UWB L1 API

Qorvo

Release R12.7.0-405-gb33c5c4272

Contents

1 Overview

The L1 component is responsible of the **Layer 1** as defined by the OSI model, also know as the **Physical layer**.

L1 provides to the Layer 2, also referred as the MAC, an abstraction of the Hardware regarding the networking features. It allows to translate logical communications requests from the MAC into **hardware-specific** operations for transmitting and receiving UWB RF signals.

1.1 Porting guide

1.1.1 Initialization

When integrating the UWB stack, L1 component must be initialized. Each of its sub-components `LLHW` and `L1 Config` have their own initialization APIs:

- *l1_config_init()*;
- *llhw_init()*.

Note: API `l1_config_init` must always be called prior to `llhw_init`.

```
#include "l1_config.h"
#include "llhw.h"

[...]

enum qerr r = l1_config_init(NULL);
if (r)
      return r;
r = llhw_init();
if (r)
      return;
```

Important: Some platform specific ops can be registered when initializing L1 Config (here empty since set to `NULL`). For more details, please refer to *Platform specific operations*.

When deinitializing the UWB stack, L1 component must be deinitialized as well. This must be done by calling the following APIs:

- *l1_config_deinit()*;
- *llhw_deinit()*.

```
#include "l1_config.h"
#include "llhw.h"

[...]

llhw_deinit();
l1_config_deinit();
```

1.1.2 L1 Config options

1.1.2.1 Volatile or persistent configuration

Depending on customer needs, the L1 Config can be volatile or persistent.

- When it is **volatile**, the key values are stored in **SRAM only**, and will be lost at next reboot. Values are thus always reset to default after reboot.
- When it is **persistent**, the key values are stored in **persistent memory**, such as NVRAM, and loaded in RAM after each reboot. A configured value is not lost after a reboot.

Compilation flag `CONFIG_L1_CONFIG_VOLATILE` allows to activate the volatile mode. If not set, the persistent mode is used.

1.1.2.1.1 Persistent storage APIs

When L1 configuration is **non-volatile** (`CONFIG_L1_CONFIG_VOLATILE` = n), functions allowing to store to and read from the persistent memory must be implemented.

Here are the APIs required:

1.1.2.1.2 l1_config_load_from_persistent_memory

enum qerr **l1_config_load_from_persistent_memory**(struct *l1_config* *l1_config)

Load Configuration and calibration data structure from persistent memory to RAM.

Parameters

- **l1_config** (`struct l1_config*`) – pointer to RAM Configuration and Calibration data.

1.1.2.1.3 Return

QERR_SUCCESS or error.

1.1.2.1.4 l1_config_store_to_persistent_memory

enum qerr **l1_config_store_to_persistent_memory**(struct *l1_config* *l1_config)

Store Configuration and calibration data structure to persistent memory.

Parameters

- **l1_config** (struct *l1_config**) – pointer to RAM Configuration and Calibration data.

1.1.2.1.5 Return

QERR_SUCCESS or error.

1.1.2.1.6 l1_config_get_persistent_memory

const uint32_t ***l1_config_get_persistent_memory**(void)

Get address of L1 configuration structure in persistent memory.

Parameters

- **void** – no arguments

1.1.2.1.7 Return

structure address in persistent memory.

1.1.2.1.8 Persistent storage default implementation

The implementation of persistent storage APIs is platform specific.

A **default implementation** of those functions, based on the `qflash` **QHAL APIs**, is available in the file `l1_config_custom.c`. Enabling the compilation flag `CONFIG_L1_CONFIG_CUSTOM_DEFAULT` allows to build that default implementation.

However, it is also possible to provide **an external implementation** of those functions.

When using the default implementation, two options are provided regarding the section where the configuration is stored in persistent memory. The choice is made by enabling one of the two following compilation flags:

- `CONFIG_L1_CONFIG_CUSTOM_DEFAULT_USE_TEXT_SECTION`: the L1 configuration is stored in the **same section as the code**.

> **Warning:** That option cannot be used with a signed firmware, because updating the configuration values would cause the signature check to fail at next reboot.

- `CONFIG_L1_CONFIG_CUSTOM_DEFAULT_USE_DEDICATED_SECTION`: the L1 configuration is stored in its **dedicated section**, named `.l1_config_persist_storage`.

1.1.2.2 Size of itemized keys

Some of the configuration keys are itemized. As an example, multiple antenna sets can be defined for different use cases (Radar, PDoA, Ranging with no PDoA, etc.). All their related keys start with `ant_set` followed by the index of the itemized key:

- Antenna set 0: `ant_set0.tx_ant_path`, `ant_set0.rx_ants`, `ant_set0.*`
- Antenna set 1: `ant_set1.tx_ant_path`, `ant_set1.rx_ants`, `ant_set1.*`

All the projects do not have the same needs regarding the number of itemized parameters. For example, some customer boards will contain only one antenna, so having multiple entries for antennas and antenna sets is useless. On the contrary, some boards will contain 5 different antennas, so many combinations could be needed.

For that reason, the size of some itemized keys is configurable via compilation flags:

- `CONFIG_L1_CONFIG_ANT_NUM`: defines the maximum number of **Antenna Paths**;
- `CONFIG_L1_CONFIG_ANT_PAIR_NUM`: defines the maximum number of **Antenna Pairs**.
- `CONFIG_L1_CONFIG_ANT_SET_NUM`: defines the maximum number of **Antenna Sets**.
- `CONFIG_L1_CONFIG_PDOA_LUT_NUM`: defines the maximum number of PDoA Lookup Tables.

1.1.3 Platform specific operations

Some platform specific operations can be registered when initializing L1 Config, see *l1_config_init()*.

The available operations are defined by the *struct l1_config_platform_ops*:

- `reset_to_default`: called when configuration is reset to default, at the end of the process.

Note: If there is no need to register specific platform operations, `l1_config_init` API can be called with empty argument `ops`:

```
r = l1_config_init(NULL);
```

1.1.3.1 Reset to default

API `uwbmac_reset_calibration` allows to restore the whole configuration, i.e. resetting all the parameters to their default values. That behaviour is common to all the platforms.

However sometimes there is a need to **overload those default** values depending on the project, or to **perform specific actions**.

Such a typical need is when devices are calibrated in factory, where some of the configuration parameters, for example the TX power or the antenna delays, are calibrated per device, and values are stored in the OTP memory. In that case, when restoring the configuration, it is required to **read values from the OTP**, and to store them in the configuration.

Note: The values stored in the OTP are here **default values only**. They must be read and copied to L1 config (the SRAM memory, and optionally the Flash in case of persistent mode) when the configuration is reset (=restored) to default. But once reset, those parameters can still be overwritten by the user, using common API `uwbmac_set_calibration`.

Since all projects do not calibrate their chips, nor contain the same OTP map, that mecanism is platform specific, and must be **implemented by the integrator** or the UWB stack. That specific implementation must be done in ops `reset_to_default` from *struct l1_config_platform_ops*.

Here is an example of an ops which is reading the crystal trim value from the OTP, at a specific address.

```
#include "l1_config.h"
#include "qotp.h"

enum qerr l1_config_platform_ops_reset_to_default(void)
{
    enum qerr r;
    uint32_t val32;
    uint8_t xtal_trim;

    /* Read xtal trim. Ignore value if 0. */
    r = qotp_read(OTP_XTAL_TRIM_ADDRESS, &val32, 1);
    if (r)
        return r;
    xtal_trim = val32 & XTAL_TRIM_MASK;
    if (xtal_trim != 0)
        l1_config_store_key("xtal_trim", &xtal_trim, sizeof(xtal_trim));

    return QERR_SUCCESS;
}

struct l1_config_platform_ops l1_config_platform_ops = {
    .reset_to_default = l1_config_platform_ops_reset_to_default,
};

void main(void) {
      enum qerr r;
      r = l1_config_init(&l1_config_platform_ops);
      if (r)
            return;
      /* [...] */
}
```

1.1.3.1.1 QM33 specific

For QM33 and DW3000, the **factory calibration procedure** uses the **legacy** TX Power control format, instead of the linear l1_tx_power_index. Those calibration values are stored in the OTP. They can be read in the l1_config platform specific ops *Reset to default*, in order to overwrite default values when resetting the configuration.

But since the configuration model uses TX Power indexes, the TX Power control values read from the OTP cannot be used as is. They must first be converted to their corresponding TX Power index. The API *llhw_convert_tx_power_to_index()* allows to convert a one-byte TX Power value into its corresponding one-byte TX Power index.

Note: It is considered that the same one-byte TX power control value is used for all parts of the frame (STS, SHR, PHR and DATA payload).

Following is an example of a *Reset to default* allowing to read a TX Power value from the OTP, and overwrite L1 config TX Power index parameter.

```c
#include "l1_config.h"
#include "qotp.h"

enum qerr l1_config_platform_ops_reset_to_default(void)
{
    enum qerr r;
    uint32_t tx_power_otp;
    uint32_t tx_power_idx;
    uint8_t tx_power_idx_u8;
    uint8_t tx_power_u8;

    /* Will configure the TX power index for Ant Path 0, and Channel 5. */
    char key_str_tx_power_idx[] = "ant0.ch5.ref_frame0.tx_power_index";

    /* Read TX power for Channel 5 from the OTP. */
    r = qotp_read(OTP_CH5_TXPOWER_ADDRESS, &tx_power_otp, 1);
    if (r)
        return r;

    /* Conversion from TX power to index requires that same value is calibrated for all TX power
     * sections. */
    tx_power_u8 = tx_power_otp & 0xFF;
    if ((((tx_power_otp >> 8) & 0xFF) != tx_power_u8) ||
        (((tx_power_otp >> 16) & 0xFF) != tx_power_u8) ||
        (((tx_power_otp >> 24) & 0xFF) != tx_power_u8))
        return QERR_EINVAL;

    r = llhw_convert_tx_power_to_index(channel, tx_power_u8, &tx_power_idx_u8);
    if (r)
        return r;

    tx_power_idx = tx_power_idx_u8 | ((uint32_t)tx_power_idx_u8 << 8) |
                   ((uint32_t)tx_power_idx_u8 << 16) | ((uint32_t)tx_power_idx_u8 << 24);

    l1_config_store_key(key_str_tx_power_idx, &tx_power_idx, sizeof(tx_power_idx));

    return QERR_SUCCESS;
}

struct l1_config_platform_ops l1_config_platform_ops = {
    .reset_to_default = l1_config_platform_ops_reset_to_default,
};
```

2 L1 API

2.1 L1 config API

2.1.1 struct l1_config_platform_ops

struct **l1_config_platform_ops**

Platform specific callback to used by L1 config plugin.

2.1.1.1 Definition

```
struct l1_config_platform_ops {
    enum qerr (*reset_to_default)(void);
}
```

2.1.1.2 Members

reset_to_default
Called when configuration is reset to default.

Return: QERR_SUCCESS or error.

2.1.2 l1_config_init

enum qerr **l1_config_init**(struct *l1_config_platform_ops* *platform_ops)
Initialize L1 config.

Parameters

- **platform_ops** (struct *l1_config_platform_ops**) – Platform specific operations.

2.1.2.1 Description

Initialize the plugin and register platform specific ops, if any. When configuration uses persistent mode, structure from persistent memory is copied to RAM cache.

2.1.2.2 Return

QERR_SUCCESS or error.

2.1.3 l1_config_deinit

void **l1_config_deinit**(void)
Deinitialize L1 config.

Parameters

- **void** – no arguments

2.2 L1 core API

2.2.1 llhw_init

enum qerr **llhw_init**(void)
Init Low-Level Hardware layer.

Parameters

- **void** – no arguments

2.2.1.1 Return

QERR_SUCCESS or error.

2.2.2 llhw_deinit

void **llhw_deinit**(void)

DeInit Low-Level Hardware layer.

Parameters

- **void** – no arguments

2.2.3 llhw_lpm_enter

void **llhw_lpm_enter**(void)

Prepare llhw to enter in Low Power Mode.

Parameters

- **void** – no arguments

2.2.4 llhw_lpm_exit

void **llhw_lpm_exit**(void)

DeInit Low-Level Hardware layer.

Parameters

- **void** – no arguments

2.2.5 llhw_convert_tx_power_to_index

enum qerr **llhw_convert_tx_power_to_index**(int channel, uint8_t tx_power, uint8_t *tx_power_idx)

Convert a TX power value into its corresponding power index.

Parameters

- **channel** (int) – the channel for which the TX power index must be calculated.
- **tx_power** (uint8_t) – the TX power to convert.
- **tx_power_idx** (uint8_t*) – the returned TX power index.

2.2.5.1 Description

The TX power index is a linear value where one unit means an attenuation of 0.25 dB compared to the maximum emitted power. The correspondance between an index and its corresponding TX power has been characterized in order to reduce the LO leakeage and maximize effective dynamics. The L1 configuration uses TX power indexes. For compatibility with chips which were calibrated using old model (TX power instead of index), the conversion from TX power to index is required.

2.2.5.2 Note

it is considered that the same TX power value is used for all parts of the frame.

2.2.5.3 Return

QERR_SUCCESS or error.

Index

L