO overview, IIO subsystem overview
- https://wiki.analog.com/resources/tools-software/linux-software/libiio, What is libiio
- https://wiki.analog.com/resources/tools-software/linux-software/libiio_internals, About libiio
- https://wiki.analog.com/resources/tools-software/linux-software/libiio/iio_readdev, iio_readdev
- https://github.com/analogdevicesinc/libiio, libiio download link

Application programming interface

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

Industrial I/O Linux subsystem

Linux® is a registered trademark of Linus Torvalds.

symetric multiprocessing

Stable: 17.02.2021 - 16:18 / Revision: 17.02.2021 - 16:11

A quality version of this page, approved on *17 February 2021*, was based off this revision.

This article gives information about the Linux®IIO framework.
It explains how to activate the IIO interface and, based on examples, how to use it.

Contents

1 Framework purpose

IIO (Industrial I/O) is a subsystem for Analog to Digital Converters (ADCs), Digital to Analog Converters (DACs) and various types of sensors. It can be used on high speed, high data rates industrial devices. Until recently, it was mostly focused on user-space abstraction. It also includes in-kernel API for other drivers.

The Industrial I/O Linux® subsystem offers a unified framework to communicate (read and write) with drivers covering many different types of embedded sensors and a few actuators. It also offers a standard interface to user space applications manipulating sensors through sysfs and devfs.

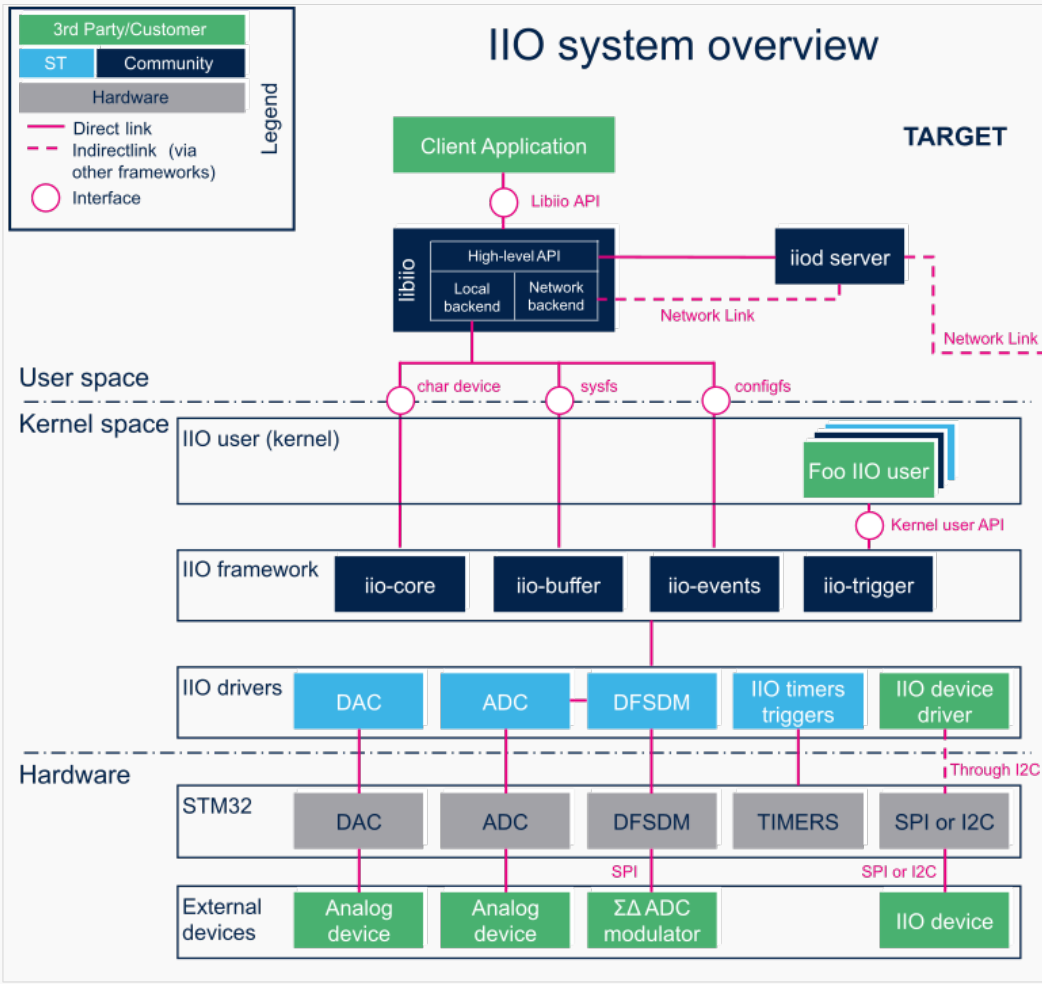
Here are some examples of supported sensor types in IIO:

- ADC / DAC
- accelerometers
- magnetometers
- gyroscopes
- pressure
- humidity
- temperature
- light and proximity

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

- Low speed acquisition for slow varying input signal (example: log temperature to a file)
- High speed acquisition using ADC, DFSDM or external devices (example: audio, power meter)
- Read the position of a rotary element using TIM or LPTIM quadrature encoder interface
- Driving an analog source through a DAC
- External devices connected via SPI or I2C.

2 System overview



IIO Implementation architecture

2.1 Components description

From client application to hardware

- **Client Application** (User Space): An application that configures, read or write data samples to/from *IIO device(s)* via libiio.
- **iiod server** (User Space): It is optional. Applications based on libiio can benefit from a remote access via *IIO Daemon* server, to IIO "local" backend through a network link.
- **libiio** (User Space): libiio is a complete library offering an API for developping an application. It's composed of a high-level API, and two backends:

1. The “local” backend, interfacing with the Linux kernel through the IIO API
2. The “network” backend, interfacing with the iiod server through a network link.

- **User Space interface**: It is composed of a standard **char device**, **sysfs**, **configfs** and debugfs (see API description).
- **Kernel Space user**: It can be any kernel space IIO consumer, like STM32 DFSDM audio driver or IIO hwmon driver (See How to use IIO kernel API).
- **Kernel Space interface**: It is composed of a standard API
- **IIO framework** (Kernel Space): It's composed of a core. It manages data buffers, userspace events, triggers. It also handles clients (either in kernel or in User Space).
- **IIO drivers** (Kernel Space): Linux kernel drivers to handle internal peripherals or external devices. They includes an interface that provides controls and data to the user (examples: ADC Linux driver, DAC Linux driver, DFSDM Linux driver, TIM Linux driver, LPTIM Linux driver, IIO device driver connected on SPI or I2C).
- **STM32 peripherals** (Hardware): connected to the external devices through a specific interface (examples: ADC, DAC, DFSDM , TIM, LPTIM, SPI, or I2C)
- **External devices** (Hardware): connected to the STM32 front-end through a specific interface. These can be analog devices (such as accelerometers, Inertial Measurement Units...), a Sigma Delta ADC Modulator (for audio record, energy measurements...), IIO devices on SPI or I2C...

2.2 API description

Depending on needs and location (Kernel Space or User Space), several APIs are available to control an IIO device.

2.2.1 libiio

Libiio provides a user space high-level API for client applications[1]. The library 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 the *user space interface* (sysfs and char device) provided by the kernel.

2.2.2 User space interface

The IIO framework provides several interfaces:

- **iio device sysfs interface**: It is used to **configure which events and data** should come out of the character device, e.g. */sys/bus/iio/devices/iio:deviceX*.

It can be used to **read** (poll) or **write data** directly **at low rates**.

The IIO sysfs ABI is documented in: *Documentation/ABI/testing/sysfs-bus-iio*[2].

See How to use the IIO user space interface and How to access information in sysfs for further details.

- **character device**[3]: It is optional in IIO. It is used to output **events and sensor data**, e.g. */dev/iio:deviceX*.

It is basically a file from application point of view. Standard file API allows to access it: open(), read(), write(), close()...

See How to use triggered buffer mode and The Linux driver implementer's API guide - Industrial I/O Buffers for further details.

- **configfs**: It allows to configure additional IIO features like *software and hrtimer triggers*.

The IIO configfs interface is documented in: *Documentation/ABI/testing/configfs-iio*[4] and *Documentation/iio/iio_configfs.txt*[5].

Note: STM32 already provides *hardware triggers* (See How to use the IIO timers triggers).

- Debugfs: May provide some debug conveniences (like *direct_reg_access* entry to read/write registers) depending on the IIO device driver in use.

2.2.3 Kernel space interface

Useful kernel API for users:

- **devm_iio_channel_get_all()** or iio_channel_get_all() / iio_channel_release_all(): Used to lookup, get, then release IIO channels.
- **iio_get_channel_type()**: get the type of a channel, such as IIO_VOLTAGE, IIO_TEMP...
- **iio_read_channel_processed()**: read channel processed value, e.g. like in micro-volts for voltage, milli-degree for temperature...
- ...

Available routines can be found in kernel header file: include/linux/iio/consumer.h[6].

3 Configuration

3.1 Kernel configuration

IIO is 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 IIO in kernel configuration with Linux Menuconfig tool: Menuconfig or how to configure kernel

```
Device Drivers  --->
  <*> Industrial I/O support  --->
    [*]   Enable buffer support within IIO
     < >     IIO callback buffer used for push in-kernel interfaces
     <*>     Industrial I/O HW buffering
     <*>     Industrial I/O buffering based on kfifo
     < >   Enable IIO configuration via configfs
     [*]   Enable triggered sampling support
     (2)     Maximum number of consumers per trigger
     < >   Enable software triggers support
           Accelerometers  --->
           Analog to digital converters  --->
           Amplifiers  --->
           Chemical Sensors  --->
           Hid Sensor IIO Common  ----
           SSP Sensor Common  --->
           Digital to analog converters  --->
           IIO dummy driver  --->
           Frequency Synthesizers DDS/PLL  --->
           Digital gyroscope sensors  --->
           Health Sensors  --->
           Humidity sensors  --->
           Inertial measurement units  --->
           Light sensors  --->
           Magnetometer sensors  --->
           Inclinometer sensors  ----
           Triggers - standalone  --->
           Digital potentiometers  --->
           Pressure sensors  --->
           Lightning sensors  --->
           Proximity sensors  --->
           Temperature sensors  --->
```

IIO supports several types of sensors and devices. User can select from there any driver among the supported devices.

Please refer to ADC Linux driver, DAC Linux driver, DFSDM Linux driver, TIM Linux driver, LPTIM Linux driver articles for each peripheral.

3.2 Device tree configuration

IIO bindings[7] documentation deals with all required or optional IIO generic DT properties.

It also introduces **IIO providers** and **IIO consumers**.
Example with STM32 ADC:

```
&adc {
        adc2: adc@100 {                         /* IIO provider example */
                ...
                #io-channel-cells = <1>;
                st,adc-channels = <12>;         /* channel 12 in use */
        };
};
/ {
        consumer_device {                       /* IIO consumer example */
                io-channels = <&adc2 12>;
                io-channel-names = "example";   /* IIO consumer driver side: devm_iio_chann
el_get(&dev, "example"); */
        };

        iio-hwmon {                             /* iio_hwmon[8] is another consumer example
(See SENSORS_IIO_HWMON in kernel configuration) */
                compatible = "iio-hwmon";       /* See Documentation/devicetree/bindings
/iio/iio-bindings.txt[7]
                io-channels = <&adc2 12>;
        };
};
```

Detailed DT configuration for STM32 internal peripherals:

- ADC device tree configuration
- DAC device tree configuration
- DFSDM device tree configuration
- TIM device tree configuration
- LPTIM device tree configuration

Linux kernel provides many other supported devices[9] in **Documentation/devicetree/bindings/iio** directory.

4 How to use the framework

This section describes how to use the IIO framework from:

- User space interface: Please refer to libiio and IIO Linux kernel tools that run on top of **sysfs** and **character device** (How to use the IIO user space interface)
- Kernel space interface: How to use IIO kernel API

4.1 How to use the IIO user space interface

Please see examples based on the following use cases:

- How to read a data: How to do a simple ADC conversion using the sysfs interface
- How to write a data: How to do a simple DAC conversion using the sysfs interface
- How to setup a trigger source: How to set up a TIM or LPTIM trigger using the sysfs interface
- How to use a trigger source: How to perform multiple ADC conversions in triggered buffer mode
- How to register on an event: How to get ADC analog watchdog events
- How to use quadrature encoder: How to use the quadrature encoder with the sysfs interface

4.2 How to use IIO kernel API

Several in-kernel drivers use kernel IIO API. See HWMON client example for IIO devices, and STM32 DFSDM audio ALSA IIO client:

- iio_hwmon: *drivers/hwmon/iio_hwmon.c*[8]. See also device tree configuration example to read voltage from ADC.

```
$ cat /sys/class/hwmon/hwmon0/in1_input
1809                                    # iio_hwmon calls iio_read_channel_processed():
ADC result is in mV.
```

- stm32-adfsdm: See DFSDM Linux driver and ALSA overview for further details

5 How to trace and debug the framework

5.1 How to trace with dynamic debug

By default there is no kernel log that shows activity on IIO. However the user could enable dynamic debug for the IIO core and the IIO drivers.

```
Board $> dmesg -n8
Board $> echo "file drivers/iio/* +p" > /sys/kernel/debug/dynamic_debug/control
Board $> echo "file drivers/iio/adc/* +p" > /sys/kernel/debug/dynamic_debug/control
Board $> echo "file drivers/iio/dac/* +p" > /sys/kernel/debug/dynamic_debug/control
```

See dynamic debug for more details.

5.2 How to debug with debugfs

IIO proposes an optional debugfs entry to access registers. It is up to the IIO device driver to implement it (e.g. *debugfs_reg_access()*). When it is available:

```
$ cd /sys/kernel/debug/iio/iio:deviceX
```

To read a register from the device:

```
$ echo [register offset] > direct_reg_access
$ cat direct_reg_access
0xhhhh                                          # Register content
```

To write a register:

```
$ echo [register offset] [register value] > direct_reg_access
```

6 To go further

6.1 How to write a kernel IIO device driver

The Linux Kernel community provides all the documents needed to develop an IIO device driver :

- The Linux driver implementer's API guide - Industrial I/O[10]: This guide provides the API provided by kernel IIO core components.
- IIO staging documentation[11], included in the Kernel sources (**drivers/staging/iio/Documentation**).
- Linux Kernel IIO dummy driver example source code[12]: Dummy driver source code, included in the kernel sources (**drivers/iio/dummy/iio_simple_dummy.c**).

6.2 Trainings documents

- IIO a new subsystem[13] : Presentation of Kernel IIO subsystem
- Industrial I/O Subsystem: The Home of Linux Sensors[14]: Why IIO? What is it? Sensor types...
- Software Defined Radio using the Linux Industrial IO framework[15] : User Guide describing how to implement an application by using Linux Industrial IO framework
- Linux Device Drivers, Third Edition[16] : Reference book for linux device drivers development, for IIO see Chapter 3, Char Drivers.

7 References

- libiio High-Level API, libiio API Documentation (Library for interfacing with IIO devices)
- sysfs-bus-iio ABI, Linux standard sysfs IIO interface
- character device interface, *Linux Kernel and Driver Development* training document, see **Character drivers** and **Kernel frameworks for device drivers** chapter
- configfs-iio ABI, Linux standard configfs IIO interface
- iio_configfs interface, Linux standard configfs interface
- include/linux/iio/consumer.h , IIO 'inkern' API
- 7.07.1 Documentation/devicetree/bindings/iio/iio-bindings.txt , Linux Foundation, IIO Generic DT bindings
- 8.08.1 drivers/hwmon/iio_hwmon.c IIO HWMON, consumer driver example (kernel space)
- Kernel DT documentation IIO bindings , Linux Foundation, IIO DT bindings documents included in the Kernel sources
- Industrial I/O, The Linux driver implementer's API guide
- IIO staging documentation , Linux Foundation, IIO documents included in the Kernel sources
- drivers/iio/dummy/iio_simple_dummy.c , Linux Foundation, IIO dummy driver example source code
- IIO a new subsystem, Free Electrons, Presentation of Kernel IIO subsystem
- Industrial I/O Subsystem: The Home of Linux Sensors, Linux Foundation, IIO training
- Software Defined Radio using the Linux Industrial IO framework, Linux Foundation, User Guide describing how to implement an application by using Linux Industrial IO framework
- Linux Device Drivers, Third Edition, Pdf book, Authors Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Industrial I/O Linux subsystem

Application programming interface

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

Device File System (See https://en.wikipedia.org/wiki/Device_file#DEVFS for more details)

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Serial Peripheral Interface

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

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

Digital Filter for Sigma-Delta Modulator

Application binary interface. (In computer software, an application binary interface (ABI) describes the low-level interface between a computer program and the operating system or another program.)

Secure Secret Provisioning

Secure secrets provisioning

Device Tree

Advanced Linux sound architecture

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

input/output

Stable: 30.03.2021 - 15:44 / Revision: 29.03.2021 - 09:55

A quality version of this page, approved on *30 March 2021*, was based off this revision.

Contents

1 Article purpose

This article introduces the LPTIM Linux® driver for the LPTIM internal peripheral[1]:

- Which LPTIM features are supported by the driver
- How to configure, use and debug the driver
- What is the driver structure, and where the source code can be found.

2 Short description

The *LPTIM*[1]Linux driver (kernel space) is based on the PWM, IIO and counter frameworks. It provides several functionalities:

MFD driver:

- handles common resources (registers, clock)

PWM driver:

- handles the **PWM output** channel (single channel)

IIO hardware trigger driver:

- handles hardware **trigger sources** (synchronously with PWM) for other internal peripherals such as ADC[2], DAC[3], DFSDM [4]

Counter driver:

- handles the **quadrature encoder** interface[5] as well as the external event counter.

3 Configuration

3.1 Kernel configuration

Activate the LPTIM[1]Linux driver in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel.

Enable the following configurations (as well as their dependencies):

- CONFIG_MFD_STM32_LPTIMER
- CONFIG_PWM_STM32_LP
- CONFIG_IIO_STM32_LPTIMER_TRIGGER
- CONFIG_STM32_LPTIMER_CNT

```
Device Drivers  --->
   -> Multifunction device drivers  --->
      <*> Support for STM32 Low-Power Timer
   -> Pulse-width modulation (PWM) support  --->
      <*> STMicroelectronics STM32 PWM LP
   -> Industrial I/O support  --->
      -> Triggers - standalone  --->
         <*> STM32 Low-Power Timer Trigger
   -> Counter support  --->
      <*> STM32 LP Timer encoder counter driver
```

3.2 Device tree

Refer to the LPTIM device tree configuration article when configuring the LPTIM Linux kernel driver.

4 How to use

How to use PWM with sysfs interface

How to set up a TIM or LPTIM trigger using the sysfs interface

How to use the quadrature encoder with the sysfs interface

5 How to trace and debug

The LPTIM Linux driver can access LPTIM registers through REGMAP.

It comes with debugfs[6] entries, which allow dumping registers:

```
$ cd /sys/kernel/debug/regmap
$ ls
40004000.timer  40009000.timer

$ cd 40009000.timer
$ cat registers
000: 00000003
004: 00000000
008: 00000000
...
```

It also comes with tracepoints[7]:

```
$ cd /sys/kernel/debug/tracing
$ cat available_events | grep regmap
...
regmap:regmap_reg_read
regmap:regmap_reg_write
```

6 Source code location

The LPTIM Linux driver is composed of:

- stm32-lptimer.c driver to handle common resources, such as registers and clock.
- pwm-stm32-lp.c driver to handle PWM channel
- stm32-lptimer-trigger.c driver to handle trigger sources for other internal peripherals
- stm32-lptimer-cnt.c driver to handle quadrature encoder and external event counter
- include/linux/mfd/stm32-lptimer.h and include/linux/iio/timer/stm32-lptim-trigger.h header files

7 References

- 1.01.11.2 LPTIM internal peripheral
- ADC internal peripheral
- DAC internal peripheral
- DFSDM internal peripheral
- Incremental encoder overview
- Debugfs
- Ftrace

low-power timer (STM32 specific)

Linux® is a registered trademark of Linus Torvalds.

Multifunction device

Pulse Width Modulation

Industrial I/O Linux subsystem

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Digital Filter for Sigma-Delta Modulator

Low Power (MIPI® Alliance DSI standard)

Register map (Linux registers map abstraction API)

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

Stable: 30.03.2021 - 15:16 / Revision: 29.03.2021 - 09:53

A quality version of this page, approved on *30 March 2021*, was based off this revision.

Contents

1 Article purpose

The purpose of this article is to explain how to configure the low-power timer (*LPTIM*)[1] **when the peripheral is assigned to Linux®OS**, and in particular:

- how to configure the LPTIM **peripheral** to enable PWM, trigger, event counter and quadrature encoder
- how to configure the **board**, e.g. LPTIM input/output pins

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

It is used by the LPTIM Linux driver that registers relevant information in PWM, IIO and counter frameworks.

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 LPTIM[1] is a multifunction device (MFD).

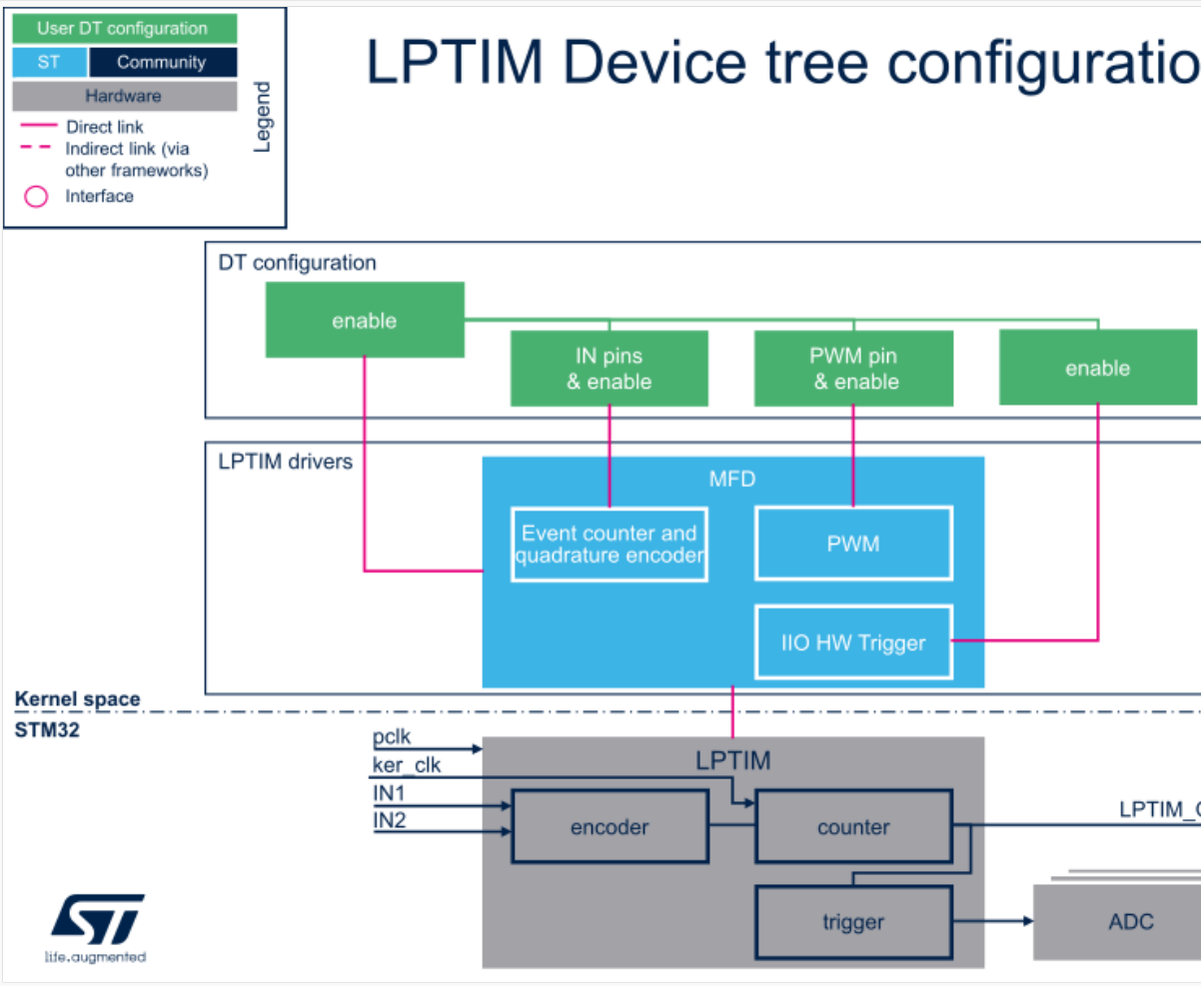
Each function is represented by a separate binding document:

- *STM32 LPTIM MFD device tree bindings*[3] deals with core resources (e.g. registers, clock)
- *STM32 LPTIM PWM device tree bindings*[4] deals with **PWM** interface resources (e.g. PWM pins)
- *STM32 LPTIM trigger device tree bindings*[5] deals with **LPTIM triggers** resources (e.g. trigger output connected to other STM32 internal peripherals)
- *STM32 LPTIM counter device tree bindings*[6] deals with **LPTIM counter and quadrature encoder** resources (e.g. counter /encoder IN[1..2] pins)

3 DT configuration

This hardware description is a combination of STM32 microprocessor and board device tree files. See Device tree for more explanations on device tree file split.

The **STM32CubeMX** can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.



LPTIM DT configuration

3.1 DT configuration (STM32 level)

LPTIM nodes are declared in stm32mp151.dtsi[7].

DT root node (aka lptimer1, ..., lptimer5) and **DT child nodes** describe the LPTIM features such as:

- PWM
- trigger
- event counter and quadrature encoder.

They also describe hardware parameters such as register addresses and clock.

```
lptimer1: timer@40009000 {
        compatible = "st,stm32-lptimer";                        /* lptimer's common
resources */
        ...
        pwm {
                compatible = "st,stm32-pwm-lp";                 /* PWM part of LPTIM */
        };
        trigger@0 {
                compatible = "st,stm32-lptimer-trigger";        /* trigger part of LPTIM */
                reg = <0>;                                      /* trigger identifier (e.g.
0 for LPTIM1 trigger, 1 for LPTIM2... */
        };
        counter {
                compatible = "st,stm32-lptimer-counter";        /* quadrature encoder &
event counter part of LPTIM */
        };
};
```

Warning

This device tree part is related to STM32 microprocessors. It should be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

Follow the below sequence to configure and enable the LPTIM on your board STM32MP15 microprocessor:

- Enable **DT root node** for the LPTIM instance in use (e.g lptimer1, ..., lptimer5), with **status = "okay";**
- Enable **DT child node(s)** for the feature(s) in use (PWM output, trigger, event counter and quadrature encoder), with **status = "okay";**
- Configure the pins in use via pinctrl, with **pinctrl-0**, **pinctrl-1** and **pinctrl-names**.

3.3 DT configuration examples

3.3.1 LPTIM1 configured as PWM and trigger source

The example below shows how to configure LPTIM1 to act as:

- PWM output on PG13 (See pinctrl device tree configuration and GPIO internal peripheral)
- trigger source (in Synchronous mode with PWM) for other internal peripherals such as ADC[8], DAC[9], and DFSDM[10]

```
lppwm1_pins_a: lppwm1-0 {
         pins {
                 pinmux = <STM32_PINMUX('G', 13, AF1)>;    /* configure 'PG13' as
alternate 1 for LPTIM1_OUT mode of operation */
                 bias-pull-down;
                 drive-push-pull;
                 slew-rate = <0>;
         };
};
lppwm1_sleep_pins_a: lppwm1-sleep-0 {
        pins {
                 pinmux = <STM32_PINMUX('G', 13, ANALOG)>; /* configure 'PG13' as analog
for low power mode */
        };
};
```

```
&lptimer1 {
        status = "okay";
        pwm {
                pinctrl-0 = <&lppwm1_pins_a>;                   /* configure PWM on LPTIM1_OUT
*/
                pinctrl-1 = <&lppwm1_sleep_pins_a>;
                pinctrl-names = "default", "sleep";
                status = "okay";                                /* enable PWM on LPTIM1 */
        };
        trigger@0 {
                status = "okay";                                /* enable LPTIM1_OUT trigger
source */
        };
};
```

3.3.2 LPTIM2 configured as counter and quadrature encoder

The example below shows how to configure LPTIM2 to act as counter and/or quadrature encoder, with LPTIM2_IN1 and LPTIM2_IN2 pins configured as inputs on PD12 and PD11, respectively (See pinctrl device tree configuration and GPIO internal peripheral)

```
# part of pin-controller dt node
lptim2_in_pins_a: lptim2-in-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('D', 12, AF3)>, /* LPTIM2_IN1 */
                         <STM32_PINMUX('D', 11, AF3)>; /* LPTIM2_IN2 */
                bias-disabled;
        };
};
lptim2_sleep_in_pins_a: lptim2-sleep-in-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('D', 12, ANALOG)>, /* LPTIM2_IN1 */
                         <STM32_PINMUX('D', 11, ANALOG)>; /* LPTIM2_IN2 */
        };
};
```

```
&lptimer2 {
        status = "okay";
        counter {
                pinctrl-0 = <&lptim2_in_pins_a>;                /* configure LPTIM2 counter
/encoder pins */
                pinctrl-1 = <&lptim2_sleep_in_pins_a>;
                pinctrl-names = "default", "sleep";
                status = "okay";                                /* enable counter/encoder on
LPTIM2 */
        };
};
```

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

For additional information, refer to the following links:

- 1.01.1 LPTIM internal peripheral
- Device tree
- Documentation/devicetree/bindings/mfd/stm32-lptimer.txt , STM32 LPTIM MFD device tree bindings
- Documentation/devicetree/bindings/pwm/pwm-stm32-lp.txt , STM32 LPTIM PWM device tree bindings
- Documentation/devicetree/bindings/iio/timer/stm32-lptimer-trigger.txt , STM32 LPTIM trigger device tree bindings
- Documentation/devicetree/bindings/counter/stm32-lptimer-cnt.txt , STM32 LPTIM counter/encoder device tree bindings
- STM32MP151 device tree file
- ADC internal peripheral
- DAC internal peripheral
- DFSDM internal peripheral

low-power timer (STM32 specific)

Linux® is a registered trademark of Linus Torvalds.

Operating System

Pulse Width Modulation

Device Tree

Multifunction device

also known as

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Digital Filter for Sigma-Delta Modulator

Stable: 12.03.2021 - 09:55 / Revision: 17.02.2021 - 14:14

A quality version of this page, approved on *12 March 2021*, was based off this revision.

Contents

1 Article purpose

This article introduces the TIM Linux® driver for the TIM internal peripheral[1]:

- Which TIM features are supported by the driver
- How to configure, use and debug the driver
- What is the driver structure, and where the source code can be found.

2 Short description

The *TIM*[1]Linux driver (kernel space) is based on the PWM, IIO and counter frameworks. It provides several functionalities:

MFD driver:

- handles registers, clock and DMA[2] resources
- detects the TIM counter resolution, e.g. 16 or 32 bits.

PWM driver:

- detects the number of TIM channels.
- handles **PWM output** channels.
- handles **PWM capture** channels (input). Note that the PWM capture relies on DMA, which is handled by the MFD core.

IIO driver:

- handles hardware **trigger sources** (synchronously with PWM) for other internal peripherals such as ADC[3], DAC[4], DFSDM [5].

counter driver:

- handles the **quadrature encoder** interface[6].

3 Configuration

3.1 Kernel configuration

Activate the TIM[1]Linux driver in the kernel configuration using the Linux Menuconfig tool: Menuconfig or how to configure kernel.

Enable the following configurations (and their dependencies):

- CONFIG_MFD_STM32_TIMERS
- CONFIG_PWM_STM32
- CONFIG_IIO_STM32_TIMER_TRIGGER
- CONFIG_STM32_TIMER_CNT

```
Device Drivers  --->
  -> Multifunction device drivers  --->
     <*> Support for STM32 Timers
  -> Pulse-width modulation (PWM) support  --->
     <*> STMicroelectronics STM32 PWM
  -> Industrial I/O support  --->
     -> Triggers - standalone  --->
        <*> STM32 timer trigger
  -> Counter support  --->
     <*> STM32 Timer encoder counter driver
```

3.2 Device tree

Refer to the TIM device tree configuration article when configuring the TIM Linux kernel driver.

4 How to use

How to use PWM with sysfs interface

How to set up a TIM or LPTIM trigger using the sysfs interface

How to use the quadrature encoder with the sysfs interface

5 How to trace and debug

The *TIM*[1]Linux driver can access the timer registers through REGMAP.

It comes with debugfs[7] entries, which allow dumping registers:

```
$ cd /sys/kernel/debug/regmap
$ ls
40004000.timer  44000000.timer

$ cd 44000000.timer
$ cat registers
000: 00000081
004: 00000000
008: 00000000
00c: 00000000
...
```

It also comes with tracepoints[8]:

```
$ cd /sys/kernel/debug/tracing
$ cat available_events | grep regmap
...
regmap:regmap_reg_read
regmap:regmap_reg_write
```

6 Source code location

The TIM Linux driver source code is composed of:

- stm32-timers.c MFD driver to handle common resources: registers, clock, dmas.
- pwm-stm32.c PWM driver to handle PWM channel(s).
- stm32-timer-trigger.c IIO driver to handle trigger source for other internal peripherals.
- stm32-timer-cnt.c counter driver to handle the quadrature encoder interface.
- include/linux/mfd/stm32-timers.h and include/linux/iio/timer/stm32-timer-trigger.h header files

7 References

- 1.01.11.21.3 TIM internal peripheral
- DMA_internal_peripheral
- ADC internal peripheral
- DAC internal peripheral
- DFSDM internal peripheral
- Incremental encoder Incremental encoder overview
- Debugfs
- Ftrace

Linux® is a registered trademark of Linus Torvalds.

Multifunction device

Direct Memory Access

Pulse Width Modulation

Industrial I/O Linux subsystem

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Digital Filter for Sigma-Delta Modulator

Register map (Linux registers map abstraction API)

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

Stable: 11.03.2021 - 14:36 / Revision: 22.02.2021 - 14:45

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 explain how to configure the *timer (TIM)*[1] **when the peripheral is assigned to Linux®OS**:

- Configuring the timer **peripheral** to enable PWM, trigger or quadrature encoder.
- Configuring the **board**, e.g. TIM pins.

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

It is used by the TIM Linux driver that registers relevant information in PWM, IIO and counter frameworks.

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 *TIM internal peripheral*[1] is a multifunction device (MFD).

Each function is represented by a separate DT binding document:

- *STM32 TIM MFD device tree bindings*[3] document deals with core resources (e.g. registers, clock, DMAs)
- *STM32 TIM PWM device tree bindings*[4] document deals with PWM resources (e.g. PWM input/output pins)
- *STM32 TIM IIO trigger device tree bindings*[5] document deals with other internal peripheral triggering resources
- *STM32 TIM quadrature encoder device tree bindings*[6] document deals with quadrature encoder resources

3 DT configuration

This hardware description is a combination of both STM32 microprocessor and board device tree files. Refer to Device tree for more explanations about device tree file split.

The **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)

TIM nodes are declared in stm32mp151.dtsi[7].

DT root node (e.g. timers1...) and **DT child nodes** describe the TIM features such as:

- PWM
- trigger and quadrature encoder

They also describe hardware parameters such as registers address, clock and DMA.

```
timers1: timer@address {
        /* timer common resources */
        compatible = "st,stm32-timers";
        ...
        pwm {
                /* PWM*/
                compatible = "st,stm32-pwm";
        };
        timer@0 {
                compatible = "st,stm32h7-timer-trigger";
                /* trigger identifier (e.g. 0 for TIM1 triggers, 1 for TIM2... */
                reg = <0>;
        };
        counter {
                compatible = "st,stm32-timer-counter";
        };
};
```

Warning

This device tree part is related to STM32 microprocessors. It must be kept as is, without being modified by the end-user.

3.2 DT configuration (board level)

This part is used to configure and enable the TIM hardware used on the board:

- Enabling **DT root node** for the TIM instances in use (e.g timers1...) by setting **status = "okay";**
- Enabling **DT child node(s)** for the feature(s) in use (PWM input/output, trigger and quadrature encoder) by setting **status = "okay";**
- Configuring pins in use via pinctrl through **pinctrl-0**, **pinctrl-1** and **pinctrl-names**.

To enable PWM capture on the board (optional), DMA must be configured:

- Enable DMA channel(s) corresponding to the PWM input(s) by setting **dmas = <...>, <...>;** and matching **dma-names = "ch1", "ch3";**.

When PWM capture isn't used, it's recommended to disable DMA channels by default, to spare them for other usage:

- Disable DMA channels by setting **/delete-property/dmas** and **/delete-property/dma-names**

3.3 DT configuration examples

3.3.1 TIM configured in PWM mode

The example below shows how to configure **TIM1 channel 1** to act as:

- **PWM output on PE9**, e.g. TIM1_CH1 (See pinctrl device tree configuration and GPIO internal peripheral)
- PWM device tree provider (e.g. TIM1_CH1) used by a device tree consumer (e.g. like "pwm-leds"[8]).

```
/* select TIM1_CH1 alternate function 1 on 'PE9' */
pwm1_pins_a: pwm1-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, AF1)>;
                bias-pull-down;
                drive-push-pull;
                slew-rate = <0>;
        };
};

/* configure 'PE9' as analog input in low-power mode */
pwm1_sleep_pins_a: pwm1-sleep-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, ANALOG)>;
        };
};
```

Information

The PWM output doesn't require any DMA channel. Disable them if they are configured by default in the .dtsi file.

```
/* PWM DT provider on TIM1: "pwm1" */
&timers1 {
        status = "okay";
        /* spare all DMA channels since they are not needed for PWM output */
        /delete-property/dmas;
        /delete-property/dma-names;
        /* define pwm1 label */
        pwm1: pwm {
                /* configure PWM pins on TIM1_CH1 */
                pinctrl-0 = <&pwm1_pins_a>;
                pinctrl-1 = <&pwm1_sleep_pins_a>;
                pinctrl-names = "default", "sleep";
                /* enable PWM on TIM1 */
                status = "okay";
        };
};
```

- PWM DT user example

Information

The TIM PWM DT user specifier encodes 3 cells:

- PWM **number** (0 for CH1, 1 for CH2 and so on)
- PWM **period** in nanoseconds
- PWM **polarity** (0 for normal polarity or *PWM_POLARITY_INVERTED*)

```
/ {
        ...
        /* PWM DT user on TIM1_CH1: "pwm1", example with "pwm-leds"[8] */
        pwmleds {
                compatible = "pwm-leds";
                example {
                        label = "stm32-pwm-leds-example";
                        /* Use pwm1 channel 0 (e.g. TIM1_CH1) */
                        /* period in nanoseconds (500000), normal polarity (0) */
                        pwms = <&pwm1 0 500000 0>;
                        max-brightness = <127>;
                };
        };
};
```

3.3.2 TIM configured in PWM mode and trigger source

The example below shows how to configure **TIM1 channel 1** to act as:

- **PWM output on PE9**, e.g. TIM1_CH1 (See pinctrl device tree configuration and GPIO internal peripheral)
- **trigger source** (synchronous with PWM) for other internal peripheral such as STM32 ADC

```
/* select TIM1_CH1 alternate function 1 on 'PE9' */
pwm1_pins_a: pwm1-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, AF1)>;
                bias-pull-down;
                drive-push-pull;
                slew-rate = <0>;
        };
};

/* configure 'PE9' as analog input in low-power mode */
pwm1_sleep_pins_a: pwm1-sleep-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, ANALOG)>;
        };
};
```

Information

The PWM output doesn't require any DMA channel. Disable them if they are configured by default in the .dtsi file.

```
&timers1 {
        status = "okay";
        /* spare all DMA channels since they are not needed for PWM output */
        /delete-property/dmas;
        /delete-property/dma-names;
```

```
        pwm {
                /* configure PWM on TIM1_CH1 */
                pinctrl-0 = <&pwm1_pins_a>;
                pinctrl-1 = <&pwm1_sleep_pins_a>;
                pinctrl-names = "default", "sleep";
                /* enable PWM on TIM1 */
                status = "okay";
        };
        timer@0 {
                /* enable trigger on TIM1 */
                status = "okay";
        };
};
```

3.3.3 TIM configured in PWM input capture mode

The example below shows how to configure **TIM1 channel 1** in PWM input capture mode (e.g. period and duty cycle):

- Configure **PWM input on PE9**, e.g. TIM1_CH1 (See pinctrl device tree configuration and GPIO internal peripheral)

```
/* select TIM1_CH1 alternate function 1 on 'PE9' */
pwm1_in_pins_a: pwm1-in-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, AF1)>;
                bias-disable;
        };
};

/* configure 'PE9' as analog input in low-power mode */
pwm1_in_sleep_pins_a: pwm1-in-sleep-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, ANALOG)>;
        };
};
```

A DMA channel is required and must be configured depending on the PWM input channel:

- Select **DMA channel 1**, "ch1", to capture **PWM input channel 1 and/or 2**
- Select **DMA channel 3**, "ch3", to capture **PWM input channel 3 and/or 4**
- Select both "ch1" and "ch3" dmas to enable capture on all PWM input channels

```
&timers1 {
        status = "okay";
        /* Enable DMA "ch1" for PWM input on TIM1_CH1 */
        dmas = <&dmamux1 11 0x400 0x5>;
        dma-names = "ch1";
        pwm {
                /* configure PWM input pins, e.g. TIM1_CH1 */
                pinctrl-0 = <&pwm1_in_pins_a>;
                pinctrl-1 = <&pwm1_in_sleep_pins_a>;
                pinctrl-names = "default", "sleep";
                /* enable PWM on TIM1 */
                status = "okay";
        };
};
```

Information

DMA channels 1 and/or 3 for each TIM can be picked from the **"dmas"** list in stm32mp151.dtsi[7] file

3.3.4 TIM configured as quadrature encoder interface

The example below shows how to configure **TIM1** to interface with a quadrature encoder:

- **Configure PE9 and PE11 as encoder input pins**, e.g. TIM1_CH1, TIM1_CH2 (see pinctrl device tree configuration and GPIO internal peripheral)

Information

On the STM32MP157X-DKX discovery board, TIM1_CH1 and TIM1_CH2 signals are accessible via the D6 and D10 pins of the Arduino Uno connector.

```
tim1_in_pins_a: tim1-in-pins-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, AF1)>, /* TIM1_CH1 */
                         <STM32_PINMUX('E', 11, AF1)>; /* TIM1_CH2 */
                bias-disable;
        };
};

tim1_in_pins_sleep_a: tim1-in-pins-sleep-0 {
        pins {
                pinmux = <STM32_PINMUX('E', 9, ANALOG)>, /* TIM1_CH1 */
                         <STM32_PINMUX('E', 11, ANALOG)>; /* TIM1_CH2 */
        };
};
```

```
&timers1 {
        status = "okay";
        /delete-property/dmas;                          /* spare all DMA channels since they
are not required for quadrature encoder interface */
        /delete-property/dma-names;
        counter {
                pinctrl-0 = <&tim1_in_pins_a>;          /* configure TIM1_CH1 and TIM1_CH2
as encoder input pins */
                pinctrl-1 = <&tim1_in_pins_sleep_a>;
                pinctrl-names = "default", "sleep";
                status = "okay";                        /* enable Encoder interface mode on
TIM1 */
        };
};
```

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

Please refer to the following links for additional information:

- 1.01.1 TIM internal peripheral
- Device tree
- Documentation/devicetree/bindings/mfd/stm32-timers.txt , STM32 TIM MFD device tree bindings
- Documentation/devicetree/bindings/pwm/pwm-stm32.txt , STM32 TIM PWM device tree bindings
- Documentation/devicetree/bindings/iio/timer/stm32-timer-trigger.txt , STM32 TIM trigger device tree bindings
- Documentation/devicetree/bindings/counter/stm32-timer-cnt.txt , STM32 TIM quadrature encoder device tree bindings
- 7.07.1 stm32mp151.dtsi , STM32.dtsi file
- 8.08.1 Documentation/devicetree/bindings/leds/leds-pwm.txt , PWM LEDs device tree bindings

Operating System

Pulse Width Modulation

Device Tree

Multifunction device

Industrial I/O Linux subsystem

Direct Memory Access