libhackrf

Generated by Doxygen 1.9.4

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

Module Documentation

3.1 Library related functions and enums

Library initialization, exit, error handling, etc.

Enumerations

- enum hackrf_error {
  HACKRF_SUCCESS = 0 ,
  HACKRF_TRUE = 1 ,
  HACKRF_ERROR_INVALID_PARAM = -2 ,
  HACKRF_ERROR_NOT_FOUND = -5 ,
  HACKRF_ERROR_BUSY = -6 ,
  HACKRF_ERROR_NO_MEM = -11 ,
  HACKRF_ERROR_LIBUSB = -1000 ,
  HACKRF_ERROR_THREAD = -1001 ,
  HACKRF_ERROR_STREAMING_THREAD_ERR = -1002 ,
  HACKRF_ERROR_STREAMING_STOPPED = -1003 ,
  HACKRF_ERROR_STREAMING_EXIT_CALLED = -1004 ,
  HACKRF_ERROR_USB_API_VERSION = -1005 ,
  HACKRF_ERROR_NOT_LAST_DEVICE = -2000 ,
  HACKRF_ERROR_OTHER = -9999 }

  error enum, returned by many libhackrf functions

Functions

- int hackrf_init ()

  Initialize libhackrf.
- int hackrf_exit ()

  Exit libhackrf.
- const char * hackrf_library_version ()

  Get library version.

- const char * hackrf_library_release ()

  Get library release.
- const char * hackrf_error_name (enum hackrf_error errcode)

  Convert hackrf_error into human-readable string.
- size_t hackrf_get_transfer_buffer_size (hackrf_device *device)

  Get USB transfer buffer size.
- uint32_t hackrf_get_transfer_queue_depth (hackrf_device *device)

  Get the total number of USB transfer buffers.

3.1.1 Detailed Description

3.1.2 Library initialization & exit

The libhackrf library needs to be initialized in order to use most of its functions. This can be achieved via the function hackrf_init. This initializes internal state and initializes `libusb`. You should only call this function on startup, but it's safe to call it later as well, only it does nothing.

When exiting the program, a call to hackrf_exit should be called. This releases all resources, stops background thread and exits `libusb`. This function should only be called if all streaming is stopped and all devices are closed via hackrf_close, else the error HACKRF_ERROR_NOT_LAST_DEVICE is returned.

3.1.3 Error handling

Many of the functions in libhackrf can signal errors via returning hackrf_error. This enum is backed by an integer, thus these functions are declared to return an int, but they in fact return an enum variant. The special case HACKRF_SUCCESS signals no errors, so return values should be matched for that. It is also set to the value 0, so boolean conversion can also be used. The function hackrf_error_name can be used to convert the enum into a human-readable string, useful for logging the error.

There is a special variant HACKRF_TRUE, used by some functions that return boolean. This fact is explicitly mentioned at those functions.

Typical error-handling code example:

```
result = hackrf_init();
if (result != HACKRF_SUCCESS) {
    fprintf(stderr,
        "hackrf_init() failed:  %s (%d)\n",
        hackrf_error_name(result),
        result);
    return EXIT_FAILURE;
}
```

Instead of `if (result != HACKRF_SUCCESS)` the line `if (result)` can also be used with the exact same behaviour.

The special case HACKRF_TRUE is only used by hackrf_is_streaming

3.1.4 Enum conversion

Most of the enums defined in libhackrf have a corresponding `_name` function that converts the enum value into a human-readable string. All strings returned by these functions are statically allocated and do not need to be `free`d. An example is the already mentioned hackrf_error_name function for the hackrf_error enum.

3.1.5 Library internals

The library uses `libusb` (version 1.0) to communicate with HackRF hardware. It uses both the synchronous and asynchronous API for communication (asynchronous for streaming data to/from the device, and synchronous for everything else). The asynchronous API requires to periodically call a variant of `libusb_handle_events`, so the library creates a new "transfer thread" for each device doing that using the `pthread` library. The library uses multiple transfers for each device (hackrf_get_transfer_queue_depth).

3.1.6 USB API versions

As all functionality of HackRF devices requires cooperation between the firmware and the host, both devices can have outdated software. If host machine software is outdated, the new functions will be unalaviable in `hackrf.h`, causing linking errors. If the device firmware is outdated, the functions will return HACKRF_ERROR_USB_API_VERSION. Since device firmware and USB API are separate (but closely related), USB API has its own version numbers. Here is a list of all the functions that require a certain minimum USB API version, up to version 0x0107

3.1.6.1 0x0102

- hackrf_set_hw_sync_mode
- hackrf_init_sweep
- hackrf_set_operacake_ports
- hackrf_reset

3.1.6.2 0x0103

- hackrf_spiflash_status
- hackrf_spiflash_clear_status
- hackrf_set_operacake_ranges
- hackrf_set_operacake_freq_ranges
- hackrf_set_clkout_enable
- hackrf_operacake_gpio_test
- hackrf_cpld_checksum

3.1.6.3 0x0104

- hackrf_set_ui_enable
- hackrf_start_rx_sweep

3.1.6.4 0x0105

- hackrf_get_operacake_boards
- hackrf_set_operacake_mode
- hackrf_get_operacake_mode
- hackrf_set_operacake_dwell_times

3.1.6.5 0x0106

- hackrf_get_m0_state
- hackrf_set_tx_underrun_limit
- hackrf_set_rx_overrun_limit
- hackrf_get_clkin_status
- hackrf_board_rev_read
- hackrf_supported_platform_read

3.1.6.6 0x0107

- hackrf_set_leds

3.1.7 Enumeration Type Documentation

3.1.7.1 hackrf_error

`enum hackrf_error`

Many functions that are specified to return INT are actually returning this enum

Enumerator

HACKRF_SUCCESS	no error happened
HACKRF_TRUE	TRUE value, returned by some functions that return boolean value. Only a few functions can return this variant, and this fact should be explicitly noted at those functions.
HACKRF_ERROR_INVALID_PARAM	The function was called with invalid parameters.
HACKRF_ERROR_NOT_FOUND	USB device not found, returned at opening.
HACKRF_ERROR_BUSY	Resource is busy, possibly the device is already opened.
HACKRF_ERROR_NO_MEM	Memory allocation (on host side) failed.

Enumerator

HACKRF_ERROR_LIBUSB	LibUSB error, use hackrf_error_name to get a human-readable error string (using `libusb_strerror`)
HACKRF_ERROR_THREAD	Error setting up transfer thread (pthread-related error)
HACKRF_ERROR_STREAMING_THREAD_ERR	Streaming thread could not start due to an error.
HACKRF_ERROR_STREAMING_STOPPED	Streaming thread stopped due to an error.
HACKRF_ERROR_STREAMING_EXIT_CALLED	Streaming thread exited (normally)
HACKRF_ERROR_USB_API_VERSION	The installed firmware does not support this function.
HACKRF_ERROR_NOT_LAST_DEVICE	Can not exit library as one or more HackRFs still in use.
HACKRF_ERROR_OTHER	Unspecified error.

3.1.8 Function Documentation

3.1.8.1 hackrf_error_name()

```
const char * hackrf_error_name (
            enum hackrf_error errcode )
```

Parameters

errcode	enum to convert

Returns

human-readable name of error

3.1.8.2 hackrf_exit()

```
int hackrf_exit ( )
```

Should be called before exit. No other libhackrf functions should be called after it. Can be safely called multiple times.

Returns

HACKRF_SUCCESS on success or HACKRF_ERROR_NOT_LAST_DEVICE if not all devices were closed properly.

3.1.8.3 hackrf_get_transfer_buffer_size()

```
size_t hackrf_get_transfer_buffer_size (
            hackrf_device * device )
```

Parameters

in	*device*	unused

Returns

size in bytes

3.1.8.4 hackrf_get_transfer_queue_depth()

```
uint32_t hackrf_get_transfer_queue_depth (
            hackrf_device * device )
```

Parameters

in	*device*	unused

Returns

number of buffers

3.1.8.5 hackrf_init()

```
int hackrf_init ( )
```

Should be called before any other libhackrf function. Initializes libusb. Can be safely called multiple times.

Returns

HACKRF_SUCCESS on success or HACKRF_ERROR_LIBUSB

3.1.8.6 hackrf_library_release()

```
const char * hackrf_library_release ( )
```

Can be called before hackrf_init

Returns

library version as a human-readable string.

3.1.8.7 hackrf_library_version()

```
const char * hackrf_library_version ( )
```

Can be called before hackrf_init

Returns

library version as a human-readable string

3.2 Device listing, opening, closing and querying

Managing HackRF devices and querying information about them.

Data Structures

- struct read_partid_serialno_t

  MCU (LPC43xx) part ID and serial number.
- struct hackrf_bias_t_user_settting_req

  User settings for user-supplied bias tee defaults.
- struct hackrf_device_list_t

  List of connected HackRF devices.

Macros

- #define HACKRF_BOARD_REV_GSG (0x80)

  Made by GSG bit in hackrf_board_rev enum and in platform ID.
- #define HACKRF_PLATFORM_JAWBREAKER (1 << 0)

  JAWBREAKER platform bit in result of hackrf_supported_platform_read.
- #define HACKRF_PLATFORM_HACKRF1_OG (1 << 1)

  HACKRF ONE (pre r9) platform bit in result of hackrf_supported_platform_read.
- #define HACKRF_PLATFORM_RAD1O (1 << 2)

  RAD1O platform bit in result of hackrf_supported_platform_read.
- #define HACKRF_PLATFORM_HACKRF1_R9 (1 << 3)

  HACKRF ONE (r9 or later) platform bit in result of hackrf_supported_platform_read.
- #define BOARD_ID_HACKRF_ONE (BOARD_ID_HACKRF1_OG)

  These deprecated board ID names are provided for API compatibility.
- #define BOARD_ID_INVALID (BOARD_ID_UNDETECTED)

  These deprecated board ID names are provided for API compatibility.

Typedefs

- typedef struct hackrf_device hackrf_device

  Opaque struct for hackrf device info.

Enumerations

- enum hackrf_board_id {
  BOARD_ID_JELLYBEAN = 0 ,
  BOARD_ID_JAWBREAKER = 1 ,
  BOARD_ID_HACKRF1_OG = 2 ,
  BOARD_ID_RAD1O = 3 ,
  BOARD_ID_HACKRF1_R9 = 4 ,
  BOARD_ID_UNRECOGNIZED = 0xFE ,
  BOARD_ID_UNDETECTED = 0xFF }

  HACKRF board id enum.
- enum hackrf_board_rev {
  BOARD_REV_HACKRF1_OLD = 0 ,
  BOARD_REV_HACKRF1_R6 = 1 ,
  BOARD_REV_HACKRF1_R7 = 2 ,
  BOARD_REV_HACKRF1_R8 = 3 ,
  BOARD_REV_HACKRF1_R9 = 4 ,
  BOARD_REV_HACKRF1_R10 = 5 ,
  BOARD_REV_GSG_HACKRF1_R6 = 0x81 ,
  BOARD_REV_GSG_HACKRF1_R7 = 0x82 ,
  BOARD_REV_GSG_HACKRF1_R8 = 0x83 ,
  BOARD_REV_GSG_HACKRF1_R9 = 0x84 ,
  BOARD_REV_GSG_HACKRF1_R10 = 0x85 ,
  BOARD_REV_UNRECOGNIZED = 0xFE ,
  BOARD_REV_UNDETECTED = 0xFF }

  Board revision enum.
- enum hackrf_usb_board_id {
  USB_BOARD_ID_JAWBREAKER = 0x604B ,
  USB_BOARD_ID_HACKRF_ONE = 0x6089 ,
  USB_BOARD_ID_RAD1O = 0xCC15 ,
  USB_BOARD_ID_INVALID = 0xFFFF }

  USB board ID (product ID) enum.

Functions

- hackrf_device_list_t * hackrf_device_list ()

  List connected HackRF devices.
- int hackrf_device_list_open (hackrf_device_list_t *list, int idx, hackrf_device **device)

  Open a hackrf_device from a device list.
- void hackrf_device_list_free (hackrf_device_list_t *list)

  Free a previously allocated hackrf_device_list list.
- int hackrf_open (hackrf_device **device)

  Open first available HackRF device.
- int hackrf_open_by_serial (const char *const desired_serial_number, hackrf_device **device)

  Open HackRF device by serial number.
- int hackrf_close (hackrf_device *device)

  Close a previously opened device.
- int hackrf_board_id_read (hackrf_device *device, uint8_t *value)

  Read hackrf_board_id from a device.
- int hackrf_version_string_read (hackrf_device *device, char *version, uint8_t length)

Read HackRF firmware version as a string.

- int hackrf_usb_api_version_read (hackrf_device *device, uint16_t *version)

  Read HackRF USB API version.
- int hackrf_board_partid_serialno_read (hackrf_device *device, read_partid_serialno_t *read_partid_serialno)

  Read board part ID and serial number.
- const char * hackrf_board_id_name (enum hackrf_board_id board_id)

  Convert hackrf_board_id into human-readable string.
- uint32_t hackrf_board_id_platform (enum hackrf_board_id board_id)

  Lookup platform ID (HACKRF_PLATFORM_xxx) from board id (hackrf_board_id)
- const char * hackrf_usb_board_id_name (enum hackrf_usb_board_id usb_board_id)

  Convert hackrf_usb_board_id into human-readable string.
- int hackrf_reset (hackrf_device *device)

  Reset HackRF device.
- int hackrf_set_ui_enable (hackrf_device *device, const uint8_t value)

  Enable / disable UI display (RAD1O, PortaPack, etc.)
- int hackrf_board_rev_read (hackrf_device *device, uint8_t *value)

  Read board revision of device.
- const char * hackrf_board_rev_name (enum hackrf_board_rev board_rev)

  Convert board revision name.
- int hackrf_supported_platform_read (hackrf_device *device, uint32_t *value)

  Read supported platform of device.
- int hackrf_set_leds (hackrf_device *device, const uint8_t state)

  Turn on or off (override) the LEDs of the HackRF device.
- int hackrf_set_user_bias_t_opts (hackrf_device *device, hackrf_bias_t_user_settting_req *req)

  Configure bias tee behavior of the HackRF device when changing RF states.

3.2.1 Detailed Description

The libhackrf library interacts via HackRF hardware through a hackrf_device handle. This handle is opaque, meaning its fields are internal to the library and should not be accessed by user code. To use a device, it first needs to be opened, than it can be interacted with, and finally the device needs to be closed via hackrf_close.

3.2.2 Opening devices

3.2.2.1 Open first device

hackrf_open opens the first USB device (chosen by libusb). Useful if only one HackRF device is expected to be present.

3.2.2.2 Open by serial

hackrf_open_by_serial opens a device by a given serial (suffix). If no serial is specified it defaults to hackrf_open

3.2.2.3 Open by listing

All connected HackRF devices can be listed via hackrf_device_list. The list must be freed by hackrf_device_list_free.

This struct lists all devices and their serial numbers. Any one of them can be opened by hackrf_device_list_open. All the fields should be treated read-only!

3.2.3 Closing devices

If the device is not needed anymore, then it can be closed via hackrf_close. Closing a device terminates all ongoing transfers, and resets the device to IDLE mode.

3.2.4 Querying device information

3.2.4.1 Board ID

Board ID identifies the type of HackRF board connected. See the enum hackrf_board_id for possible values. The value can be read by hackrf_board_id_read and converted into a human-readable string using hackrf_board_id_name. When reading, the initial value of the enum should be BOARD_ID_UNDETECTED.

3.2.4.2 Version string

Version string identifies the firmware version on the board. It can be read with the function hackrf_version_string_read

3.2.4.3 USB API version

USB API version identifies the USB API supported by the device's firmware. It is coded as a xx.xx 16-bit value, and can be read by hackrf_usb_api_version_read

Example of reading firmware and USB API version (from hackrf_info.c):

```
result = hackrf_version_string_read(device, &version[0], 255);
if (result != HACKRF_SUCCESS) {
    fprintf(stderr,
        "hackrf_version_string_read() failed:  %s (%d)\n",
        hackrf_error_name(result),
        result);
    return EXIT_FAILURE;
}
result = hackrf_usb_api_version_read(device, &usb_version);
if (result != HACKRF_SUCCESS) {
    fprintf(stderr,
        "hackrf_usb_api_version_read() failed:  %s (%d)\n",
        hackrf_error_name(result),
        result);
    return EXIT_FAILURE;
}
printf("Firmware Version:  %s (API:%x.%02x)\n",
    version,
    (usb_version » 8) & 0xFF,
    usb_version & 0xFF);
```

3.2.4.4 Part ID and serial number

The part ID and serial number of the MCU. Read via hackrf_board_partid_serialno_read. See the documentation of the MCU for details.

3.2.4.5 Board revision

Board revision identifies revision of the HackRF board inside a device. Read via hackrf_board_rev_read and converted into a human-readable string via hackrf_board_rev_name. See hackrf_board_rev for possible values. When reading, the value should be initialized with BOARD_REV_UNDETECTED

3.2.4.6 Supported platform

Identifies the platform supported by the firmare of the HackRF device. Read via hackrf_supported_platform_read. Returns a bitfield. Can identify bad firmware version on device.

3.2.5 Macro Definition Documentation

3.2.5.1 BOARD_ID_HACKRF_ONE

```
#define BOARD_ID_HACKRF_ONE (BOARD_ID_HACKRF1_OG)
```

3.2.5.2 BOARD_ID_INVALID

```
#define BOARD_ID_INVALID (BOARD_ID_UNDETECTED)
```

3.2.5.3 HACKRF_BOARD_REV_GSG

```
#define HACKRF_BOARD_REV_GSG (0x80)
```

3.2.5.4 HACKRF_PLATFORM_HACKRF1_OG

```
#define HACKRF_PLATFORM_HACKRF1_OG (1 << 1)
```

3.2.5.5 HACKRF_PLATFORM_HACKRF1_R9

```
#define HACKRF_PLATFORM_HACKRF1_R9 (1 << 3)
```

3.2.5.6 HACKRF_PLATFORM_JAWBREAKER

```
#define HACKRF_PLATFORM_JAWBREAKER (1 << 0)
```

3.2.5.7 HACKRF_PLATFORM_RAD1O

```
#define HACKRF_PLATFORM_RAD1O (1 << 2)
```

3.2.6 Typedef Documentation

3.2.6.1 hackrf_device

```
typedef struct hackrf_device hackrf_device
```

Object can be created via hackrf_open, hackrf_device_list_open or hackrf_open_by_serial and be destroyed via hackrf_close

3.2.7 Enumeration Type Documentation

3.2.7.1 hackrf_board_id

```
enum hackrf_board_id
```

Returned by hackrf_board_id_read and can be converted to a human-readable string using hackrf_board_id_name

Enumerator

BOARD_ID_JELLYBEAN	Jellybean (pre-production revision, not supported)
BOARD_ID_JAWBREAKER	Jawbreaker (beta platform, 10-6000MHz, no bias-tee)
BOARD_ID_HACKRF1_OG	HackRF One (prior to rev 9, same limits: 1-6000MHz, 20MSPS, bias-tee)
BOARD_ID_RAD1O	RAD1O (Chaos Computer Club special edition with LCD & other features. 50M-4000MHz, 20MSPS, no bias-tee)
BOARD_ID_HACKRF1_R9	HackRF One (rev. 9 & later. 1-6000MHz, 20MSPS, bias-tee)
BOARD_ID_UNRECOGNIZED	Unknown board (failed detection)

3.2.7.2 hackrf_board_rev

`enum hackrf_board_rev`

Returned by hackrf_board_rev_read and can be converted into human-readable name by hackrf_board_rev_name. MSB (`board_rev & HACKRF_BOARD_REV_GSG`) should signify if the board was built by GSG or not. hackrf_board_rev_name ignores this information.

Enumerator

BOARD_REV_HACKRF1_OLD	Older than rev6.
BOARD_REV_HACKRF1_R6	board revision 6, generic
BOARD_REV_HACKRF1_R7	board revision 7, generic
BOARD_REV_HACKRF1_R8	board revision 8, generic
BOARD_REV_HACKRF1_R9	board revision 9, generic
BOARD_REV_HACKRF1_R10	board revision 10, generic
BOARD_REV_GSG_HACKRF1_R6	board revision 6, made by GSG
BOARD_REV_GSG_HACKRF1_R7	board revision 7, made by GSG
BOARD_REV_GSG_HACKRF1_R8	board revision 8, made by GSG
BOARD_REV_GSG_HACKRF1_R9	board revision 9, made by GSG
BOARD_REV_GSG_HACKRF1_R10	board revision 10, made by GSG
BOARD_REV_UNRECOGNIZED	unknown board revision (detection failed)
BOARD_REV_UNDETECTED	unknown board revision (detection not yet attempted)

3.2.7.3 hackrf_usb_board_id

`enum hackrf_usb_board_id`

Contains USB-IF product id (field `idProduct` in `libusb_device_descriptor`). Can be used to identify general type of hardware. Only used in hackrf_device_list::usb_board_ids field of hackrf_device_list, and can be converted into human-readable string via hackrf_usb_board_id_name.

Enumerator

USB_BOARD_ID_JAWBREAKER	Jawbreaker (beta platform) USB product id.
USB_BOARD_ID_HACKRF_ONE	HackRF One USB product id.
USB_BOARD_ID_RAD1O	RAD1O (custom version) USB product id.
USB_BOARD_ID_INVALID	Invalid / unknown USB product id.

3.2.8 Function Documentation

3.2.8.1 hackrf_board_id_name()

```
const char * hackrf_board_id_name (
            enum hackrf_board_id board_id )
```

Parameters

board_id	enum to convert

Returns

human-readable name of board id

3.2.8.2 hackrf_board_id_platform()

```
uint32_t hackrf_board_id_platform (
            enum hackrf_board_id board_id )
```

Parameters

board_id	hackrf_board_id enum variant to convert

Returns

HACKRF_PLATFORM_JAWBREAKER, HACKRF_PLATFORM_HACKRF1_OG, HACKRF_PLATFORM_RAD1O, HACKRF_PLATFORM_HACKRF1_R9 or 0

3.2.8.3 hackrf_board_id_read()

```
int hackrf_board_id_read (
            hackrf_device * device,
            uint8_t * value )
```

The result can be converted into a human-readable string via hackrf_board_id_name

Parameters

in	*device*	device to query
out	*value*	hackrf_board_id enum value

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.4 hackrf_board_partid_serialno_read()

```
int hackrf_board_partid_serialno_read (
            hackrf_device * device,
            read_partid_serialno_t * read_partid_serialno )
```

Read MCU part id and serial number. See the documentation of the MCU for details!

Parameters

in	*device*	device to query
out	*read_partid_serialno*	result of query

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.5 hackrf_board_rev_name()

```
const char * hackrf_board_rev_name (
            enum hackrf_board_rev board_rev )
```

Parameters

board_rev	board revision enum from hackrf_board_rev_read

Returns

human-readable name of board revision. Discards GSG bit.

3.2.8.6 hackrf_board_rev_read()

```
int hackrf_board_rev_read (
            hackrf_device * device,
            uint8_t * value )
```

Requires USB API version 0x0106 or above!

Parameters

in	*device*	device to read board revision from
out	*value*	revision enum, will become one of hackrf_board_rev. Should be initialized with BOARD_REV_UNDETECTED

Returns

HACKRF_SUCCESS on success or HACKRF_ERROR_LIBUSB

3.2.8.7 hackrf_close()

```
int hackrf_close (
            hackrf_device * device )
```

Parameters

in	*device*	device to close

Returns

HACKRF_SUCCESS on success or variant of hackrf_error

3.2.8.8 hackrf_device_list()

```
hackrf_device_list_t * hackrf_device_list ( )
```

Returns

list of connected devices. The list should be freed with hackrf_device_list_free

3.2.8.9 hackrf_device_list_free()

```
void hackrf_device_list_free (
            hackrf_device_list_t * list )
```

Parameters

in	*list*	list to free

3.2.8.10 hackrf_device_list_open()

```
int hackrf_device_list_open (
            hackrf_device_list_t * list,
            int idx,
            hackrf_device ** device )
```

Parameters

in	*list*	device list to open device from
in	*idx*	index of the device to open
out	*device*	device handle to open

Returns

HACKRF_SUCCESS on success, HACKRF_ERROR_INVALID_PARAM on invalid parameters or other hackrf_error variant

3.2.8.11 hackrf_open()

```
int hackrf_open (
            hackrf_device ** device )
```

Parameters

out	*device*	device handle

Returns

HACKRF_SUCCESS on success, HACKRF_ERROR_INVALID_PARAM if `device` is NULL, HACKRF_ERROR_NOT_FOUND if no HackRF devices are found or other hackrf_error variant

3.2.8.12 hackrf_open_by_serial()

```
int hackrf_open_by_serial (
            const char *const desired_serial_number,
            hackrf_device ** device )
```

Parameters

in	*desired_serial_number*	serial number of device to open. If NULL then default to first device found.
out	*device*	device handle

Returns

HACKRF_SUCCESS on success, HACKRF_ERROR_INVALID_PARAM if `device` is NULL, HACKRF_ERROR_NOT_FOUND if no HackRF devices are found or other hackrf_error variant

3.2.8.13 hackrf_reset()

```
int hackrf_reset (
            hackrf_device * device )
```

Requires USB API version 0x0102 or above!

Parameters

device	device to reset

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.14 hackrf_set_leds()

```
int hackrf_set_leds (
            hackrf_device * device,
            const uint8_t state )
```

This function can turn on or off the LEDs of the device. There are 3 controllable LEDs on the HackRF one: USB, RX and TX. On the Rad1o, there are 4 LEDs. Each LED can be set individually, but the setting might get overridden by other functions.

The LEDs can be set via specifying them as bits of a 8 bit number `state`, bit 0 representing the first (USB on the HackRF One) and bit 3 or 4 representing the last LED. The upper 4 or 5 bits are unused. For example, binary value 0bxxxxx101 turns on the USB and TX LEDs on the HackRF One.

Requires USB API version 0x0107 or above!

Parameters

device	device to query
state	LED states as a bitfield

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.15 hackrf_set_ui_enable()

```
int hackrf_set_ui_enable (
            hackrf_device * device,
            const uint8_t value )
```

Enable or disable the display on display-enabled devices (Rad1o, PortaPack)

Requires USB API version 0x0104 or above!

Parameters

device	device to enable/disable UI on
value	Enable UI. Must be 1 or 0

Returns

HACKRF_SUCCESS on success or HACKRF_ERROR_LIBUSB on usb error

3.2.8.16 hackrf_set_user_bias_t_opts()

```
int hackrf_set_user_bias_t_opts (
            hackrf_device * device,
            hackrf_bias_t_user_settting_req * req )
```

This function allows the user to configure bias tee behavior so that it can be turned on or off automatically by the HackRF when entering the RX, TX, or OFF state. By default, the HackRF switches off the bias tee when the RF path switches to OFF mode.

The bias tee configuration is specified via a bitfield:

0000000TmmRmmOmm

Where setting T/R/O bits indicates that the TX/RX/Off behavior should be set to mode 'mm', 0=don't modify

mm specifies the bias tee mode:

00 - do nothing 01 - reserved, do not use 10 - disable bias tee 11 - enable bias tee

Parameters

device	device to configure
state	Bias tee states, as a bitfield

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.17 hackrf_supported_platform_read()

```
int hackrf_supported_platform_read (
            hackrf_device * device,
            uint32_t * value )
```

Returns a combination of HACKRF_PLATFORM_JAWBREAKER | HACKRF_PLATFORM_HACKRF1_OG | HACKRF_PLATFORM_RAD1O | HACKRF_PLATFORM_HACKRF1_R9

Requires USB API version 0x0106 or above!

Parameters

in	*device*	device to query
out	*value*	supported platform bitfield

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.18 hackrf_usb_api_version_read()

```
int hackrf_usb_api_version_read (
            hackrf_device * device,
            uint16_t * version )
```

Read version as MM.mm 16-bit value, where MM is the major and mm is the minor version, encoded as the hex digits of the 16-bit number.

Example code from `hackrf_info.c` displaying the result:

```
printf("Firmware Version:  %s (API:%x.%02x)\n",
           version,
           (usb_version » 8) & 0xFF,
           usb_version & 0xFF);
```

Parameters

in	*device*	device to query
out	*version*	USB API version

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.2.8.19 hackrf_usb_board_id_name()

```
const char * hackrf_usb_board_id_name (
            enum hackrf_usb_board_id usb_board_id )
```

Parameters

usb_board_id	enum to convert

Returns

human-readable name of board id

3.2.8.20 hackrf_version_string_read()

```
int hackrf_version_string_read (
            hackrf_device * device,
            char * version,
            uint8_t length )
```

Parameters

in	*device*	device to query
out	*version*	version string
in	*length*	length of allocated string **without null byte** (so set it to `length(arr)-1`)

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3 Configuration of the RF hardware

Configuring gain, sample rate, filter bandwidth, etc.

Enumerations

- enum rf_path_filter {
  RF_PATH_FILTER_BYPASS = 0 ,
  RF_PATH_FILTER_LOW_PASS = 1 ,
  RF_PATH_FILTER_HIGH_PASS = 2 }

  RF filter path setting enum.

Functions

- int hackrf_set_baseband_filter_bandwidth (hackrf_device *device, const uint32_t bandwidth_hz)

  Set baseband filter bandwidth.
- int hackrf_set_freq (hackrf_device *device, const uint64_t freq_hz)

  Set the center frequency.
- int hackrf_set_freq_explicit (hackrf_device *device, const uint64_t if_freq_hz, const uint64_t lo_freq_hz, const enum rf_path_filter path)

  Set the center frequency via explicit tuning.
- int hackrf_set_sample_rate_manual (hackrf_device *device, const uint32_t freq_hz, const uint32_t divider)

  Set sample rate explicitly.
- int hackrf_set_sample_rate (hackrf_device *device, const double freq_hz)

  Set sample rate.
- int hackrf_set_amp_enable (hackrf_device *device, const uint8_t value)

  Enable/disable 14dB RF amplifier.
- int hackrf_set_lna_gain (hackrf_device *device, uint32_t value)

  Set LNA gain.
- int hackrf_set_vga_gain (hackrf_device *device, uint32_t value)

  Set baseband RX gain of the MAX2837 transceier IC ("BB" or "VGA" gain setting) in decibels.
- int hackrf_set_txvga_gain (hackrf_device *device, uint32_t value)

  Set RF TX gain of the MAX2837 transceiver IC ("IF" or "VGA" gain setting) in decibels.
- int hackrf_set_antenna_enable (hackrf_device *device, const uint8_t value)

  Enable / disable bias-tee (antenna port power)
- const char * hackrf_filter_path_name (const enum rf_path_filter path)

  Convert rf_path_filter into human-readable string.
- uint32_t hackrf_compute_baseband_filter_bw_round_down_lt (const uint32_t bandwidth_hz)

  Compute nearest valid baseband filter bandwidth lower than a specified value.
- uint32_t hackrf_compute_baseband_filter_bw (const uint32_t bandwidth_hz)

  Compute nearest valid baseband filter bandwidth to specified value.
- int hackrf_set_clkout_enable (hackrf_device *device, const uint8_t value)

  Enable / disable CLKOUT.
- int hackrf_get_clkin_status (hackrf_device *device, uint8_t *status)

  Get CLKIN status.

3.3.1 Detailed Description

3.3.2 Amplifiers and gains

There are 5 different amplifiers in the HackRF One. Most of them have variable gain, but some of them can be either enabled / disabled. Please note that most of the gain settings are not precise, and they depend on the used frequency as well.

(image taken from https://hackrf.readthedocs.io/en/latest/hardware_components.html)

3.3.2.1 RX path

- baseband gain in the MAX2837 ("BB" or "VGA") - 0-62dB in 2dB steps, configurable via the hackrf_set_vga_gain function
- RX IF gain in the MAX2837 ("IF") - 0-40dB with 8dB steps, configurabe via the hackrf_set_lna_gain function
- RX RF amplifier near the antenna port ("RF") - 0 or $\sim$11dB, either enabled or disabled via the hackrf_set_amp_enable (same function is used for enabling/disabling the TX RF amp in TX mode)

3.3.2.2 TX path

- TX IF gain in the MAX2837 ("IF" or "VGA") - 0-47dB in 1dB steps, configurable via hackrf_set_txvga_gain
- TX RF amplifier near the antenna port ("RF") - 0 or $\sim$11dB, either enabled or disabled via the hackrf_set_amp_enable (same function is used for enabling/disabling the RX RF amp in RX mode)

3.3.3 Tuning

The HackRF One can tune to nearly any frequency between 1-6000MHz (and the theoretical limit is even a bit higher). This is achieved via up/downconverting the RF section of the MAX2837 transceiver IC with the RFFC5072 mixer/synthesizer's local oscillator. The mixer produces the sum and difference frequencies of the IF and LO frequencies, and a LPF or HPF filter can be used to select one of the resulting frequencies. There is also the possibility to bypass the filter and use the IF as-is. The IF and LO frequencies can be programmed independently, and the behaviour is selectable. See the function hackrf_set_freq_explicit for more details on it.

There is also the convenience function hackrf_set_freq that automatically select suitable LO and IF frequencies and RF path for a desired frequency. It should be used in most cases.

3.3.4 Filtering

The MAX2837 has an internal selectable baseband filter for both RX and TX. Its width can be set via hackrf_set_baseband_filter_bandwidth, but only some values are valid. Valid values can be acquired via the functions hackrf_compute_baseband_filter_bw_round_down_lt and hackrf_compute_baseband_filter_bw.

NOTE in order to avoid aliasing, the bandwidth should not exceed the sample rate. As a sensible default, the firmware auto-sets the baseband filter bandwidth to a value $\leq 0.75 \cdot F_s$ whenever the sample rate is changed, thus setting a custom value should be done after setting the samplerate.

3.3.5 Sample rate

The sample rate of the ADC/DAC can be set between 2-20MHz via hackrf_set_sample_rate or hackrf_set_sample_rate_manual. This also automatially adjusts the baseband filter bandwidth to a suitable value.

3.3.6 Clocking

The HackRF one has external clock input and clock output connectors for 10MHz 3.3V clock signals. It automatically switches to the external clock if it's detected, and its status is readable with hackrf_get_clkin_status. The external clock can be enabled by the hackrf_set_clkout_enable function.

3.3.7 Bias-tee

The HackRF one has a built in bias-tee (also called (antenna) port power in some of the documentation) capable of delivering 50mA@3V3 for powering small powered antennas or amplifiers. It can be enabled via the hackrf_set_antenna_enable function. Please note that when the device is returning to IDLE mode, the firmware automatically disables this feature. This means it can't be enabled permanently like with the RTL-SDR, and all software using the HackRF must enable this separatlely.

3.3.8 Enumeration Type Documentation

3.3.8.1 rf_path_filter

```
enum rf_path_filter
```

Used only when performing explicit tuning using hackrf_set_freq_explicit, or can be converted into a human readable string using hackrf_filter_path_name. This can select the image rejection filter (U3, U8 or none) to use - using switches U5, U6, U9 and U11. When no filter is selected, the mixer itself is bypassed.

Enumerator

RF_PATH_FILTER_BYPASS	No filter is selected, **the mixer is bypassed**, $f_{center} = f_{IF}$.
RF_PATH_FILTER_LOW_PASS	LPF is selected, $f_{center} = f_{IF} - f_{LO}$.
RF_PATH_FILTER_HIGH_PASS	HPF is selected, $f_{center} = f_{IF} + f_{LO}$.

3.3.9 Function Documentation

3.3.9.1 hackrf_compute_baseband_filter_bw()

```
uint32_t hackrf_compute_baseband_filter_bw (
            const uint32_t bandwidth_hz )
```

The result can be used via hackrf_set_baseband_filter_bandwidth

Parameters

bandwidth_hz	desired filter bandwidth in Hz

Returns

nearest valid filter bandwidth in Hz

3.3.9.2 hackrf_compute_baseband_filter_bw_round_down_lt()

```
uint32_t hackrf_compute_baseband_filter_bw_round_down_lt (
            const uint32_t bandwidth_hz )
```

The result can be used via hackrf_set_baseband_filter_bandwidth

Parameters

bandwidth_hz	desired filter bandwidth in Hz

Returns

the highest valid filter bandwidth lower than `bandwidth_hz` in Hz

3.3.9.3 hackrf_filter_path_name()

```
const char * hackrf_filter_path_name (
            const enum rf_path_filter path )
```

Parameters

path	enum to convert

Returns

human-readable name of filter path

3.3.9.4 hackrf_get_clkin_status()

```
int hackrf_get_clkin_status (
            hackrf_device * device,
            uint8_t * status )
```

Check if an external clock signal is detected on the CLKIN port.

Requires USB API version 0x0106 or above!

Parameters

in	*device*	device to read status from
out	*status*	external clock detected (0/1)

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.5 hackrf_set_amp_enable()

```
int hackrf_set_amp_enable (
            hackrf_device * device,
            const uint8_t value )
```

Enable / disable the ∼11dB RF RX/TX amplifiers U13/U25 via controlling switches U9 and U14.

Parameters

device	device to configure
value	enable (1) or disable (0) amplifier

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.6 hackrf_set_antenna_enable()

```
int hackrf_set_antenna_enable (
            hackrf_device * device,
            const uint8_t value )
```

Enable or disable the **3.3V (max 50mA)** bias-tee (antenna port power). Defaults to disabled.

NOTE: the firmware auto-disables this after returning to IDLE mode, so a perma-set is not possible, which means all software supporting HackRF devices must support enabling bias-tee, as setting it externally is not possible like it is with RTL-SDR for example.

Parameters

device	device to configure
value	enable (1) or disable (0) bias-tee

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.7 hackrf_set_baseband_filter_bandwidth()

```
int hackrf_set_baseband_filter_bandwidth (
            hackrf_device * device,
            const uint32_t bandwidth_hz )
```

Possible values: 1.75, 2.5, 3.5, 5, 5.5, 6, 7, 8, 9, 10, 12, 14, 15, 20, 24, 28MHz, default $\leq 0.75 \cdot F_s$ The functions hackrf_compute_baseband_filter_bw and hackrf_compute_baseband_filter_bw_round_down_lt can be used to get a valid value nearest to a given value.

Setting the sample rate causes the filter bandwidth to be (re)set to its default $\leq 0.75 \cdot F_s$ value, so setting sample rate should be done before setting filter bandwidth.

Parameters

device	device to configure
bandwidth_hz	baseband filter bandwidth in Hz

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.8 hackrf_set_clkout_enable()

```
int hackrf_set_clkout_enable (
            hackrf_device * device,
            const uint8_t value )
```

Requires USB API version 0x0103 or above!

Parameters

device	device to configure
value	clock output enabled (0/1)

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.9 hackrf_set_freq()

```
int hackrf_set_freq (
            hackrf_device * device,
            const uint64_t freq_hz )
```

Simple (auto) tuning via specifying a center frequency in Hz

This setting is not exact and depends on the PLL settings. Exact resolution is not determined, but the actual tuned frequency will be quariable in the future.

Parameters

device	device to tune
freq_hz	center frequency in Hz. Defaults to 900MHz. Should be in range 1-6000MHz, but 0-7250MHz is possible. The resolution is ~50Hz, I could not find the exact number.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.10 hackrf_set_freq_explicit()

```
int hackrf_set_freq_explicit (
            hackrf_device * device,
```

```
            const uint64_t if_freq_hz,
            const uint64_t lo_freq_hz,
            const enum rf_path_filter path )
```

Center frequency is set to $f_{center} = f_{IF} + k \cdot f_{LO}$ where $k \in \{-1; 0; 1\}$, depending on the value of `path`. See the documentation of rf_path_filter for details

Parameters

device	device to tune
if_freq_hz	tuning frequency of the MAX2837 transceiver IC in Hz. Must be in the range of 2150-2750MHz
lo_freq_hz	tuning frequency of the RFFC5072 mixer/synthesizer IC in Hz. Must be in the range 84.375-5400MHz, defaults to 1000MHz. No effect if `path` is set to RF_PATH_FILTER_BYPASS
path	filter path for mixer. See the documentation for rf_path_filter for details

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.11 hackrf_set_lna_gain()

```
int hackrf_set_lna_gain (
            hackrf_device * device,
            uint32_t value )
```

Set the RF RX gain of the MAX2837 transceiver IC ("IF" gain setting) in decibels. Must be in range 0-40dB, with 8dB steps.

Parameters

device	device to configure
value	RX IF gain value in dB

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.12 hackrf_set_sample_rate()

```
int hackrf_set_sample_rate (
            hackrf_device * device,
            const double freq_hz )
```

Sample rate should be in the range 2-20MHz, with the default being 10MHz. Lower & higher values are technically possible, but the performance is not guaranteed. This function also sets the baseband filter bandwidth to a value $\leq 0.75 \cdot F_s$, so any calls to hackrf_set_baseband_filter_bandwidth should only be made after this.

Parameters

device	device to configure
freq_hz	sample rate frequency in Hz. Should be in the range 2-20MHz

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.13 hackrf_set_sample_rate_manual()

```
int hackrf_set_sample_rate_manual (
            hackrf_device * device,
            const uint32_t freq_hz,
            const uint32_t divider )
```

Sample rate should be in the range 2-20MHz, with the default being 10MHz. Lower & higher values are technically possible, but the performance is not guaranteed.

This function sets the sample rate by specifying a clock frequency in Hz and a divider, so the resulting sample rate will be `freq_hz` / `divider`. This function also sets the baseband filter bandwidth to a value $\leq 0.75 \cdot F_s$, so any calls to hackrf_set_baseband_filter_bandwidth should only be made after this.

Parameters

device	device to configure
freq_hz	sample rate base frequency in Hz
divider	frequency divider. Must be in the range 1-31

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.14 hackrf_set_txvga_gain()

```
int hackrf_set_txvga_gain (
            hackrf_device * device,
            uint32_t value )
```

Must be in range 0-47dB in 1dB steps.

Parameters

device	device to configure
value	TX IF gain value in dB

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.3.9.15 hackrf_set_vga_gain()

```
int hackrf_set_vga_gain (
            hackrf_device * device,
            uint32_t value )
```

Must be in range 0-62dB with 2dB steps.

Parameters

device	device to configure
value	RX BB gain value in dB

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4 Transmit & receive operation

RX and TX, callbacks.

Data Structures

- struct hackrf_transfer

  USB transfer information passed to RX or TX callback.

Macros

- #define SAMPLES_PER_BLOCK 8192

  Number of samples per tuning when sweeping.
- #define BYTES_PER_BLOCK 16384

  Number of bytes per tuning for sweeping.
- #define MAX_SWEEP_RANGES 10

  Maximum number of sweep ranges to be specified for hackrf_init_sweep.

Typedefs

- typedef int(∗ hackrf_sample_block_cb_fn) (hackrf_transfer ∗transfer)

  Sample block callback, used in RX and TX (set via hackrf_start_rx, hackrf_start_rx_sweep and hackrf_start_tx).
- typedef void(∗ hackrf_tx_block_complete_cb_fn) (hackrf_transfer ∗transfer, int)

  Block complete callback.
- typedef void(∗ hackrf_flush_cb_fn) (void ∗flush_ctx, int)

  Flush (end of transmission) callback.

Enumerations

- enum sweep_style {
  LINEAR = 0 ,
  INTERLEAVED = 1 }

  sweep mode enum

Functions

- int hackrf_start_rx (hackrf_device ∗device, hackrf_sample_block_cb_fn callback, void ∗rx_ctx)

  Start receiving.
- int hackrf_stop_rx (hackrf_device ∗device)

  Stop receiving.
- int hackrf_start_tx (hackrf_device ∗device, hackrf_sample_block_cb_fn callback, void ∗tx_ctx)

  Start transmitting.
- int hackrf_set_tx_block_complete_callback (hackrf_device ∗device, hackrf_tx_block_complete_cb_fn callback)

  Setup callback to be called when an USB transfer is completed.
- int hackrf_enable_tx_flush (hackrf_device ∗device, hackrf_flush_cb_fn callback, void ∗flush_ctx)

  Setup flush (end-of-transmission) callback.
- int hackrf_stop_tx (hackrf_device ∗device)

  Stop transmission.
- int hackrf_set_tx_underrun_limit (hackrf_device ∗device, uint32_t value)

  Set transmit underrun limit.
- int hackrf_set_rx_overrun_limit (hackrf_device ∗device, uint32_t value)

  Set receive overrun limit.
- int hackrf_is_streaming (hackrf_device ∗device)

  Query device streaming status.
- int hackrf_set_hw_sync_mode (hackrf_device ∗device, const uint8_t value)

  Set hardware sync mode (hardware triggering)
- int hackrf_init_sweep (hackrf_device ∗device, const uint16_t ∗frequency_list, const int num_ranges, const uint32_t num_bytes, const uint32_t step_width, const uint32_t offset, const enum sweep_style style)

  Initialize sweep mode.
- int hackrf_start_rx_sweep (hackrf_device ∗device, hackrf_sample_block_cb_fn callback, void ∗rx_ctx)

  Start RX sweep.

3.4.1 Detailed Description

3.4.1.1 Streaming

There are 3 different streaming modes supported by HackRF:

- transmitting (TX)
- receiving (RX)
- swept receiving (SWEEP)

Each mode needs to be initialized before use, then the mode needs to be entered with the `hackrf_start_*` function. Data transfer happens through callbacks.

There are 3 types of callbacks in the library:

- transfer callback
- flush callback
- block complete callback

Steps for starting an RX or TX operation:

- initialize libhackrf
- open device
- setup device (frequency, samplerate, gain, etc)
- setup callbacks, start operation (`hackrf_start_*`)
- the main program should go to sleep
- when done, the transfer callback should return non-zero value, and signal the main thread to stop
- stop operation via `hackrf_stop_*`
- close device, exit library, etc.

Data is transfered through the USB connection via setting up multiple async libusb transfers (hackrf_get_transfer_queue_depth). In TX mode, the transfers needs to be filled before submitting, and in RX mode, they need to be read out when they are done. This is done using the transfer callback - it receives a hackrf_transfer object and needs to transfer the data to/from it. As it's needed for all operations, this gets called whenever we need to move data, so every time a transfer is finished (and before the first transfer in TX mode). There's a "transfer complete callback" that only gets called when a transfer is completed. It does not need to do anything special tho, and is optional.

Streaming can be stopped via returning a non-zero value from the transfer callback, but that does NOT reset the device to IDLE mode, it only stops data transfers. In TX mode, when this happens, and the transmitter runs out of data to transmit, it will start transmitting all 0 values (but in older firmware versions, it started repeating the last buffer). To actually stop the operation, a call to `hackrf_stop_*` is needed. Since the callback operate in an async libusb context, such a call can't be made from there, only from the main thread, so it must be signaled through some means (for example, a global variable, or better, a `pthread_cond`) to stop. In RX mode, this signaling can be done from the transfer callback, but in TX mode, we must make sure that we only stop the operation when the last transfer is completed and the device transmitted it, or we might lose it. For this reason, the third **flush callback** exists, that gets called when this happens. It is adivsed to only signal the main thread to stop from this callback.

The function hackrf_is_streaming can be used to check if the device is streaming or not.

3.4.1.1.1 Transfer callback Set when starting an operation with hackrf_start_tx, hackrf_start_rx or hackrf_start_rx_sweep. This callback supplies / receives data. This function takes a hackrf_transfer struct as a parameter, and fill/read data to/from its buffer. This function runs in an async libusb context, meaning it should not iteract with the libhackrf library in other ways. The callback can return a boolean value, if its return value is non-zero then it won't be called again, meaning that no future transfers will take place, and (in TX case) the flush callback will be called shortly.

3.4.1.1.2 Block complete callback This callback is optional, and only applicable in TX mode. It gets called whenever a data transfer is finished, and can read the data. It needs to do nothing at all. This callback can be set using hackrf_set_tx_block_complete_callback

3.4.1.1.3 Flush callback This callback is optional, and only applicable in TX mode. It get called when the last transfer is completed, and it's advisable to only stop streaming via this callback. This callback can be set using hackrf_enable_tx_flush

3.4.1.1.4 Example TX code utilizing the transfer and flush callbacks.

```
// Transmit a 440Hz triangle wave through FM (144.5MHz) using the libhackrf API
// Copyright (c) 2022 László Baráth "Uncle Dino" HA7DN <https://github.com/Sasszem>
#include <libhackrf/hackrf.h>
#include <math.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <complex.h>
#include <stdint.h>
const double f_mod = 440;
const uint64_t sample_rate = 10000000;
double triangle() {
    // Generate an f_mod frequency triangle wave in the -1 - 1 region
    // each call to this function generates a single sample
    static double state;
    static uint64_t samples_generated;

    const uint64_t period_in_samples = sample_rate / f_mod;
    const double step = 4.0 / period_in_samples; // we need to go from -1 to 1 in half the period
    if (samples_generated < period_in_samples / 2 )
        state += step;
    else
        state -= step;
    // this way we don't need to modulo it
    if (samples_generated ++ == period_in_samples)
        samples_generated = 0;
    return state - 1.0;
}
volatile double complex phasor = 1.0;
int xfered_samples = 0;
int samples_to_xfer = 5*sample_rate;
volatile int should_stop = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int transfer_callback(hackrf_transfer *transfer) {
    int8_t *signed_buffer = (int8_t*)transfer->buffer;
    for (int i = 0; i<transfer->buffer_length; i+=2) {
        phasor *= cexp(I*6.28*3000 / sample_rate*triangle());
        // any IQ samples can be written here, now I'm doing FM modulation with a triangle wave
        signed_buffer[i] = 128 * creal(phasor);
        signed_buffer[i+1] = 128 * cimag(phasor);
    }
    transfer->valid_length = transfer->buffer_length;
    xfered_samples += transfer->buffer_length;
    if (xfered_samples >= samples_to_xfer) {
        return 1;
    }
    return 0;
}
void flush_callback(hackrf_transfer *transfer) {
    pthread_mutex_lock(&mutex);
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
int main() {
```

```
    hackrf_init();
    hackrf_device *device = NULL;
    hackrf_open(&device);

    hackrf_set_freq(device, 144500000);
    hackrf_set_sample_rate(device, 10000000);
    hackrf_set_amp_enable(device, 1);
    hackrf_set_txvga_gain(device, 20);
    // hackrf_set_tx_underrun_limit(device, 100000); // new-ish library function, not always available
    hackrf_enable_tx_flush(device, flush_callback, NULL);
    hackrf_start_tx(device, transfer_callback, NULL);
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex); // wait fo transfer to complete

    hackrf_stop_tx(device);
    hackrf_close(device);
    hackrf_exit();
    return 0;
}
```

This code can be compiled using `gcc -o triangle triangle.c -lm -lhackrf`. It generates and transmits a 440Hz triangle wave using FM modulation on the 2m HAM band (**check your local laws and regulations on transmitting and only transmit on bands you have license to!**).

For a more complete example, including error handling and more settings, see `hackrf_transfer.c`

3.4.1.2 Underrun and overrun

Underrun/overrun detection can be enabled using hackrf_set_tx_underrun_limit or hackrf_set_rx_overrun_limit limit. This causes the HackRF to stop operation if more than the specified amount of samples get lost, for example in case of your program crashing, USB connection faliure, etc.

3.4.1.3 Sweeping

Sweeping mode is kind of special. In this mode, the device can be programmed to a list of frequencies to tune on, record set amount of samples and then tune to the next frequency and repeat. It can be setup via hackrf_init_sweep and started with hackrf_start_rx_sweep. In this mode, **the callback does not receive raw samples**, but blocks of samples prefixed with a frequency header specifying the tuned frequency.

See `hackrf_sweep.c` for a full example, and especialy `the start of the RX callback` for parsing the frequency header.

3.4.1.4 HW sync mode

hackrf_set_hw_sync_mode can be used to setup HW sync mode (`see the documentation on this mode`). This mode allows multiple HackRF Ones to synchronize operations, or one HackRF One to synchrnonize on an external trigger source.

3.4.2 Macro Definition Documentation

3.4.2.1 BYTES_PER_BLOCK

```
#define BYTES_PER_BLOCK 16384
```

3.4.2.2 MAX_SWEEP_RANGES

```
#define MAX_SWEEP_RANGES 10
```

3.4.2.3 SAMPLES_PER_BLOCK

```
#define SAMPLES_PER_BLOCK 8192
```

3.4.3 Typedef Documentation

3.4.3.1 hackrf_flush_cb_fn

```
typedef void(* hackrf_flush_cb_fn) (void *flush_ctx, int)
```

Will be called when the last samples are transmitted and stopping transmission will result in no samples getting lost. Should signal the main thread that it should stop transmission via hackrf_stop_tx

3.4.3.2 hackrf_sample_block_cb_fn

```
typedef int(* hackrf_sample_block_cb_fn) (hackrf_transfer *transfer)
```

In each mode, it is called when data needs to be handled, meaning filling samples in TX mode or reading them in RX modes.

In TX mode, it should refill the transfer buffer with new raw IQ data, and set hackrf_transfer::valid_length.

In RX mode, it should copy/process the contents of the transfer buffer's valid part.

In RX SWEEP mode, it receives multiple "blocks" of data, each with a 10-byte header containing the tuned frequency followed by the samples. See hackrf_init_sweep for more info.

The callback should return 0 if it wants to be called again, and any other value otherwise. Stopping the RX/TX/SWEEP is still done with hackrf_stop_rx and hackrf_stop_tx, and those should be called from the main thread, so this callback should signal the main thread that it should stop. Signaling the main thread to stop TX should be done from the flush callback in order to guarantee that no samples are discarded, see hackrf_flush_cb_fn

3.4.3.3 hackrf_tx_block_complete_cb_fn

```
typedef void(* hackrf_tx_block_complete_cb_fn) (hackrf_transfer *transfer, int)
```

Set via hackrf_set_tx_block_complete_callback, called when a transfer is finished to the device's buffer, regardless if the transfer was successful or not. It can signal the main thread to stop on failure, can catch USB transfer errors and can also gather statistics about the transfered data.

3.4.4 Enumeration Type Documentation

3.4.4.1 sweep_style

```
enum sweep_style
```

Used by hackrf_init_sweep, to set sweep parameters.

Linear mode is no longer used by the hackrf_sweep command line tool and in general the interleaved method is always preferable, but the linear mode remains available for backward compatibility and might be useful in some special circumstances.

Enumerator

LINEAR	step_width is added to the current frequency at each step.
INTERLEAVED	each step is divided into two interleaved sub-steps, allowing the host to select the best portions of the FFT of each sub-step and discard the rest.

3.4.5 Function Documentation

3.4.5.1 hackrf_enable_tx_flush()

```
int hackrf_enable_tx_flush (
            hackrf_device * device,
            hackrf_flush_cb_fn callback,
            void * flush_ctx )
```

This callback will be called when all the data was transmitted and all data transfers were completed. First parameter is supplied context, second parameter is success flag.

Parameters

device	device to configure
callback	callback to call when all transfers were completed
flush_ctx	context (1st parameter of callback)

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.2 hackrf_init_sweep()

```
int hackrf_init_sweep (
            hackrf_device * device,
            const uint16_t * frequency_list,
            const int num_ranges,
            const uint32_t num_bytes,
            const uint32_t step_width,
            const uint32_t offset,
            const enum sweep_style style )
```

In this mode, in a single data transfer (single call to the RX transfer callback), multiple blocks of size `num_bytes` bytes are received with different center frequencies. At the beginning of each block, a 10-byte frequency header is present in `0x7F - 0x7F - uint64_t frequency (LSBFIRST, in Hz)` format, followed by the actual samples.

Requires USB API version 0x0102 or above!

Parameters

device	device to configure
frequency_list	list of start-stop frequency pairs in MHz
num_ranges	length of array `frequency_list` (in pairs, so total array length / 2!). Must be less than MAX_SWEEP_RANGES
num_bytes	number of bytes to capture per tuning, must be a multiple of BYTES_PER_BLOCK
step_width	width of each tuning step in Hz
offset	frequency offset added to tuned frequencies. sample_rate / 2 is a good value
style	sweep style

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.3 hackrf_is_streaming()

```
int hackrf_is_streaming (
            hackrf_device * device )
```

Parameters

device	device to query

Returns

HACKRF_TRUE if the device is streaming, else one of HACKRF_ERROR_STREAMING_THREAD_ERR, HACKRF_ERROR_STREAMING_STOPPED or HACKRF_ERROR_STREAMING_EXIT_CALLED

3.4.5.4 hackrf_set_hw_sync_mode()

```
int hackrf_set_hw_sync_mode (
            hackrf_device * device,
            const uint8_t value )
```

See the documentation on hardware triggering for details

Requires USB API version 0x0102 or above!

Parameters

device	device to configure
value	enable (1) or disable (0) hardware triggering

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.5 hackrf_set_rx_overrun_limit()

```
int hackrf_set_rx_overrun_limit (
            hackrf_device * device,
            uint32_t value )
```

When this limit is set, after the specified number of samples (bytes, not whole IQ pairs) missing the device will automatically return to IDLE mode, thus stopping operation. Useful for handling cases like program/computer crashes or other problems. The default value 0 means no limit.

Requires USB API version 0x0106 or above!

Parameters

device	device to configure
value	number of samples to wait before auto-stopping

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.6 hackrf_set_tx_block_complete_callback()

```
int hackrf_set_tx_block_complete_callback (
            hackrf_device * device,
            hackrf_tx_block_complete_cb_fn callback )
```

This callback will be called whenever an USB transfer to the device is completed, regardless if it was successful or not (indicated by the second parameter).

Parameters

device	device to configure
callback	callback to call when a transfer is completed

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.7 hackrf_set_tx_underrun_limit()

```
int hackrf_set_tx_underrun_limit (
            hackrf_device * device,
            uint32_t value )
```

When this limit is set, after the specified number of samples (bytes, not whole IQ pairs) missing the device will automatically return to IDLE mode, thus stopping operation. Useful for handling cases like program/computer crashes or other problems. The default value 0 means no limit.

Requires USB API version 0x0106 or above!

Parameters

device	device to configure
value	number of samples to wait before auto-stopping

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.8 hackrf_start_rx()

```
int hackrf_start_rx (
            hackrf_device * device,
            hackrf_sample_block_cb_fn callback,
            void * rx_ctx )
```

Should be called after setting gains, frequency and sampling rate, as these values won't get reset but instead keep their last value, thus their state is unknown.

The callback is called with a hackrf_transfer object whenever the buffer is full. The callback is called in an async context so no libhackrf functions should be called from it. The callback should treat its argument as read-only.

Parameters

device	device to configure
callback	rx_callback
rx_ctx	User provided RX context. Not used by the library, but available to `callback` as hackrf_transfer::rx_ctx.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.9 hackrf_start_rx_sweep()

```
int hackrf_start_rx_sweep (
            hackrf_device * device,
            hackrf_sample_block_cb_fn callback,
            void * rx_ctx )
```

See hackrf_init_sweep for more info

Requires USB API version 0x0104 or above!

Parameters

device	device to start sweeping
callback	rx callback processing the received data
rx_ctx	User provided RX context. Not used by the library, but available to `callback` as hackrf_transfer::rx_ctx.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.10 hackrf_start_tx()

```
int hackrf_start_tx (
            hackrf_device * device,
            hackrf_sample_block_cb_fn callback,
            void * tx_ctx )
```

Should be called after setting gains, frequency and sampling rate, as these values won't get reset but instead keep their last value, thus their state is unknown. Setting flush function (using hackrf_enable_tx_flush) and/or setting block complete callback (using hackrf_set_tx_block_complete_callback) (if these features are used) should also be done before this.

The callback is called with a hackrf_transfer object whenever a transfer buffer is needed to be filled with samples. The callback is called in an async context so no libhackrf functions should be called from it. The callback should treat its argument as read-only, except the hackrf_transfer::buffer and hackrf_transfer::valid_length.

Parameters

device	device to configure
callback	tx_callback
tx_ctx	User provided TX context. Not used by the library, but available to `callback` as hackrf_transfer::tx_ctx.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.11 hackrf_stop_rx()

```
int hackrf_stop_rx (
            hackrf_device * device )
```

Parameters

device	device to stop RX on

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.4.5.12 hackrf_stop_tx()

```
int hackrf_stop_tx (
            hackrf_device * device )
```

Parameters

device	device to stop TX on

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5 Firmware flashing & debugging

Firmware flashing and directly accessing hardware components.

Data Structures

- struct hackrf_m0_state

  State of the SGPIO loop running on the M0 core.

Functions

- int hackrf_get_m0_state (hackrf_device *device, hackrf_m0_state *value)

  Get the state of the M0 code on the LPC43xx MCU.
- int hackrf_max2837_read (hackrf_device *device, uint8_t register_number, uint16_t *value)

  Directly read the registers of the MAX2837 transceiver IC.
- int hackrf_max2837_write (hackrf_device *device, uint8_t register_number, uint16_t value)

  Directly write the registers of the MAX2837 transceiver IC.
- int hackrf_si5351c_read (hackrf_device *device, uint16_t register_number, uint16_t *value)

  Directly read the registers of the Si5351C clock generator IC.
- int hackrf_si5351c_write (hackrf_device *device, uint16_t register_number, uint16_t value)

  Directly write the registers of the Si5351 clock generator IC.
- int hackrf_rffc5071_read (hackrf_device *device, uint8_t register_number, uint16_t *value)

  Directly read the registers of the RFFC5071/5072 mixer-synthesizer IC.
- int hackrf_rffc5071_write (hackrf_device *device, uint8_t register_number, uint16_t value)

  Directly write the registers of the RFFC5071/5072 mixer-synthesizer IC.
- int hackrf_spiflash_erase (hackrf_device *device)

  Erase firmware image on the SPI flash.

- int hackrf_spiflash_write (hackrf_device *device, const uint32_t address, const uint16_t length, unsigned char *const data)

  Write firmware image on the SPI flash.
- int hackrf_spiflash_read (hackrf_device *device, const uint32_t address, const uint16_t length, unsigned char *data)

  Read firmware image on the SPI flash.
- int hackrf_spiflash_status (hackrf_device *device, uint8_t *data)

  Read the status registers of the W25Q80BV SPI flash chip.
- int hackrf_spiflash_clear_status (hackrf_device *device)

  Clear the status registers of the W25Q80BV SPI flash chip.
- int hackrf_cpld_write (hackrf_device *device, unsigned char *const data, const unsigned int total_length)

  Write configuration bitstream into the XC2C64A-7VQ100C CPLD.

3.5.1 Detailed Description

3.5.2 Firmware flashing

IMPORTANT You should try to use the existing flashing utilities (`hackrf_spiflash`) to flash new firmware to the device! Incorrect usage of the SPIFLASH functions (especially hackrf_spiflash_erase an hackrf_spiflash_write) can brick the device, and DFU mode will be needed to unbrick it!

Firmware flashing can be achieved via writing to the SPI flash holding the firmware of the ARM microcontroller. This can be achieved by the `hackrf_spiflash_*` functions.

The Spartan II CPLD inside the HackRF One devices could also be reconfigured in the past, but in newer firmwares, the ARM MCU automatically reconfigures it on startup with a bitstream baked into the firmware image, thus the function hackrf_cpld_write has no effect, and CPLD flashing can only be done by building a custom firmware (or the automatic loading can be disabled this way as well). The function hackrf_cpld_write and the util `hackrf_cpldjtag` are **deprecated** and only kept for backward compatibility with older firmware versions.

3.5.3 Debugging

The functions in this section can be used to directly read/write internal registers of the chips inside a HackRF One unit. See the page `Hardware Components` for more details on them.

Here's a brief introduction on the various chips in the HackRF One unit:

3.5.3.1 MAX2837 2.3 to 2.7 GHz transceiver

This transceiver chip is the RF modulator/demodulator of the HackRF One. This chip sends/receives analoge I/Q samples to/from the MAX5864 ADC/DAC chip.

Its registers are accessible through the functions hackrf_max2837_read and hackrf_max2837_write

3.5.3.2 MAX5864 ADC/DAC

This chip converts received analgoe I/Q samples to digital and transmitted I/Q samples to analoge. It connects to the main ARM MCU through the CPLD. No configuration is needed for it, only the sample rate can be set via the clock generator IC.

3.5.3.3 Si5351C Clock generator

This chip supplies clock signals to all of the other chips. It can synthesize a wide range of frequencies from its clock inputs (internal or external). It uses a fixed 800-MHz internal clock (synthesized via a PLL).

Its registers are accessible through the functions hackrf_si5351c_read and hackrf_si5351c_write

3.5.3.4 RFFC5072 Synthesizer/mixer

This mixer mixes the RF signal with an internally synthesized local oscillator signal and thus results in the sum and difference frequencies. Combined with the LPF or HPF filters and the frequency setting in the MAX2837 IC it can be used to tune to any frequency in the 0-6000MHz range.

Its registers are accessible through the functions hackrf_rffc5071_read and hackrf_rffc5071_write

3.5.3.5 LPC4320 ARM MCU

This is the main processor of the unit. It's a multi-core ARM processor. It's configured to boot from a W25Q80B SPI flash, but can also be booted from DFU in order to unbrick a bricked unit. It communicated with the host PC via USB.

Some operation details are available via the function hackrf_get_m0_state

3.5.3.6 W25Q80B SPI flash

This chip holds the firmware for the LPC4320 ARM MCU.

It's accessible through the functions hackrf_spiflash_read, hackrf_spiflash_write, hackrf_spiflash_erase, hackrf_spiflash_status and hackrf_spiflash_clear_status

3.5.3.7 XC2C64A CPLD

This CPLD sits between the MAX5864 ADC/DAC and the main MCU, and mainly performs data format conversion and some synchronisation.

Its bitstream is auto-loaded on reset by the ARM MCU (from the firmware image), but in older versions, it was possible to reconfigure it via hackrf_cpld_write, and the (since temporarly removed) `hackrf_cpld_checksum` function could verify the firmware in the configuration flash (again, overwritten on startup, so irrelevant).

See `issue 608`, `issue 1140` and `issue 1141` for some more details on this!

3.5.4 Function Documentation

3.5.4.1 hackrf_cpld_write()

```
int hackrf_cpld_write (
            hackrf_device * device,
            unsigned char *const data,
            const unsigned int total_length )
```

Parameters

device	device to configure
data	CPLD bitstream data
total_length	length of the bitstream to write

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.2 hackrf_get_m0_state()

```
int hackrf_get_m0_state (
            hackrf_device * device,
            hackrf_m0_state * value )
```

Requires USB API version 0x0106 or above!

Parameters

in	*device*	device to query
out	*value*	MCU code state

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.3 hackrf_max2837_read()

```
int hackrf_max2837_read (
            hackrf_device * device,
            uint8_t register_number,
            uint16_t * value )
```

Intended for debugging purposes only!

Parameters

in	*device*	device to query
in	*register_number*	register number to read
out	*value*	value of the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.4 hackrf_max2837_write()

```
int hackrf_max2837_write (
            hackrf_device * device,
            uint8_t register_number,
            uint16_t value )
```

Intended for debugging purposes only!

Parameters

device	device to write
register_number	register number to write
value	value to write in the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.5 hackrf_rffc5071_read()

```
int hackrf_rffc5071_read (
            hackrf_device * device,
```

```
            uint8_t register_number,
            uint16_t * value )
```

Intended for debugging purposes only!

Parameters

in	*device*	device to query
in	*register_number*	register number to read
out	*value*	value of the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.6 hackrf_rffc5071_write()

```
int hackrf_rffc5071_write (
            hackrf_device * device,
            uint8_t register_number,
            uint16_t value )
```

Intended for debugging purposes only!

Parameters

in	*device*	device to write
in	*register_number*	register number to write
out	*value*	value to write in the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.7 hackrf_si5351c_read()

```
int hackrf_si5351c_read (
            hackrf_device * device,
            uint16_t register_number,
            uint16_t * value )
```

Intended for debugging purposes only!

Parameters

in	*device*	device to query
in	*register_number*	register number to read
out	*value*	value of the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.8 hackrf_si5351c_write()

```
int hackrf_si5351c_write (
            hackrf_device * device,
            uint16_t register_number,
            uint16_t value )
```

Intended for debugging purposes only!

Parameters

in	*device*	device to write
in	*register_number*	register number to write
out	*value*	value to write in the specified register

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.9 hackrf_spiflash_clear_status()

```
int hackrf_spiflash_clear_status (
            hackrf_device * device )
```

See the datasheet for details of the status registers.

Requires USB API version 0x0103 or above!

Parameters

device	device to clear

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.10 hackrf_spiflash_erase()

```
int hackrf_spiflash_erase (
            hackrf_device * device )
```

Should be followed by writing a new image, or the HackRF will be soft-bricked (still rescuable in DFU mode)

Parameters

device	device to ersase

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.11 hackrf_spiflash_read()

```
int hackrf_spiflash_read (
            hackrf_device * device,
            const uint32_t address,
            const uint16_t length,
            unsigned char * data )
```

Should only be used for firmware verification.

Parameters

device	device to read from
address	address to read from. Firmware should start at 0
length	length of data to read. Must be at most 256.
data	pointer to buffer

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.12 hackrf_spiflash_status()

```
int hackrf_spiflash_status (
            hackrf_device * device,
            uint8_t * data )
```

See the datasheet for details of the status registers. The two registers are read in order.

Requires USB API version 0x0103 or above!

Parameters

in	*device*	device to query
out	*data*	char[2] array of the status registers

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.5.4.13 hackrf_spiflash_write()

```
int hackrf_spiflash_write (
            hackrf_device * device,
            const uint32_t address,
            const uint16_t length,
            unsigned char *const data )
```

Should only be used for firmware updating. Can brick the device, but it's still rescuable in DFU mode.

Parameters

device	device to write on
address	address to write to. Should start at 0
length	length of data to write. Must be at most 256.
data	data to write

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6 Opera Cake add-on board functions

Various functions related to the Opera Cake add-on boards.

Data Structures

- struct hackrf_operacake_dwell_time

  Opera Cake port setting in OPERACAKE_MODE_TIME operation.
- struct hackrf_operacake_freq_range

  Opera Cake port setting in OPERACAKE_MODE_FREQUENCY operation.

Macros

- #define HACKRF_OPERACAKE_ADDRESS_INVALID 0xFF

  Invalid Opera Cake add-on board address, placeholder in hackrf_get_operacake_boards.
- #define HACKRF_OPERACAKE_MAX_BOARDS 8

  Maximum number of connected Opera Cake add-on boards.
- #define HACKRF_OPERACAKE_MAX_DWELL_TIMES 16

  Maximum number of specifiable dwell times for Opera Cake add-on boards.
- #define HACKRF_OPERACAKE_MAX_FREQ_RANGES 8

  Maximum number of specifiable frequency ranges for Opera Cake add-on boards.

Enumerations

- enum operacake_ports {
  OPERACAKE_PA1 = 0 ,
  OPERACAKE_PA2 = 1 ,
  OPERACAKE_PA3 = 2 ,
  OPERACAKE_PA4 = 3 ,
  OPERACAKE_PB1 = 4 ,
  OPERACAKE_PB2 = 5 ,
  OPERACAKE_PB3 = 6 ,
  OPERACAKE_PB4 = 7 }

  Opera Cake secondary ports (A1-A4, B1-B4)
- enum operacake_switching_mode {
  OPERACAKE_MODE_MANUAL ,
  OPERACAKE_MODE_FREQUENCY ,
  OPERACAKE_MODE_TIME }

  Opera Cake port switching mode.

Functions

- int hackrf_get_operacake_boards (hackrf_device *device, uint8_t *boards)

  Query connected Opera Cake boards.
- int hackrf_set_operacake_mode (hackrf_device *device, uint8_t address, enum operacake_switching_mode mode)

  Setup Opera Cake operation mode.
- int hackrf_get_operacake_mode (hackrf_device *device, uint8_t address, enum operacake_switching_mode *mode)

  Query Opera Cake mode.
- int hackrf_set_operacake_ports (hackrf_device *device, uint8_t address, uint8_t port_a, uint8_t port_b)

Setup Opera Cake ports in OPERACAKE_MODE_MANUAL mode operation.

- int hackrf_set_operacake_dwell_times (hackrf_device *device, hackrf_operacake_dwell_time *dwell_times, uint8_t count)

  Setup Opera Cake dwell times in OPERACAKE_MODE_TIME mode operation.
- int hackrf_set_operacake_freq_ranges (hackrf_device *device, hackrf_operacake_freq_range *freq_ranges, uint8_t count)

  Setup Opera Cake frequency ranges in OPERACAKE_MODE_FREQUENCY mode operation.
- int hackrf_set_operacake_ranges (hackrf_device *device, uint8_t *ranges, uint8_t num_ranges)

  Setup Opera Cake frequency ranges in OPERACAKE_MODE_FREQUENCY mode operation.
- int hackrf_operacake_gpio_test (hackrf_device *device, uint8_t address, uint16_t *test_result)

  Perform GPIO test on an Opera Cake addon board.

3.6.1 Detailed Description

These boards are versatile RF switching boards capable of switching two primary ports (A0 and B0) to any of 8 (A1-A4 and B1-B4) secondary ports (with the only rule that A0 and B0 can not be connected to the same side/bank of secondary ports at the same time).

There are 3 operating modes:

- manual setup
- frequency-based setup
- time-based setup

3.6.1.0.1 Manual setup This mode allows A0 and B0 to be connected to any of the secondary ports. This mode is configured with hackrf_set_operacake_ports.

3.6.1.0.2 Frequency-based setup In this mode the Opera Cake board automatically switches A0 to a port depending on the tuning frequency. Up to HACKRF_OPERACAKE_MAX_FREQ_RANGES frequency ranges can be setup using hackrf_set_operacake_freq_ranges, in a priority order. Port B0 mirrors A0 on the opposite side (but both B and A side ports can be specified for connections to A0)

3.6.1.0.3 Time-based setup In this mode the Opera Cake board automatically switches A0 to a port for a set amount of time (specified in samples). Up to HACKRF_OPERACAKE_MAX_DWELL_TIMES times can be setup via hackrf_set_operacake_dwell_times. Port B0 mirrors A0 on the opposite side.

3.6.1.1 Opera Cake setup

Opera Cake boards can be listed with hackrf_get_operacake_boards, but if only one board is connected, than using address 0 defaults to it.

Opera Cake mode can be setup via hackrf_set_operacake_mode, then the corresponding configuration function can be called.

3.6.1.2 Multiple boards

There can be up to HACKRF_OPERACAKE_MAX_BOARDS boards connected to a single HackRF One. They can be assigned individual addresses via onboard jumpers, see the documentation page for details. **Note**: the operating modes of the boards can be set individually via hackrf_set_operacake_mode, but in frequency or time mode, every board configured to that mode will use the same switching plan!

3.6.2 Macro Definition Documentation

3.6.2.1 HACKRF_OPERACAKE_ADDRESS_INVALID

```
#define HACKRF_OPERACAKE_ADDRESS_INVALID 0xFF
```

3.6.2.2 HACKRF_OPERACAKE_MAX_BOARDS

```
#define HACKRF_OPERACAKE_MAX_BOARDS 8
```

3.6.2.3 HACKRF_OPERACAKE_MAX_DWELL_TIMES

```
#define HACKRF_OPERACAKE_MAX_DWELL_TIMES 16
```

3.6.2.4 HACKRF_OPERACAKE_MAX_FREQ_RANGES

```
#define HACKRF_OPERACAKE_MAX_FREQ_RANGES 8
```

3.6.3 Enumeration Type Documentation

3.6.3.1 operacake_ports

```
enum operacake_ports
```

Enumerator

OPERACAKE_PA1	
OPERACAKE_PA2	
OPERACAKE_PA3	
OPERACAKE_PA4	
OPERACAKE_PB1	
OPERACAKE_PB2	
OPERACAKE_PB3	
OPERACAKE_PB4	

3.6.3.2 operacake_switching_mode

```
enum operacake_switching_mode
```

Set via hackrf_set_operacake_mode and quaried via hackrf_get_operacake_mode

Enumerator

OPERACAKE_MODE_MANUAL	Port connections are set manually using hackrf_set_operacake_ports. Both ports can be specified, but not on the same side.
OPERACAKE_MODE_FREQUENCY	Port connections are switched automatically when the frequency is changed. Frequency ranges can be set using hackrf_set_operacake_freq_ranges. In this mode, B0 mirrors A0
OPERACAKE_MODE_TIME	Port connections are switched automatically over time. dwell times can be set with hackrf_set_operacake_dwell_times. In this mode, B0 mirrors A0

3.6.4 Function Documentation

3.6.4.1 hackrf_get_operacake_boards()

```
int hackrf_get_operacake_boards (
            hackrf_device * device,
            uint8_t * boards )
```

Returns a HACKRF_OPERACAKE_MAX_BOARDS size array of addresses, with HACKRF_OPERACAKE_ADDRESS_INVALID as a placeholder

Requires USB API version 0x0105 or above!

Parameters

in	*device*	device to query
out	*boards*	list of boards

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.2 hackrf_get_operacake_mode()

```
int hackrf_get_operacake_mode (
            hackrf_device * device,
            uint8_t address,
            enum operacake_switching_mode * mode )
```

Requires USB API version 0x0105 or above!

Parameters

in	*device*	device to query from
in	*address*	address of add-on board to query
out	*mode*	operation mode of the selected add-on board

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.3 hackrf_operacake_gpio_test()

```
int hackrf_operacake_gpio_test (
            hackrf_device * device,
            uint8_t address,
            uint16_t * test_result )
```

Value 0xFFFF means "GPIO mode disabled", and hackrf_operacake advises to remove additional add-on boards and retry. Value 0 means all tests passed. In any other values, a 1 bit signals an error. Bits are grouped in groups of 3. Encoding: 0 - u1ctrl - u3ctrl0 - u3ctrl1 - u2ctrl0 - u2ctrl1

Requires USB API version 0x0103 or above!

Parameters

in	*device*	device to perform test on
in	*address*	address of Opera Cake board to test
out	*test_result*	result of tests

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.4 hackrf_set_operacake_dwell_times()

```
int hackrf_set_operacake_dwell_times (
            hackrf_device * device,
            hackrf_operacake_dwell_time * dwell_times,
            uint8_t count )
```

Should be called after hackrf_set_operacake_mode

Note: this configuration applies to all Opera Cake boards in OPERACAKE_MODE_TIME mode

Requires USB API version 0x0105 or above!

Parameters

device	device to configure
dwell_times	list of dwell times to setup
count	number of dwell times to setup. Must be at most HACKRF_OPERACAKE_MAX_DWELL_TIMES.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.5 hackrf_set_operacake_freq_ranges()

```
int hackrf_set_operacake_freq_ranges (
            hackrf_device * device,
            hackrf_operacake_freq_range * freq_ranges,
            uint8_t count )
```

Should be called after hackrf_set_operacake_mode

Note: this configuration applies to all Opera Cake boards in OPERACAKE_MODE_FREQUENCY mode

Requires USB API version 0x0103 or above!

Parameters

device	device to configure
freq_ranges	list of frequency ranges to setup
count	number of ranges to setup. Must be at most HACKRF_OPERACAKE_MAX_FREQ_RANGES.

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.6 hackrf_set_operacake_mode()

```
int hackrf_set_operacake_mode (
            hackrf_device * device,
            uint8_t address,
            enum operacake_switching_mode mode )
```

Requires USB API version 0x0105 or above!

Parameters

device	device to configure
address	address of Opera Cake add-on board to configure
mode	mode to use

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.7 hackrf_set_operacake_ports()

```
int hackrf_set_operacake_ports (
            hackrf_device * device,
            uint8_t address,
            uint8_t port_a,
            uint8_t port_b )
```

Should be called after hackrf_set_operacake_mode. A0 and B0 must be connected to opposite sides (A->A and B->B or A->B and B->A but not A->A and B->A or A->B and B->B)

Requires USB API version 0x0102 or above!

Parameters

device	device to configure
address	address of add-on board to configure
port_a	port for A0. Must be one of operacake_ports
port_b	port for B0. Must be one of operacake_ports

Returns

HACKRF_SUCCESS on success or hackrf_error variant

3.6.4.8 hackrf_set_operacake_ranges()

```
int hackrf_set_operacake_ranges (
            hackrf_device * device,
            uint8_t * ranges,
            uint8_t num_ranges )
```

Old function to set ranges with. Use hackrf_set_operacake_freq_ranges instead!

Note: this configuration applies to all Opera Cake boards in OPERACAKE_MODE_FREQUENCY mode

Requires USB API version 0x0103 or above!

Parameters

device	device to configure
ranges	ranges to setup. Should point to a valid hackrf_operacake_freq_range array.
num_ranges	length of ranges to setup, must be number of ranges ∗ 5. Must be at most 8∗5=40. (internally called len_ranges, possible typo)

Returns

HACKRF_SUCCESS on success or hackrf_error variant

Chapter 4

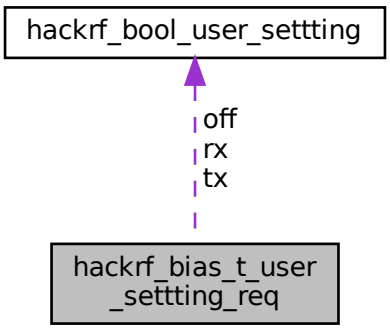
Data Structure Documentation

4.1 hackrf_bias_t_user_settting_req Struct Reference

User settings for user-supplied bias tee defaults.

`#include <hackrf.h>`

Collaboration diagram for hackrf_bias_t_user_settting_req:



Data Fields

- hackrf_bool_user_settting tx
- hackrf_bool_user_settting rx
- hackrf_bool_user_settting off

4.1.1 Field Documentation

4.1.1.1 off

hackrf_bool_user_settting hackrf_bias_t_user_settting_req::off

4.1.1.2 rx

hackrf_bool_user_settting hackrf_bias_t_user_settting_req::rx

4.1.1.3 tx

hackrf_bool_user_settting hackrf_bias_t_user_settting_req::tx

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.2 hackrf_bool_user_settting Struct Reference

Helper struct for hackrf_bias_t_user_setting.

`#include <hackrf.h>`

Data Fields

- bool do_update
- bool change_on_mode_entry
- bool enabled

4.2.1 Detailed Description

If 'do_update' is true, then the values of 'change_on_mode_entry' and 'enabled' will be used as the new default. If 'do_update' is false, the current default will not change.

4.2.2 Field Documentation

4.2.2.1 change_on_mode_entry

```
bool hackrf_bool_user_settting::change_on_mode_entry
```

4.2.2.2 do_update

```
bool hackrf_bool_user_settting::do_update
```

4.2.2.3 enabled

```
bool hackrf_bool_user_settting::enabled
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.3 hackrf_device_list_t Struct Reference

List of connected HackRF devices.

```
#include <hackrf.h>
```

Data Fields

- char ** serial_numbers
  Array of human-readable serial numbers.
- enum hackrf_usb_board_id * usb_board_ids
  ID of each board, based on USB product ID.
- int * usb_device_index
  USB device index for a given HW entry.
- int devicecount
  Number of connected HackRF devices, the length of arrays serial_numbers, usb_board_ids and usb_device_index.
- void ** usb_devices
  All USB devices (as `libusb_device**` array)
- int usb_devicecount
  Number of all queried USB devices.

4.3.1 Detailed Description

Acquired via hackrf_device_list and should be freeed via hackrf_device_list_free. Individual devices can be opened via hackrf_device_list_open

4.3.2 Field Documentation

4.3.2.1 devicecount

```
int hackrf_device_list_t::devicecount
```

4.3.2.2 serial_numbers

```
char** hackrf_device_list_t::serial_numbers
```

Each entry can be NULL!

4.3.2.3 usb_board_ids

```
enum hackrf_usb_board_id* hackrf_device_list_t::usb_board_ids
```

Can be used for general HW identification without opening the device.

4.3.2.4 usb_device_index

```
int* hackrf_device_list_t::usb_device_index
```

Intended for internal use only.

4.3.2.5 usb_devicecount

```
int hackrf_device_list_t::usb_devicecount
```

Length of array usb_devices.

4.3.2.6 usb_devices

```
void** hackrf_device_list_t::usb_devices
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.4 hackrf_m0_state Struct Reference

State of the SGPIO loop running on the M0 core.

```
#include <hackrf.h>
```

Data Fields

- uint16_t requested_mode

  Requested mode.
- uint16_t request_flag

  Request flag, 0 means request is completed, any other value means request is pending.
- uint32_t active_mode

  Active mode.
- uint32_t m0_count

  Number of bytes transferred by the M0.
- uint32_t m4_count

  Number of bytes transferred by the M4.
- uint32_t num_shortfalls

  Number of shortfalls.
- uint32_t longest_shortfall

  Longest shortfall in bytes.
- uint32_t shortfall_limit

  Shortfall limit in bytes.
- uint32_t threshold

  Threshold m0_count value (in bytes) for next mode change.
- uint32_t next_mode

  Mode which will be switched to when threshold is reached.
- uint32_t error

  Error, if any, that caused the M0 to revert to IDLE mode.

4.4.1 Field Documentation

4.4.1.1 active_mode

```
uint32_t hackrf_m0_state::active_mode
```

Possible values are the same as in hackrf_m0_state::requested_mode

4.4.1.2 error

```
uint32_t hackrf_m0_state::error
```

Possible values are 0 (NONE), 1 (RX_TIMEOUT) and 2(TX_TIMEOUT)

4.4.1.3 longest_shortfall

```
uint32_t hackrf_m0_state::longest_shortfall
```

4.4.1.4 m0_count

```
uint32_t hackrf_m0_state::m0_count
```

4.4.1.5 m4_count

```
uint32_t hackrf_m0_state::m4_count
```

4.4.1.6 next_mode

```
uint32_t hackrf_m0_state::next_mode
```

Possible values are the same as in hackrf_m0_state::requested_mode

4.4.1.7 num_shortfalls

```
uint32_t hackrf_m0_state::num_shortfalls
```

4.4.1.8 request_flag

```
uint16_t hackrf_m0_state::request_flag
```

4.4.1.9 requested_mode

```
uint16_t hackrf_m0_state::requested_mode
```

Possible values are 0(IDLE), 1(WAIT), 2(RX), 3(TX_START), 4(TX_RUN)

4.4.1.10 shortfall_limit

```
uint32_t hackrf_m0_state::shortfall_limit
```

4.4.1.11 threshold

```
uint32_t hackrf_m0_state::threshold
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.5 hackrf_operacake_dwell_time Struct Reference

Opera Cake port setting in OPERACAKE_MODE_TIME operation.

```
#include <hackrf.h>
```

Data Fields

- uint32_t dwell

  Dwell time for port (in number of samples)
- uint8_t port

  Port to connect A0 to (B0 mirrors this choice) Must be one of operacake_ports.

4.5.1 Field Documentation

4.5.1.1 dwell

```
uint32_t hackrf_operacake_dwell_time::dwell
```

4.5.1.2 port

```
uint8_t hackrf_operacake_dwell_time::port
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.6 hackrf_operacake_freq_range Struct Reference

Opera Cake port setting in OPERACAKE_MODE_FREQUENCY operation.

```
#include <hackrf.h>
```

Data Fields

- uint16_t freq_min
  Start frequency (in MHz)
- uint16_t freq_max
  Stop frequency (in MHz)
- uint8_t port
  Port (A0) to use for that frequency range.

4.6.1 Field Documentation

4.6.1.1 freq_max

```
uint16_t hackrf_operacake_freq_range::freq_max
```

4.6.1.2 freq_min

```
uint16_t hackrf_operacake_freq_range::freq_min
```

4.6.1.3 port

```
uint8_t hackrf_operacake_freq_range::port
```

Port B0 mirrors this. Must be one of operacake_ports

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.7 hackrf_transfer Struct Reference

USB transfer information passed to RX or TX callback.

```
#include <hackrf.h>
```

Data Fields

- hackrf_device * device
  HackRF USB device for this transfer.
- uint8_t * buffer
  transfer data buffer (interleaved 8 bit I/Q samples)
- int buffer_length
  length of data buffer in bytes
- int valid_length
  number of buffer bytes that were transferred
- void * rx_ctx
  User provided RX context.
- void * tx_ctx
  User provided TX context.

4.7.1 Detailed Description

A callback should treat all these fields as read-only except that a TX callback should write to the data buffer and may write to valid_length to indicate that a smaller number of bytes is to be transmitted.

4.7.2 Field Documentation

4.7.2.1 buffer

```
uint8_t* hackrf_transfer::buffer
```

4.7.2.2 buffer_length

```
int hackrf_transfer::buffer_length
```

4.7.2.3 device

```
hackrf_device* hackrf_transfer::device
```

4.7.2.4 rx_ctx

```
void* hackrf_transfer::rx_ctx
```

Not used by the library, but available to transfer callbacks for use. Set along with the transfer callback using hackrf_start_rx or hackrf_start_rx_sweep

4.7.2.5 tx_ctx

```
void* hackrf_transfer::tx_ctx
```

Not used by the library, but available to transfer callbacks for use. Set along with the transfer callback using hackrf_start_tx

4.7.2.6 valid_length

```
int hackrf_transfer::valid_length
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

4.8 read_partid_serialno_t Struct Reference

MCU (LPC43xx) part ID and serial number.

```
#include <hackrf.h>
```

Data Fields

- uint32_t part_id [2]

  MCU part ID register value.
- uint32_t serial_no [4]

  MCU device unique ID (serial number)

4.8.1 Detailed Description

See the documentation of the MCU for details! Read via hackrf_board_partid_serialno_read

4.8.2 Field Documentation

4.8.2.1 part_id

```
uint32_t read_partid_serialno_t::part_id[2]
```

4.8.2.2 serial_no

```
uint32_t read_partid_serialno_t::serial_no[4]
```

The documentation for this struct was generated from the following file:

- /tmp/host/libhackrf/src/hackrf.h

Index