DOCUMENTATION

Classification | PUBLIC

UWB-Stack Documentation

Qorvo

Release R11.9.2

Contents

1 UWB MAC APIs **1**
1.1 Architecture . . . 1
1.2 UWB MAC Public APIs . . . 2

2 UCI Documentation **53**
2.1 UCI Public APIs . . . 53

3 Power Saving Using Deep Sleep State **85**
3.1 Introduction . . . 85
3.2 Low Power States . . . 85
3.3 Timestamps and Durations . . . 86
3.4 DTU Implementation Considerations . . . 87
3.5 Giving the RCTU Timestamp in Advance . . . 90
3.6 MCPS Handling of Power Saving . . . 90

Index **95**

1 UWB MAC APIs

1.1 Architecture

Qorvo architecture contains a unified API to access to UWB MAC operations from different target configuration.

We call this unified layer the `UWBMAC API` and it also has a specific open/close architecture to allow additions of new protocols in the future.

This is achieved by having a core module, contained in `uwbmac/uwbmac.h` containing function abstracting a call to the UWB MAC.

Aside of this core module comes what we call the `helpers` which expose all the MAC operation associated with a specific protocol.

Note: At the moment our current SDK only contains the FiRa helper exposing FiRa operations

1.1.1 UWBMAC FLAVORS

While the API is unified, different “flavors” implementations exists depending on hardware and on where client’s code is running compared to where the UWB MAC is running.

All these implementations respect the same APIs, but they may differ in their initialization.

Note: At the moment our SDK only contains the EMBEDDED implementation.

1.1.1.1 UWBMAC EMBEDDED

Uwbmac embedded is used when MAC is located inside a dedicated chip. This implementation communicates witch the MAC directly.

This implementation doest not have additional interface for initialization.

As of now, the message layer is using skbuff with a fake netlink implementation. It is planned to move to a cbor implementation in the future.

1.2 UWB MAC Public APIs

1.2.1 UWB MAC API

1.2.1.1 UWBMAC_MAX_CHANNEL_COUNT

UWBMAC_MAX_CHANNEL_COUNT()
Maximum number of channels in use at the same time.

1.2.1.2 uwbmac_call_region_cb_t

void **uwbmac_call_region_cb_t**(void **user_data*, uint32_t *call_id*, struct *uwbmac_msg *call_params*)
Receive a region call callback.

Parameters

- **user_data** (*void **) – data given when registering this callback.
- **call_id** (*uint32_t*) – the region call identifier.
- **call_params** (*struct uwbmac_msg **) – the payload of the callback.

1.2.1.2.1 Return

nothing.

1.2.1.3 uwbmac_tracing_cb_t

void **uwbmac_tracing_cb_t**(const char **fmt*, ...)
Receive a tracing callback.

Parameters

- **fmt** (*const char **) – string format of the trace.
- **ellipsis** (*ellipsis*) – variable list of arguments.

1.2.1.3.1 Return

nothing.

1.2.1.4 uwbmac_get_device_count

int **uwbmac_get_device_count**(struct uwbmac_context **context*, int **count*)
Get the number of uwb chips available.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **count** (*int **) – Number of uwb devices.

1.2.1.4.1 Return

UWBMAC_SUCCESS or error.

1.2.1.5 uwbmac_get_supported_channels

int **uwbmac_get_supported_channels**(struct uwbmac_context **context*, uint16_t **channels*)
Get the supported UWB channels

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **channels** (*uint16_t **) – (out parameter) bitmask for supported channels. First bit is for channel 0, and so on.

1.2.1.5.1 Return

UWBMAC_SUCCESS or error.

1.2.1.6 uwbmac_init_device

int **uwbmac_init_device**(struct uwbmac_context **context*, unsigned int *idx*)
Fill the corresponding device information.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **idx** (*unsigned int*) – index of the device.

1.2.1.6.1 NOTE

use *struct uwbmac_get_device_count* to check how many devices are present.

1.2.1.6.2 Return

UWBMAC_SUCCESS or error.

1.2.1.7 uwbmac_channel_create

int **uwbmac_channel_create**(struct uwbmac_context **context*, struct uwbmac_channel **channel*)
Create a new channel.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **channel** (*struct uwbmac_channel **) – The channel to be created.

1.2.1.7.1 Return

UWBMAC_SUCCESS or error.

1.2.1.8 uwbmac_channel_release

int **uwbmac_channel_release**(struct uwbmac_channel **channel*)
Release a channel.

Parameters

- **channel** (*struct uwbmac_channel **) – The channel to be released.

1.2.1.8.1 Return

UWBMAC_SUCCESS or error.

1.2.1.9 uwbmac_channel_set_timeout

int **uwbmac_channel_set_timeout**(struct uwbmac_channel **channel*, int *timeout*)
Set a timeout on a channel.

Parameters

- **channel** (*struct uwbmac_channel **) – The channel .
- **timeout** (*int*) – The timeout in seconds.

1.2.1.9.1 Return

UWBMAC_SUCCESS or error.

1.2.1.10 uwbmac_channel_receive

int **uwbmac_channel_receive**(struct uwbmac_channel **channel*)
Ask channel to process incoming messages if any.

Parameters

- **channel** (*struct uwbmac_channel **) – The channel that should process the messages.

1.2.1.10.1 Return

UWBMAC_SUCCESS or error.

1.2.1.11 uwbmac_register_report_callback

int **uwbmac_register_report_callback**(struct uwbmac_channel **channel*, *uwbmac_call_region_cb_t msg_cb*, void **user_data*)

Register a region callback for a specific channel.

Parameters

- **channel** (*struct uwbmac_channel **) – The channel associated with this callback.
- **msg_cb** (uwbmac_call_region_cb_t) – Callback to call when a report is available on this channel.
- **user_data** (*void **) – Context to give back to callback.

1.2.1.11.1 Description

This function registers the callback to call in case of a mac event.

1.2.1.11.2 NOTE

In embedded application, the callback might be called from MAC context, large treatments should be deferred.

1.2.1.11.3 Return

UWBMAC_SUCCESS or error.

1.2.1.12 uwbmac_init

int **uwbmac_init**(struct uwbmac_context ***context*)

Initialize the UWB MAC and return an UWB MAC context.

Parameters

- **context** (*struct uwbmac_context ***) – UWB MAC context.

1.2.1.12.1 NOTE

Some flavors of uwbmac have their own init method in their dedicated headers.

1.2.1.12.2 Return

UWBMAC_SUCCESS or error.

1.2.1.13 uwbmac_exit

void **uwbmac_exit**(struct uwbmac_context **context*)
Free the UWB MAC.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.

1.2.1.14 uwbmac_start

int **uwbmac_start**(struct uwbmac_context **context*)
Start the device.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.

1.2.1.14.1 Return

UWBMAC_SUCCESS or error.

1.2.1.15 uwbmac_stop

int **uwbmac_stop**(struct uwbmac_context **context*)
Stop the device.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.

1.2.1.15.1 Return

UWBMAC_SUCCESS or error.

1.2.1.16 uwbmac_is_started

bool **uwbmac_is_started**(struct uwbmac_context **context*)
Return the state of UWB MAC.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.

1.2.1.16.1 Return

true if UWB MAC is started, false otherwise.

1.2.1.17 uwbmac_poll_events

int **uwbmac_poll_events**(struct uwbmac_context **context*, uint64_t *timeout_us*)
Poll next event.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **timeout_us** (*uint64_t*) – Timeout, in micro-seconds, for the poll.

1.2.1.17.1 Description

This function is only available if you passed a NULL `event_loop_ops` to *uwbmac_init()*.

Passing 0 for `timeout_us` will make the call non-bloquing: existent pending event will be consume, and if there is no event the function will return instead of blocking.

Passing a value greated than 0 will make the function block until the timeout is reached when there is no pending event.

1.2.1.17.2 Return

UWBMAC_SUCCESS or error.

1.2.1.18 uwbmac_set_channel

int **uwbmac_set_channel**(struct uwbmac_context **context*, int *channel*)
Set UWB channel to use.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **channel** (*int*) – Uwb channel, supported channels depend on driver/hardware.

1.2.1.18.1 Return

UWBMAC_SUCCESS or error.

1.2.1.19 uwbmac_get_channel

int **uwbmac_get_channel**(struct uwbmac_context **context*, int **channel*)

Get used UWB channel.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **channel** (*int **) – Uwb channel, supported channels depend on driver/hardware.

1.2.1.19.1 Return

UWBMAC_SUCCESS or error.

1.2.1.20 uwbmac_set_calibration

int **uwbmac_set_calibration**(struct uwbmac_context **context*, const char **key*, void **value*, size_t *value_size*)

Send a calibration key and its value

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **key** (*const char **) – the calibration key name
- **value** (*void **) – the value for the specified calibration key
- **value_size** (*size_t*) – the size of the calibration key's value

1.2.1.20.1 Return

UWBMAC_SUCCESS or error.

1.2.1.21 uwbmac_get_calibration

int **uwbmac_get_calibration**(struct uwbmac_context **context*, const char **key*, void **value*, int **length*, size_t *max_length*)

Retrieve a calibration value.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **key** (*const char **) – The calibration key name.
- **value** (*void **) – The output array for the specified calibration key.
- **length** (*int **) – The length of the the resulting array.
- **max_length** (*size_t*) – Capacity of the array given.

1.2.1.21.1 Return

UWBMAC_SUCCESS or error.

1.2.1.22 struct uwbmac_list_calibration_context

struct **uwbmac_list_calibration_context**
context for listing calibration keys

1.2.1.22.1 Definition

```
struct uwbmac_list_calibration_context {
    const char *const *list;
    size_t key_count;
    void (*dealloc_cb)( struct uwbmac_list_calibration_context *list_calibration_ctx);
}
```

1.2.1.22.2 Members

list list of retrieved calibration keys

key_count count of retrieved calibration keys

dealloc_cb callback for freeing memory buffer

1.2.1.23 uwbmac_list_calibrations

int **uwbmac_list_calibrations**(struct uwbmac_context **context*, struct *uwbmac_list_calibration_context* **list_calibration_ctx*)
Retrieve the list calibration keys.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **list_calibration_ctx** (*struct uwbmac_list_calibration_context **) – Operation context.

1.2.1.23.1 Description

The list must be freed by client by calling list_calibration_ctx->dealloc_cb.

1.2.1.23.2 Return

UWBMAC_SUCCESS or error.

1.2.1.24 uwbmac_set_pan_id

int **uwbmac_set_pan_id**(struct uwbmac_context **context*, uint16_t *pan_id*)
Set pan id to use.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **pan_id** (*uint16_t*) – Pan id.

1.2.1.24.1 NOTE

HW Filtering is disabled if promiscuous mode is enabled.

1.2.1.24.2 Return

UWBMAC_SUCCESS or error.

1.2.1.25 uwbmac_set_short_addr

int **uwbmac_set_short_addr**(struct uwbmac_context **context*, uint16_t *short_addr*)
Set short address to use.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **short_addr** (*uint16_t*) – Short address.

1.2.1.25.1 NOTE

HW Filtering is disabled if promiscuous mode is enabled.

1.2.1.25.2 Return

UWBMAC_SUCCESS or error.

1.2.1.26 uwbmac_set_extended_addr

int **uwbmac_set_extended_addr**(struct uwbmac_context **context*, uint64_t *extended_addr*)

Set extended address to use.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **extended_addr** (*uint64_t*) – extended address.

1.2.1.26.1 NOTE

HW Filtering is disabled if promiscuous mode is enabled.

1.2.1.26.2 Return

UWBMAC_SUCCESS or error.

1.2.1.27 uwbmac_set_promiscuous_mode

int **uwbmac_set_promiscuous_mode**(struct uwbmac_context **context*, bool *on*)

Set promiscuous mode.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **on** (*bool*) – True to enable promiscuous mode.

1.2.1.27.1 Description

Control hardware filtering, if promiscuous mode is enabled, the hardware filtering is disabled.

1.2.1.27.2 Return

UWBMAC_SUCCESS or error.

1.2.1.28 uwbmac_set_scheduler

int **uwbmac_set_scheduler**(struct uwbmac_context **context*, const char **name*, const struct *uwbmac_msg *params*)

Set the scheduler responsible for managing the schedule, and configure its parameters.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **name** (*const char **) – Scheduler name.
- **params** (*const struct uwbmac_msg **) – Scheduler paraameters.

1.2.1.28.1 Description

Device should not be started for the moment.

1.2.1.28.2 Return

UWBMAC_SUCCESS or error.

1.2.1.29 uwbmac_get_scheduler

int **uwbmac_get_scheduler**(struct uwbmac_context **context*, char **name*, int *max_length*)

Get the scheduler name in use.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **name** (*char **) – The buffer to fill with the scheduler name.
- **max_length** (*int*) – Length of provided buffer.

1.2.1.29.1 Return

UWBMAC_SUCCESS or error.

1.2.1.30 uwbmac_close_scheduler

int **uwbmac_close_scheduler**(struct uwbmac_context **context*)

Close the current scheduler and all regions.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.

1.2.1.30.1 Return

UWBMAC_SUCCESS or error.

1.2.1.31 uwbmac_set_scheduler_parameters

int **uwbmac_set_scheduler_parameters**(struct uwbmac_context **context*, const char **name*, const struct *uwbmac_msg *params*)

Set the scheduler parameters.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **name** (*const char **) – Scheduler name.
- **params** (*const struct uwbmac_msg **) – Scheduler parameters.

1.2.1.31.1 Return

UWBMAC_SUCCESS or error.

1.2.1.32 uwbmac_get_scheduler_parameters

int **uwbmac_get_scheduler_parameters**(struct uwbmac_context **context*, const char **name*, struct *uwbmac_msg *reply*)

Get the scheduler parameters.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **name** (*const char **) – Scheduler name.
- **reply** (*struct uwbmac_msg **) – Message filled with the parameters.

1.2.1.32.1 Return

UWBMAC_SUCCESS or error.

1.2.1.33 uwbmac_set_regions

int **uwbmac_set_regions**(struct uwbmac_context **context*, const char **scheduler_name*, uint32_t *region_id*, const char **region_name*, const struct *uwbmac_msg *params*)

Set regions that populate the schedule.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **scheduler_name** (*const char **) – Scheduler name.
- **region_id** (*uint32_t*) – Identifier of the region, scheduler specific.
- **region_name** (*const char **) – Name of region to attach to the scheduler.
- **params** (*const struct uwbmac_msg **) – Region parameters.

1.2.1.33.1 Return

UWBMAC_SUCCESS or error.

1.2.1.34 uwbmac_set_region_parameters

int **uwbmac_set_region_parameters**(struct uwbmac_context **context*, const char **scheduler_name*, uint32_t *region_id*, const char **region_name*, const struct *uwbmac_msg *params*)

Set region parameters.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **scheduler_name** (*const char **) – Scheduler name.

- **region_id** (*uint32_t*) – Identifier of the region, scheduler specific.
- **region_name** (*const char **) – Name of region to attach to the scheduler.
- **params** (*const struct uwbmac_msg **) – Region parameters.

1.2.1.34.1 Return

UWBMAC_SUCCESS or error.

1.2.1.35 uwbmac_get_region_parameters

int **uwbmac_get_region_parameters**(struct uwbmac_context **context*, const char **scheduler_name*, uint32_t *region_id*, struct *uwbmac_msg *reply*)

Get region parameters.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **scheduler_name** (*const char **) – Scheduler name.
- **region_id** (*uint32_t*) – Identifier of the region, scheduler specific.
- **reply** (*struct uwbmac_msg **) – Replied region parameters.

1.2.1.35.1 Return

UWBMAC_SUCCESS or error.

1.2.1.36 uwbmac_call_scheduler

int **uwbmac_call_scheduler**(struct uwbmac_context **context*, const char **name*, uint32_t *call_id*, const struct *uwbmac_msg *params*, const struct uwbmac_channel **channel*)

Call scheduler specific procedure.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **name** (*const char **) – Scheduler name.
- **call_id** (*uint32_t*) – Identifier of the procedure, scheduler specific.
- **params** (*const struct uwbmac_msg **) – Scheduler call parameters.
- **channel** (*const struct uwbmac_channel **) – Channel to get response.

1.2.1.36.1 Return

UWBMAC_SUCCESS or error.

1.2.1.37 uwbmac_call_region

int **uwbmac_call_region**(struct uwbmac_context **context*, const char **scheduler_name*, uint32_t *region_id*, const char **region_name*, uint32_t *call_id*, const struct *uwbmac_msg *params*, const struct uwbmac_channel **channel*, struct *uwbmac_msg *reply*)

Call region specific procedure.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **scheduler_name** (*const char **) – Scheduler name.
- **region_id** (*uint32_t*) – Identifier of the region, scheduler specific.
- **region_name** (*const char **) – Name of the region to call.
- **call_id** (*uint32_t*) – Identifier of the procedure, region specific.
- **params** (*const struct uwbmac_msg **) – Region call parameters.
- **channel** (*const struct uwbmac_channel **) – Channel to get response if reply is not NULL.
- **reply** (*struct uwbmac_msg **) – If not NULL, wait for a reply and store its payload here.

1.2.1.37.1 NOTE

most calls to this function do not trigger a response, so reply must only be given when a reply is expected, in which case uwbmac_call_region_free must be called on the reply when done.

1.2.1.37.2 Return

UWBMAC_SUCCESS or error.

1.2.1.38 uwbmac_call_region_free

void **uwbmac_call_region_free**(struct *uwbmac_msg *reply*)

Free internal resources after uwbmac_call_region.

Parameters

- **reply** (*struct uwbmac_msg **) – The reply filled in by a call to uwbmac_call_region.

1.2.1.39 uwbmac_get_version

const char *__uwbmac_get_version__(void)
Get the uwbmac release version.

Parameters

- **void** – no arguments

1.2.1.39.1 Return

The release version string.

1.2.1.40 uwbmac_strerror

const char *__uwbmac_strerror__(uwbmac_error *error*)
Return a description of the given error code.

Parameters

- **error** (*uwbmac_error*) – The UWBMAC error code.

1.2.1.40.1 Return

a human-readable description of the error.

1.2.1.41 uwbmac_error_to_errno

int __uwbmac_error_to_errno__(uwbmac_error *error*)
Return a errno code.

Parameters

- **error** (*uwbmac_error*) – The UWBMAC error code.

1.2.1.41.1 Return

Equivalent errno code

1.2.1.42 uwbmac_set_scanning_mode

int __uwbmac_set_scanning_mode__(struct uwbmac_context **context*, bool *enabled*)
Enable or disable scanning.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **enabled** (*bool*) – True to enable ieee 802.15.4 scanning.

1.2.1.42.1 Description

This mode is only used for IEEE 802.15.4 scanning, actual control must be handled by the MLME running on the client side.

1.2.1.42.2 Return

UWBMAC_SUCCESS or error.

1.2.1.43 uwbmac_testmode_cb_t

void **uwbmac_testmode_cb_t**(void **user_data*, void **data*, int *length*)

Receive a testmode call response.

Parameters

- **user_data** (*void **) – data given when registering this callback.
- **data** (*void **) – response given.
- **length** (*int*) – length of data.

1.2.1.43.1 Return

nothing.

1.2.1.44 uwbmac_register_testmode_callback

int **uwbmac_register_testmode_callback**(struct uwbmac_context **context*, *uwbmac_testmode_cb_t msg_cb*, void **user_data*)

Register a testmode callback.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **msg_cb** (uwbmac_testmode_cb_t) – Callback to call when the result of the test is available.
- **user_data** (*void **) – Context to give back to callback.

1.2.1.44.1 Description

This function registers the callback to call in case of a mac event. The callback is called from MAC context, big treatments should be deferred.

1.2.1.44.2 NOTE

The msg sent to the callback should be freed by the APP using uwbmac_buf_free.

1.2.1.44.3 Return

UWBMAC_SUCCESS or error.

1.2.1.45 uwbmac_call_testmode

int **uwbmac_call_testmode**(struct uwbmac_context **context*, void **data*, int *length*)
Call a test mode function.

Parameters

- **context** (*struct uwbmac_context **) – UWB MAC context.
- **data** (*void **) – Test data.
- **length** (*int*) – Size of test data.

1.2.1.45.1 Description

Test mode allows to directly call the driver. This is expected to be called for tests. Test mode may be disabled in a device.

1.2.1.45.2 Return

UWBMAC_SUCCESS or error.

1.2.1.46 uwbmac_set_trace_cb

void **uwbmac_set_trace_cb**(*uwbmac_tracing_cb_t cb*)
Set the trace callback.

Parameters

- **cb** (uwbmac_tracing_cb_t) – Trace callback.

1.2.1.46.1 NOTE

That API is only required for embedded systems, and should only be called when not using zephyr OS. For zephyr OS, the trace function is implemented internally so that it is best optimized.

1.2.1.47 struct uwbmac_trace_info_t

struct **uwbmac_trace_info_t**
Trace module information

1.2.1.47.1 Definition

```
struct uwbmac_trace_info_t {
    char name[UWBMAC_TRACE_MODULE_NAME_MAX_SIZE];
    bool enable;
}
```

1.2.1.47.2 Members

name name of the trace module

enable true is trace module enabled, false otherwise

1.2.1.48 uwbmac_trace_enable

int **uwbmac_trace_enable**(char **module_name*, bool *enable*)
Enable/disable trace for a specific module

Parameters

- **module_name** (*char **) – Name of the module to set trace of.
- **enable** (*bool*) – true to enable, false to disable.

1.2.1.48.1 NOTE

That API is only required for embedded systems.

1.2.1.48.2 Return

UWBMAC_SUCCESS or error.

1.2.1.49 uwbmac_get_trace_modules

int **uwbmac_get_trace_modules**(struct *uwbmac_trace_info_t* ***info*, int **nb_modules*)
Retrieve info of all trace modules available

Parameters

- **info** (*struct uwbmac_trace_info_t ***) – output param where trace module informations are stored.
- **nb_modules** (*int **) – output param where number of modules is stored.

1.2.1.49.1 NOTE

That API is only required for embedded systems.

1.2.1.49.2 Return

UWBMAC_SUCCESS or error.

1.2.2 UWB MAC EMBEDDED API

1.2.2.1 uwbmac_region_call_reply

int **uwbmac_region_call_reply**(struct ieee802154_hw **hw*, struct sk_buff **reply*)
Reply to a region call.

Parameters

- **hw** (*struct ieee802154_hw **) – Pointer to MCPS hw instance.
- **reply** (*struct sk_buff **) – Reply message.

1.2.2.1.1 Return

0 or error.

1.2.2.2 uwbmac_event_report

int **uwbmac_event_report**(struct ieee802154_hw **hw*, u32 *port_id*, struct sk_buff **report*)
Report an event.

Parameters

- **hw** (*struct ieee802154_hw **) – Pointer to MCPS hw instance.
- **port_id** (*u32*) – Port id to use to notify upper layer.
- **report** (*struct sk_buff **) – Event report.

1.2.2.2.1 Return

0 or error.

1.2.2.3 uwbmac_handle_report

void **uwbmac_handle_report**(struct report_data **report*)
Handle reported event.

Parameters

- **report** (*struct report_data **) – Event report.

1.2.2.3.1 NOTE

This method is only used by embedded flavor. It must be used by the platform to process an event received from the MAC. It is recommended to handle events from a dedicated thread, as processing the event may imply reentrancy in the MAC.

1.2.2.4 uwbmac_testmode_reply

int **uwbmac_testmode_reply**(struct ieee802154_hw **hw*, struct uwbmac_buf **reply*)
Reply to a testmode call.

Parameters

- **hw** (*struct ieee802154_hw **) – Pointer to MCPS hw instance.
- **reply** (*struct uwbmac_buf **) – Reply message.

1.2.2.4.1 NOTE

This method is only used by embedded flavor.

1.2.2.4.2 Return

0 or error.

1.2.3 Fira helper

1.2.3.1 struct measurement_sequence_step

struct **measurement_sequence_step**
Fira measurement sequence step.

1.2.3.1.1 Definition

```
struct measurement_sequence_step {
    enum fira_measurement_type type;
    uint8_t n_measurements;
    uint8_t rx_ant_set_nonranging;
    uint8_t rx_ant_sets_ranging[2];
    uint8_t tx_ant_set_nonranging;
    uint8_t tx_ant_set_ranging;
}
```

1.2.3.1.2 Members

type Kind of ranging measurement done in this step.

n_measurements Number ranging round done in this step.

rx_ant_set_nonranging Antenna set ID, used to receive non-RFRAMEs.

rx_ant_sets_ranging Array of sets used to receive RFRAMEs.

tx_ant_set_nonranging Antenna set ID, used to tansmit non-RFRAMEs.

tx_ant_set_ranging Array of sets used to tansmit RFRAMEs.

1.2.3.1.3 Description

This structure contains a step of the measurement sequence executed by the region. It can be used to configure the region when inserted in a measurement sequence.

1.2.3.2 struct measurement_sequence

struct **measurement_sequence**
Fira measurement sequence.

1.2.3.2.1 Definition

```
struct measurement_sequence {
    size_t n_steps;
    struct measurement_sequence_step steps[FIRA_MEASUREMENT_SEQUENCE_STEP_MAX];
}
```

1.2.3.2.2 Members

n_steps Number of steps in the schedule.

steps Steps of the schedule.

1.2.3.2.3 Description

This structure contains the measurement sequence executed by the region.

1.2.3.3 struct session_parameters

struct **session_parameters**
Fira session parameters.

1.2.3.3.1 Definition

```
struct session_parameters {
    uint8_t device_type;
    uint8_t device_role;
    uint8_t ranging_round_usage;
    uint8_t sts_config;
    uint8_t multi_node_mode;
    uint16_t short_addr;
    uint16_t destination_short_address;
    uint32_t initiation_time_ms;
    uint32_t slot_duration_rstu;
    uint32_t round_duration_slots;
    uint32_t block_duration_ms;
    uint32_t block_stride_length;
    bool round_hopping;
    uint8_t priority;
    bool result_report_phase;
    uint8_t embedded_mode;
    uint16_t max_number_of_measurements;
    uint32_t max_rr_retry;
    uint8_t channel_number;
    uint8_t preamble_code_index;
    uint8_t rframe_config;
    uint8_t preamble_duration;
    uint8_t sfd_id;
    uint8_t psdu_data_rate;
    uint8_t phr_data_rate;
    uint8_t vupper64[FIRA_VUPPER64_SIZE];
    uint8_t session_key[FIRA_KEY_SIZE_MIN];
    uint8_t key_rotation;
    uint8_t key_rotation_rate;
    uint8_t aoa_result_req;
    uint8_t report_rssi;
    uint8_t report_tof;
    uint8_t report_aoa_azimuth;
    uint8_t report_aoa_elevation;
    uint8_t report_aoa_fom;
    uint32_t data_vendor_oui;
    uint8_t mac_fcs_type;
    uint8_t tx_adaptative_payload_power;
    uint8_t prf_mode;
    uint8_t cap_size_min;
    uint8_t cap_size_max;
    uint8_t number_of_sts_segments;
    struct measurement_sequence meas_seq;
    bool enable_diagnostics;
    uint32_t diags_frame_reports_fields;
    uint8_t sts_length;
}
```

1.2.3.3.2 Members

device_type Type of the device.

See *enum fira_device_type*.

device_role **[NOT IMPLEMENTED]** Role played by the device.

This parameter is not used in the current implementation.

Current implementation does not support decorrelation between the device's role and the device's type. The controller is always the initiator and the controlee is always the responder.

See *enum fira_device_role*.

ranging_round_usage The ranging mode used during a round.

See *enum fira_ranging_round_usage*.

sts_config it configures how system shall generate the STS. Possible values:

- 0x00: Static STS (default).
- 0x01: Dynamic STS.
- 0x02: RFU (Dynamic STS - Individual Key).
- 0x03: Provisioned STS.
- 0x04: RFU (Provisioned STS - Individual Key).

See *enum fira_sts_config*.

multi_node_mode The multi-node mode used during a round.

Current implementation does not support FIRA_MULTI_NODE_MODE_MANY_TO_MANY mode.

See enum *struct fira_multi_node_mode*.

short_addr Short address of the local device.

destination_short_address Short address of the destination controller.

initiation_time_ms Initiation time of the session in milliseconds.

slot_duration_rstu Duration of a slot in RSTU (1200RSTU=1ms).

round_duration_slots Number of slots per ranging round.

block_duration_ms Block size in unit of 1200 RSTU (same as ms).

block_stride_length Number of blocks to stride.

round_hopping Enable FiRa round hopping.

priority Priority of the session.

result_report_phase Enable result report phase.

Current implementation does not support enabling RRRM on controller/initiator, it works only on controlee/responder.

embedded_mode Message embedding behaviour. Possible values:

- 0: MODE_DEFERRED - Ranging messages do not embed control messages. Additional messages are required.
- 1: MODE_NON_DEFERRED - Ranging messages embed control messages

max_number_of_measurements Max number of measurements

max_rr_retry Number of failed ranging round attempts before stopping the session.

The value zero disable the feature.

channel_number UWB channel for this session.

preamble_code_index UWB preamble code index.

Possible values:

- 9-24: BPRF
- 25-32: HPRF

rframe_config The configuration of the frame.

Current implementation only supports deferred mode.

see enum *struct fira_rframe_config*.

preamble_duration

Possible values:

- 0x00: 32 symbols
- 0x01: 64 symbols (default)

See *enum fira_preambule_duration*.

sfd_id

Possible values:

- 0 or 2 in BPRF
- 1-4 in HPRF

See *enum fira_sfd_id*.

psdu_data_rate

Possible values:

- 0: 6.81Mbps (default)
- 1: 7.80 Mbps
- 2: 27.2 Mbps
- 3: 31.2 Mbps

See *enum fira_psdu_data_rate*.

phr_data_rate

Possible values:

- 0: 850 kbit/s
- 1: 6.81 Mbit/s.

See *enum fira_phr_data_rate*.

vupper64 vUpper64 used during Static STS ranging.

session_key session key used during Provisioned STS ranging.

key_rotation Enable/disable key rotation feature duringDynamic or Provisioned STS ranging.

Possible values:

- false: No key rotation.
- true: Key rotation enabled and period set by key_rotation_rate.

key_rotation_rate defines n, with 2^n being the rotation rate of somekeys used during Dynamic or Provisioned STS Ranging, n shall be in the range of 0<=n<=15.

aoa_result_req Activate local AoA report Possible values:

- false: No local AoA report
- true: -90 to +90

report_rssi Activate rssi report Possible values:

- 0: no rssi report
- 1: activate rssi report

report_tof Activate ToF report in RRRM. Possible values:

- false: No ToF report in RRRM
- true: ToF Report in RRRM

report_aoa_azimuth Activate AoA azimuth report in RRRM. Possible values:

- false: No AoA azimuth report in RRRM
- true: AoA azimuth Report in RRRM

report_aoa_elevation Activate AoA elevation report in RRRM. Possible values:

- false: No AoA elevation report in RRRM
- true: AoA elevation Report in RRRM

report_aoa_fom No Report AoA FOM in result message (0) Possible values:

- false: No AoA FOM report in RRRM
- true: AoA FOM Report in RRRM

data_vendor_oui Vendor OUI used to send and receive data using theranging frames.

mac_fcs_type **[NOT IMPLEMENTED]** The length of the Frame Check Sequence in the session.

Possible values:

- 0x00: CRC 16 (default)
- 0x01: CRC 32
- Values 0x02 to 0xFF: RFU

This parameter is not used in the current implementation.

See *enum fira_mac_fcs_type*.

tx_adaptative_payload_power **[NOT IMPLEMENTED]** Activate TX adaptive power.

Possible values:

- 0x00: Disabled
- 0x01: Enabled
- Values 0x02 to 0xFF: RFU

This parameter is not used in the current implementation.

prf_mode

Possible values:

- 0x00: 62.4 MHz PRF. BPRF mode (default)
- 0x01: 124.8 MHz PRF. HPRF mode.
- 0x02: 249.6 MHz PRF. HPRF mode with data rate 27.2 and 31.2 Mbps

See *enum fira_prf_mode*.

cap_size_min Contention access period minimum value. Default: 5

cap_size_max Contention access period maximum value. Default: round_duration_slots - 1

number_of_sts_segments **[NOT IMPLEMENTED]** Number of STS segments.

Possible values:

- 0x01: 1 STS Segment (default)
- 0x02: 2 STS Segments (HPRF only)
- 0x03: 3 STS Segments (HPRF only)
- 0x04: 4 STS Segments (HPRF only)
- Values 0x05 to 0xFF: RFU

This parameter is not used in the current implementation.

meas_seq sequence of measurement sequence steps,configures the Antenna Flexibility features.

enable_diagnostics activate the diagnostics for each round.

diags_frame_reports_fields select the fields to activate in the framereports stored in the diags. If the ENABLE_DIAGNOSTICS is not true this parameters does not activate the diags itself.

sts_length Number of symbols in a STS segment. Possible values:

- 0x00: 32 symbols
- 0x01: 64 symbols (default)
- 0x02: 128 symbols
- Values 0x03 to 0xFF: RFU

1.2.3.3.3 Description

This structure contains the session parameters sent to the Fira region. Current implementation does not use all the parameters defined below.

1.2.3.4 struct data_parameters

struct **data_parameters**
Data parameters.

1.2.3.4.1 Definition

```
struct data_parameters {
    uint8_t data_payload[FIRA_DATA_PAYLOAD_SIZE_MAX];
    int data_payload_len;
}
```

1.2.3.4.2 Members

data_payload Data payload to send.

data_payload_len Length of data to send.

1.2.3.5 struct controlee_parameters

struct **controlee_parameters**
Controlee parameters.

1.2.3.5.1 Definition

```
struct controlee_parameters {
    uint16_t address;
}
```

1.2.3.5.2 Members

address Controlee short address.

1.2.3.6 struct controlees_parameters

struct **controlees_parameters**
Controlees list parameters.

1.2.3.6.1 Definition

```
struct controlees_parameters {
    struct controlee_parameters controlees[FIRA_CONTROLEES_MAX];
    int n_controlees;
}
```

1.2.3.6.2 Members

controlees List of controlees.

n_controlees Number of controlees in the list.

1.2.3.7 enum aoa_measurements_index

enum **aoa_measurements_index**
AOA measurements.

1.2.3.7.1 Definition

```
enum aoa_measurements_index {
    FIRA_HELPER_AOA_AZIMUTH,
    FIRA_HELPER_AOA,
    FIRA_HELPER_AOA_ELEVATION,
    FIRA_HELPER_AOA_NB
};
```

1.2.3.7.2 Constants

FIRA_HELPER_AOA_AZIMUTH Retrieve AOA azimuth.

FIRA_HELPER_AOA Retrieve AOA (same as azimuth).

FIRA_HELPER_AOA_ELEVATION Retrieve AOA elevation.

FIRA_HELPER_AOA_NB Enum members number.

1.2.3.8 struct aoa_measurements

struct **aoa_measurements**
Fira Angle of Arrival measurements.

1.2.3.8.1 Definition

```
struct aoa_measurements {
    uint8_t rx_antenna_pair;
    uint8_t aoa_fom;
    int16_t aoa_2pi;
    int16_t pdoa_2pi;
}
```

1.2.3.8.2 Members

rx_antenna_pair Antenna pair index.

aoa_fom Estimation of local AoA reliability.

aoa_2pi Estimation of reception angle.

pdoa_2pi Estimation of reception phase difference.

1.2.3.8.3 Description

Contains the different results of the AOA measurements.

1.2.3.9 struct ranging_measurements

struct **ranging_measurements**
Fira ranging measurements.

1.2.3.9.1 Definition

```
struct ranging_measurements {
    uint16_t short_addr;
    uint8_t status;
    uint8_t slot_index;
    bool stopped;
    bool nlos;
    bool los;
    int32_t distance_mm;
    int16_t remote_aoa_azimuth_2pi;
    int16_t remote_aoa_elevation_pi;
    uint8_t remote_aoa_azimuth_fom;
    uint8_t remote_aoa_elevation_fom;
    struct aoa_measurements local_aoa_measurements[FIRA_HELPER_AOA_NB];
    uint8_t sp1_data[FIRA_DATA_PAYLOAD_SIZE_MAX];
    int sp1_data_len;
    uint8_t rssi;
    uint32_t payload_seq_sent;
}
```

1.2.3.9.2 Members

short_addr Address of the participating device.

status Zero if ok, or error reason.

slot_index in case of error, slot index where the error was detected.

stopped Ranging was stopped as requested [controller only].

nlos **[NOT IMPLEMENTED]** Not in line of sight indicator.

The current implementation does not compute nLOS.

los **[NOT IMPLEMENTED]** Line of sight indicator.

The current implementation does not compute LOS.

distance_mm Distance in mm.

remote_aoa_azimuth_2pi Estimation of reception angle in the azimuthof the participating device.

remote_aoa_elevation_pi Estimation of reception angle in theelevation of the participating device.

remote_aoa_azimuth_fom Estimation of azimuth reliability of theparticipating device.

remote_aoa_elevation_fom Estimation of elevation of theparticipating device.

local_aoa_measurements Table of estimations of local measurements.

sp1_data SP1 received data payload

sp1_data_len Length of received data.

rssi computed rssi

payload_seq_sent Data sequence number.

1.2.3.10 struct ranging_results

struct **ranging_results**
Fira ranging results.

1.2.3.10.1 Definition

```
struct ranging_results {
    uint8_t stopped_reason;
    uint32_t session_id;
    uint32_t block_index;
    uint32_t ranging_interval_ms;
    uint64_t timestamp_ns;
    int n_measurements;
    struct ranging_measurements measurements[FIRA_CONTROLEES_MAX];
    struct diagnostic_info *diagnostic;
}
```

1.2.3.10.2 Members

stopped_reason 0x00: Session was stopped due to stop request (0).

0x01 (controlee only): Session was stopped using in band signaling from the controller.

0x02 (controller only): Session was stopped due to maximum attempts reached with no response.

0xff: Session is running.

session_id Session id of the ranging result.

block_index Current block index.

ranging_interval_ms Current ranging interval in unit of ms.formula: (block size * (stride + 1))

timestamp_ns **[NOT IMPLEMENTED]** Timestamp in nanoseconds in the CLOCK_MONOTONIC time reference.

The current implementation does not provide any timestamp.

n_measurements Number of measurements stored in the measurements table.

measurements Ranging measurements information.

diagnostic Debug informations

1.2.3.11 ranging_result_free

void **ranging_result_free**(struct *ranging_results* *ranging_results)

Free diagnostics data attached to ranging results.

Parameters

- **ranging_results** (*struct ranging_results **) – Pointer to Fira ranging results.

1.2.3.11.1 NOTE

ranging results structure itself will no be freed.

1.2.3.11.2 Return

Nothing.

1.2.3.12 fira_helper_notification_cb_t

void **fira_helper_notification_cb_t**(const struct *ranging_results* *results, void *user_data)

Notification callback type.

Parameters

- **results** (*const struct ranging_results **) – Fira ranging results.
- **user_data** (*void **) – User data pointer given to fira_helper_open.

1.2.3.12.1 Return

Nothing.

1.2.3.13 fira_helper_open

int **fira_helper_open**(struct fira_context **context*, struct uwbmac_context **uwbmac_context*, *fira_helper_notification_cb_t notification_cb*, const char **scheduler*, int *region_id*, void **user_data*)

Initialize the internal resources of the helper.

Parameters

- **context** (*struct fira_context **) – Fira context to initialize.
- **uwbmac_context** (*struct uwbmac_context **) – UWB MAC context.
- **notification_cb** (fira_helper_notification_cb_t) – Callback to call when a notification is available.
- **scheduler** (*const char **) – In which scheduler the region will be
- **region_id** (*int*) – Which id the region will have in the scheduler
- **user_data** (*void **) – User data pointer to give back in callback.

1.2.3.13.1 NOTE

This function must be called first. fira_helper_close must be called at the end of the application to ensure resources are freed. The channel will be managed by the helper, this means you should neither use uwbmac_channel_create nor uwbmac_channel_release.

1.2.3.13.2 Return

0 or error.

1.2.3.14 fira_helper_close

void **fira_helper_close**(struct fira_context **context*)

Free all internal resources of the helper.

Parameters

- **context** (*struct fira_context **) – Fira context to free.

1.2.3.15 fira_helper_set_scheduler

int **fira_helper_set_scheduler**(struct fira_context **context*)
Set the scheduler and the region of fira.

Parameters

- **context** (*struct fira_context **) – Fira context.

1.2.3.15.1 NOTE

This function must be called while the UWB MAC is stopped.

1.2.3.15.2 Return

0 or error.

1.2.3.16 fira_helper_get_capabilities

int **fira_helper_get_capabilities**(struct fira_context **context*, struct fira_capabilities **capabilites*)
Get the FiRa region capabilities.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **capabilites** (*struct fira_capabilities **) – Fira capabilites.

1.2.3.16.1 Return

0 or error.

1.2.3.17 fira_helper_init_session

int **fira_helper_init_session**(struct fira_context **context*, uint32_t *session_id*)
Initialize a fira session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.

1.2.3.17.1 Description

This function must be called first to create and initialize the fira session.

1.2.3.17.2 Return

0 or error.

1.2.3.18 fira_helper_start_session

int **fira_helper_start_session**(struct fira_context **context*, uint32_t *session_id*)
Start a fira session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.

1.2.3.18.1 Description

This function must be called after fira session was initialized.

1.2.3.18.2 Return

0 or error.

1.2.3.19 fira_helper_stop_session

int **fira_helper_stop_session**(struct fira_context **context*, uint32_t *session_id*)
Stop a fira session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.

1.2.3.19.1 Description

This function stop the session ranging.

1.2.3.19.2 Return

0 or error.

1.2.3.20 fira_helper_deinit_session

int **fira_helper_deinit_session**(struct fira_context **context*, uint32_t *session_id*)

Deinitialize a fira session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.

1.2.3.20.1 Description

This function is called to free all memory allocated by the session. This function must be called when the session is stopped.

1.2.3.20.2 Return

0 or error.

1.2.3.21 fira_helper_set_session_parameters

int **fira_helper_set_session_parameters**(struct fira_context **context*, uint32_t *session_id*, const struct *session_parameters* **session_params*)

Set session parameters.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **session_params** (*const struct session_parameters **) – Session parameters.

1.2.3.21.1 Return

0 or error.

1.2.3.22 fira_helper_get_session_parameters

int **fira_helper_get_session_parameters**(struct fira_context **context*, uint32_t *session_id*, struct *session_parameters* **session_params*)

Get session parameters.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **session_params** (*struct session_parameters **) – Session parameters.

1.2.3.22.1 Return

0 or error.

1.2.3.23 fira_helper_session_get_count

int **fira_helper_session_get_count**(struct fira_context **context*, int **count*)

Get sessions count.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **count** (*int **) – Session count.

1.2.3.23.1 Return

0 or error.

1.2.3.24 fira_helper_session_get_state

int **fira_helper_session_get_state**(struct fira_context **context*, uint32_t *session_id*, int **state*)

Get session state.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session ID.
- **state** (*int **) – Session state.

1.2.3.24.1 Return

0 or error.

1.2.3.25 fira_helper_set_controlees

int **fira_helper_set_controlees**(struct fira_context **context*, uint32_t *session_id*, const struct *controlees_parameters *controlees*)

Set controlees to a specific session. This API can be used only when session is not active.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **controlees** (*const struct controlees_parameters **) – List of controlees to add.

1.2.3.25.1 Return

0 or error.

1.2.3.26 fira_helper_add_controlees

int **fira_helper_add_controlees**(struct fira_context **context*, uint32_t *session_id*, const struct *controlees_parameters* **controlees*)

Add controlees to a specific session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **controlees** (*const struct controlees_parameters **) – List of controlees to add.

1.2.3.26.1 Return

0 or error.

1.2.3.27 fira_helper_delete_controlees

int **fira_helper_delete_controlees**(struct fira_context **context*, uint32_t *session_id*, const struct *controlees_parameters* **controlees*)

Delete controlees from a specific session.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **controlees** (*const struct controlees_parameters **) – List of controlees to delete.

1.2.3.27.1 Return

0 or error.

1.2.3.28 fira_helper_get_controlees

int **fira_helper_get_controlees**(struct fira_context **context*, uint32_t *session_id*, struct *controlees_parameters* **controlees*)

Get controlees list.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **controlees** (*struct controlees_parameters **) – List of controlees to write.

1.2.3.28.1 Return

0 or error.

1.2.3.29 fira_helper_send_data

int **fira_helper_send_data**(struct fira_context **context*, uint32_t *session_id*, const struct *data_parameters* **data_params*)

Send custom parameters.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **session_id** (*uint32_t*) – Session identifier.
- **data_params** (*const struct data_parameters **) – Custom data parameters.

1.2.3.29.1 Return

0 or error.

1.2.3.30 struct session_parameters_builder

struct **session_parameters_builder**

Tool to send partial session configuration.

1.2.3.30.1 Definition

```
struct session_parameters_builder {
    struct uwbmac_msg msg;
    struct uwbmac_msg request;
    struct uwbmac_msg session_params;
}
```

1.2.3.30.2 Members

msg msg constructed by the builder.

request first nested part of msg constructed.

session_params second nested part of msg constructed.

1.2.3.31 session_parameters_builder_init

int **session_parameters_builder_init**(struct fira_context **context*, struct *session_parameters_builder* **builder*, uint32_t *session_id*)

Initialize the UWB MAC message stored in the builder.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **builder** (*struct session_parameters_builder **) – Builder to initialize.
- **session_id** (*uint32_t*) – Session identifier.

1.2.3.31.1 Return

0 or error.

1.2.3.32 session_parameters_builder_finish

struct *uwbmac_msg* ***session_parameters_builder_finish**(struct *session_parameters_builder* **builder*)

Generate the UWB MAC message to send.

Parameters

- **builder** (*struct session_parameters_builder **) – Builder which will build the message.

1.2.3.32.1 Return

Built message, or NULL on error.

1.2.3.33 fira_helper_set_partial_session_parameters

int **fira_helper_set_partial_session_parameters**(struct fira_context **context*, struct *session_parameters_builder* **builder*)

Set session parameters defined with builder.

Parameters

- **context** (*struct fira_context **) – Fira context.
- **builder** (*struct session_parameters_builder **) – Session parameters.

1.2.3.33.1 Return

0 or error.

1.2.3.34 enum fira_device_type

enum **fira_device_type**
Type of a device.

1.2.3.34.1 Definition

```
enum fira_device_type {
    FIRA_DEVICE_TYPE_CONTROLEE,
    FIRA_DEVICE_TYPE_CONTROLLER
};
```

1.2.3.34.2 Constants

FIRA_DEVICE_TYPE_CONTROLEE The device is a controlee.

FIRA_DEVICE_TYPE_CONTROLLER The device is a controller.

1.2.3.35 enum fira_device_role

enum **fira_device_role**
[NOT IMPLEMENTED] Role played by a device.

1.2.3.35.1 Definition

```
enum fira_device_role {
    FIRA_DEVICE_ROLE_RESPONDER,
    FIRA_DEVICE_ROLE_INITIATOR
};
```

1.2.3.35.2 Constants

FIRA_DEVICE_ROLE_RESPONDER The device acts as a responder.

FIRA_DEVICE_ROLE_INITIATOR The device acts as an initiator.

1.2.3.35.3 Description

Current implementation does not support decorrelation between the device's role and the device's type. The controller is always the initiator and the controlee is always the responder.

This enum is not used in the current implementation.

1.2.3.36 enum fira_ranging_round_usage

enum **fira_ranging_round_usage**
Ranging mode.

1.2.3.36.1 Definition

```
enum fira_ranging_round_usage {
    FIRA_RANGING_ROUND_USAGE_OWR,
    FIRA_RANGING_ROUND_USAGE_SSTWR,
    FIRA_RANGING_ROUND_USAGE_DSTWR
};
```

1.2.3.36.2 Constants

FIRA_RANGING_ROUND_USAGE_OWR One Way Ranging mode (unused, not in FiRa 1.1).

FIRA_RANGING_ROUND_USAGE_SSTWR Single-Sided Two Way Ranging mode.

FIRA_RANGING_ROUND_USAGE_DSTWR Dual-Sided Two Way Ranging mode.

1.2.3.37 enum fira_multi_node_mode

enum **fira_multi_node_mode**
Multi-node mode.

1.2.3.37.1 Definition

```
enum fira_multi_node_mode {
    FIRA_MULTI_NODE_MODE_UNICAST,
    FIRA_MULTI_NODE_MODE_ONE_TO_MANY,
    FIRA_MULTI_NODE_MODE_MANY_TO_MANY
};
```

1.2.3.37.2 Constants

FIRA_MULTI_NODE_MODE_UNICAST Ranging between one initiator and one responder.

FIRA_MULTI_NODE_MODE_ONE_TO_MANY Ranging between one initiator and multiple responders.

FIRA_MULTI_NODE_MODE_MANY_TO_MANY Ranging between multiple initiators and multiple responders.

1.2.3.38 enum fira_measurement_report

enum **fira_measurement_report**
[NOT IMPLEMENTED] Transmission of a Ranging Measurement Report Message (MRM) option.

1.2.3.38.1 Definition

```
enum fira_measurement_report {
    FIRA_MEASUREMENT_REPORT_AT_RESPONDER,
    FIRA_MEASUREMENT_REPORT_AT_INITIATOR
};
```

1.2.3.38.2 Constants

FIRA_MEASUREMENT_REPORT_AT_RESPONDER The initiator emits a MRM.

FIRA_MEASUREMENT_REPORT_AT_INITIATOR The responder emits a MRM.

1.2.3.38.3 Description

In the current implementation measurement report is always available at responder.

This enum is not used in the current implementation.

1.2.3.39 enum fira_embedded_mode

enum **fira_embedded_mode**
[NOT IMPLEMENTED] Message embedding behaviour.

1.2.3.39.1 Definition

```
enum fira_embedded_mode {
    FIRA_EMBEDDED_MODE_DEFERRED,
    FIRA_EMBEDDED_MODE_NON_DEFERRED
};
```

1.2.3.39.2 Constants

FIRA_EMBEDDED_MODE_DEFERRED Ranging messages do not embed control messages. Additional messages are required.

FIRA_EMBEDDED_MODE_NON_DEFERRED Ranging messages embed control messages.

1.2.3.39.3 Description

The current implementation only supports deferred mode.

This enum is not used in the current implementation.

1.2.3.40 enum fira_rframe_config

enum **fira_rframe_config**
Rframe configuration used to transmit/receive ranging messages.

1.2.3.40.1 Definition

```
enum fira_rframe_config {
    FIRA_RFRAME_CONFIG_SP0,
    FIRA_RFRAME_CONFIG_SP1,
    FIRA_RFRAME_CONFIG_SP2,
    FIRA_RFRAME_CONFIG_SP3
};
```

1.2.3.40.2 Constants

FIRA_RFRAME_CONFIG_SP0 Use SP0 mode.

FIRA_RFRAME_CONFIG_SP1 Use SP1 mode.

FIRA_RFRAME_CONFIG_SP2 RFU

FIRA_RFRAME_CONFIG_SP3 Use SP3 mode.

1.2.3.41 enum fira_prf_mode

enum **fira_prf_mode**
Pulse Repetition Frequency mode

1.2.3.41.1 Definition

```
enum fira_prf_mode {
    FIRA_PRF_MODE_BPRF,
    FIRA_PRF_MODE_HPRF,
    FIRA_PRF_MODE_HPRF_HIGH_RATE
};
```

1.2.3.41.2 Constants

FIRA_PRF_MODE_BPRF Base Pulse Repetition Frequency.

FIRA_PRF_MODE_HPRF Higher Pulse Repetition Frequency.

FIRA_PRF_MODE_HPRF_HIGH_RATE Higher Pulse Repetition Frequency allows high data rate (27.2 Mbps and 31.2 Mbps).

1.2.3.41.3 Description

This enum is not used in the current implementation.

1.2.3.42 enum fira_preambule_duration

enum **fira_preambule_duration**
Duration of preamble in symbols.

1.2.3.42.1 Definition

```
enum fira_preambule_duration {
    FIRA_PREAMBULE_DURATION_32,
    FIRA_PREAMBULE_DURATION_64
};
```

1.2.3.42.2 Constants

FIRA_PREAMBULE_DURATION_32 32 symbols duration.

FIRA_PREAMBULE_DURATION_64 64 symbols duration.

1.2.3.43 enum fira_sfd_id

enum **fira_sfd_id**
Start-of-frame delimiter.

1.2.3.43.1 Definition

```
enum fira_sfd_id {
    FIRA_SFD_ID_0,
    FIRA_SFD_ID_1,
    FIRA_SFD_ID_2,
    FIRA_SFD_ID_3,
    FIRA_SFD_ID_4
};
```

1.2.3.43.2 Constants

FIRA_SFD_ID_0 Delimiter is [0 +1 0 –1 +1 0 0 –1]

FIRA_SFD_ID_1 Delimiter is [–1 –1 +1 –1]

FIRA_SFD_ID_2 Delimiter is [–1 –1 –1 +1 –1 –1 +1 –1]

FIRA_SFD_ID_3 Delimiter is [–1 –1 –1 –1 –1 +1 +1 –1 –1 +1 –1 +1 –1 –1 +1 –1]

FIRA_SFD_ID_4 Delimiter is [–1 –1 –1 –1 –1 –1 –1 +1 –1 –1 +1 –1 –1 +1 –1 +1 –1 +1 –1 –1 –1 +1 +1 –1 –1 –1 +1 –1 +1 +1 –1 –1]

1.2.3.44 enum fira_sts_segments

enum **fira_sts_segments**
Number of STS segments.

1.2.3.44.1 Definition

```
enum fira_sts_segments {
    FIRA_STS_SEGMENTS_0,
    FIRA_STS_SEGMENTS_1,
    FIRA_STS_SEGMENTS_2,
    FIRA_STS_SEGMENTS_3,
    FIRA_STS_SEGMENTS_4
};
```

1.2.3.44.2 Constants

FIRA_STS_SEGMENTS_0 No STS Segment (Rframe config SP0).

FIRA_STS_SEGMENTS_1 1 STS Segment.

FIRA_STS_SEGMENTS_2 2 STS Segments.

FIRA_STS_SEGMENTS_3 3 STS Segments.

FIRA_STS_SEGMENTS_4 4 STS Segments.

1.2.3.45 enum fira_psdu_data_rate

enum **fira_psdu_data_rate**
Data rate used to exchange PSDUs.

1.2.3.45.1 Definition

```
enum fira_psdu_data_rate {
    FIRA_PSDU_DATA_RATE_6M81,
    FIRA_PSDU_DATA_RATE_7M80,
    FIRA_PSDU_DATA_RATE_27M2,
    FIRA_PSDU_DATA_RATE_31M2
};
```

1.2.3.45.2 Constants

FIRA_PSDU_DATA_RATE_6M81 6.8Mb/s rate.

FIRA_PSDU_DATA_RATE_7M80 7.8Mb/s rate.

FIRA_PSDU_DATA_RATE_27M2 27.2Mb/s rate.

FIRA_PSDU_DATA_RATE_31M2 31.2Mb/s rate.

1.2.3.46 enum fira_phr_data_rate

enum **fira_phr_data_rate**
Data rate used to exchange PHR.

1.2.3.46.1 Definition

```
enum fira_phr_data_rate {
    FIRA_PHR_DATA_RATE_850K,
    FIRA_PHR_DATA_RATE_6M81
};
```

1.2.3.46.2 Constants

FIRA_PHR_DATA_RATE_850K 850kb/s rate.

FIRA_PHR_DATA_RATE_6M81 6.8Mb/s rate.

1.2.3.46.3 Description

This enum is not used in the current implementation.

1.2.3.47 enum fira_mac_fcs_type

enum **fira_mac_fcs_type**
Length of Frame Check Sequence.

1.2.3.47.1 Definition

```
enum fira_mac_fcs_type {
    FIRA_MAC_FCS_TYPE_CRC_16,
    FIRA_MAC_FCS_TYPE_CRC_32
};
```

1.2.3.47.2 Constants

FIRA_MAC_FCS_TYPE_CRC_16 2 bytes sequence.

FIRA_MAC_FCS_TYPE_CRC_32 4 bytes sequence.

1.2.3.48 enum fira_rssi_report_type

enum **fira_rssi_report_type**
Mode used to sum up individual frames RSSI in report.

1.2.3.48.1 Definition

```
enum fira_rssi_report_type {
    FIRA_RSSI_REPORT_NONE,
    FIRA_RSSI_REPORT_MINIMUM,
    FIRA_RSSI_REPORT_AVERAGE
};
```

1.2.3.48.2 Constants

FIRA_RSSI_REPORT_NONE No RSSI value in report.

FIRA_RSSI_REPORT_MINIMUM Report minimum RSSI

FIRA_RSSI_REPORT_AVERAGE Report average RSSI

1.2.3.49 enum fira_sts_config

enum **fira_sts_config**
Scrambled Timestamp Sequence configuration.

1.2.3.49.1 Definition

```
enum fira_sts_config {
    FIRA_STS_CONFIG_STATIC,
    FIRA_STS_CONFIG_DYNAMIC,
    FIRA_STS_CONFIG_DYNAMIC_INDIVIDUAL_KEY,
    FIRA_STS_CONFIG_PROVISIONED,
    FIRA_STS_CONFIG_PROVISIONED_INDIVIDUAL_KEY
};
```

1.2.3.49.2 Constants

FIRA_STS_CONFIG_STATIC Use a static STS configuration.

FIRA_STS_CONFIG_DYNAMIC Use a dynamic STS configuration.

FIRA_STS_CONFIG_DYNAMIC_INDIVIDUAL_KEY Use a dynamic STS configuration with individual controlee key.

FIRA_STS_CONFIG_PROVISIONED Use a provisioned STS configuration.

FIRA_STS_CONFIG_PROVISIONED_INDIVIDUAL_KEY Use a provisioned STS configuration with individual controlee key.

1.2.3.50 enum fira_ranging_status

enum **fira_ranging_status**
The ranging status.

1.2.3.50.1 Definition

```
enum fira_ranging_status {
    FIRA_STATUS_RANGING_INTERNAL_ERROR,
    FIRA_STATUS_RANGING_SUCCESS,
    FIRA_STATUS_RANGING_TX_FAILED,
    FIRA_STATUS_RANGING_RX_TIMEOUT,
    FIRA_STATUS_RANGING_RX_PHY_DEC_FAILED,
    FIRA_STATUS_RANGING_RX_PHY_TOA_FAILED,
    FIRA_STATUS_RANGING_RX_PHY_STS_FAILED,
    FIRA_STATUS_RANGING_RX_MAC_DEC_FAILED,
    FIRA_STATUS_RANGING_RX_MAC_IE_DEC_FAILED,
    FIRA_STATUS_RANGING_RX_MAC_IE_MISSING
};
```

1.2.3.50.2 Constants

FIRA_STATUS_RANGING_INTERNAL_ERROR Implementation specific error.

FIRA_STATUS_RANGING_SUCCESS Ranging info are valid.

FIRA_STATUS_RANGING_TX_FAILED Failed to transmit UWB packet.

FIRA_STATUS_RANGING_RX_TIMEOUT No UWB packet detected by the receiver.

FIRA_STATUS_RANGING_RX_PHY_DEC_FAILED UWB packet channel decoding error.

FIRA_STATUS_RANGING_RX_PHY_TOA_FAILED Failed to detect time of arrival of the UWB packet from CIR samples.

FIRA_STATUS_RANGING_RX_PHY_STS_FAILED UWB packet STS segment mismatch.

FIRA_STATUS_RANGING_RX_MAC_DEC_FAILED MAC CRC or syntax error.

FIRA_STATUS_RANGING_RX_MAC_IE_DEC_FAILED IE syntax error.

FIRA_STATUS_RANGING_RX_MAC_IE_MISSING Expected IE missing in the packet.

1.2.3.51 enum fira_session_state

enum **fira_session_state**
Session state.

1.2.3.51.1 Definition

```
enum fira_session_state {
    FIRA_SESSION_STATE_INIT,
    FIRA_SESSION_STATE_DEINIT,
    FIRA_SESSION_STATE_ACTIVE,
    FIRA_SESSION_STATE_IDLE
};
```

1.2.3.51.2 Constants

FIRA_SESSION_STATE_INIT Initial state, session is not ready yet.

FIRA_SESSION_STATE_DEINIT Session does not exist.

FIRA_SESSION_STATE_ACTIVE Session is currently active.

FIRA_SESSION_STATE_IDLE Session is ready to start, but not currently active.

1.2.3.52 enum fira_measurement_type

enum **fira_measurement_type**
The different type of available measurements.

1.2.3.52.1 Definition

```
enum fira_measurement_type {
    FIRA_MEASUREMENT_TYPE_RANGE,
    FIRA_MEASUREMENT_TYPE_AOA,
    FIRA_MEASUREMENT_TYPE_AOA_AZIMUTH,
    FIRA_MEASUREMENT_TYPE_AOA_ELEVATION,
    FIRA_MEASUREMENT_TYPE_AOA_AZIMUTH_ELEVATION,
    __FIRA_MEASUREMENT_TYPE_AFTER_LAST
};
```

1.2.3.52.2 Constants

FIRA_MEASUREMENT_TYPE_RANGE Measure only range.

FIRA_MEASUREMENT_TYPE_AOA Measure range + unspecified AoA.

FIRA_MEASUREMENT_TYPE_AOA_AZIMUTH Measure range + azimuth.

FIRA_MEASUREMENT_TYPE_AOA_ELEVATION Measure range + elevation.

FIRA_MEASUREMENT_TYPE_AOA_AZIMUTH_ELEVATION Measure range+azimuth+elevation.

__FIRA_MEASUREMENT_TYPE_AFTER_LAST Internal use.

1.2.3.53 enum fira_ranging_diagnostics_frame_report_flags

enum **fira_ranging_diagnostics_frame_report_flags**
Activation flags for different frame diagnostics information.

1.2.3.53.1 Definition

```
enum fira_ranging_diagnostics_frame_report_flags {
    FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_NONE,
    FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_RSSIS,
    FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_AOAS,
    FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_CIRS,
    __FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_AFTER_LAST
};
```

1.2.3.53.2 Constants

FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_NONE No specific frame diagnostic report requested.

FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_RSSIS Report RSSI in frame diagnostics.

FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_AOAS Report AOA in frame diagnostics.

FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_CIRS Report CIR in frame diagnostics.

__FIRA_RANGING_DIAGNOSTICS_FRAME_REPORT_AFTER_LAST Internal use.

1.2.3.54 enum fira_sts_length

enum **fira_sts_length**
Number of symbols in a STS segment.

1.2.3.54.1 Definition

```
enum fira_sts_length {
    FIRA_STS_LENGTH_32,
    FIRA_STS_LENGTH_64,
    FIRA_STS_LENGTH_128
};
```

1.2.3.54.2 Constants

FIRA_STS_LENGTH_32 The STS length is 32 symbols.

FIRA_STS_LENGTH_64 The STS length is 64 symbols.

FIRA_STS_LENGTH_128 The STS length is 128 symbols.

2 UCI Documentation

2.1 UCI Public APIs

2.1.1 Introduction

UCI is a FiRa defined communication protocol between an Host and the UWB Sub-system running on a co-processor.

It also allows for vendor implementations outside of the FiRa defined protocol.

Our UCI server for the UWB-subsystem is implemented as several layers:

- Physical transport and framing: it handles the physical interface, synchronisation and packet length.
- Core: it implements packet parsing, packets generation, error processing, segmentation, reassembly and routing.
- Backends: they implement the high level logic. They are split in several modules, for example one for each group identifier.

For controlling different sessions type (Fira), since the same set of commands are used (as per Fira specification), a backend called uci_backend_manager was designed to dispatch session commands based on session type.

2.1.2 UCI Core

2.1.2.1 Overview

The UCI core implements packets parsing, packets generation, error processing, segmentation, reassembly and routing. It can be used to implement a UCI server (which receives commands) or UCI client (which receives responses and notifications).

The packets are sent and received using a transport channel which is responsible for framing, and communication with the transport driver.

Received commands are given to message handlers which decode the payload.

2.1.2.2 Memory handling

To allow zero copy, buffers used to store packet data are allocated dynamically. This can be backed with simple heap allocation, or using a limited pool of memory blocks.

To limit memory fragmentation, packet buffers can be split in smaller blocks. This is decided by the memory allocator. In this case, a packet can be composed of several blocks which are chained together. The first block contains the information of the full packet length.

2.1.2.3 UCI_MAX_PAYLOAD_SIZE

UCI_MAX_PAYLOAD_SIZE()
Maximum size of a UCI payload.

2.1.2.4 UCI_PACKET_HEADER_SIZE

UCI_PACKET_HEADER_SIZE()
Size of a UCI packet header.

2.1.2.4.1 Note

this also the minimum packet size.

2.1.2.5 UCI_MAX_PACKET_SIZE

UCI_MAX_PACKET_SIZE()
Maximum size of a UCI packet.

2.1.2.6 UCI_STATUS_PACKET_SIZE

UCI_STATUS_PACKET_SIZE()
Size of a UCI status packet.

2.1.2.7 UCI_MT_GID_OID

UCI_MT_GID_OID(*mt*, *gid*, *oid*)
Macro to build *uci_message_handler.mt_gid_oid*.

Parameters

- **mt** – Message type.
- **gid** – Group identifier.
- **oid** – Opcode identifier.

2.1.2.7.1 Return

Integer suitable for use in *uci_message_handler.mt_gid_oid*.

2.1.2.8 UCI_GID_OID

UCI_GID_OID(*gid*, *oid*)
Macro to build `gid_oid` parameter.

Parameters

- **gid** – Group identifier.
- **oid** – Opcode identifier.

2.1.2.8.1 Return

Integer suitable for use as `gid_oid` parameter.

2.1.2.9 UCI_MT

UCI_MT(*mt_gid_oid*)
Macro to extract message type from a `mt_gid_oid` parameter.

Parameters

- **mt_gid_oid** – Union of message type, group identifier and opcode identifier.

2.1.2.9.1 Return

Message type.

2.1.2.10 UCI_GID

UCI_GID(*mt_gid_oid*)
Macro to extract group identifier from a `mt_gid_oid` parameter.

Parameters

- **mt_gid_oid** – Union of message type, group identifier and opcode identifier.

2.1.2.10.1 Return

Group identifier.

2.1.2.11 UCI_OID

UCI_OID(*mt_gid_oid*)
Macro to extract opcode identifier from a `mt_gid_oid` parameter.

Parameters

- **mt_gid_oid** – Union of message type, group identifier and opcode identifier.

2.1.2.11.1 Return

Opcode identifier.

2.1.2.12 UCI_CONTAINER_OF

UCI_CONTAINER_OF(*ptr*, *type*, *member*)
Get a struct pointer from one of its members pointer.

Parameters

- **ptr** – Pointer to the member.
- **type** – Type of the parent struct.
- **member** – Name to the member in the parent struct.

2.1.2.12.1 Return

Pointer to the parent struct.

2.1.2.13 enum uci_blk_flags

enum **uci_blk_flags**
Packet buffer block flag.

2.1.2.13.1 Definition

```
enum uci_blk_flags {
    UCI_BLK_FLAGS_STATIC,
    UCI_BLK_FLAGS_DESTRUCTIBLE,
    UCI_BLK_FLAGS_HEADER_RESERVED
};
```

2.1.2.13.2 Constants

UCI_BLK_FLAGS_STATIC Do not release this block.

UCI_BLK_FLAGS_DESTRUCTIBLE Do not release this block with the free method from the given uci_allocator, but use a custom destructor. In this case the block should be cast to *struct uci_blk_destructible* to access the destructor.

UCI_BLK_FLAGS_HEADER_RESERVED Set if there are 4 reserved bytes for header before payload at the beginning of the data buffer.

2.1.2.14 struct uci_blk

struct **uci_blk**
Packet buffer block.

2.1.2.14.1 Definition

```
struct uci_blk {
    struct uci_blk *next;
    uint8_t *data;
    uint16_t len;
    uint16_t total_len;
    uint16_t size;
    uint8_t flags;
}
```

2.1.2.14.2 Members

next Pointer to next block, or NULL if last one.

data Pointer to data in this block. This can be changed during thelifetime of the buffer and should not be used for memory release.

len Length of data in this block.

total_len Length of the full packet or payload. This is set in thefirst block only, and zero for other blocks.

size Allocation size for data in this block.

flags Block flags, see *enum uci_blk_flags*.

2.1.2.15 struct uci_blk_destructible

struct **uci_blk_destructible**
Packet buffer block with a destructor.

2.1.2.15.1 Definition

```
struct uci_blk_destructible {
    struct uci_blk blk;
    void (*destructor)(void *arg, struct uci_blk_destructible *blk);
    void *destructor_arg;
}
```

2.1.2.15.2 Members

blk Block common part.

destructor Function to call instead of *uci_allocator_ops.free*.

destructor_arg First argument of *uci_blk_destructible.destructor*.

2.1.2.15.3 Description

This is not allocated using the UCI allocator. It can be used by an external code to embed data inside a packet and avoid copy.

2.1.2.16 struct uci_allocator_ops

struct **uci_allocator_ops**
UCI allocator operations.

2.1.2.16.1 Definition

```
struct uci_allocator_ops {
    struct uci_blk *(*alloc)(struct uci_allocator *allocator, size_t size_hint);
    void (*free)(struct uci_allocator *allocator, struct uci_blk *blk);
}
```

2.1.2.16.2 Members

alloc Allocate one block. The returned block can be smaller orlarger than the size_hint. Return NULL if memory is exhausted.

free Release one block.

2.1.2.17 struct uci_allocator

struct **uci_allocator**
UCI allocator instance.

2.1.2.17.1 Definition

```
struct uci_allocator {
    const struct uci_allocator_ops *ops;
}
```

2.1.2.17.2 Members

ops Allocator operations.

2.1.2.18 uci_init

uwbmac_error **uci_init**(struct *uci* *uci, struct *uci_allocator* *allocator, bool *is_client*)
Initialize UCI core.

Parameters

- **uci** (*struct uci **) – UCI context.
- **allocator** (*struct uci_allocator **) – Allocator for buffers.
- **is_client** (*bool*) – True if this is a client.

2.1.2.18.1 Return

UWBMAC_SUCCESS on success, or an error code.

2.1.2.19 uci_uninit

void **uci_uninit**(struct *uci* *uci)
Uninitialize UCI core.

Parameters

- **uci** (*struct uci **) – UCI context.

2.1.2.19.1 Description

Note that uninit does not cleanup the handlers linked list, which should no longer be used outside of the library.

2.1.2.20 uci_blk_alloc

struct *uci_blk* ***uci_blk_alloc**(struct *uci* *uci, size_t *size_hint*)

Allocate buffer block.

Parameters

- **uci** (*struct uci **) – UCI context.
- **size_hint** (*size_t*) – Hint on the required size.

2.1.2.20.1 Description

Allocate a buffer, usually used to send a message. The size hint can give an indication of the required size. The returned buffer can be smaller or larger due to allocation constrains. In all cases, only one block is returned.

The UCI message API should be preferred to build a message to send.

2.1.2.20.2 Return

The allocated buffer or NULL if no memory is available.

2.1.2.21 uci_blk_free_all

void **uci_blk_free_all**(struct *uci* *uci, struct *uci_blk* *blks)

Release a chain of buffer blocks.

Parameters

- **uci** (*struct uci **) – UCI context.
- **blks** (*struct uci_blk **) – Chain of buffer blocks, or NULL.

2.1.2.21.1 Description

Release a chain of buffer blocks which were allocated from *uci_blk_alloc()* or received by a message handler.

Does nothing if called with a NULL pointer.

2.1.2.22 uci_message_handler_function

uwbmac_error **uci_message_handler_function**(struct *uci* *uci, uint16_t *mt_gid_oid*, const struct *uci_blk* *payload, void *user_data)

Handle a received message.

Parameters

- **uci** (*struct uci **) – UCI context.
- **mt_gid_oid** (*uint16_t*) – Union of message type, group identifier and opcode identifier.
- **payload** (*const struct uci_blk **) – Reassembled message payload, can span several buffer blocks.
- **user_data** (*void **) – User data given at registration.

2.1.2.22.1 Description

If the handler returns 0, it musts take care of the response if needed, during its execution, or later.

If the message is a command and the return value signals an error, a response with status FAILED will be sent.

2.1.2.22.2 Return

UWBMAC_SUCCESS on success, or an uwbmac error code. The special value UWBMAC_UCI_MSG_HANDLED forces the packet receiver to stop calling after this handler.

2.1.2.23 struct uci_message_handler

struct **uci_message_handler**
Definition of a message handler.

2.1.2.23.1 Definition

```
struct uci_message_handler {
    uint16_t mt_gid_oid;
    uci_message_handler_function handler;
}
```

2.1.2.23.2 Members

mt_gid_oid Union of message type, group identifier and opcodeidentifier. Use the *UCI_MT_GID_OID()* macro.

handler Function called to handle a received message.

2.1.2.24 struct uci_message_handlers

struct **uci_message_handlers**
Definition of several message handlers belonging to the same module.

2.1.2.24.1 Definition

```
struct uci_message_handlers {
    struct uci_message_handlers *next;
    const struct uci_message_handler *handlers;
    size_t n_handlers;
    void *user_data;
}
```

2.1.2.24.2 Members

next Pointer to next message handlers definition, or NULL if thisis the last one. Filled at registration.

handlers Pointer to array of message handler definitions. NOTE: **This must be sorted** by message type, then group identifier, then opcode identifier.

n_handlers Number of message handlers.

user_data User data given to message handlers when a message isreceived.

2.1.2.25 uci_message_handlers_register

void **uci_message_handlers_register**(struct *uci* *uci, struct *uci_message_handlers* *handlers)

Register message handlers.

Parameters

- **uci** (*struct uci **) – UCI context.
- **handlers** (*struct uci_message_handlers **) – Information on message handlers.

2.1.2.25.1 Description

Be sure that the uci_message_handlers provided only contains handlers for only one GID. The method will not accept it otherwise and will register nothing.

2.1.2.25.2 NOTE

Handlers memory is managed by the client, but its representation is managed as a linked list by the library. Its memory is assumed to be usable `struct until` *uci_uninit()*.

2.1.2.26 uci_packet_recv_alloc

struct *uci_blk* ***uci_packet_recv_alloc**(struct *uci* *uci, uint16_t *size_hint*)

Allocate buffer for reception.

Parameters

- **uci** (*struct uci **) – UCI context.
- **size_hint** (*uint16_t*) – Hint on the required size.

2.1.2.26.1 Description

This function should be called by the transport channel to build packets for received data. The size hint can give an indication of the required size in case the packet size is known. The returned buffer can be smaller or larger due to allocation constrains. In all cases, only one block is returned.

2.1.2.26.2 Return

The allocated buffer or NULL if no memory is available.

2.1.2.27 uci_packet_recv_free_all

void **uci_packet_recv_free_all**(struct *uci* *uci, struct *uci_blk* *blks)
Release unused reception buffer.

Parameters

- **uci** (*struct uci **) – UCI context.
- **blks** (*struct uci_blk **) – Chain of buffer blocks to release, or NULL.

2.1.2.27.1 Description

This function should be called by the transport channel with unused buffers when a packet reception was aborted due to an error, or when the channel is shut down.

Does nothing if called with a NULL pointer.

2.1.2.28 uci_packet_recv

void **uci_packet_recv**(struct *uci* *uci, struct *uci_blk* *packet)
Hand off valid received packet.

Parameters

- **uci** (*struct uci **) – UCI context.
- **packet** (*struct uci_blk **) – Received packet (can be composed of several blocks).

2.1.2.28.1 Description

This function should be called by the transport channel when a packet has been successfully received. The UCI core takes ownership of the associated memory.

2.1.2.29 struct uci_transport_ops

struct **uci_transport_ops**
UCI transport channel callbacks.

2.1.2.29.1 Definition

```
struct uci_transport_ops {
    void (*attach)(struct uci_transport *uci_tr, struct uci *uci);
    void (*detach)(struct uci_transport *uci_tr);
    void (*packet_send_ready)(struct uci_transport *uci_tr);
}
```

2.1.2.29.2 Members

attach Callback invoked when the transport channel is attached.

detach Callback invoked when the transport channel is detached. Alllent buffers should be released (using `uci_packet_recv_free()` and *uci_packet_send_done()*).

packet_send_ready Callback invoked when UCI wants to send a packetinitially or after `struct uci_packet_get_ready`() returned NULL. It allows to restart the sending data pump.

2.1.2.30 struct uci_transport

struct **uci_transport**
UCI generic transport channel.

2.1.2.30.1 Definition

```
struct uci_transport {
    const struct uci_transport_ops *ops;
}
```

2.1.2.30.2 Members

ops Transport channel callbacks.

2.1.2.31 uci_transport_attach

uwbmac_error **uci_transport_attach**(struct *uci* *uci, struct *uci_transport* *uci_tr)
Attach a transport channel.

Parameters

- **uci** (*struct uci **) – UCI context.
- **uci_tr** (*struct uci_transport **) – Transport channel to attach.

2.1.2.31.1 Return

UWBMAC_SUCCESS on success, or UWBMAC_EBUSY if a transport is already attached.

2.1.2.32 uci_transport_detach

uwbmac_error **uci_transport_detach**(struct *uci* *uci)
Detach a transport channel.

Parameters

- **uci** (*struct uci **) – UCI context.

2.1.2.32.1 Return

UWBMAC_SUCCESS on success, or UWBMAC_EINVAL if no transport is attached.

2.1.2.33 uci_send_message

void **uci_send_message**(struct *uci* *uci, uint16_t *mt_gid_oid*, struct *uci_blk* *payload)
Send a message.

Parameters

- **uci** (*struct uci **) – UCI context.
- **mt_gid_oid** (*uint16_t*) – Union of message type, group identifier and opcode identifier. Use the *UCI_MT_GID_OID()* macro.
- **payload** (*struct uci_blk **) – Message payload which can be composed of several buffer blocks, or NULL if there is none. Ownership is transferred to UCI core.

2.1.2.33.1 Description

The message payload is segmented, packets are built and queued to be sent. If you use the UCI message API, this is done efficiently as room is reserved during message construction for headers.

2.1.2.34 uci_send_status

void **uci_send_status**(struct *uci* *uci, uint16_t *gid_oid*, uint8_t *status*)
Send a status response.

Parameters

- **uci** (*struct uci **) – UCI context.
- **gid_oid** (*uint16_t*) – Union of group identifier and opcode identifier. Use the *UCI_GID_OID()* macro, message type is ignored.
- **status** (*uint8_t*) – Status code.

2.1.2.35 uci_packet_send_get_ready

struct *uci_blk* ***uci_packet_send_get_ready**(struct *uci* **uci*)
Retrieve the packet ready to be sent.

Parameters

- **uci** (*struct uci **) – UCI context.

2.1.2.35.1 Description

Packet buffers are lent to the transport channel, they should be returned once transmission is done using *uci_packet_send_done()*.

If there is no packet to send, NULL is returned and UCI core will signal any new pending packet using the `uci_tr_ops.packet_send_ready` callback.

2.1.2.35.2 Return

The first block of the packet to send, or NULL if no packet to send.

2.1.2.36 uci_packet_send_done

void **uci_packet_send_done**(struct *uci* **uci*, struct *uci_blk* **packet*, int *status*)
Signal the packet has been transmitted.

Parameters

- **uci** (*struct uci **) – UCI context.
- **packet** (*struct uci_blk **) – Packet that was sent, or that failed to be sent.
- **status** (*int*) – 0 if transmission was successful, or a negative error code.

2.1.2.36.1 Description

Once a packet has been transmitted, or after a non recoverable failure, packet should be returned by the transport channel using this function (see `struct uci_packet_get_ready` and `uci_tr_ops.packet_send_ready`).

2.1.2.37 uci_packet_response_expire

void **uci_packet_response_expire**(struct *uci* **uci*)
Signal the response timer expired.

Parameters

- **uci** (*struct uci **) – UCI context.

2.1.2.37.1 Description

This is only needed for UCI client.

The transport channel should reset a timer each time a packet is sent on the transport channel. If the timer expires, it should call this function. This is used to identify problem with a missing response.

The timer expiration will be ignored if UCI core has a pending packet which was not given back using *uci_packet_send_done()*.

The timer duration is specific to the transport channel.

2.1.2.38 struct uci

struct **uci**
UCI core context.

2.1.2.38.1 Definition

```
struct uci {
    struct uci_allocator *allocator;
    struct uci_message_handlers *handlers_head;
    struct uci_blk *rx;
    struct uci_blk *rx_last;
    struct uci_blk *tx;
    struct uci_blk *tx_last;
    struct uci_transport *tr;
    struct uci_blk status_blk;
    uint8_t status_data[UCI_STATUS_PACKET_SIZE];
    bool is_client;
    uint16_t known_gid;
}
```

2.1.2.38.2 Members

allocator Allocator to use to allocate buffer blocks.

handlers_head Head of the list of handlers.

rx Used to collect segmented messages before processing.

rx_last Last received segmented message before processing.

tx Pointer to first message to send of TX queue.

tx_last Pointer to last message to send of TX queue.

tr Transport channel attached to this context or NULL.

status_blk Buffer block reserved for status signaling. This shouldnot fail to signal error status so it does not depend on dynamic allocation.

status_data Data for error buffer block.

is_client True if this is a client.

known_gid Bitfield of known GID in message handlers list.

2.1.3 UCI Backend

2.1.3.1 UCI Backend Manager

2.1.3.1.1 Manager overview

The uci_backend_manager implements command dispatching for the submodules it manages. Various sub backends exists to handle ranging commands in a modular way. Thus, the manager will parse an UCI command and select the correct sub-backend to call based on session type.

2.1.3.1.2 struct uci_backend_manager

struct **uci_backend_manager**
Definition of this backend context.

2.1.3.1.2.1 Definition

```
struct uci_backend_manager {
    struct uci_session *head;
    struct uwbmac_context *uwbmac_context;
    uint8_t channel_number;
    struct uci *uci;
    const uint8_t *core_device_info_vendor_data;
    size_t core_device_info_vendor_length;
    struct uci_session_controller *controller_head;
    bool uwb_disabled;
}
```

2.1.3.1.2.2 Members

head Linked list of sessions.

uwbmac_context UWB MAC context.

channel_number Channel number in use.

uci uci context used in callbacks from the helper.

core_device_info_vendor_data vendor data to include in core device info.

core_device_info_vendor_length length of vendor_data.

controller_head Head of the list of uci session operations.

uwb_disabled Wether uwb is disabled or not

2.1.3.1.3 uci_backend_manager_init

void **uci_backend_manager_init**(struct *uci_backend_manager* **context*, struct *uci* **uci*, struct uwbmac_context **uwbmac*)

Initialize the manager and attach this backend to the uci context.

Parameters

- **context** (*struct uci_backend_manager **) – This backend context.
- **uci** (*struct uci **) – UCI context.
- **uwbmac** (*struct uwbmac_context **) – UWB MAC context.

2.1.3.1.4 uci_backend_manager_release

void **uci_backend_manager_release**(struct *uci_backend_manager* **context*)

Free ressources allocated.

Parameters

- **context** (*struct uci_backend_manager **) – This backend context.

2.1.3.1.5 struct uci_core_controller_ops

struct **uci_core_controller_ops**

The operations on core that are dispatched.

2.1.3.1.5.1 Definition

```
struct uci_core_controller_ops {
    void (*get_device_info)(uint8_t *mac_major, uint8_t *mac_minor,uint8_t *phy_major, uint8_
↪t *phy_minor, void *user_data);
}
```

2.1.3.1.5.2 Members

get_device_info Get device information.

2.1.3.1.6 struct uci_session_controller_ops

struct **uci_session_controller_ops**

The operations on session that are dispatched.

2.1.3.1.6.1 Definition

```
struct uci_session_controller_ops {
    uwbmac_error (*init)(struct uci *uci, uint16_t mt_gid_oid, uint32_t session_id, void␣
↪*user_data);
    uwbmac_error (*deinit)(struct uci *uci, uint16_t mt_gid_oid, uint32_t session_id, void␣
↪*user_data);
    uwbmac_error (*start)(struct uci *uci, uint16_t mt_gid_oid, uint32_t session_id, void␣
↪*user_data);
    uwbmac_error (*stop)(struct uci *uci, uint16_t mt_gid_oid, uint32_t session_id, void␣
↪*user_data);
    uwbmac_error (*set_app_config)(struct uci *uci, uint16_t mt_gid_oid,struct uci_session␣
↪*session,const struct uci_blk *payload, void *user_data);
    uwbmac_error (*get_app_config)(struct uci *uci, uint16_t mt_gid_oid,const struct uci_blk␣
↪*payload, void *user_data);
}
```

2.1.3.1.6.2 Members

init Initialize a session. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`session_id`: The session identifier that shall be initialized.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

deinit Deinitialize a session. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`session_id`: The session identifier that shall be deinitialized.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

start Start the ranging of given session. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`session_id`: The identifier for the ranging that should start.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

stop Stop the ranging of the given session. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`session_id`: The identifier for the ranging that should stop.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

set_app_config Configure a session. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`payload`: The whole uci message to process.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

get_app_config Get session configuration. Arguments:

`uci`: uci context needed to send uci messages.

`mt_gid_oid`: Union of message type, group identifier and opcode identifier.

`payload`: The whole uci message to process.

`user_data`: user data previously passed in uci_backend_manager_register.

Return: UWBMAC_SUCCESS or an error.

2.1.3.1.7 struct uci_session_controller

struct **uci_session_controller**
Definition of controller (or a sub-backend).

2.1.3.1.7.1 Definition

```
struct uci_session_controller {
    enum uci_session_type type;
    struct uci_core_controller_ops *core_ops;
    struct uci_session_controller_ops *session_ops;
    void *user_data;
    struct uci_session_controller *next;
}
```

2.1.3.1.7.2 Members

type Type of session associated with this controller.

core_ops Core ops of this controller.

session_ops Session ops of this controller.

user_data Data to pass to controller when calling an operation.

next Pointer to next controller, or NULL if thisis the last one. Filled at registration.

2.1.3.1.8 uci_backend_manager_register

void **uci_backend_manager_register**(struct *uci_backend_manager* **manager*, struct *uci_core_controller_ops* **core_ops*, struct *uci_session_controller_ops* **session_ops*, enum *uci_session_type type*, void **user_data*)

Register a controller.

Parameters

- **manager** (*struct uci_backend_manager **) – backend context
- **core_ops** (*struct uci_core_controller_ops **) – Controller core ops to register.
- **session_ops** (*struct uci_session_controller_ops **) – Controller session ops to register.
- **type** (*enum uci_session_type*) – Type of session to associate with ops.
- **user_data** (*void **) – Data to give ops when calling it, used to pass along their context.

2.1.3.1.9 uci_backend_manager_unregister

void **uci_backend_manager_unregister**(struct *uci_backend_manager* **manager*, enum *uci_session_type type*)

Remove all handlers associated to a session type.

Parameters

- **manager** (*struct uci_backend_manager **) – backend context
- **type** (*enum uci_session_type*) – Session type.

2.1.3.1.10 uci_backend_manager_disable_uwb

void **uci_backend_manager_disable_uwb**(struct *uci_backend_manager* **manager*, bool *disable*)

Disable uwb because it is used by another stack.

Parameters

- **manager** (*struct uci_backend_manager **) – context.
- **disable** (*bool*) – true to disable, false to enable.

2.1.3.1.10.1 Description

On some hardware two stacks have access to the chipset. This is used to disable this stack when necessary. It is enabled by default.

2.1.3.1.11 struct uci_session

struct **uci_session**
Holds information on a session.

2.1.3.1.11.1 Definition

```
struct uci_session {
    struct uci_session *next;
    void (*destructor)(struct uci_session *);
    uint32_t id;
    uint8_t type;
    uint8_t channel_number;
    bool is_active;
    int n_measurements;
    uint8_t nb_range;
    uint8_t nb_elevation;
    uint8_t nb_azimuth;
    uint8_t aoa_result_req;
}
```

2.1.3.1.11.2 Members

next Next element in list.NOTE: Private member

destructor Function to call to release all memory for this struct.

id Unique ID.NOTE: Private member

type Type as given at init.

channel_number Channel number in use by the session, or 0 if none.NOTE: Private member

is_active Session is currently active, i.e. ranging.NOTE: Private member

n_measurements Cached current number of measurements.

nb_range Number of range measurements.

nb_elevation Number of elevation measurements.

nb_azimuth Number of azimuth measurements.

aoa_result_req The type of aoa report requested.

2.1.3.1.12 uci_session_destructor

void **uci_session_destructor**(struct *uci_session* *session)
Simple destructor that just frees.

Parameters

- **session** (*struct uci_session **) – Session to destroy.

2.1.3.1.13 uci_session_add

enum *uci_status_code* **uci_session_add**(struct *uci_backend_manager* **manager*, uint32_t *session_id*, uint8_t *session_type*, void (**destructor*)(struct *uci_session* *), struct *uci_session* **session*)

Add a new session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.
- **session_type** (*uint8_t*) – Session type.
- **(*destructor)(struct uci_session *)** (*void*) – Function to call to release all memory for this struct.
- **session** (*struct uci_session **) – Session to add.

2.1.3.1.13.1 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_DUPLICATE (session with this ID exists).

2.1.3.1.14 uci_session_remove

enum *uci_status_code* **uci_session_remove**(struct *uci_backend_manager* **manager*, uint32_t *session_id*, uint8_t *session_type*)

Remove and destroy a session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.
- **session_type** (*uint8_t*) – Session type.

2.1.3.1.14.1 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_NOT_EXIST (session does not exist).

2.1.3.1.15 uci_session_get

enum *uci_status_code* **uci_session_get**(struct *uci_backend_manager* **manager*, uint32_t *session_id*, uint8_t *session_type*, struct *uci_session* ***session*)

Get a session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.
- **session_type** (*uint8_t*) – Session type.
- **session** (*struct uci_session ***) – Pointer to session structure (can be NULL if not needed).

2.1.3.1.15.1 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_NOT_EXIST (session does not exist) or UCI_STATUS_ERROR_SESSION_DUPLICATE (exists with a different type).

2.1.3.1.16 uci_session_get_current_channel_number

uint8_t **uci_session_get_current_channel_number**(struct *uci_backend_manager* *manager)
Get channel number in use.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.

2.1.3.1.16.1 Return

channel number or 0 if none in use.

2.1.3.1.17 uci_session_set_channel_number

enum *uci_status_code* **uci_session_set_channel_number**(struct *uci_backend_manager* *manager, uint32_t *session_id*, uint8_t *channel_number*)
Set channel number for session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.
- **channel_number** (*uint8_t*) – Channel number specified in app config.

2.1.3.1.17.1 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_NOT_EXIST (session does not exist).

2.1.3.1.18 uci_set_channel

enum *uci_status_code* **uci_set_channel**(struct *uci_backend_manager* *manager, uint32_t *session_id*)
set channel number in the device.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.

2.1.3.1.18.1 Return

UCI_STATUS_OK or UCI_STATUS_FAILED or UCI_STATUS_ERROR_ACTIVE_SESSIONS_ONGOING (another channel is already in use).

2.1.3.1.19 uci_session_start

enum *uci_status_code* **uci_session_start**(struct *uci_backend_manager* **manager*, uint32_t *session_id*)

Start ranging for session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.

2.1.3.1.19.1 If no channel is un use

- if session has specified a channel number, use it
- if session has not specified a channel number, use default

return UCI_STATUS_OK

If a channel number is in use (some sessions active), compare to session's: - if session has none or same channel number, return UCI_STATUS_OK - if session has same, return UCI_STATUS_ERROR_ACTIVE_SESSIONS_ONGOING

2.1.3.1.19.2 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_NOT_EXIST (session does not exist) or UCI_STATUS_ERROR_SESSION_ACTIVE (session is already active) or UCI_STATUS_ERROR_ACTIVE_SESSIONS_ONGOING (another channel is already in use).

2.1.3.1.20 uci_session_stop

enum *uci_status_code* **uci_session_stop**(struct *uci_backend_manager* **manager*, uint32_t *session_id*)

Stop ranging for session.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **session_id** (*uint32_t*) – Session ID.

2.1.3.1.20.1 Return

UCI_STATUS_OK or UCI_STATUS_ERROR_SESSION_NOT_EXIST (session does not exist) or UCI_STATUS_ERROR_SESSION_NOT_CONFIGURED (session is not active).

2.1.3.1.21 uci_session_count

int **uci_session_count**(struct *uci_backend_manager* **manager*, int *type*)

Get sessions number.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **type** (*int*) – Type to count sessions for, or UCI_SESSION_TYPE_ALL for all types.

2.1.3.1.21.1 Return

number of sessions in manager.

2.1.3.1.22 uci_session_clear

void **uci_session_clear**(struct *uci_backend_manager* **manager*, int *type*, void (**cb*)(struct *uci_session* *, void *), void **user_data*)

Remove and destroy all sessions for a given type.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **type** (*int*) – Type to remove sessions for, or UCI_SESSION_TYPE_ALL for all types.
- **(*cb)(struct uci_session *, void *)** (*void*) – Function to run before destroying session.
- **user_data** (*void **) – Cookie to pass to the callback above.

2.1.3.1.23 uci_backend_manager_set_vendor_data

void **uci_backend_manager_set_vendor_data**(struct *uci_backend_manager* **manager*, const uint8_t **data*, size_t *length*)

Set for core device info vendor data.

Parameters

- **manager** (*struct uci_backend_manager **) – Session manager to use.
- **data** (*const uint8_t **) – Pointer to vendor data to include in core device info response.
- **length** (*size_t*) – The vendor data length.

2.1.3.2 UCI Backend Coordinator

2.1.3.2.1 Coordinator overview

The uci_backend_coordinator implements MAC scheduler and region selection for the protocol backends. It also implements device state management.

The various backends are not aware of the state of scheduler or region when a session of a certain type is required, or what state the device is in (active or not). This interface provides a way for them to request a certain state. The coordinator implementation is meant to be custom depending on the project. It is meant to vary depending on the use case, target and so on. A default implementation is provided.

2.1.3.2.2 struct uci_backend_coordinator_ops

struct **uci_backend_coordinator_ops**
Operations provided by the coordinator.

2.1.3.2.2.1 Definition

```
struct uci_backend_coordinator_ops {
    uwbmac_error (*request_region)(struct uci_backend_coordinator *coord,struct uwbmac_
↪context *context, const char *region_name);
    void (*release_region)(struct uci_backend_coordinator *coord,struct uwbmac_context␣
↪*context, const char *region_name);
    uwbmac_error (*request_start)(struct uci_backend_coordinator *coord, struct uwbmac_
↪context *context);
    void (*suggest_stop)(struct uci_backend_coordinator *coord, struct uwbmac_context␣
↪*context);
    const char *(*get_scheduler)(struct uci_backend_coordinator *coord, const char *region_
↪name);
    int (*get_region_id)(struct uci_backend_coordinator *coord, const char *region_name);
}
```

2.1.3.2.2.2 Members

request_region Callback invoked when a backend is requesting a certain region.

release_region Callback invoked when a backend no longer needs a certain region.

request_start Callback invoked when a backend needs the device to be started.

suggest_stop Callback invoked when a backend no longer needs the device.

get_scheduler Get the scheduler used for the given region.

get_region_id Get the region id used for the given region.

2.1.3.2.3 struct uci_backend_coordinator

struct **uci_backend_coordinator**
UCI generic backend coordinator.

2.1.3.2.3.1 Definition

```
struct uci_backend_coordinator {
    const struct uci_backend_coordinator_ops *ops;
}
```

2.1.3.2.3.2 Members

ops Coordinator operations.

2.1.3.2.3.3 Description

Coordinator is a mechanism for backends to handle device state and region state without knowing each others.

2.1.3.3 UCI Backend Fira

2.1.3.3.1 Backend FiRa overview

The uci_backend_fira implements the control of ranging type session.

For every UCI commands relative to a session, the backend manager will dispatch it to this fira backend if the session is of type ranging. Then this fira backend will realise the command through uwbmac API using the fira helper interface. This will then be realized by the fira region inside the MAC.

2.1.3.3.2 enum meas_seq_template_type

enum **meas_seq_template_type**
Measurement step type.

2.1.3.3.2.1 Definition

```
enum meas_seq_template_type {
    TEMPLATE_RANGING,
    TEMPLATE_COMBO,
    TEMPLATE_AZIMUTH,
    TEMPLATE_ELEVATION,
    TEMPLATE_NB
};
```

2.1.3.3.2.2 Constants

TEMPLATE_RANGING Ranging measure.

TEMPLATE_COMBO Both azimuth and elevation measure.

TEMPLATE_AZIMUTH Azimuth measure.

TEMPLATE_ELEVATION Elevation measure.

TEMPLATE_NB Internal use.

2.1.3.3.3 struct antenna_parameters

struct **antenna_parameters**
Definition of antennas parameters.

2.1.3.3.3.1 Definition

```
struct antenna_parameters {
    struct measurement_sequence_step ch5[TEMPLATE_NB];
    struct measurement_sequence_step ch9[TEMPLATE_NB];
    uint8_t aoa_capability;
}
```

2.1.3.3.3.2 Members

ch5 Fira measurement sequence step for channel 5.

ch9 Fira measurement sequence step for channel 9.

aoa_capability Angle Of Arrival capability: 0 (no AOA), 1 (Azimuth only), 2 (Azimuth and Elevation).

2.1.3.3.4 struct stop_ntf_policy

struct **stop_ntf_policy**
Definition of stop notification policy.

2.1.3.3.4.1 Definition

```
struct stop_ntf_policy {
    int64_t session_id;
    int16_t stopped_reason;
}
```

2.1.3.3.4.2 Members

session_id the session being stopped, or negative value if no session are being stopped.

stopped_reason reason why the session is being stopped:0x01: UCI_SESSION_REASON_MAX_NUMBER_OF_MEASURE 0x02: UCI_SESSION_REASON_MAX_RANGING_ROUND_RETRY_COUNT_REACHED. negative value: session is not being stopped.

2.1.3.3.5 struct device_params

struct **device_params**
Definition of device specific parameters

2.1.3.3.5.1 Definition

```
struct device_params {
    enum uci_device_state device_state;
    uint8_t low_power_mode;
    uint8_t default_channel;
}
```

2.1.3.3.5.2 Members

device_state current uci device state.

low_power_mode current low power mode:0x00: disabled. 0x01: enabled.

default_channel the device default channel.

2.1.3.3.6 struct uci_backend_fira_context

struct **uci_backend_fira_context**
Definition of this backend context.

2.1.3.3.6.1 Definition

```
struct uci_backend_fira_context {
    struct fira_context fira_context;
    struct uwbmac_context *uwbmac_context;
    struct antenna_parameters *antennas;
    void (*core_device_reset_cb)(uint8_t reason, void *user_data);
    void *core_device_reset_cb_user_data;
    size_t running_sessions;
    struct stop_ntf_policy stop_ntf_policy;
    struct device_params device_param;
    struct uci *uci;
    struct uci_backend_coordinator *coord;
    struct uci_backend_manager *sess_man;
}
```

2.1.3.3.6.2 Members

fira_context context for fira helper.

uwbmac_context UWB MAC context.

antennas antennas parameters.

core_device_reset_cb callback for core device reset.

core_device_reset_cb_user_data parameters for core_device_reset_cb.

running_sessions number of running sessions.

stop_ntf_policy stop notification policy.

device_param devices parameters.

uci uci context used in callbacks from the helper.

coord uci backend coordinator.

sess_man the root backend manager.

2.1.3.3.7 uci_backend_fira_init

uwbmac_error **uci_backend_fira_init**(struct *uci_backend_fira_context* *context, struct *uci* *uci, struct uwbmac_context *uwbmac_context, struct *uci_backend_coordinator* *coord, struct *uci_backend_manager* *sess_man)

Attach this backend to the uci context to bridge uci communication to the MAC.

Parameters

- **context** (*struct uci_backend_fira_context **) – FIRA context.
- **uci** (*struct uci **) – UCI context.
- **uwbmac_context** (*struct uwbmac_context **) – UWB MAC context.
- **coord** (*struct uci_backend_coordinator **) – backend coordinator.
- **sess_man** (*struct uci_backend_manager **) – session manager.

2.1.3.3.7.1 Return

UWBMAC_SUCCESS or an error code.

2.1.3.3.8 uci_backend_fira_set_reset_callback

void **uci_backend_fira_set_reset_callback**(struct *uci_backend_fira_context* *context, void (*cb)(uint8_t reason, void *user_data), void *user_data)

Set for core device reset callback.

Parameters

- **context** (*struct uci_backend_fira_context **) – This backend context.
- **(*cb)(uint8_t reason, void *user_data)** (*void*) – The callback to call on reset, or NULL.

- **user_data** (*void **) – The callback private data.

2.1.3.3.9 uci_backend_fira_send_reset_response

void **uci_backend_fira_send_reset_response**(struct *uci* **uci*, bool *success*)
Send response to core device reset.

Parameters

- **uci** (*struct uci **) – UCI context.
- **success** (*bool*) – Status of the reset command.

2.1.3.3.9.1 Description

Must be called once done resetting the chip, to send the UCI response back.

2.1.3.3.10 uci_backend_fira_set_antenna_conf

void **uci_backend_fira_set_antenna_conf**(struct *uci_backend_fira_context* **context*, struct *antenna_parameters* **antennas_params*)
Give backend the antenna conf to configure session according to aoa requests.

Parameters

- **context** (*struct uci_backend_fira_context **) – This backend context.
- **antennas_params** (*struct antenna_parameters **) – The antenna configuration.

2.1.3.3.11 uci_backend_fira_release

void **uci_backend_fira_release**(struct *uci_backend_fira_context* **context*)
Free global ressources used.

Parameters

- **context** (*struct uci_backend_fira_context **) – This backend context.

3 Power Saving Using Deep Sleep State

3.1 Introduction

The goal of this development is describe what must be done to put the DW3xxx in a low power state as often as possible and provide a general interface for power saving needs for other chips.

3.2 Low Power States

There are five candidates for the DW3xxx:

- `IDLE_RC`: In this state, a fast RC clock (~120 MHz) is used to clock the chip, allowing communications at full SPI speed. The crystal oscillator is still running so that transitions to `IDLE_PLL` is fast (~20 us).
- `INIT_RC` (actually `IDLE_RC` with clock ÷ 4): In this state, a fast RC clock divided by four (~120 MHz ÷ 4 = ~30 MHz) is used to clock the chip, SPI clock is limited to 7 MHz. Crystal oscillator is still running, but not used. Transition to `IDLE_RC` is fast (time of SPI access).
- `SLEEP`: In this state, a slow RC clock is kept activated which can be used to wake up the chip. The timer granularity is large and the clock imprecise. We see no use for this state.
- `DEEP_SLEEP`: In this state, all clocks are shut down, Only VDD1 is present. To transition to `IDLE_RC`, first a signal (`WAKEUP` or `CS`) must be asserted for more than 500 us, then it takes about 1 ms to wake up, restore saved registers, and start the oscillator. The chip can transition to `IDLE_PLL` once the crystal oscillator is stabilized, which depend on the connected crystal. Full wake up can take up to about 2 ms.
- `OFF`: In this state, the chip is powered off. This is only possible if there is a external switch or regulator with enable signal. Wake up time is similar to `DEEP_SLEEP`.

In all those states, the clock used for ranging can not be used. This means that if ranging is active, state must be `IDLE_PLL`.

Also, in all those states, the DW3xxx system time is not maintained.

Corresponding power consumption (Dual rail, VDD1 = VDD2 = 2.5 V, VDD3 = 1.6 V, PMIC efficiency = 90%), from datasheet:

State	Consumption
IDLE_PLL	31.7 mW (ch5), 56.3 mW (ch9)
IDLE_RC	22 mW
INIT_RC	8.25 mW (from measurements)
SLEEP	2337 nW
DEEP_SLEEP	715 nW
OFF	~0

First step will be to use the `DEEP_SLEEP` state, other states will come as a bonus.

3.3 Timestamps and Durations

There are several timestamp and duration units handled by MCPS:

- RCTU (ranging counter time unit) is used only for timestamping frames for ranging. It corresponds to a counter running at about 64 GHz (499.2 MHz × 128) as specified by 802.15.4.
- RSTU (ranging slot time unit) is a coarser unit used to define superframe structure for example. It corresponds to a counter running at 1.2 MHz (499.2 MHz ÷ 416) as specified by 802.15.4z.
- DTU (device time unit) is used to sequence frames. Its unit is provided by the driver, it must correspond to a counter running at an integer frequency in Hz.

3.3.1 Device Time Unit Definition

The MCPS has no constraint on the relation between RCTU and DTU.

For precise timings, RSTU durations should be convertible to DTU durations without loss. More precisely, the tolerance of DTU durations with respect to the PHY clock must be within ±100 ppm (this holds for a ranging block duration according to 802.15.4z), but better accuracy is desirable. Having RSTU being an integer multiple of a DTU is the best choice.

The DTU timestamps related to frames (TX frame timestamps, RX frame timestamps) must respect the same precision.

The DTU timestamps related to current time, or RX activation time can have a lower precision (for example, it's acceptable to enable RX a little bit earlier, as long as the requested RX enable window is included in the effective RX enable window).

DTU timestamps must be stored as 32-bit integers, and value must wraparound without discontinuity.

The maximum value divided by two must be large enough to express the maximum duration between two accesses (about 24 seconds for CCC, about 66 seconds for FiRa, for example).

The counter giving the current DTU timestamps value must never stop while the UWB interface is up. It is however acceptable to stop it when the UWB interface is completely shut down, all timestamps values are invalidated in this case.

Giving RMARKER timestamp in advance can add another constraint on the DTU see *Giving the RCTU Timestamp in Advance* for details.

A non-goal is to be able to transmit a frame at a very precise time after a received frame. The precision we can reach with the current hardware is not sufficient to allow ranging without reply time communication anyway. The precision of the response will be limited to the size of the DTU.

3.3.2 DTU Counter Implementation Using DW3xxx System Time

In the current implementation, DTU corresponds to a counter running at 249.6 MHz, so that it can be used directly with DW3xxx system time.

The DW3xxx system time is a 32 bit register, with the least significant bit always 0.

As 249.6 MHz (and even 124.8 MHz, because of the least significant bit) is an integer multiple of 1.2 MHz, the precision requirement is respected.

As the counter is a overflowing 32 bit register, the overflow requirement is respected.

The half period is 2^31 / 249.6 MHz = 8.6 s, which is too short.

The counter stops as soon as a low power mode is entered, this is not good too.

Therefore DTU counter must be replaced with an external counter.

3.4 DTU Implementation Considerations

3.4.1 Synchronization

As the DW3xxx uses its own counter as a source for all used timestamps, DTU must be synchronized with the DW3xxx system time (`SYS_TIME`) every time the DW3xxx is woken up.

Unless using dedicated hardware (which is not the case for DW3xxx), this procedure may imply a loss of precision. To avoid drifting, synchronization must only be done when the DW3xxx is waking up, *not* when put into sleep. This way, precision loss does not accumulate.

If the synchronisation procedure is reproducible (always respect the same timing), it can use the following algorithm:

- Read the current value of the DTU counter.
- Read the current value of the DW3xxx system time.
- Store a representation of both values for future conversions.

If the synchronisation procedure is not reproducible, the following algorithm can be used:

- Time the following procedure:
  - Read the current value of the DTU counter.
  - Read the current value of the DW3xxx system time.
- If the procedure took too much time to guarantee requested precision:
  - Restart up to a fixed number of time,
  - Or, accept the precision loss (degraded mode),
  - Or return an error.
- Store a representation of both values for future conversions.

To improve accuracy, the value read from the DW3xxx system time can be applied an offset due to the delay between the moment the DTU counter is read and the moment the system time is sampled inside the component.

Instead of reading the DTU counter, the source counter used to derive the DTU counter can be read and the conversion done at the end of the procedure. The source counter or the DTU counter can also be used to time the procedure.

If reading the current value of the DTU counter is not precise enough for the requested precision (for example, the counter least significant bits are not updated), the procedure must wait for a value change, using polling or an interrupt.

Once the synchronisation is done, there is no need to read the DW3xxx system time until the next wake up. When MCPS requests the current timestamp, it can be directly returned using the DTU counter. This is also the case when DW3xxx is asleep of course.

Every time a DTU value is given to the driver, or given by the driver, the driver is responsible to convert the value to/from the DW3xxx system time if needed.

The DW3xxx also uses other various units, the driver is responsible to do the conversion.

In the rest of this document, the DTU counter value used for synchronisation is called DTU_{sync}, the corrected system time is called $SysTime_{sync}$.

3.4.2 Converting Between DTU and DW3xxx System Time

If the DTU is an integer multiple of the DW3xxx system time unit, then the conversion is easy. It's even easier if this integer is a power of two, 2^N.

All operations are defined for integers arithmetic modulo 2^{32}, $\ll$ is the left shift and $\gg$ is the logical right shift.

When converting a DTU timestamp to DW3xxx system time:

$$SysTime = (DTU - DTU_{sync} \ll N) + SysTime_{sync}$$

Or equivalent, but less intuitive:

$$SysTime = (DTU \ll N) + (SysTime_{sync} - (DTU_{sync} \ll N))$$

When converting a DW3xxx system time to a DTU timestamp, this is a little bit more complicated because there are missing bits, given DTU_{near} a timestamp in the neighborhood of the system time to convert:

$$\begin{aligned}
DTU_{\times 2^N} &= SysTime - (SysTime_{sync} - (DTU_{sync} \ll N)) \\
DTU_{top} &= DTU_{\times 2^N} \gg 30 \\
DTU_{neartop} &= (DTU_{near} \gg 30 - N) \wedge 11_2 \\
DTU_{lsb} &= DTU_{\times 2^N} \gg N \\
DTU_{msb} &= \begin{cases} (DTU_{near} \gg 32 - N) + 1 & \text{if } DTU_{neartop} = 11_2 \text{ and } DTU_{top} = 00_2 \\ (DTU_{near} \gg 32 - N) - 1 & \text{if } DTU_{neartop} = 00_2 \text{ and } DTU_{top} = 11_2 \\ DTU_{near} \gg 32 - N & \text{otherwise} \end{cases} \\
DTU &= (DTU_{msb} \ll 32 - N) \vee DTU_{lsb}
\end{aligned}$$

If the relative position to DTU_{near} is known, a simpler formula can be used, for example, if DTU_{near} is known to be before the system time to convert:

$$\begin{aligned}
DTU_{lsb} &= SysTime - (SysTime_{sync} - (DTU_{sync} \ll N)) \gg N \\
DTU_{add} &= ((\neg DTU_{lsb} \wedge DTU_{near}) \wedge 2^{31-N}) \ll 1 \\
mask &= 2^{32-N} - 1 \\
DTU &= ((DTU_{near} \wedge \neg mask) \vee DTU_{lsb}) + DTU_{add}
\end{aligned}$$

3.4.3 Converting Between Source Counter and DTU

3.4.3.1 When Frequency Ratio Is Not an Integer

With F_{DTU} the DTU counter frequency and F_{Source} the source frequency:

$$F_{Source} = \frac{N}{D} F_{DTU}$$

If the computation can be done without overflow, the conversion is simple:

$$DTU = \left(\frac{N(Source - Source_0)}{D} + DTU_0 \right) \wedge (2^{32} - 1)$$

DTU	timestamp in DTU
DTU_0	reference timestamp in DTU, can be 0
$Source$	timestamp using source counter
$Source_0$	reference timestamp using source counter, can be 0

But usually, that is not the case. A solution is to do the conversion incrementally, using an algorithm inspired from Bresenham's line algorithm, but using division.

$$\begin{aligned}
DTU_0 &= 0 \\
Source_0 &= \text{initial source counter value} \\
Error_0 &= 0 \\
&\vdots \\
Source_n &= \text{current source counter value} \\
q_n, r_n &= \operatorname{divmod}(Error_{n-1} + N(Source_n - Source_{n-1}), D) \\
DTU_n &= DTU_{n-1} + q_n \\
Error_n &= r_n
\end{aligned}$$

With $\operatorname{divmod}(a, b)$ returning the quotient and remainder of the euclidean division of a by b.

A new step must be computed each time a DTU timestamp is needed and at regular interval to avoid any overflow.

3.4.3.2 When Source Counter Range Is Too Short

There are two solutions:

- Regularly sample the source counter to extend the counter value range (increment the most significant par on each counter overflow).
- Use the algorithm presented in the previous section for incremental conversion.

3.4.3.3 When Source Counter Frequency Is Too Low

This is not too big of a problem:

- When requesting the current value, the returned DTU timestamp must be "pessimistic", and rounded up.
- When programming or reporting an event, the DTU timestamp is converted to/from the DW3xxx system time and therefore it will have the requested precision.
- When waking up the DW3xxx, the margin must take into account the low frequency of the source counter and wake the DW3xxx earlier.

3.5 Giving the RCTU Timestamp in Advance

The DW3xxx is able to transmit a frame at a precise timestamp allowing to include, in the message payload, information computed from the timestamp of this very message. This feature enables non-deferred mode ranging. This timestamp can be computed from the DW3xxx system time transmission timestamp, and the antenna delay.

When the DW3xxx is in a low power mode, the system time is not running and therefore the timestamp can not be determined. However, there is nothing preventing the driver to guarantee the relative values of the returned timestamps.

When the MCPS requests the RMARKER timestamp for a frame to be transmitted and the ranging clock is not running:

- If this is the first time this is requested, the returned value must be determined (can be zero added to the SHR duration and the antenna delay for example) and the associated timestamp in DTU must be recorded.
- If this is not the first time, the returned value must be computed using the stored information.

Once the ranging clock is running:

- The stored information and the synchronization information must be used to adjust returned timestamps.

When the ranging clock is later stopped, or on a `stop`, `rx_disable`, `reset` or `idle` callback:

- The stored information is discarded and the process starts anew.

For this to work, it must be possible to compute the difference between two RMARKER timestamps in RCTU given the difference between two transmission timestamps in DTU and a defined SHR format.

For the DW3xxx, this means that:

- The DTU must not be smaller than the system time unit, unless special precaution is taken when synchronisation is done to always align transmission time the same way. This is not a concern as system time unit is too small for DTU anyway.
- The DTU must be an integer multiple of the system time unit, which is a sane assumption anyway.

3.6 MCPS Handling of Power Saving

3.6.1 Problem Statement

Power saving state can be entered if no ranging is currently running and the next action is far enough in the future to allow waking up.

When a Tx is done, there is usually no surprise for the calling code, and therefore the state which can be entered after the Tx can be known before the Tx is requested. This allows to program the device to automatically enter a low power state if possible right after the frame has been transmitted.

When an Rx is done, the following action usually depends on whether a frame was received and the eventual frame content. In this case, a conservative decision must be done.

Currently, the MCPS handles one access at a time, in the future, it could be able to anticipate the next access in order to refine the latency requirement after the last step of the current access which will improve a little bit the power saving.

In all cases, surprises can happen (change of schedule, memory exhaustion, unrelated error...) and the MCPS needs a way to signal that no event is expected in the near future and a sleep state can be entered.

Also, as MCPS is able to run concurrent protocols with different precision requirements, the assumption is that an access must always be done precisely (by respecting the DTU precision constraints), unless the access is immediate.

For immediate accesses, the latency can be of several milliseconds.

3.6.2 Anticipation

The driver declares in `anticip_dtu` the time it needs to program any action. This includes the interrupt latency, an eventual wake up delay, the time needed for processing, and the time needed for hardware access.

This anticipation must be given for active state, not for low power state, this is the best the driver can do.

Currently, this is used by the MCPS to determine the first timestamp that can be used after the current timestamp has been read.

In the future, this value will also be used when programming an access after an access is done, and to refuse unrealistic constraints from applications.

3.6.3 New API

For all API taking a timestamp in DTU, the driver must use a low power mode as long as the other constraints are respected (need for a ranging clock and timestamp precision). This must also be done for vendor specific commands taking a timestamp.

The power state after an action must be determined according to the following requirements. If possible, the device can be programmed to switch to a low power state automatically after an action is done without software intervention.

For the `tx_frame` callback:

- `MCPS802154_TX_FRAME_CONFIG_RANGING` flag: In addition to its current meaning, the clock used for ranging must be started at the frame transmission if not active yet.
- New `MCPS802154_TX_FRAME_CONFIG_KEEP_RANGING_CLOCK` flag: When set, the clock used for ranging must be kept active after the frame has been transmitted.
- In case `tx_frame` returns an error, the clock must be left in the state it was before the call.
- New `next_delay_dtu` argument: This is the expected delay between the *start* of the transmitted frame and the next action. This is a best effort delay, and the next action is not guaranteed to pass successfully. It can be zero to request a minimum delay.

For the `rx_enable` callback:

- `MCPS802154_RX_FRAME_CONFIG_RANGING` flag: In addition to its current meaning, the clock used for ranging must be started at the frame reception if not active yet.
- New `MCPS802154_RX_FRAME_CONFIG_KEEP_RANGING_CLOCK` flag: When set, the clock used for ranging must be kept active after the frame has been received, or after an error or a timeout was signaled.
- In case `rx_enable` returns an error, the clock must be left in the state it was before the call.
- New `next_delay_dtu` argument: This is the expected delay between the *start* of the received frame or timeout event and the next action. This is a best effort delay, and the next action is not guaranteed to pass successfully. It can be zero to request a minimum delay.

The `next_delay_dtu` argument is adapted to slot based communication protocols. In a future API change, negative value could be supported to give the delay between the *end* of the transmitted frame and the next action, or if `info->rx_enable_after_tx_dtu` is set and for `rx_enable`, the delay between the *end* of the received frame or timeout event. For the moment, this is not supported and a 0 value must be used for such use cases.

For the `start`, `rx_disable` and `reset` callbacks, the delay is an implementation detail. The driver may keep the device in a low power mode. In these cases, the `get_current_timestamp_dtu` must always be used by MCPS prior to request an action at a specific timestamp.

Please note that when reception is disabled automatically due to programming another action, the delay after disabling the reception is also an implementation detail, but it must be chosen to match the requirement of the programmed action.

For vendor commands without more precise specification, the delay is always `anticip_dtu` after the end of the command.

There is a new `idle` callback to signal that there will be no action until the given timestamp. The driver is free to use any low power state as long as it calls the `mcps802154_timer_expired` long enough before the given timestamp so that a new action can be programmed.

There can be several strategies to implement the `idle` callback:

- The naive approach: call back as soon as possible, this could trigger an busy loop and should be avoided.
- The power hungry approach: wake up before the given date as late as possible.
- The power saving approach: use the lowest power possible state and call back soon enough to allow wake up if needed, this should be the chosen solution.

It is possible that MCPS decides to interrupt an idle state. In this case, it will request a new action, which can be any action, and even a new idle condition. In this case, if the requested action is at a specific timestamp, it will call `get_current_timestamp_dtu` callback before.

If the `mcps802154_timer_expired` function is called while the MCPS is not idle, it will be ignored. This handles the situation where there was a race condition between the call back and an idle condition interruption. If the MCPS interrupted an idle condition, requested a new idle condition, and there was a race condition leading to `mcps802154_timer_expired` being called for the first idle condition, the MCPS will handle it gracefully.

The `get_current_timestamp_dtu` has a new meaning when the device is currently in a low power state: the wake up delay must be added to the returned value so that the next possible action timestamp can always be computed using the returned result plus `anticip_dtu`.

The new `idle_dtu` value in `mcps802154_llhw` gives a duration considered long enough to prefer to call the `idle` callback rather that try to find a valid access. If this value is too large, this could add a latency in case a new data is to be sent as the schedule never go backward. It should be set long enough to allow usage of a low power state while still minimizing latency.

The `tx_timestamp_dtu_to_rmarker_rctu` must be implemented to allow requesting the RMARKER timestamp even when the ranging clock is not active, see *Giving the RCTU Timestamp in Advance* for details.

3.6.4 Nothing to Do

There are several occasions where the MCPS has nothing to do. Previously, this was a tedious situation as it was not able to wait. There was a “nothing” access type to handle this situation, but is has no end and therefore is not suitable to handle medium sharing between several regions.

Previously, a region handler was not allowed to return no allocation at the start of a region. This was done to avoid an infinite loop where time would go forward with nothing to stop it.

With the new `idle` callback, MCPS is able to handle the situation gracefully. This implies the following changes:

- The “nothing” access type must be removed, instead, the region `get_access` handler must return `NULL`.
- Returning `NULL` from `get_access` is always permitted. The schedule will continue to be explored until an access is found, or the next timestamp is `idle_dtu` in the future.

- When no access is possible, the “nothing” FProc FSM must be used and the idle condition be signaled to the driver.

3.6.5 Removed API

The following API are no longer needed:

- `timestamp_dtu_to_rctu`
- `timestamp_rctu_to_dtu`

The following API are deprecated and will be changed in the future (not now!):

- RCTU timestamps being a 64 bit value, a 32 bit value cover 67 ms, which is really large for a ranging session. Regions needing larger range must extend range using a technique similar to the one used to convert System Time to DTU.
- `difference_timestamp_rctu`, once RCTU will be a 32 bit value.

Index

A

antenna_parameters (*C type*), 80
aoa_measurements (*C type*), 29
aoa_measurements_index (*C type*), 29

C

controlee_parameters (*C type*), 28
controlees_parameters (*C type*), 28

D

data_parameters (*C type*), 28
device_params (*C type*), 81

F

fira_device_role (*C type*), 41
fira_device_type (*C type*), 41
fira_embedded_mode (*C type*), 43
fira_helper_add_controlees (*C function*), 38
fira_helper_close (*C function*), 33
fira_helper_deinit_session (*C function*), 36
fira_helper_delete_controlees (*C function*), 38
fira_helper_get_capabilities (*C function*), 34
fira_helper_get_controlees (*C function*), 38
fira_helper_get_session_parameters (*C function*), 36
fira_helper_init_session (*C function*), 34
fira_helper_notification_cb_t (*C function*), 32
fira_helper_open (*C function*), 33
fira_helper_send_data (*C function*), 39
fira_helper_session_get_count (*C function*), 37
fira_helper_session_get_state (*C function*), 37
fira_helper_set_controlees (*C function*), 37
fira_helper_set_partial_session_parameters (*C function*), 40
fira_helper_set_scheduler (*C function*), 34
fira_helper_set_session_parameters (*C function*), 36
fira_helper_start_session (*C function*), 35
fira_helper_stop_session (*C function*), 35
fira_mac_fcs_type (*C type*), 48
fira_measurement_report (*C type*), 43
fira_measurement_type (*C type*), 51
fira_multi_node_mode (*C type*), 42
fira_phr_data_rate (*C type*), 47
fira_preambule_duration (*C type*), 45
fira_prf_mode (*C type*), 44
fira_psdu_data_rate (*C type*), 46
fira_ranging_diagnostics_frame_report_flags (*C type*), 51
fira_ranging_round_usage (*C type*), 42
fira_ranging_status (*C type*), 49
fira_rframe_config (*C type*), 44
fira_rssi_report_type (*C type*), 48
fira_session_state (*C type*), 50
fira_sfd_id (*C type*), 45
fira_sts_config (*C type*), 48
fira_sts_length (*C type*), 52
fira_sts_segments (*C type*), 46

M

meas_seq_template_type (*C type*), 79
measurement_sequence (*C type*), 22
measurement_sequence_step (*C type*), 21

R

ranging_measurements (*C type*), 30
ranging_result_free (*C function*), 32
ranging_results (*C type*), 31

S

session_parameters (*C type*), 22
session_parameters_builder (*C type*), 39
session_parameters_builder_finish (*C function*), 40
session_parameters_builder_init (*C function*), 40
stop_ntf_policy (*C type*), 80

U

uci (*C type*), 67
uci_allocator (*C type*), 59
uci_allocator_ops (*C type*), 58
uci_backend_coordinator (*C type*), 79
uci_backend_coordinator_ops (*C type*), 78
uci_backend_fira_context (*C type*), 81
uci_backend_fira_init (*C function*), 82
uci_backend_fira_release (*C function*), 83

uci_backend_fira_send_reset_response (*C function*), 83
uci_backend_fira_set_antenna_conf (*C function*), 83
uci_backend_fira_set_reset_callback (*C function*), 82
uci_backend_manager (*C type*), 68
uci_backend_manager_disable_uwb (*C function*), 72
uci_backend_manager_init (*C function*), 69
uci_backend_manager_register (*C function*), 72
uci_backend_manager_release (*C function*), 69
uci_backend_manager_set_vendor_data (*C function*), 77
uci_backend_manager_unregister (*C function*), 72
uci_blk (*C type*), 57
uci_blk_alloc (*C function*), 60
uci_blk_destructible (*C type*), 57
uci_blk_flags (*C type*), 56
uci_blk_free_all (*C function*), 60
UCI_CONTAINER_OF (*C macro*), 56
uci_core_controller_ops (*C type*), 69
UCI_GID (*C macro*), 55
UCI_GID_OID (*C macro*), 55
uci_init (*C function*), 59
UCI_MAX_PACKET_SIZE (*C function*), 54
UCI_MAX_PAYLOAD_SIZE (*C function*), 54
uci_message_handler (*C type*), 61
uci_message_handler_function (*C function*), 60
uci_message_handlers (*C type*), 61
uci_message_handlers_register (*C function*), 62
UCI_MT (*C macro*), 55
UCI_MT_GID_OID (*C macro*), 54
UCI_OID (*C macro*), 56
UCI_PACKET_HEADER_SIZE (*C function*), 54
uci_packet_recv (*C function*), 63
uci_packet_recv_alloc (*C function*), 62
uci_packet_recv_free_all (*C function*), 63
uci_packet_response_expire (*C function*), 66
uci_packet_send_done (*C function*), 66
uci_packet_send_get_ready (*C function*), 66
uci_send_message (*C function*), 65
uci_send_status (*C function*), 65
uci_session (*C type*), 73
uci_session_add (*C function*), 74
uci_session_clear (*C function*), 77
uci_session_controller (*C type*), 71
uci_session_controller_ops (*C type*), 69
uci_session_count (*C function*), 77
uci_session_destructor (*C function*), 73
uci_session_get (*C function*), 74
uci_session_get_current_channel_number (*C function*), 75
uci_session_remove (*C function*), 74
uci_session_set_channel_number (*C function*), 75
uci_session_start (*C function*), 76
uci_session_stop (*C function*), 76
uci_set_channel (*C function*), 75
UCI_STATUS_PACKET_SIZE (*C function*), 54
uci_transport (*C type*), 64
uci_transport_attach (*C function*), 64
uci_transport_detach (*C function*), 65
uci_transport_ops (*C type*), 63
uci_uninit (*C function*), 59
uwbmac_call_region (*C function*), 15
uwbmac_call_region_cb_t (*C function*), 2
uwbmac_call_region_free (*C function*), 15
uwbmac_call_scheduler (*C function*), 14
uwbmac_call_testmode (*C function*), 18
uwbmac_channel_create (*C function*), 3
uwbmac_channel_receive (*C function*), 4
uwbmac_channel_release (*C function*), 4
uwbmac_channel_set_timeout (*C function*), 4
uwbmac_close_scheduler (*C function*), 12
uwbmac_error_to_errno (*C function*), 16
uwbmac_event_report (*C function*), 20
uwbmac_exit (*C function*), 6
uwbmac_get_calibration (*C function*), 8
uwbmac_get_channel (*C function*), 8
uwbmac_get_device_count (*C function*), 2
uwbmac_get_region_parameters (*C function*), 14
uwbmac_get_scheduler (*C function*), 12
uwbmac_get_scheduler_parameters (*C function*), 13
uwbmac_get_supported_channels (*C function*), 3
uwbmac_get_trace_modules (*C function*), 19
uwbmac_get_version (*C function*), 16
uwbmac_handle_report (*C function*), 20
uwbmac_init (*C function*), 5
uwbmac_init_device (*C function*), 3
uwbmac_is_started (*C function*), 6
uwbmac_list_calibration_context (*C type*), 9
uwbmac_list_calibrations (*C function*), 9
UWBMAC_MAX_CHANNEL_COUNT (*C function*), 2
uwbmac_poll_events (*C function*), 7
uwbmac_region_call_reply (*C function*), 20
uwbmac_register_report_callback (*C function*), 5
uwbmac_register_testmode_callback (*C function*), 17
uwbmac_set_calibration (*C function*), 8
uwbmac_set_channel (*C function*), 7
uwbmac_set_extended_addr (*C function*), 11
uwbmac_set_pan_id (*C function*), 10
uwbmac_set_promiscuous_mode (*C function*), 11
uwbmac_set_region_parameters (*C function*), 13

uwbmac_set_regions (*C function*), 13
uwbmac_set_scanning_mode (*C function*), 16
uwbmac_set_scheduler (*C function*), 11
uwbmac_set_scheduler_parameters (*C function*), 12
uwbmac_set_short_addr (*C function*), 10
uwbmac_set_trace_cb (*C function*), 18
uwbmac_start (*C function*), 6
uwbmac_stop (*C function*), 6
uwbmac_strerror (*C function*), 16
uwbmac_testmode_cb_t (*C function*), 17
uwbmac_testmode_reply (*C function*), 21
uwbmac_trace_enable (*C function*), 19
uwbmac_trace_info_t (*C type*), 19
uwbmac_tracing_cb_t (*C function*), 2