IIO libiio

Contents

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

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: 30.03.2021 - 13:19 / Revision: 29.03.2021 - 09:31

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-v3

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

```
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
```

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

```
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

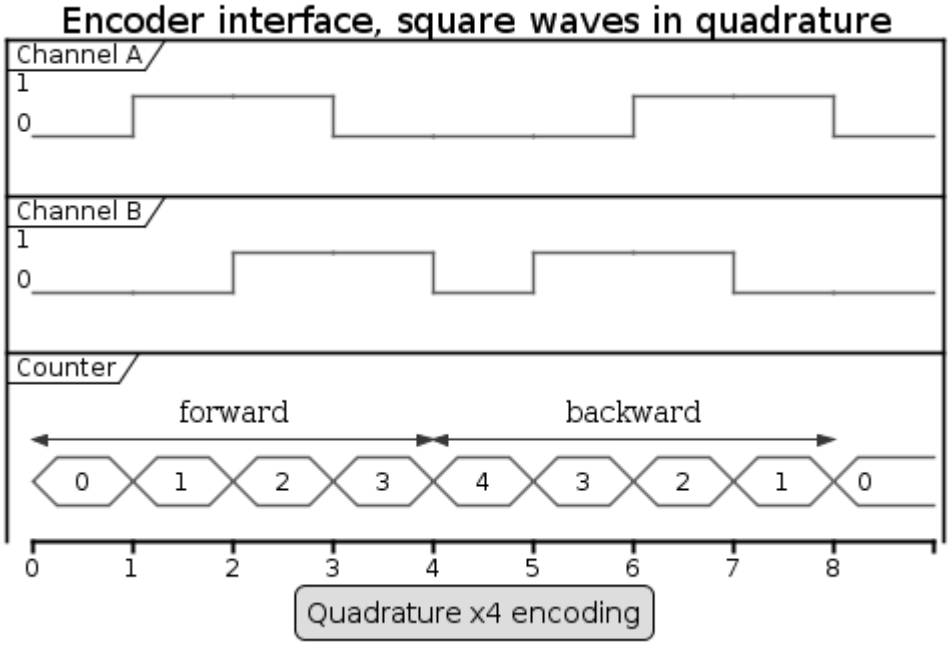
On the STM32MP157X-DKX discovery board, PD1, PG3, TIM1_CH1 and TIM1_CH2 signals are accessible via respectively the D7, D8, D6 and D10 pins of the Arduino Uno connector.

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: 17.02.2021 - 16:24 / Revision: 17.02.2021 - 16:22

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.

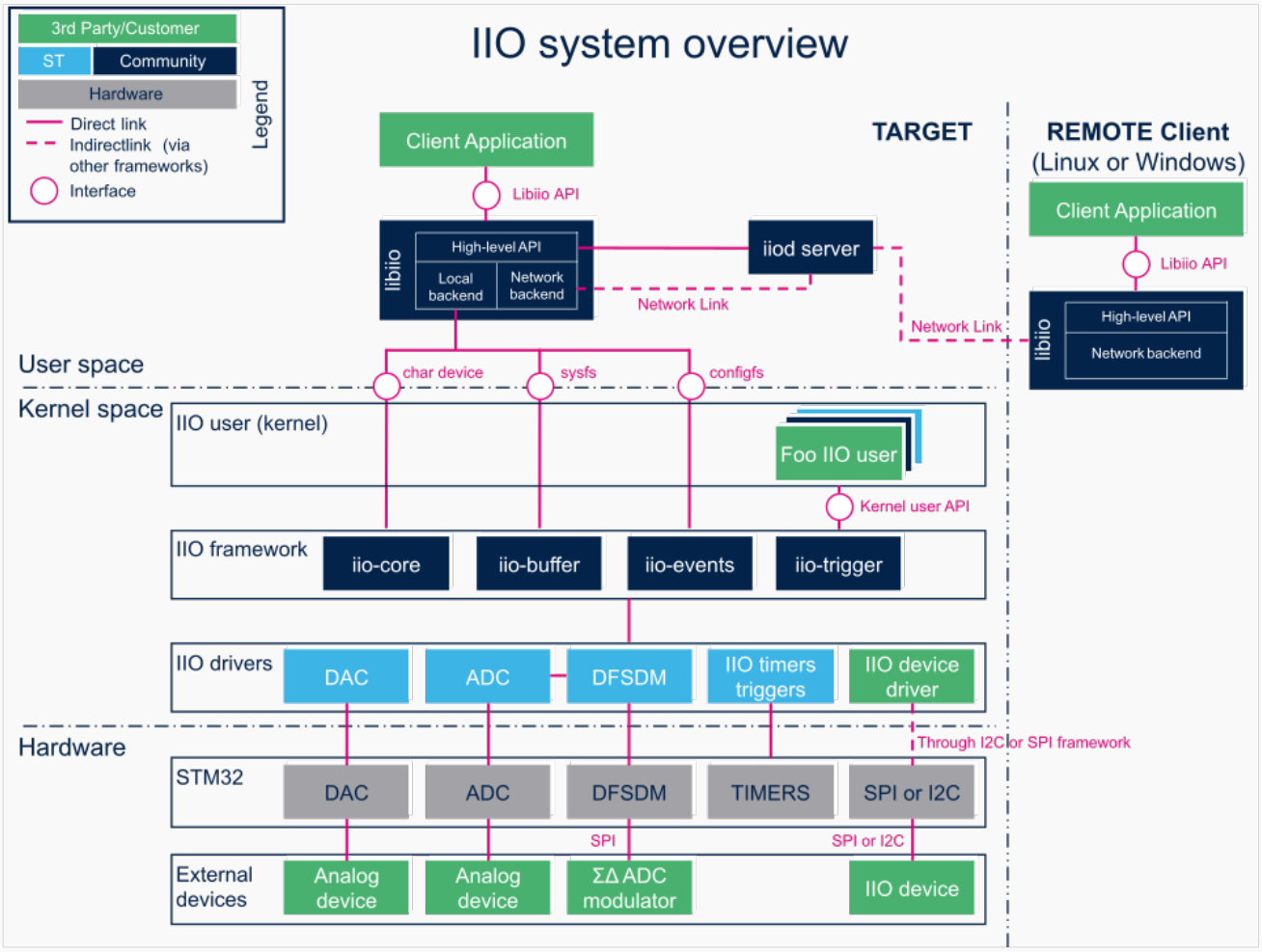
Here are some examples of supported sensor types in IIO:

- 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

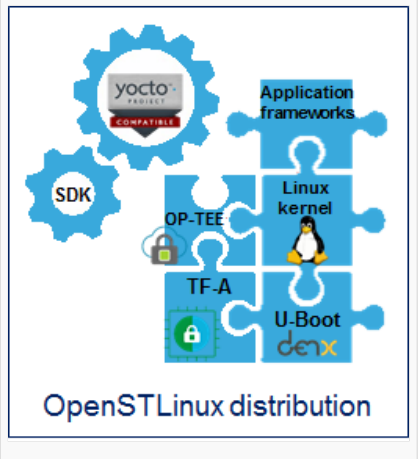
Stable: 17.11.2021 - 07:46 / Revision: 19.10.2021 - 17:08

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

Contents

1 What is the OpenSTLinux distribution?

The OpenSTLinux distribution, running on the Arm®Cortex®-A processor(s), is a sub-part of the STM32MPU Embedded Software distribution.



OpenSTLinux is a Linux® distribution based on the OpenEmbedded build framework.

It includes the following collection of software components:

- OpenSTLinux BSP (OP-TEE secure OS, boot chain and Linux kernel)
- Application frameworks such as the following Linux application frameworks (non-exhaustive list):
  - Wayland-Weston as a display/graphic framework
  - Gstreamer as a multimedia framework
  - Advanced Linux Sound Architecture (ALSA) libraries

As explained in the OpenEmbedded article, the files used to build an image are stored in layers that come from different sources, and that are configured for this image.
Only layers specific to the OpenSTLinux distribution (for example the OpenSTLinux Board Support Package layer) are detailed in the Layers chapter below. Community layers referenced in this article are not listed.

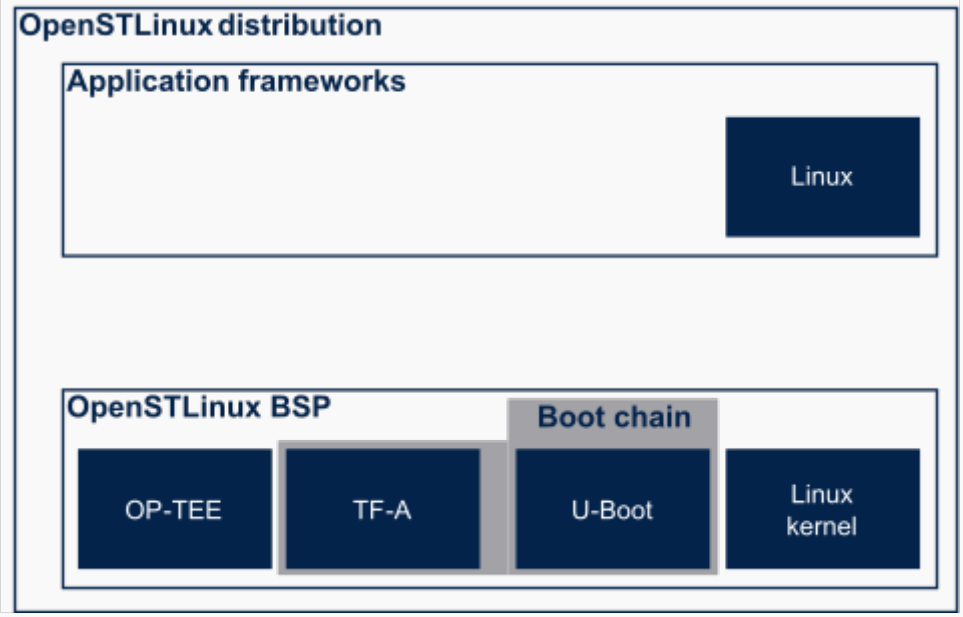
The Machines chapter introduces the machine metadata (the information needed to build a kernel for specific target boards), while the Images chapter specifies the images that contain the information needed to build the user space. The Distros chapter describes the available distros configurations.

2 Software architecture overview

The figure below gives an overview of the OpenSTLinux architecture. You can obtain more details by clicking on one of the sub-levels.

The **Linux application frameworks** (or Linux user space components) that rely on the OpenSTLinux BSP, mainly come from open-source communities.

OpenEmbedded core components (community layers) combined with STMicroelectronics-specific layers (for example the BSP layer) give a consistent starting point to develop customer applications based on a standardized interface.



OpenSTLinux architecture overview

3 OpenSTLinux concept

To build an OpenSTLinux based software, a combination of machine, image and distro must be selected.
Available machines, images and distros are listed below.

More information on how to compile and use an OpenSTLinux distribution can be found in the **Distribution Package** articles.

3.1 Layers

This chapter describes the layers developed by STMicroelectonics.

STMicroelectronics' strategy is to organize the layers in order to split BSP descriptions from applications and frameworks.
By doing this, any BSP can be combined with any framework, or no framework at all.

Layer	Description
meta-st-stm32mp	BSP Layer for stm32mp
meta-st-openstlinux	OpenSTLinux layer - framework/image settings

For the detailed content of the layers, please check STM32MP15_OpenSTLinux_release_note#Detailed delivery content

In addition, in OpenSTLinux distribution, a layer (meta-st-stm32mp-addons) has been created to manage STM32CubeMX integration.
If necessary, you can also create your own layer. The procedure to do so is explained in How_to_create_a_new_open_embedded_layer.

Layer	Description
meta-st-stm32mp-addons	BSP Layer addons for stm32mp (CubeMX Machine)
meta-***my-custo-layer***	framework addons/customization

3.2 Machines

Since ecosystem release v3.1.0

Machine	Description
stm32mp1	Machine configuration for STM32MP1 microprocessor device boards
stm32mp15-eval	Machine configuration for STM32MP1 evaluation board (Only STM32MP157C-EV1 in trusted boot mode on SDCard)
stm32mp15-disco	Machine configuration for STM32MP1 disco board (Only STM32MP157C-DK2 in trusted boot mode on SDCard)

For ecosystem release v3.0.0

Machine	Description
stm32mp1	Machine configuration for STM32MP1 microprocessor device boards
	Machine configuration for STM32MP1 evaluation board (Only STM32MP157C-

Machine	Description
stm32mp1-eval	EV1 in trusted boot mode on SDCard)
stm32mp1-disco	Machine configuration for STM32MP1 disco board (Only STM32MP157C-DK2 in trusted boot mode on SDCard)

3.3 Images

Image	Description
Official images	
st-image-weston	OpenSTLinux weston image with basic Wayland support (if enabled in distro)
Supported images	
st-image-core	OpenSTLinux core image
Images, proposed as example only	
st-example-image-qt	ST example of image based on QT framework
st-example-image-qtwayland	ST example of image based on QT framework (available for ecosystem release v2.1.0)

3.4 Distros

For further details on distributions (distros), read "Creating a distribution" [1]

Distro	Description
openstlinux-eglfs	OpenSTLinux featuring eglfs - no X11, no Wayland
openstlinux-weston	OpenSTLinux featuring Weston/Wayland
nodistro	*** DEFAULT OPENEMBEDDED SETTING : DISTRO is not defined ***

3.5 Reference source code

The TF-A, U-Boot, OP-TEE and kernel components have been configured by default to use **tarball + patches** for source code.

If you prefer to use **github**® as a source code reference, you need to update your local.conf file

Refer to the section ‘Configure STM32MP default version to github’ in local.conf :

```
# =========================================================================
# Configure STM32MP default version to github
# =========================================================================
#STM32MP_SOURCE_SELECTION_pn-linux-stm32mp = "github"
#STM32MP_SOURCE_SELECTION_pn-optee-os-stm32mp = "github"
#STM32MP_SOURCE_SELECTION_pn-tf-a-stm32mp = "github"
#STM32MP_SOURCE_SELECTION_pn-u-boot-stm32mp = "github"
```

Then just uncomment the line to switch the component to github® for source code.

4 How to get the software for this distribution

The OpenSTLinux distribution is available through the three Packages (Starter, Developer, and Distribution) of the STM32MPU Embedded Software distribution.

5 References

- Creating a distribution

Open Portable Trusted Execution Environment

Operating System

Board support package

Trusted Firmware for Arm® Cortex®-A

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