MAVID-3M

API Reference Guide

Revision: 0.1

Libre Wireless Technologies Private Limited

librewireless.com

Circuit diagrams and other information relating to Libre Wireless Technologies products are included as a means of illustrating typical applications. Consequently, complete information sufficient for construction purposes is not necessarily given. Although the information has been checked and is believed to be accurate, no responsibility is assumed for inaccuracies. Libre Wireless Technologies reserves the right to make changes to specifications and product descriptions at any time without notice. Contact your local Libre Wireless Technologies sales office to obtain the latest specifications before placing your product order. The provision of this information does not convey to the purchaser of the described semiconductor devices any licenses under any patent rights or other intellectual property rights of Libre Wireless Technologies or others. All sales are expressly conditional on your agreement to the terms and conditions of the most recently dated version of Libre Wireless Technologies standard Terms of Sale Agreement dated before the date of your order (the "Terms of Sale Agreement"). The product may contain design defects or errors known as anomalies which may cause the product's functions to deviate from published specifications. Anomaly sheets are available upon request. Libre Wireless Technologies products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an Officer of Libre Wireless Technologies and further testing and/or modification will be fully at the risk of the customer. Copies of this document or other Libre Wireless Technologies literature, as well as the Terms of Sale Agreement, may be obtained by visiting Libre Wireless Technologies website.

LIBRE WIRELESS TECHNOLOGIES DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT AND THE LIKE, AND ANY AND ALL WARRANTIES ARISING FROM ANY COURSE OF DEALING OR USAGE OF TRADE. IN NO EVENT SHALL LIBRE WIRELESS TECHNOLOGIES BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES; OR FOR LOST DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND; REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF LIBRE WIRELESS TECHNOLOGIES OR OTHERS; STRICT LIABILITY; BREACH OF WARRANTY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER OR NOT LIBRE WIRELESS TECHNOLOGIES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES

Table of Contents

1. Document Information

1.1. Abstract

Application Programming Interfaces (APIs) enable the reuse of libraries and frameworks in software development. An API is a contract between the component providing a functionality and the component using that functionality.

MAVID-3M SDK provides platform for developers to make their own Alexa voice-based application in different environments. The platform supports hardware abstraction layers, peripheral drivers, FreeRTOS, Wi-Fi, lwIP modules and user applications. This API reference document describes list of APIs for each of the supported modules.

1.2. Document Convention

Icon	Meaning	Description
	Note	Provides information good to know
	Caution	Indicates situation that might result in loss of data or hardware damage

1.3. Document Revision History

Revision	Date	Description of change	Author
0.1	June 11, 2021	Initial Draft	Jay and Sachin

2. API Reference

This chapter describes the features of and how to use the APIs for each of the supported modules.

2.1. Init_service

This section describes the initial program services for system initialization.

Example: hardware, nvdm, logging and random seed.

2.1.1. Sys_init

Header File

Examples/project/aw7698_evk/inc/sys_init.h

Function

void system_init(void);

This function will initialize the clock and required peripherals.

2.1.2. SDK_init

Header File

Examples/project/aw7698_evk/inc/SDK_init.h

Function

int32_t Wifi_preconfig_init(char *ssid, char *password, int pre_config);

Wifi init function will initialize the Wi-Fi module with SSID and password.

Parameter:

ssid: For ssid.

password: For password.

pre_config: Flag to configure where to get SSID and password, from defined macro or from CLI command.

Return:

Status for success or failure.

2.1.3. i2c_master_init

Header File

Examples/project/aw7698_evk/inc/i2c_master_init.h

Function

I2C_STATUS_t i2c_master_init(hal_i2c_port_t i2c_port, hal_i2c_frequency_t frequency);

It is used to initialize i2c master parameters.

Parameter:

i2c_port: i2c port name.

frequency: i2c communication frequency.

Return:

i2c init status.

Function

I2C_STATUS_t i2c_master_deinit (hal_i2c_port_t i2c_port);

It is used to deinitialize i2c master parameters.

Parameter:

i2c_port: i2c port name.

Return:

i2c deinit status.

2.2. Network Config

This section describes the network configuration.

2.2.1. wifi_nvdm_config

Header File

Examples/project/aw7698_evk/inc/netInit/inc/wifi_nvdm_config.h

Function

int32_t wifi_config_init(wifi_cfg_t *wifi_config);

Wi-Fi configuration initialization.

Parameter:

wifi_config: Wi-Fi parameters init.

Return:

Wi-Fi config status.

2.2.2. wifi_lwip_helper

Header File

Examples/project/aw7698_evk/inc/netInit/inc/wifi_lwip_helper.h

Function

void lwip_network_init(uint8_t opmode);

Initialize Wi-Fi and lwip configuration.

Parameter:

opmode: The target operation mode.

Function

void lwip_net_start(uint8_t opmode);

Start lwip service in a certain operation mode.

Parameter:

opmode: The target operation mode.

Function

void lwip_net_stop(uint8_t opmode);

Stop lwip service in a certain operation mode.

Parameter:

opmode: The current operation mode.

Function

void lwip_net_ready(void);

If Wi-Fi and IP are ready, it will return the Wi-Fi connectivity and IP values.

Function

uint8_t wifi_set_opmode(uint8_t target_mode);

Change operation mode dynamically.

Parameter:

target_mode: The target switched operation mode.

Return:

Status of the Wi-Fi opmode set.

2.2.3. wifi_events

Header File

Examples/project/aw7698_evk/inc/avsExt/wifi_events.h

Function

void subscribe_for_events(events_callback_t cb);

Subscribe for Wi-Fi events.

Parameter:

cb: Subscribe events callback.

Function

void unsubscribe_for_events(events_callback_t cb);

Unsubscribe for Wi-Fi events.

Parameter:

cb: Unsubscribe events callback.

2.3. AVS

This section describes the Amazon Login and competitivity with Amazon Alexa skills.

2.3.1. avs

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs.h

Function

void avs_app_init(void);

AVS app initialization.

Function

error_e avsInit(AvsContext *context);

AVS parameter initialization.

Parameter:

context: AVS context capability event.

Return:

Proper error message.

Function

error_e avsSetProductId(AvsContext *context, const char_t *productId);

AVS product ID setting.

Parameter:

context: AVS context capability event.

productId: To set desired product ID.

Return:

Proper error message.

Function

error_e avsSetProductSerialNum(AvsContext *context, const char_t *productSerialNum);

To set AVS product ID serial number.

Parameter:

context: AVS context capability event.

productId: To set desired product serial number.

Return:

Proper error message.

Function

error_e avsSetClientId(AvsContext *context, const char_t *clientId);

Set AVS client ID.

Parameter:

context: AVS context capability event.

productId: To set desired client ID.

Return:

Proper error message.

Function

error_e avsSetClientSecret(AvsContext *context, const char_t *clientSecret);

Set AVS client secret ID.

Parameter:

context: AVS context capability event.

productId: To set desired client secret ID.

Return:

Proper error message.

Function

void avs_stream_send_rst_frame(int stream_id);

This function is used to send reset frame for AVS stream.

Parameter:

stream_id: To give stream ID.

Function

int avs_stream_get_effective_local_window(int stream_id);

This function is used to get effective local value for AVS stream ID.

Parameter:

stream_id: To get stream ID.

Return:

Stream effective local window value.

Function

void avs_stream_set_local_window(int stream_id, int val);

This function is used to set local window for AVS stream ID.

Parameter:

stream_id: To give proper stream ID.

val: Stream effective local window value.

Function

int avs_check_and_trigger_kwDetect();

This function is used to check and trigger KW detect for AVS.

Return:

Trigger event start or exit value.

Function

void avsTask(void *param);

This function will schedule AVS task.

Function

void avsTerminateSession(void);

This function will terminate AVS session.

Function

error_e avsStart(AvsContext *context);

This function will start AVS event.

Parameter:

Context: AVS context capability event.

Return:

Proper error message.

Function

void getDeviceID(char *deviceID);

Get AVS device ID.

Parameter:

deviceID: To give payload for device ID.

Function

void setAppCode(char *appCode);

This function is used to set AVS app code.

Parameter:

appCode: To give appcode.

Function

void ClearAppCode();

This function is used to clear AVS app code.

Function

void show_window_size(nghttp2_session *session, int stream_id);

This API is used to show AVS window size.

Parameter:

session: For networking session.

stream_id: To give AVS stream effective stream id.

2.3.2. avs_auth

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_auth.h

Function

char *get_barer_token_copy();

This API is used to copy Barer token.

Return:

Get copied stream.

Function

void avsBarerMemFree(void *ptr);

This API is used to free AVS barer memory.

Function

error_e avsAuthTick(systime_t time, AvsContext *context);

AVS Authentication.

Parameter:

time: System tick.

context: AVS capabilities context.

Return:

Error message.

Function

error_e avsGetAccessToken(AvsContext *context, AvsGrantType grantType, char_t *response);

Get AVS access token.

Parameter:

context: AVS capabilities context.

grantType: AVS grant type.

response: Response for access token.

Return:

Error message.

Function

error_e authServerDecodePercentEncodedString(const char_t *input,char_t *output, size_t outputSize);

AVS Auth server decode encode percent string.

Parameter:

input: For input stream buffer.

output: For output stream buffer.

outputSize: For output buffer size.

Return:

Error message.

2.3.3. avs_alerts

Header file

Examples/project/aw7698_evk/inc/avsClient/inc/avs_alerts.h

Function

error_e avsInitAlerts(void* pvt);

This function is used to initialize alerts interface.

Function

json_t *avsFormatAlertsContext(void);

Format alerts context.

Return:

JSON-encoded context.

Function

error_e avsAlertsTick(AvsContext *context);

Format alert context.

Parameter:

context: For AVS context capabilities.

Return:

JSON-encoded context.

Function

error_e avsProcessAlertsDirective(const char_t *name, json_t *payload, int stream_id);

This function is used to process alerts directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessSetAlertDirective(json_t *payload);

This function is used to process SetAlert directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

int avsAlerts_is_alert_waiting();

This function is used to give waiting alert to player for AVS.

Return:

Success or failure value.

Function

error_e avsProcessDeleteAlertDirective(json_t *payload, int stream_id);

This function is used to process the delete alert directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsSendAlertsEvent(const char *name, const char *token, int run_as_thread);

This function is used to send an alert event to AVS.

Parameter:

name: Alert event name.

token: Opaque token provided by the SetAlert directive.

Return:

Error code.

Function

json_t *avsCapabilitiesAlertInterface();

This function is used to provide alert interface capabilities to AVS.

Return:

JSON-encoded context.

Function

void avsAlertCancel();

This function is used to cancel AVS alert.

Function

bool avsAlert_IsInited();

This function is to check AVS alert initialization.

Function

void avsAlert_DeleteAllEntries();

This function is to delete all AVS alert entries.

2.3.4. avs_audio_player

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_audio_player.h

Function

error_e avsInitAudioPlayer(void);

This function is used to initialize audio player interface.

Function

json_t *avsFormatAudioPlayerContext(char *token);

This function is used to format audio player context.

Return:

JSON-encoded context.

Function

error_e avsProcessAudioPlayerDirective(const char_t *name, json_t *payload,int *stop_processing);

This function is used to process audio player directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsPlayDirectiveProcess(json_t *payload,int *stop_processing);

This function is used to process the play directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessStopDirective(json_t *payload);

This function is used to process stop directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessClearQueueDirective(json_t *payload);

This function is used to process clear queue directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

void AudioPlayer_Notify(char *token,int msg);

This function is used to provide audio player notification.

parameter:

token: For stream.

msg: For message ID.

Function

json_t *avsCapabilitiesAudioPlayerInterface();

This function is used to provide audio player capabilities interface for AVS.

Return:

Json context.

2.3.5. avs_audio_in

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_audio_in.h

Function

error_e audioInInit(AudioInContext *inContext);

This function is used to initialize audio in.

Parameter:

context: Pointer to the audio in context.

Return:

Error code.

Function

size_t audioReadInStream(AudioInContext *inContext,AudioMonoPcmSample *data, size_t length);

This function is used to read input from audio stream.

Parameter:

param: Pointer to the audio in context.

data: Buffer where to copy the PCM samples (16-bit, mono, 16kHz).

Number: Number of PCM samples to be read.

Return:

Error code.

Function

size_t audioWriteInStream(AudioInContext *inContext,AudioMonoPcmSample *data, size_t write_len);

This function is used to write input to audio stream.

Parameter:

param: Pointer to the audio in context.

data: Buffer of the PCM samples (16-bit, mono, 16kHz).

Number: Number of PCM samples to write.

Return:

Error code.

Function

error_e audioFlushInStream(AudioInContext *inContext);

This function is used to flush input from audio stream.

Parameter:

param: Pointer to the audio in context.

Return:

Error code.

Function

error_e audioClearInStream(AudioInContext *inContext,size_t reqlen);

This function is used to clear data input from audio stream.

Parameter:

param: Pointer to the audio in context.

Number: Number of PCM samples to be cleared.

Return:

Error code.

Function

void audioInCopyBuffer(AudioInContext *context);

This function is used to copy data input from audio stream.

Parameter:

param: Pointer to the audio in context.

2.3.6. avs_capabilities

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_capabilities.h

Function

json_t *avsCapabilitiesGeneric(char *type, char *interface, char* version);

This function is used to provide generic capabilities to AVS.

Parameter:

type: To provide types of capabilities.

interface: To give interface type.

version: For capabilities version.

Return:

Json context.

Function

void avsCapabilitiesSendMsg();

This function is used to send generic capabilities message for AVS.

Function

void avsCapabilitiesInit();

This function is used to initialize generic capabilities for AVS.

Function

json_t *avsCapabilitiesGeneric(char *type, char *interface, char* version);

This function is used to configure generic capabilities for AVS.

Parameter:

type: To provide types of capabilities.

interface: To give interface type.

version: For capabilities version.

Return:

Json context.

Function

char* avsCapabilitiesMsgCreate(void);

This function is used to create generic capabilities message for AVS.

Return:

Created message.

Function

void avsCapabilityApiSend(AvsContext *context);

This API is used to send generic capabilities for AVS.

Parameter:

context: AVS capabilities context.

2.3.7. avs_directives

Header file

Examples/project/aw7698_evk/inc/avsClient/inc/avs_directives.h

Function

void avsDirectiveTask(void *param);

This function is used to schedule directive task for AVS.

Function

error_e avsProcessDirective(const char_t *message, json_t *root,int stream_id, int *stop_processing);

This function is used to process AVS directive.

Parameter:

message: JSON-encoded string that holds the directive.

root: JSON-encoded string.

stream_id: Stream ID.

stop_processing: Stop processing command.

Return:

Error code.

Function

error_e avsProcessException(uint_t statusCode, const char_t *message);

This function is used to process AVS exception.

Parameter:

statusCode: HTTP status code.

message: JSON-encoded string that holds the exception.

Return:

Error code.

2.3.8. avs_events

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_events.h

Function

error_e avsSendEvent(const char_t *message, int run_as_thread, int *http_status_code, char *name);

This function is used to send an event to AVS.

Parameter:

message: NULL-terminated string that containes the event to be sent.

Return:

Error code.

Function

void avsEventProcInit();

This function is used to initialize proc event for AVS.

Function

void avsEventInit();

This function is used to initialize events for AVS.

Function

void avs_event_printf(char* info, int stream_id,char* msg);

This function is used to print event for AVS.

Parameter:

info: Information of a message.

stream_id: Stream ID type of a message.

msg: Message string.

2.3.9. avs_expect_speech

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_expect_speech.h

Function

void avsExpectSpeechInit();

This function is used to initialize expected speech for AVS.

Function

void expectSpeechTimerCallback(void const *arg);

This function is used to provide timer call back for expected speech.

Function

void avsProcExpectSpeechTimeout(void);

This function is used to provide timer call back for proc expected speech.

Function

void avsStartExpectSpeechTimeoutTimer();

This function is used to start expected speech timeout timer for AVS.

Function

void avsStopExpectSpeechTimeoutTimer();

This function is used to stop expected speech timeout timer for AVS.

Function

int avsExpectSpeechProc();

This function is used to call expected speech proc for AVS.

Function

error_e avsProcessExpectSpeechDirective(json_t *payload);

This function is used to process expected speech directive for AVS.

2.3.10. avs_http2

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_http2.h

Function

void avs_http2_deliver_data(int stream_id, uint8_t *data, size_t len);

This function is used to deliver data through http2 for AVS.

Parameter:

stream_id: For data stream ID.

data: For data stream.

len: Data stream length.

Function

int avs_http2_wait_for_data(int stream_id);

This function is used to wait for data through http2 for AVS.

Parameter:

stream_id: For data stream ID.

Return:

AVS data wait status.

Function

void avs_http2_read_all_data_in_stream(int stream_id);

This function is used to read all the data in stream for AVS.

Parameter:

stream_id: For data stream ID.

Function

void avs_http2_deliver_header_vals(int stream_id, uint8_t *name, size_t nameLen, uint8_t *value, size_t valueLen);

This function is used to deliver the data header value through http2 for AVS.

Parameter:

stream_id: For data stream ID.

name: Type of the data stream.

namelen: Type string length.

value: Stream value.

valuelen: Stream value length.

Function

void avs_http2_deliver_header(int stream_id);

This function is used to deliver data header through http2 for AVS.

Parameter:

stream_id: For data stream ID.

Function

int avs_http2_wait_for_header(int stream_id, int *len);

This function is used to wait for header through http2 for AVS.

Parameter:

stream_id: For data stream ID.

len: Data stream length.

Return:

Status value.

Function

void avs_http2_deliver_close(int stream_id);

This function is used to deliver the data close command through http2 for AVS.

Parameter:

stream_id: For data stream ID.

Function

void avs_http2_wait_for_close(int stream_id);

This function is used to deliver the data wait close command through http2 for AVS.

Parameter:

stream_id: For data stream ID.

Function

void avs_http2_print_streams();

This function is used to print data stream through http2 for AVS.

Function

void avs_http2_add(int stream_id, void *buff, char *name, int len);

This function is used to add data through http2 for AVS.

Parameter:

stream_id: For data stream ID.

buff: Add the stream.

name: Stream data name.

len: Data stream length.

Function

void avs_http2_cleanup();

This function is used to cleanup the data for AVS.

Function

void avs_http2_remove_entry(int stream_id, conn_info_t *info);

This function is used to remove the data entry through http2 for AVS.

Parameter:

stream_id: For data stream ID.

info: Remove entry info.

Function

void avs_http2_init();

This function is used to initialize the parameter through http2 for AVS.

Function

void avs_http2_wakeup_directive_task(int stream_id);

This function is used to deliver wakeup directive talk through http2 for AVS.

Parameter:

stream_id: For data stream ID.

Function

void avs_http2_orphan_stream(int stream_id);

This function is used to create orphan stream through http2 for AVS.

Parameter:

stream_id: For data stream ID.

2.3.11. avs_interactionModel

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_InteractionModel.h

Function

error_e avsProcessInteractionModelDirective(const char_t *name, json_t *payload);

This function is used to process the interaction model directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

json_t *avsCapabilitiesInteractionModelInterface();

This function is used to interface AVS capabilities interaction model.

Return:

Json context.

2.3.12. avs_io

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_io_events.h

Function

void avs_io_init();

This API is used to initialize input/output for AVS.

Function

void avs_io_de_init();

This API is used to deinitialize input/output for AVS.

Function

void avs_io_check_and_trigger();

This API is used to check and trigger the input/output for AVS.

Function

void avs_io_proc(int avs_sock, int delay);

This API is used to provide proc command to input/output for AVS.

2.3.13. avs_io_events

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_io_events.h

Function

void notify_avs_events(AvsState avs_events);

This function is used to notify AVS event.

Parameter:

avs_events : AVS event state.

Function

void init_avs_event_semaphore(void);

This function is used to initialize AVS event semaphore.

Function

int subscribe_for_avs_events(avs_events_callback_t cb);

This function is used to subscribe AVS event.

Parameter:

cb: AVS event callback.

Return:

AVS subscribe event status.

Function

void unsubscribe_for_avs_events(avs_events_callback_t cb);

This function is used to unsubscribe AVS event.

Parameter:

cb: AVS event callback.

2.3.14. avs_lib

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_lib.h

Function

error_e strSafeCopy(char_t *, const char_t *, size_t);

This function is used to copy the string.

Parameter:

dest: Pointer to the destination string.

src: Pointer to the source string.

destSize: Size of the buffer allocated for the destination string.

Return:

Error code.

Function

int parse_uri(struct URI*, const char*);

This function is used for parsing URI value.

Parameter:

uri: URI struct to update.

url: URI value stream.

Return:

Success of failure.

Function

void request_free(struct Request *req);

Request to free the structure.

Parameter:

req: Request value.

Function

void request_init(struct Request *req, struct URI *uri);

This function is used to request initialization.

Parameter:

req: Request value.

uri: URI struct to update.

Function

void print_request(struct Request *req);

This function is used to request for print.

Parameter:

req: Request value.

2.3.15. avs_misc_cmds

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_misc_cmds.h

Function

void avsSendMiscCmd(AVS_MISC_CMD cmd);

This function is used to send misc command to AVS.

Parameter:

cmd: Misc command for AVS.

Function

void avsMiscCmdQueueInit();

This API is used to initialize queue and send misc command for AVS.

Function

bool avsMiscCmdCheckAndProc();

This API is used to check and provide proc command for AVS misc.

Return:

Check status for AVS misc command.

2.3.16. avs_ping

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_ping.h

Function

error_e avsPingTick(systime_t time_now);

This function is used to ping AVS tick.

Parameter:

time_now: System current time.

Return:

Error code.

Function

error_e avsSendPingRequest();

This function is used to send request to ping.

Return:

Error code.

2.3.17. avs_playback_controller

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_playback_controller.h

Function

void avsPlaybackController_PlayCommandIssued();

AVS playback controller play command.

Function

void avsPlaybackController_PauseCommandIssued();

AVS playback controller pause command.

Function

void avsPlaybackController_NextCommandIssued();

AVS playback controller next command.

Function

void avsPlaybackController_PreviousCommandIssued();

AVS playback controller previous command.

Function

void avsPlaybackControllerInit();

AVS playback controller initialization.

Function

json_t *avsCapabilitiesPlaybackControllerInterface();

AVS playback controller capabilities initialization.

2.3.18. avs_reply_parser

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_reply_parser.h

Function

void print_initial_data(char *buff, int len);

This function is used to print initial parser data.

Parameter:

buff: Data buffer.

len: Length of data buffer.

Function

int proc_msg_from_avs(int stream_id, char *buffer, int length, int *tts_present);

This function is used to proc message from AVS.

Parameter:

stream_id: Data stream ID.

buff: Data buffer.

len: Length of data buffer.

tts_present: tts present stream.

Return:

Status of message from AVS.

Function

void avs_consume_trailing_data(int stream_id, int statusCode, char *buff, const char *caller);

This function is used to consume trailing data for AVS.

Parameter:

stream_id: Data stream ID.

statusCode: Status of the code.

buff: Data buffer.

caller: Caller stream.

Function

int proc_reply_from_avs(int statusCode, const char *caller, int stream_id, char* buff, int max_len, int *tts_present);

This function is used for proc reply from AVS.

Parameter:

stream_id: Data stream ID.

statusCode: Status of the code.

buff: Data buffer.

caller: Caller stream.

max_len: Max length of a buffer.

tts_present: tts present stream value.

Return:

AVS proc reply status.

2.3.19. avs_speech_timeout

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_speech_timeout.h

Function

void speechTimeoutTimerCallback(void const *arg);

Speech timeout timer callback.

Function

int speechTimeoutStatusGet();

This API is used to get speech time out status.

Return:

Status value.

Function

void avsSpeechTimeoutCtxInit();

Speech timeout context initialization.

Function

void avsSpeechTimeoutCtxDeInit();

Speech timeout timer context deinitialization.

Function

void avsStartSpeechTimeoutTimer(int timeo);

Starts speech timeout timer.

Parameter:

timeo: Timer value.

Function

void avsStopSpeechTimeoutTimer();

Stop speech timeout timer.

Function

int speechTimeoutStatusSet(int val);

Set speech timeout timer status.

Parameter:

val: Timer status set value.

Return:

Speech time out status.

2.3.20. avs_system

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_system.h

Function

error_e avsProcessSystemDirective(const char_t *name, json_t *payload);

Process System directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessResetUserInactivityDirective(json_t *payload);

Process Reset User Inactivity directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsSendSynchronizeStateEvent(void);

Send synchronize state event.

Return:

Error code.

Function

char const *avsSystemGetFirmwareVersion();

Get AVS system firmware version.

Return:

Firmware version.

Function

void avsSystemBootComlpete();

AVS system boot complete.

Function

systime_t avsSystemGetLastActivityTime();

This function is used to get the last activity time from AVS.

Return:

System time.

Function

void avsSystemSetLastActivityTime(systime_t time);

This function is used to set the last activity time from AVS.

Parameter:

time: System time.

Function

void avsSystemUpdateLastActivityTime();

This function is used to update the last activity time.

Function

json_t *avsCapabilitiesSystemInterface();

This function is used to interface AVS system capabilities.

Return:

JASON-Encoded context.

Function

json_t *avsCapabilitiesApiGateway();

This function is used to interface AVS API gateway capabilities.

Return:

JASON-Encoded context.

Function

error_e avsProcessApiGatewayDirective(const char_t *name, json_t *payload);

This function is used to process AVS API Gateway directive.

Parameter:

name: Directive name.

payload: Directive payload stream.

Return:

Error code.

Function

error_e avsSendSystemEvent(char* namespace, char *name, char *data, uint8_t run_as_thread);

This function is used to send AVS system event.

Parameter:

namespace: Send system event.

name: Send system event name.

data: Send data.

run_as_thread: Data send method.

Return:

Error code.

2.3.21. date_time

Header File

Examples/project/aw7698_evk/inc/avsDep/date_time.h

Function

const char_t *formatSystemTime(systime_t time, char_t *str);

This function is used to format system time.

Parameter:

time: System time.

str: NULL-terminated string representing the specified time.

Return:

Pointer to the formatted string.

Function

const char_t *formatDate(const DateTime *date, char_t *str);

This function is used to format date.

Parameter:

date: Pointer to a structure representing the date.

str: NULL-terminated string representing the specified date.

Return:

Pointer to the formatted string.

Function

void getCurrentDate(DateTime *date);

This function is used to get current date and time.

Parameter:

date: Pointer to a structure representing the date and time.

Function

time_t getCurrentUnixTime(void);

This function is used to get current time.

Return:

Unix timestamp.

Function

void convertUnixTimeToDate(time_t t, DateTime *date);

This function is used to convert Unix timestamp to date.

Parameter:

t: Unix timestamp.

date: Pointer to a structure representing the date and time.

Function

time_t convertDateToUnixTime(const DateTime *date);

This function is used to convert date to Unix timestamp.

Parameter:

date: Pointer to a structure representing the date and time.

Return:

Unix timestamp.

Function

uint8_t computeDayOfWeek(uint16_t y, uint8_t m, uint8_t d);

This function is used to calculate day of week.

Parameter:

y: Year.

m: Month of year (in range 1 to 12).

d: Day of month (in range 1 to 31).

Return:

Day of week (in range 1 to 7).

2.3.22. avs_speaker

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_speaker.h

Function

error_e avsInitSpeaker(void);

Initialize Speaker interface.

Function

json_t *avsFormatSpeakerContext(void);

Format Speaker context.

Return:

JSON-encoded context.

Function

error_e avsProcessSpeakerDirective(const char_t *name, json_t *payload);

Process Speaker directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessSetVolumeDirective(json_t *payload);

Process set volume directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessAdjustVolumeDirective(json_t *payload);

Process adjust volume directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessSetMuteDirective(json_t *payload);

Process set mute directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

void avsSendMuteChangedEvent(bool_t mute);

This function is used to change AVS mute event.

Parameter:

mute: Mute change event flag.

Function

json_t *avsCapabilitiesSpeakerInterface();

This function is used to change AVS speaker capability.

Return:

Json context.

2.3.23. avs_speech_recognizer

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_speech_recognizer.h

Function

error_e avsInitSpeechRecognizer(void);

Initialize speech recognizer interface.

Function

error_e avsProcessSpeechRecognizerDirective(const char_t *name, json_t *payload);

Process speech recognizer directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessStopCaptureDirective(json_t *payload);

Process stop capture directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsSendRecognizeEvent(void);

Send a recognize event to AVS.

Return:

Error code.

Function

void avsStopSendingMicData(bool unexpected_wakeup);

Stop sending AVS mic data.

Parameter:

unexpected_wakeup: Unexpected wakeup command value.

Function

char_t *avsFormatRecognizeEvent(void);

Format recognize event.

Return:

JSON-encoded string that contains the recognize event.

Function

error_e avsSendSpeechRecognizerEvent(const char_t *name);

This API is used to send AVS speech recognize event.

Parameter:

name: Name of the event.

Return:

JSON-encoded string that contains the Recognize event.

Function

char *avsExpectSpeechTokenGet(void);

Get AVS expect speech token.

Return:

Token stream.

Function

char *avsExpectSpeechInitiatorTypeGet(void);

Get AVS expect speech type initiator.

Return:

Token stream value.

Function

json_t *avsCapabilitiesSpeechRecognizerInterface();

AVS speech capabilities recognition interface.

Return:

JSON-encoded string that contains the Recognize event.

2.3.24. avs_speech_synthesizer

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_speech_synthesizer.h

Function

error_e avsInitSpeechSynthesizer(void);

Initialize speech synthesizer interface.

Function

json_t *avsFormatSpeechSynthesizerContext(void);

Format speech synthesizer context.

Return:

JSON-encoded context.

Function

error_e avsProcessSpeechSynthesizerDirective(const char_t *name, json_t *payload);

Process speech synthesizer directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

json_t *avsCapabilitiesSpeechSynthesizerInterface();

AVS speech synthesizer capabilities interface.

Return:

JSON-Encoded context.

Function

error_e avsProcessSpeakDirective(json_t *payload);

Process speak directive.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsSendSpeechSynthesizerEvent(const char_t *name);

Send a speech synthesizer event to AVS.

Parameter:

name: SpeechSynthesizer event name (SpeechStarted or SpeechFinished)

Return:

Error code.

Function

int avsProcessSpeakAudioData(int stream_id,char *boundary,char* buffer, int *first_data_offset_ptr, int first_data_len, int *len, int *rxDataLen_ptr);

AVS speak audio data process.

Parameter:

stream_id: Stream ID of data.

boundry: Given stream.

buffer: To store the data stream.

first_data_offset_ptr: Data offset.

first_data_len: Length of the data.

len: Length of the received data.

rxDataLen_ptr: Received data length.

Return:

AVS process speak audio data status.

2.3.25. avs_time_sync

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_time_sync.h

Function

error_e avsSetNtpServer(AvsContext *context, const char_t *name);

Set NTP server name.

Parameter:

name: NULL-terminated string that holds the NTP server name.

Return:

Error code.

Function

error_e avsSynchronizeTime(AvsContext *context);

Synchronize internal clock using NTP.

Return:

Error code.

Function

time_t avsGetCurrentTime(void);

Get current time.

Return:

Current Unix time.

2.3.26. avs_notification

Header File

Examples/project/aw7698_evk/inc/avsClient/inc/avs_ping.h

Function

error_e AvsProcessAssetObject(json_t *payload);

Process Asset Object directives.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessSetIndicator(json_t *payload);

Process set indicator directives.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

error_e avsProcessClearIndicator(json_t *payload);

Process clear indicator directives.

Parameter:

payload: Payload part of the directive.

Return:

Error code.

Function

json_t *avsFormatNotificationContext(void);

Format notification context.

Return:

JSON-encoded context.

Function

error_e avsProcessNotificationDirective(const char_t *name, json_t *payload);

Process notification directives.

Parameter:

name: NULL-terminated string that holds the directive name.

payload: Payload part of the directive.

Return:

Error code.

Function

void initNotiTimer(void);

Notification timer initialization.

Function

void avsNotificationInit();

AVS notification initialization.

Function

json_t *avsCapabilitiesNotificationsInterface();

AVS capabilities notification interface initialization.

Return:

Json context.

2.4. Mobile_App

This section describes the Wi-Fi configuration using mobile application.

2.4.1. app_sec_config

Header File

Examples/project/aw7698_evk/inc/avsDep/inc/app_sec_config.h

Function

int app_sec_init(app_sec_conf *conf);

This function is used to initialize parameters for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

Return:

SUCCESS or FAILURE.

Function

Int_app_sec_set_retry_delay(app_sec_conf *conf, int delay);

This function is used to provide retry delay for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

delay: Count of delay.

Return:

SUCCESS or FAILURE.

Function

Int_app_sec_set_source(app_sec_conf *conf, app_sec_src source);

This function is used to provide source event for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

source: Source event.

Return:

SUCCESS or FAILURE.

Function

Int_app_sec_set_max_retry(app_sec_conf *conf, unsigned char max_retry);

This function is used to provide maximum retry count for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

max_retry: Max retry count.

Return:

SUCCESS or FAILURE.

Function

Int_app_sec_set_fatal_on_fail(app_sec_conf *conf, unsigned char fatal_on_fail);

This function is used to provide fatal fail count for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

fatal_on_fail: Fatal fail count.

Return:

SUCCESS or FAILURE.

Function

int_app_sec_set_authmode(app_sec_conf *conf, app_sec_authmode authmode);

This function is used to set authentication mode for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

authmode: To set authentication mode.

Return:

SUCCESS or FAILURE.

Function

Int_app_sec_set_skipverify(app_sec_conf *conf, unsigned char skip_verify);

This function is used to set skip verification for AVS task.

Parameter:

conf: App sec configuration initialization for structure app_sec_conf.

skip_verify: To set verification mode.

Return:

SUCCESS or FAILURE.

2.4.2. ut_app

Header File

Examples/project/aw7698_evk/inc/netConfig/inc/ut_app.h

Function

void copy_str_to_addr(uint8_t *addr, const char *str);

Copy string to address.

Parameter:

addr: Address where string needs to copy.

str: String value to be copied.

Function

bt_status_t bt_app_io_callback(void *input, void *output);

This API is a callback function for Bluetooth app.

Parameter:

input: bt app input stream.

output: bt app output stream.

Return:

bt app io callback status.

2.4.3. discovery

Header File

Examples/project/aw7698_evk/inc/libreApp/discovery.h

Function

void start_discovery_thread(void);

This function is used to start discovery thread.

Function

void notify_discovery(char ch);

This function is used to notify discovery.

2.4.4. netinit

Header File

Examples/project/aw7698_evk/inc/netInit/inc/netinit.h

Function

void LibreNetWorkInit(void *args);

This function is used to schedule Libre mobile application task.

2.4.5. ssl_client

Header File

Examples/project/aw7698_evk/inc/avsExt/ssl_client.h

Function

void shutdown_and_free_ssl(void* ptr);

This function is used to shut down and free SSL.

Function

int get_ssl_fd(void* ptr);

This function is used to get SSL fd (file descriptor).

Return:

ssl get status.

Function

void* init_ssl_and_connect(char *addr, char* port, char* alpn_string, char *hostname, const unsigned char *pem, int pemlen, app_sec_conf *app_conf);

This function is used to initialize and connect with SSL.

Parameter:

addr: ssl address.

port: ssl port.

alpn_string: alpn string.

hostname: ssl hostname.

Function

int new_ssl_read(void*ptr, char *buf, int len);

This function is used to read new SSL connection.

Parameter:

ptr: ssl stream ptr.

buf: ssl stream data.

len: ssl stream data length.

Return:

ssl read new status.

Function

int new_ssl_write(void*ptr, char *buf, int len);

This function is used to write new SSL connection.

Parameter:

ptr: ssl stream ptr.

buf: ssl stream data.

len: ssl stream data length.

Return:

ssl write new status.

Function

void sslClientThreadFn(void *arg);

This function is used to schedule SSL client thread.

2.5. Command Line Interface

This section describes the command line interface for device communication.

2.5.1. cli_def

Header File

Examples/middleware/MTK/minicli/inc/cli_def.h

Function

void cli_def_create(void);

This function is used to create CLI definition.

Function

void cli_def_task(void *param);

This function is used to schedule CLI task.

Function

int cli_task_create(void);

This function is used to create CLI task.

2.6. IO (BUTTON/LED)

This section describes the input/output interface for onboard button and led.

2.6.1. bsp_led

Header File

Examples/driver/board/aw7698_evk/led/bsp_led.h

Function

bsp_led_status_t bsp_led_init(uint32_t led_number);

This function is to initialize LED setting.

Parameter:

led_number: Initializes the specified LED number.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

Function

bsp_led_status_t bsp_led_set_breath(uint32_t led_number, bsp_led_config_t *config);

This function is to configure LED to breath mode.

Parameter:

led_number: Initializes the specified LED number.

config: Specifies a user defined LED parameter.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

Function

bsp_led_status_t bsp_led_set_blink(uint32_t led_number, bsp_led_config_t *config);

This function is to configure LED to Blink mode.

Parameter:

led_number: Initializes the specified LED number.

config: Specifies a user defined LED parameter.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

Function

bsp_led_status_t bsp_led_start(uint32_t led_number);

This function is to start LED twinkle.

Parameter:

led_number: Initializes the specified LED number.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

Function

bsp_led_status_t bsp_led_stop(uint32_t led_number);

This function is to stop LED twinkle.

Parameter:

led_number: Initializes the specified LED number.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

Function

bsp_led_status_t bsp_led_deinit(uint32_t led_number);

This function is to de-initial LED.

Parameter:

led_number: Initializes the specified LED number.

Return:

#BSP_LED_OK, if the operation completed successfully.

#BSP_LED_INVALID_PARAMETER, if parameter is invalid.

2.7. IR Remote

This section describes the IR remote interface of the device.

2.7.1. ir_emulator

Header File

Examples/project/aw7698_evk/inc/irController/ir_emulator.h

Function

int send_ir(int16_t repeat_count, uint16_t *data);

This function is used to send IR code.

Parameter:

repeat_count: Repeat count.

data: IR send data stream.

Return:

Send status.

2.7.2. ir_controller

Header File

Examples/project/aw7698_evk/inc/irController/ir_controller.h

Function

IRMGR_error IRMGR_sendEvents(IR_ID_Type IdType,char* payload);

This function is used to send event for IR.

Parameter:

IdType: ID type of a message.

payload: Received payload.

Return:

Error code.

Function

IRMGR_error IRMGR_sendbutton(char *payload);

This function is used to send button event for IR.

Parameter:

payload: Received payload.

Return:

Error code.

Function

IRMGR_error IRMGR_sendkey(int rid, int appliance, char *data);

This function is used to send key event for IR.

Parameter:

rid: Remote ID.

appliance: Appliance type.

data: Received data.

Return:

Error code.

Function

IRMGR_error IRMGR_sendcode(int repeat, char *ir_code);

This function is used to send event code for IR.

Parameter:

repeat: Repetition count.

ir_code: IR code stream.

Return:

Error code.

2.8. HAL

This section describes the HAL driver interface for device peripherals.

2.8.1. hal_i2c_master

Header file

Examples/driver/chip/inc/hal_i2c_master.h

Function

hal_i2c_status_t hal_i2c_master_init(hal_i2c_port_t i2c_port, hal_i2c_config_t *i2c_config);

This function initializes the I2C master before starting a transaction.

Parameter:

i2c_port: It is the I2C master port number.

i2c_config: It is the configuration parameter to initialize the I2C.

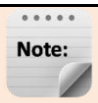
Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an invalid transfer_frequency is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.



#hal_i2c_master_deinit () must be called when the I2C is no longer in use, release the I2C resource for the other users to use this I2C master.

Function

hal_i2c_status_t hal_i2c_master_deinit(hal_i2c_port_t i2c_port);

This function releases the I2C master once the transaction is over. Call this function, if the I2C is no longer in use.

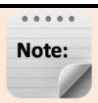
Parameter:

i2c_port: I2C master port number.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.



This function must be called when the I2C is no longer in use, release the I2C resource for the other users to use this I2C master.

Function

hal_i2c_status_t hal_i2c_master_set_frequency(hal_i2c_port_t i2c_port, hal_i2c_frequency_t frequency);

This function sets the transaction speed. Apply this function to change the transaction speed without using #hal_i2c_master_init().

Parameter:

i2c_port: I2C master port number.

frequency: Is an enum value defined in #hal_i2c_frequency_t. Only value of type #hal_i2c_frequency_t is accepted.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an invalid speed is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_set_io_config(hal_i2c_port_t i2c_port, hal_i2c_io_config_t io);

This function sets the SCL/SDA IO config.

Parameter:

i2c_port: I2C master port number.

io: Is an enum value defined in #hal_i2c_io_config_t. Only value of type #hal_i2c_io_config_t is accepted.

Return:

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR, the operation is failed.

Function

hal_i2c_status_t hal_i2c_master_register_callback(hal_i2c_port_t i2c_port, hal_i2c_callback_t i2c_callback, void *user_data);

This function registers a callback function while using DMA mode.

The callback function will be called at I2C ISR routine after the I2C triggers an interrupt.

Always call this function to register a callback function while using DMA mode.

Parameter:

i2c_port: I2C master port number.

i2c_callback: It is the user-defined callback function called at I2C ISR routine.

user_data: It is a user-defined input data returned during the callback function's call.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL function pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully.

Function

hal_i2c_status_t hal_i2c_master_send_polling(hal_i2c_port_t i2c_port, uint8_t slave_address, const uint8_t *data, uint32_t size);

This function sends data to I2C slave in polling mode.

This function will not return until the transaction is complete.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

data: Data buffer to send.

size: Data size to send. The maximum value is #HAL_I2C_MAXIMUM_POLLING_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_dma(hal_i2c_port_t i2c_port, uint8_t slave_address, const uint8_t *data, uint32_t size);

This function sends data to I2C slave in DMA mode. This function returns a value immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

data: Data buffer to send.

size: Data size to send. The maximum value is #HAL_I2C_MAXIMUM_DMA_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_polling(hal_i2c_port_t i2c_port, uint8_t slave_address, uint8_t *buffer, uint32_t size);

This function receives data from I2C slave in a polling mode.

This function will not return a value until the transaction is finished.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

buffer: Data buffer to receive.

size: Data size to receive. The maximum value is #HAL_I2C_MAXIMUM_POLLING_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_dma(hal_i2c_port_t i2c_port, uint8_t slave_address, uint8_t *buffer, uint32_t size);

This function receives data from I2C slave in a DMA mode. This function will return a value immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

buffer: Data buffer to receive.

size: Data size to receive. The maximum value is #HAL_I2C_MAXIMUM_DMA_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_polling(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_t *i2c_send_to_receive_config);

This function sends data to and then receives data from I2C slave in a polling mode. This function does not return until the transaction is finished.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config: Configuration parameter for this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_t *i2c_send_to_receive_config);

This function sends data to and then receives data from I2C slave in a DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config: Configuration parameter for this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_send_config_t *i2c_send_config);

This function sends data to I2C slave in DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_config: Configuration parameter of this API for send.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_receive_config_t *i2c_receive_config);

This function receives data from I2C slave in an extended DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_receive_config: Configuration parameter of this API for receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_ex_t *i2c_send_to_receive_config_ex);

This function sends data to and then receives data from I2C slave in extended DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config_ex: Configuration parameter of this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma_ex_none_blocking(hal_i2c_port_t i2c_port,hal_i2c_send_to_receive_config_ex_no_busy_t*i2c_send_to_receive_config_no_busy_ex);

This function sends data to and then receives data from I2C slave in an extended DMA mode with software FIFO.

This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

hal_i2c_send_to_receive_config_ex_no_busy_t: Configuration parameter of this API for both send and receive.

Return:

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR, the I2C sw FIFO is full.

Function

hal_i2c_status_t hal_i2c_master_get_running_status(hal_i2c_port_t i2c_port, hal_i2c_running_status_t *running_status);

This function gets running status of the I2C master. Call this function to check if the I2C is idle or not before transferring data. If it's not idle, then the resource is currently in use, delay the operation until the I2C is idle.

Parameter:

i2c_port: I2C master port number.

running_status:

#HAL_I2C_STATUS_BUS_BUSY, the I2C master is in busy status;

#HAL_I2C_STATUS_IDLE, the I2C master is in idle status; User can use it to transmit data immediately.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

2.8.2. hal_i2s

Header File

Examples/driver/chip/inc/hal_i2s.h

Function

hal_i2s_status_t hal_i2s_init(hal_i2s_initial_type_t i2s_initial_type);

This function initializes the I2S hardware type.

Parameter:

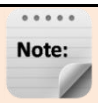
i2s_initial_type: Initial configuration parameter.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if input parameter is invalid.

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.



Set i2s_initial_type as HAL_I2S_TYPE_EXTERNAL_MODE when using external codec component in the application.

Function

hal_i2s_status_t hal_i2s_deinit(void);

This function deinitializes the I2S hardware.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_set_config(const hal_i2s_config_t *config);

This function sets the I2S configuration details.

Parameter:

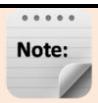
Config is the link configuration of the I2S module.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if wrong parameter is given.

#HAL_I2S_STATUS_ERROR, if one of the I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling #hal_i2s_set_config() during the I2S execution is not allowed.

Set the sampling rates of the TX and RX to the same value. Different values are not allowed.

Function

hal_i2s_status_t hal_i2s_get_config(hal_i2s_config_t *config);

This function will query the I2S configuration details.

Parameter:

Config is the link configuration of I2S module which is set in system.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the input parameter is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_disable_tx(void);

This function disables the I2S output link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_disable_rx(void);

This function disables the I2S input link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

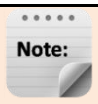
Function

hal_i2s_status_t hal_i2s_enable_audio_top(void);

This function enables uplink and downlink for FIFOs of the I2S link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function after #hal_i2s_enable_tx() or #hal_i2s_enable_rx().

Function

hal_i2s_status_t hal_i2s_disable_audio_top(void);

This function disables uplink and downlink for FIFOs of the I2S link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_enable_tx_dma_interrupt(void);

This function enables the TX VFIFO DMA interrupt.

Return:

#HAL_I2S_STATUS_ERROR, if TX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.

To avoid unpredictable system operation, calling this function is not allowed if TX callback function is not registered.

Function

hal_i2s_status_t hal_i2s_disable_tx_dma_interrupt(void);

This function disables the TX VFIFO DMA interrupt.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

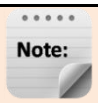
hal_i2s_status_t hal_i2s_enable_rx_dma_interrupt(void);

This function enables the RX VFIFO DMA interrupt.

Return:

#HAL_I2S_STATUS_ERROR, if RX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling this function is not allowed if RX callback function is not registered.

Function

hal_i2s_status_t hal_i2s_disable_rx_dma_interrupt(void);

This function disables the RX VFIFO DMA interrupt.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_setup_tx_vfifo(uint32_t *buffer, uint32_t threshold, uint32_t buffer_length);

This function will setup the transmit operation.

The FIFO starts to pump data from the TX VFIFO buffer into I2S TX, if the I2S TX is enabled.

VFIFO DMA will trigger an interrupt when the amount of output data in TX VFIFO is lower than the TX VFIFO threshold.

Parameter:

buffer: Pointer to memory buffer for the TX VFIFO.

threshold: Value of the TX VFIFO threshold.

buffer_length: Length to memory buffer for the TX VFIFO.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_INVALID_PARAMETER, if the buffer is a NULL pointer.

#HAL_I2S_STATUS_OK, if the operation is successful.

> **Note:** To avoid unpredictable system operation, calling #hal_i2s_setup_tx_vfifo() during the I2S execution is not allowed.
>
> Buffer is occupied by VFIFO until the transmission is complete.

Function

hal_i2s_status_t hal_i2s_setup_rx_vfifo(uint32_t *buffer, uint32_t threshold, uint32_t buffer_length);

This function will setup the receive operation.

The FIFO starts to pump data from I2S RX into the RX VFIFO buffer if the I2S RX is enabled.

VFIFO DMA will trigger an interrupt when the amount of available receive data in the RX VFIFO is larger than the RX VFIFO threshold.

Parameter:

buffer: is the pointer to memory buffer for the RX VFIFO.

threshold: is the value of the RX VFIFO threshold.

buffer_length: is the length of the memory buffer for the RX VFIFO.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_INVALID_PARAMETER, if the buffer is a NULL pointer.

#HAL_I2S_STATUS_OK, if the operation is successful.

> **Note:** To avoid unpredictable system operation, calling #hal_i2s_setup_rx_vfifo() during the I2S execution is not allowed.
>
> Buffer is occupied by VFIFO until the transmission is complete.

Function

hal_i2s_status_t hal_i2s_stop_tx_vfifo(void);

This function stops the transmit operation.

The FIFO will stop to pump data from the TX VFIFO buffer into I2S TX.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Function

hal_i2s_status_t hal_i2s_stop_rx_vfifo(void);

This function stops the receive operation.

The FIFO will stop to pump data from the I2S RX into the RX VFIFO buffer.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Function

hal_i2s_status_t hal_i2s_register_rx_vfifo_callback(hal_i2s_rx_callback_t rx_callback, void *user_data);

This function registers the callback function for input data.

Parameter:

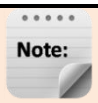
rx_callback: Callback function for the received data control.

user_data: User defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the rx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S RX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

hal_i2s_status_t hal_i2s_register_tx_vfifo_callback(hal_i2s_tx_callback_t tx_callback, void *user_data);

This function registers the callback function for output data.

Parameter:

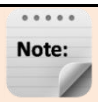
tx_callback: Pointer to the callback function to control data transmission.

user_data: User defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the tx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S TX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

hal_i2s_status_t hal_i2s_enable_tx(void);

This function enables the I2S output link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Function

hal_i2s_status_t hal_i2s_enable_rx(void);

This function enables the I2S input link.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Function

hal_i2s_status_t hal_i2s_tx_write(uint32_t data);

This function transmits data to the I2S output link.

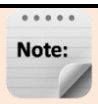
Parameter:

data: 32-bit output data to send, bit [31:16] means the data of right channel and bit [15:0] means the data of left channel.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.



Call this function after #hal_i2s_get_tx_sample_count () function to ensure there is enough free space in the TX VFIFO buffer for the write operation.

Function

hal_i2s_status_t hal_i2s_rx_read(uint32_t *data);

This function receives data from the I2S input link.

Parameter:

Data is the 32-bit data buffer to receive, bit [31:16] means the data of right channel and bit [15:0] means the data of left channel.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Note: Call this function after #hal_i2s_get_rx_sample_count() function to ensure there is data available in the RX VFIFO buffer to perform read operation.

Function

hal_i2s_status_t hal_i2s_get_tx_sample_count(uint32_t *sample_count);

This function queries the available free space in the TX VFIFO.

Parameter:

sample_count: Available free space in the TX VFIFO. (Number of data words to write)

Return:

#HAL_I2S_STATUS_ERROR, if the user does not configure the buffer length for TX VFIFO by #hal_i2s_setup_tx_vfifo ().

#HAL_I2S_STATUS_OK, if the operation is successful.

Note: Call this function before #hal_i2s_tx_write() function to ensure there is enough free space in the TX VFIFO buffer for the write operation.

Function

hal_i2s_status_t hal_i2s_get_rx_sample_count(uint32_t *sample_count);

This function queries the length of received data available in the RX VFIFO.

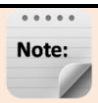
Parameter:

sample_count: Length of the received data available in the RX VFIFO.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.



Call this function before #hal_i2s_rx_read() function to ensure there is data available in the RX VFIFO buffer to perform read operation.

Function

hal_i2s_status_t hal_i2s_set_eof(void);

This function enables EOF event notification.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

The return value is reserved for further expansion.

Function

hal_i2s_status_t hal_i2s_init_ex(hal_i2s_port_t i2s_port, hal_i2s_initial_type_t i2s_initial_type);

This extended function initializes the I2S hardware type in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

i2s_initial_type: Initial configuration parameter.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if input parameter is invalid.

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.

Set i2s_initial_type as HAL_I2S_TYPE_EXTERNAL_MODE when using external codec component in the application.

User can use multiple I2S HW interface simultaneously with extended APIs.

Function

hal_i2s_status_t hal_i2s_deinit_ex(hal_i2s_port_t i2s_port);

This extended function deinitializes the I2S hardware in a multiple I2S mode.

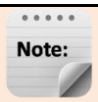
Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.



User can use multiple I2S HW interface simultaneously with extended APIs.

Function

hal_i2s_status_t hal_i2s_set_config_ex(hal_i2s_port_t i2s_port, const hal_i2s_config_t *config);

This extended function sets the I2S configuration details in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

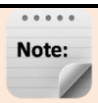
config: Link configuration of the I2S module.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if wrong parameter is given.

#HAL_I2S_STATUS_ERROR, if one of the I2S links is still available.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling #hal_i2s_set_config_ex() during the I2S execution is not allowed.

Set the sampling rates of the TX and RX to the same value. Different values are not allowed.

Function

hal_i2s_status_t hal_i2s_get_config_ex(hal_i2s_port_t i2s_port, hal_i2s_config_t *config);

This extended function queries the I2S configuration details in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

config: Link configuration of I2S module which is set in system.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the input parameter is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_enable_audio_top_ex(hal_i2s_port_t i2s_port);

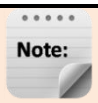
This extended function enables uplink and downlink FIFOs of the I2S link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function after #hal_i2s_enable_tx_ex() or #hal_i2s_enable_rx_ex().

Function

hal_i2s_status_t hal_i2s_disable_audio_top_ex(hal_i2s_port_t i2s_port);

This extended function disables uplink and downlink FIFOs of the I2S link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_enable_tx_dma_interrupt_ex(hal_i2s_port_t i2s_port);

This extended function enables the TX VFIFO DMA interrupt in a multiple I2S mode.

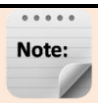
Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_ERROR, if TX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling this function is not allowed if TX callback function is not registered.

Function

hal_i2s_status_t hal_i2s_disable_tx_dma_interrupt_ex(hal_i2s_port_t i2s_port);

This extended function disables the TX VFIFO DMA interrupt in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_enable_rx_dma_interrupt_ex(hal_i2s_port_t i2s_port);

This extended function enables the RX VFIFO DMA interrupt in a multiple I2S mode.

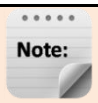
Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_ERROR, if RX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling this function is not allowed if RX callback function is not registered.

Function

hal_i2s_status_t hal_i2s_disable_rx_dma_interrupt_ex(hal_i2s_port_t i2s_port);

This extended function disables the RX VFIFO DMA interrupt in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_setup_tx_vfifo_ex(hal_i2s_port_t i2s_port, uint32_t *buffer, uint32_t threshold, uint32_t buffer_length);

This extended function sets up the transmit operation in a multiple I2S mode.

The FIFO starts to pump data from the TX VFIFO buffer into I2S TX if the I2S TX is enabled.

VFIFO DMA will trigger an interrupt when the amount of output data in TX VFIFO is lower than the TX VFIFO threshold.

Parameter:

i2s_port: I2S HW number.

buffer: Is the pointer to memory buffer for the TX VFIFO.

threshold: Is the value of the TX VFIFO threshold.

buffer_length: Is the length to memory buffer for the TX VFIFO.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_INVALID_PARAMETER, if the buffer is a NULL pointer.

#HAL_I2S_STATUS_OK, if the operation is successful.

To avoid unpredictable system operation, calling #hal_i2s_setup_tx_vfifo_ex() during the I2S execution is not allowed.

Function

hal_i2s_status_t hal_i2s_setup_rx_vfifo_ex(hal_i2s_port_t i2s_port, uint32_t *buffer, uint32_t threshold, uint32_t buffer_length);

This extended function sets up the receive operation in a multiple I2S mode.

The FIFO starts to pump data from I2S RX into the RX VFIFO buffer if the I2S RX is enabled.

VFIFO DMA will trigger an interrupt when the amount of available receive data in the RX VFIFO is larger than the RX VFIFO threshold.

Parameter:

i2s_port: I2S HW number.

buffer: Pointer to memory buffer for the RX VFIFO.

threshold: Value of the RX VFIFO threshold.

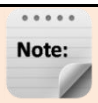
buffer_length: Length of the memory buffer for the RX VFIFO.

Return:

#HAL_I2S_STATUS_ERROR, if one of I2S links is still available.

#HAL_I2S_STATUS_INVALID_PARAMETER, if the buffer is a NULL pointer.

#HAL_I2S_STATUS_OK, if the operation is successful.



To avoid unpredictable system operation, calling #hal_i2s_setup_rx_vfifo_ex() during the I2S execution is not allowed.

Function

hal_i2s_status_t hal_i2s_stop_tx_vfifo_ex(hal_i2s_port_t i2s_port);

This extended function stops the transmit operation in a multiple I2S mode.

The FIFO will stop to pump data from the TX VFIFO buffer into I2S TX.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_stop_rx_vfifo_ex(hal_i2s_port_t i2s_port);

This extended function stops the receive operation in a multiple I2S mode.

The FIFO will stop to pump data from the I2S RX into the RX VFIFO buffer.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_register_rx_vfifo_callback_ex(hal_i2s_port_t i2s_port, hal_i2s_rx_callback_t rx_callback, void *user_data);

This extended function registers the callback function for input data in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

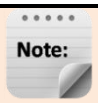
rx_callback: Callback function for the received data control.

user_data: User defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the rx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S RX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

hal_i2s_status_t hal_i2s_register_tx_vfifo_callback_ex(hal_i2s_port_t i2s_port, hal_i2s_tx_callback_t tx_callback, void *user_data);

This extended function registers the callback function for output data in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

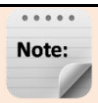
tx_callback: Pointer to the callback function to control data transmission.

user_data: User defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the tx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S TX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

hal_i2s_status_t hal_i2s_enable_tx_ex(hal_i2s_port_t i2s_port);

This extended function enables the I2S output link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_enable_rx_ex(hal_i2s_port_t i2s_port);

This extended function enables the I2S input link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_disable_tx_ex(hal_i2s_port_t i2s_port);

This extended function disables the I2S output link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_disable_rx_ex(hal_i2s_port_t i2s_port);

This extended function disables the I2S input link in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_tx_write_ex(hal_i2s_port_t i2s_port, uint32_t data);

This extended function transmits data to the I2S output link.

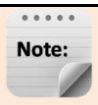
Parameter:

i2s_port: I2S HW number.

data: 32-bit output data to send, bit[31:16] means the data of right channel and bit[15:0] means the data of left channel.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function after #hal_i2s_get_tx_sample_count_ex() function to ensure there is enough free space in the TX VFIFO buffer for the write operation.

Function

hal_i2s_status_t hal_i2s_rx_read_ex(hal_i2s_port_t i2s_port, uint32_t *data);

This extended function receives data from the I2S input link.

Parameter:

i2s_port: I2S HW number.

data: 32-bit data buffer to receive, bit[31:16] means the data of right channel and bit[15:0] means the data of left channel.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Call this function after #hal_i2s_get_rx_sample_count_ex() function to ensure there is data available in the RX VFIFO buffer to perform read operation.

Function

hal_i2s_status_t hal_i2s_get_tx_sample_count_ex(hal_i2s_port_t i2s_port, uint32_t *sample_count);

This extended function queries the available free space in the TX VFIFO in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

sample_count: Available free space in the TX VFIFO. (Number of data words to write)

Return:

#HAL_I2S_STATUS_ERROR, if the user do not configure the buffer length for TX VFIFO by #hal_i2s_setup_tx_vfifo_ex().

#HAL_I2S_STATUS_OK, if the operation is successful.

> **Note:** Call this function before #hal_i2s_tx_write_ex() function to ensure there is enough free space in the TX VFIFO buffer for the write operation.

Function

hal_i2s_status_t hal_i2s_get_rx_sample_count_ex(hal_i2s_port_t i2s_port, uint32_t *sample_count);

This extended function queries the length of received data available in the RX VFIFO in a multiple I2S mode.

Parameter:

i2s_port: I2S HW number.

sample_count: Length of the received data available in the RX VFIFO.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

> **Note:** Call this function before #hal_i2s_rx_read_ex() function to ensure there is data available in the RX VFIFO buffer to perform read operation.

Function

hal_i2s_status_t hal_i2s_set_eof_ex(hal_i2s_port_t i2s_port);

This function enables EOF event notification.

Parameter:

i2s_port: I2S HW number.

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_get_memory_size(uint32_t *memory_size);

This function queries the size of the required memory to be allocated for an internal use in the I2S driver.

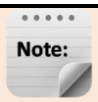
Parameter:

memory_size: Amount of memory required for the I2S driver for an internal use. (in bytes)

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the input parameter is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function at least before #hal_i2s_enable_tx() or #hal_i2s_enable_rx() to ensure there is enough memory for an internal use.

Function

hal_i2s_status_t hal_i2s_set_memory(uint8_t *memory);

This function submits the allocated memory to the I2S driver.

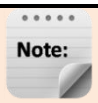
Parameter:

memory: Pointer to a memory.

Return:

#HAL_I2S_STATUS_ERROR, if an error occurred during the memory allocation in this function.

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function after user allocates enough memory greater than or equal to the size queried by #hal_i2s_get_memory_size() function.

Function

hal_i2s_status_t hal_i2s_get_memory_pointer(uint8_t **memory_pointer);

This function receives the pointer to the memory buffer.

Parameter:

memory_pointer: Pointer to an allocated memory previously provided by #hal_i2s_set_memory() function.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the input parameter is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.

Function

hal_i2s_status_t hal_i2s_register_rx_callback(hal_i2s_rx_callback_t rx_callback, void *user_data);

This function registers the callback function for input data.

Parameter:

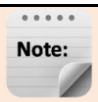
rx_callback: is a callback function for the received data control.

user_data is a user defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the rx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S RX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

hal_i2s_status_t hal_i2s_register_tx_callback(hal_i2s_tx_callback_t tx_callback, void *user_data);

This function registers the callback function for output data.

Parameter:

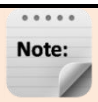
tx_callback: Pointer to the callback function to control data transmission.

user_data User defined input data.

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if the tx_callback is invalid.

#HAL_I2S_STATUS_OK, if the operation is successful.



Send I2S TX events to a user-defined task and handle the data transfer there to avoid longer performances in the NVIC ISR.

Function

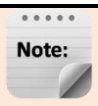
hal_i2s_status_t hal_i2s_enable_tx(void);

This function enables the I2S output link.

Return:

#HAL_I2S_STATUS_ERROR, if TX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.



Call #hal_i2s_register_tx_callback() to register a callback function before enabling the TX link to avoid unpredictable malfunctioning during the system execution.

Function

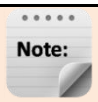
hal_i2s_status_t hal_i2s_enable_rx(void);

This function enables the I2S input link.

Return:

#HAL_I2S_STATUS_ERROR, if RX callback function is not registered.

#HAL_I2S_STATUS_OK, if the operation is successful.



Register RX link callback function by #hal_i2s_register_rx_callback() before enabling the RX link to avoid unpredictable situation occurred during the system execution.

Function

hal_i2s_status_t hal_i2s_tx_write(const void *buffer, uint32_t sample_count);

This function transmits data to the I2S output link.

Parameter:

buffer: Pointer to the output data.

sample_count: Available free space in the output buffer (in bytes).

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if input parameter sample_count is zero or greater than the size of the free space in the TX buffer.

#HAL_I2S_STATUS_ERROR, if an error occurred during data transmission.

#HAL_I2S_STATUS_OK, if the data transmission is complete.

Function

hal_i2s_status_t hal_i2s_rx_read(void *buffer, uint32_t sample_count);

This function receives data from the I2S input link.

Parameter:

buffer: Pointer to the user's data buffer.

sample_count: Number of received samples. (in bytes)

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if input parameter sample_count is zero or greater than the size of the data available in the RX buffer.

#HAL_I2S_STATUS_ERROR, when any error occurred during data read operation.

#HAL_I2S_STATUS_OK, if data read operation is complete.

Function

hal_i2s_status_t hal_i2s_get_tx_sample_count(uint32_t *sample_count);

This function queries the available free space for an output.

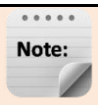
Parameter:

sample_count: Number of free samples available for an output (in bytes).

Return:

#HAL_I2S_STATUS_INVALID_PARAMETER, if a NULL pointer is given by user.

#HAL_I2S_STATUS_OK, if the operation is successful.



Call thin function before #hal_i2s_tx_write() function to ensure there is enough free space in the TX buffer for the write operation.

Function

hal_i2s_status_t hal_i2s_get_rx_sample_count(uint32_t *sample_count);

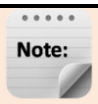
This function queries the available data for an input.

Parameter:

sample_count: Number of received samples (in bytes).

Return:

#HAL_I2S_STATUS_OK, if the operation is successful.



Call this function before #hal_i2s_rx_read() function to ensure there is data available in the RX buffer to perform read operation.

2.8.3. hal_gpio

Header file

Examples/driver/chip/inc/hal_gpio.h

Functions

hal_gpio_status_t hal_gpio_init(hal_gpio_pin_t gpio_pin);

This function initializes the GPIO hardware with basic functionality. The target pin must be initialized before use.

Parameter:

gpio_pin: Specifies the pin number to initialize.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_deinit(hal_gpio_pin_t gpio_pin);

This function deinitializes the GPIO hardware to its default status. The target pin must be deinitialized if not used.

Parameter:

gpio_pin: Specifies pin number to deinitialize.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_pinmux_status_t hal_pinmux_set_function(hal_gpio_pin_t gpio_pin, uint8_t function_index);

This function configures the pinmux of target GPIO.

Pin Multiplexer (pinmux) connects the pin and the onboard peripherals, hence the pin will operate in a specific mode once the pin is programmed to a peripheral's function.

The alternate functions of every pin are provided in hal_pinmux_define.h.

Parameter:

gpio_pin: Specifies the pin number to configure.

function_index: Specifies the function for the pin.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_PINMUX_STATUS_OK, the operation completed successfully.

If the return value is #HAL_PINMUX_STATUS_INVALID_FUNCTION, a wrong alternate function is given, the parameter must be verified.

If the return value is #HAL_PINMUX_STATUS_ERROR_PORT, invalid input pin number, the parameter must be verified.

If the return value is #HAL_PINMUX_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_get_input(hal_gpio_pin_t gpio_pin, hal_gpio_data_t *gpio_data);

This function gets the input data of target GPIO when the direction of the GPIO is input.

Parameter:

gpio_pin: Specifies the pin number to operate.

gpio_data: Input data received from the target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_output(hal_gpio_pin_t gpio_pin, hal_gpio_data_t gpio_data);

This function sets the output data of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to operate.

gpio_data: Output data of the target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_get_output(hal_gpio_pin_t gpio_pin, hal_gpio_data_t *gpio_data);

This function gets the output data of the target GPIO when the direction of the GPIO is output.

Parameter:

gpio_pin: Specifies the pin number to operate.

gpio_data: Output data of the target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_direction(hal_gpio_pin_t gpio_pin, hal_gpio_direction_t gpio_direction);

This function sets the direction of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to set.

gpio_direction: Direction of the target GPIO, the direction can be input or output.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_get_direction(hal_gpio_pin_t gpio_pin, hal_gpio_direction_t *gpio_direction);

This function gets the direction of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to operate.

gpio_direction: Direction of target GPIO, the direction can be input or output.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_high_impedance(hal_gpio_pin_t gpio_pin);

This function sets the target GPIO to high impedance state.

High impedance can prevent the target GPIO from electric leakage.

The pin in high impedance state acts as an open circuit, although it is connected to a low impedance circuit, it will not be affected.

It is recommended to put the pin into high impedance state, if the pin is not in use to optimize the power consumption.

Parameter:

gpio_pin: Specifies the pin number to set.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_clear_high_impedance(hal_gpio_pin_t gpio_pin);

This function removes the high impedance state for the target GPIO.

High impedance can prevent the target GPIO from electric leakage.

It is necessary to call this function before further configuration if the pin is in high impedance state.

Parameter:

gpio_pin: Specifies the pin number to set.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_toggle_pin(hal_gpio_pin_t gpio_pin);

This function toggles the output data of the target GPIO when the direction of the pin is output. After this function, the output data of the target GPIO will be inversed.

Parameter:

gpio_pin: Specifies the pin number to toggle.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_enable_inversion(hal_gpio_pin_t gpio_pin);

This function enables the input data inversion of the target GPIO, after this function, the input data of the target GPIO will always be inversed until the inverse function is disabled.

Parameter:

gpio_pin: Specifies the pin number to inverse.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_disable_inversion(hal_gpio_pin_t gpio_pin);

This function disables the input data inversion of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_pull_up(hal_gpio_pin_t gpio_pin);

This function sets the target GPIO to pull-up state, after this function, the input data of the target pin will be equivalent to high if the pin is disconnected.

This function operates on the pins with only one pull-up resistor.

Parameter:

gpio_pin: Specifies the pin number to set.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_pull_down(hal_gpio_pin_t gpio_pin);

This function sets the target GPIO to the pull-down state, after this function, the input data of the target pin will be equivalent to low if the pin is disconnected.

This function operates on the pin with one pull-down resistor.

Parameter:

gpio_pin: Specifies the pin number to set.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_disable_pull(hal_gpio_pin_t gpio_pin);

This function disables pull-up or pull-down of the target GPIO.

This function operates on the pins with one pull-up and one pull-down resistors.

Parameter:

gpio_pin: Specifies the pin number to set.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_pupd_register(hal_gpio_pin_t gpio_pin, uint8_t gpio_pupd, uint8_t gpio_r0, uint8_t gpio_r1);

This function sets the pull up/down state of the GPIO that has more than one pull-up or pull-down resistor.

Parameters:

gpio_pin: Specifies the pin number to configure.

gpio_pupd: Specifies the pull-up or pull-down of the target GPIO.

gpio_r0: Works with gpio_r1 to specify the pull-up and pull-down resistor of the target GPIO.

gpio_r1: Works with gpio_r0 to specify the pull-up and pull-down resistor of the target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, invalid input pin number, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_clockout(hal_gpio_clock_t gpio_clock_num, hal_gpio_clock_mode_t clock_mode);

This function sets the clock-out source of the target GPIO.

The software can configure which clock to send outside the chip.

There are 6 clock-out ports embedded in all GPIO pins, and each clock-out can be programmed to output appropriate clock source.

This function can only be used after configuring the pin to operate in output clock mode.

Parameter:

gpio_clock_num: Specifies pin clock number to set.

clock_mode: Specifies the clock mode to set to the target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_INVALID_PARAMETER, a wrong parameter (except for pin number) is given, the parameter must be verified.

If the return value is #HAL_GPIO_STATUS_ERROR, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_driving_current(hal_gpio_pin_t gpio_pin, hal_gpio_driving_current_t driving);

This function sets the driving current of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

driving: Specifies the driving current to set to target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_get_driving_current(hal_gpio_pin_t gpio_pin, hal_gpio_driving_current_t *driving);

This function gets the driving current of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

driving: Specifies the driving current to be set to target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_schmitt(hal_gpio_pin_t gpio_pin);

This function sets the schmitt of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_clear_schmitt(hal_gpio_pin_t gpio_pin);

This function clears the schmitt of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_set_slew_rate(hal_gpio_pin_t gpio_pin);

This function sets the slew rate of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_clear_slew_rate(hal_gpio_pin_t gpio_pin);

This function clears the slew rate of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

Function

hal_gpio_status_t hal_gpio_get_pull(hal_gpio_pin_t gpio_pin, hal_gpio_pull_t *pull_state);

This function gets the pull state of the target GPIO.

Parameter:

gpio_pin: Specifies the pin number to configure.

pull_state: Specifies the pull state to be set to target GPIO.

Return:

To indicate whether this function call is successful or not.

If the return value is #HAL_GPIO_STATUS_OK, the operation completed successfully.

If the return value is #HAL_GPIO_STATUS_ERROR_PIN, the operation failed.

2.8.4. hal_i2c_master

Header file

Examples/driver/chip/inc/hal_i2c_master.h

Function

hal_i2c_status_t hal_i2c_master_init(hal_i2c_port_t i2c_port, hal_i2c_config_t *i2c_config);

This function initializes the I2C master before starting a transaction.

Parameter:

i2c_port: I2C master port number.

i2c_config: Is the configuration parameter to initialize the I2C.

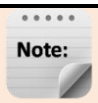
Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an invalid transfer_frequency is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.



#hal_i2c_master_deinit() must be called when the I2C is no longer in use, release the I2C resource for the other users to use this I2C master.

Function

hal_i2c_status_t hal_i2c_master_deinit(hal_i2c_port_t i2c_port);

This function releases the I2C master after the transaction is over. Call this function, if the I2C is no longer in use.

Parameter:

i2c_port: I2C master port number.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

This function must be called when the I2C is no longer in use, release the I2C resource for the other users to use this I2C master.

Function

hal_i2c_status_t hal_i2c_master_set_frequency(hal_i2c_port_t i2c_port, hal_i2c_frequency_t frequency);

This function sets the transaction speed. Apply this function to change the transaction speed without using #hal_i2c_master_init().

Parameter:

i2c_port: I2C master port number.

frequency: Is an enum value defined in #hal_i2c_frequency_t. Only value of type #hal_i2c_frequency_t is accepted.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an invalid speed is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_set_io_config(hal_i2c_port_t i2c_port, hal_i2c_io_config_t io);

This function sets the SCL/SDA IO config.

Parameter:

i2c_port: I2C master port number.

io: Is an enum value defined in #hal_i2c_io_config_t. Only value of type #hal_i2c_io_config_t is accepted.

Return:

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR, the operation is failed.

Function

hal_i2c_status_t hal_i2c_master_register_callback(hal_i2c_port_t i2c_port, hal_i2c_callback_t i2c_callback, void *user_data);

This function registers a callback function while using DMA mode.

The callback function will be called at I2C ISR routine after the I2C triggers an interrupt.

Always call this function to register a callback function while using DMA mode.

Parameter:

i2c_port: I2C master port number.

i2c_callback: Is the user-defined callback function called at I2C ISR routine.

user_data: Is a user-defined input data returned during the callback function's call.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL function pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully.

Function

hal_i2c_status_t hal_i2c_master_send_polling(hal_i2c_port_t i2c_port, uint8_t slave_address, const uint8_t *data, uint32_t size);

This function sends data to I2C slave in polling mode.

This function will not return until the transaction is complete.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

data: Data buffer to send.

size: Data size to send. The maximum value is #HAL_I2C_MAXIMUM_POLLING_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_dma(hal_i2c_port_t i2c_port, uint8_t slave_address, const uint8_t *data, uint32_t size);

This function sends data to I2C slave in DMA mode. This function returns a value immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

data: Data buffer to send.

size: Data size to send. The maximum value is #HAL_I2C_MAXIMUM_DMA_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_polling(hal_i2c_port_t i2c_port, uint8_t slave_address, uint8_t *buffer, uint32_t size);

This function receives data from I2C slave in a polling mode.

This function will not return a value until the transaction is finished.

Parameters:

i2c_port: I2C master port number.

slave_address: I2C slave address.

buffer: Data buffer to receive.

size: Data size to receive. The maximum value is #HAL_I2C_MAXIMUM_POLLING_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_dma(hal_i2c_port_t i2c_port, uint8_t slave_address, uint8_t *buffer, uint32_t size);

This function receives data from I2C slave in a DMA mode. This function will return a value immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

slave_address: I2C slave address.

buffer: Data buffer to receive.

size: Data size to receive. The maximum value is HAL_I2C_MAXIMUM_DMA_TRANSACTION_SIZE.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_polling(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_t *i2c_send_to_receive_config);

This function sends data to and then receives data from I2C slave in a polling mode. This function does not return until the transaction is finished.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config: Is the configuration parameter for this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_t *i2c_send_to_receive_config);

This function sends data to and then receives data from I2C slave in a DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config: Configuration parameter for this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_send_config_t *i2c_send_config);

This function sends data to I2C slave in DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_config: Configuration parameter of this API for send.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, an NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_receive_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_receive_config_t *i2c_receive_config);

This function receives data from I2C slave in an extended DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_receive_config: Configuration parameter of this API for receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma_ex(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_ex_t *i2c_send_to_receive_config_ex);

This function sends data to and then receives data from I2C slave in extended DMA mode. This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

i2c_send_to_receive_config_ex: Configuration parameter of this API for both send and receive.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user or either the send_packet_length or send_bytes_in_one_packet is invalid;

#HAL_I2C_STATUS_OK, the operation completed successfully;

#HAL_I2C_STATUS_ERROR_BUSY, the I2C bus is in use.

Function

hal_i2c_status_t hal_i2c_master_send_to_receive_dma_ex_none_blocking(hal_i2c_port_t i2c_port, hal_i2c_send_to_receive_config_ex_no_busy_t *i2c_send_to_receive_config_no_busy_ex);

This function sends data to and then receives data from I2C slave in an extended DMA mode with software FIFO.

This function returns immediately after configuring the hardware registers.

Parameter:

i2c_port: I2C master port number.

hal_i2c_send_to_receive_config_ex_no_busy_t: Configuration parameter of this API for both send and receive.

Return:

#HAL_I2C_STATUS_OK, the operation completed successfully.

#HAL_I2C_STATUS_ERROR, the I2C sw fifo is full.

Function

hal_i2c_status_t hal_i2c_master_get_running_status(hal_i2c_port_t i2c_port, hal_i2c_running_status_t *running_status);

This function gets running status of the I2C master. Call this function to check if the I2C is idle or not before transferring data. If it's not idle, then the resource is currently in use, delay the operation until the I2C is idle.

Parameter:

i2c_port: I2C master port number.

running_status: #HAL_I2C_STATUS_BUS_BUSY, the I2C master is in busy status;

#HAL_I2C_STATUS_IDLE, the I2C master is in idle status; User can use it to transmit data immediately.

Return:

#HAL_I2C_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_STATUS_OK, the operation completed successfully.

2.8.5. hal_i2c_slave

Header file

Examples/driver/chip/inc/hal_i2c_slave.h

Function

hal_i2c_slave_status_t hal_i2c_slave_init(hal_i2c_slave_port_t slave_port);

This function initializes the I2C slave before starting a transaction.

Parameter:

slave_port: I2C slave port number.

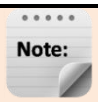
Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_INVALID_PARAMETER, an invalid transfer_frequency is given;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave is in use.



#hal_i2c_slave_deinit() must be called when the I2C is no longer in use, release the I2C resource for the other users to use this I2C slave.

In a multi-task applications, if #hal_i2c_slave_init() returns error #HAL_I2C_STATUS_ERROR_BUSY, it is suggested to call functions that can yield CPU and try again later.

Function

hal_i2c_slave_status_t hal_i2c_slave_deinit(hal_i2c_slave_port_t slave_port);

This function releases the I2C slave after the transaction is over. Call this function, if the I2C slave is no longer in use.

Parameter:

slave_port: I2C slave port number.

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave is in use.

This function must be called when the I2C slave is no longer in use, release the I2C slave resource for the other users to use this I2C slave.

Function

hal_i2c_slave_status_t hal_i2c_slave_send_polling(hal_i2c_slave_port_t slave_port, hal_i2c_slave_send_config_t *send_config, uint32_t timeout_ms, int32_t *success_size);

This function sends data to I2C master in polling mode. This function will not return until the transaction is complete.

Parameter:

slave_port: I2C slave port number.

send_config: Configuration parameter of this API for send.

timeout_ms: Time (ms) to wait for the master transaction. If the transaction is not complete in the time out value, a timeout error will occurs.

success_size: Real transaction sizes(bytes).

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully;

#HAL_I2C_SLAVE_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave bus is in use.

Function

hal_i2c_slave_status_t hal_i2c_slave_receive_polling(hal_i2c_slave_port_t slave_port, hal_i2c_slave_receive_config_t *receive_config, uint32_t timeout_ms, int32_t *success_size);

This function receives data to I2C master in polling mode. This function will not return until the transaction is complete.

Parameter:

slave_port: is the I2C slave port number.

receive_config: is the configuration parameter of this API for receive.

timeout_ms: is the time (ms) to wait for the master transaction. If the transaction is not complete in the time out value, a timeout error will occurs.

success_size: is the real transaction sizes(bytes).

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully;

#HAL_I2C_SLAVE_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave bus is in use.

Function

hal_i2c_slave_status_t hal_i2c_slave_send_dma(hal_i2c_slave_port_t slave_port, hal_i2c_slave_send_config_t *send_config, uint32_t timeout_ms, hal_i2c_slave_callback_config_t *callback_config);

This function sends data to I2C master in DMA mode.

This function will not return until the transaction is complete.

Parameter:

slave_port: I2C slave port number.

send_config: Configuration parameter of this API for send.

timeout_ms: Time (ms) to wait for the master transaction. If the transaction is not complete in the time out value, a timeout error will occurs.

callback_config: Configuration parameter of the callback function.

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully;

#HAL_I2C_SLAVE_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave bus is in use.

Function

hal_i2c_slave_status_t hal_i2c_slave_receive_dma(hal_i2c_slave_port_t slave_port, hal_i2c_slave_receive_config_t *receive_config, uint32_t timeout_ms, hal_i2c_slave_callback_config_t *callback_config);

This function receives data to I2C master in DMA mode. This function will not return until the transaction is complete.

Parameter:

slave_port: I2C slave port number.

receive_config: Configuration parameter of this API for receive.

timeout_ms: Time (ms) to wait for the master transaction. If the transaction is not complete in the time out value, a timeout error will occurs.

callback_config: Configuration parameter of the callback function.

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_INVALID_PARAMETER, a NULL buffer pointer is given by user;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully;

#HAL_I2C_SLAVE_STATUS_ERROR, a hardware error occurred during the transaction.

#HAL_I2C_SLAVE_STATUS_ERROR_BUSY, the I2C slave bus is in use.

Function

hal_i2c_slave_status_t hal_i2c_slave_get_running_status(hal_i2c_slave_port_t slave_port, hal_i2c_slave_running_status_t *running_status);

This function gets running status of the I2C slave. Call this function to check if the I2C slave is in ready or not before transferring data. If it's not in ready state, then the resource is currently in use, delay the operation until the I2C slave is in ready state.

Parameter:

slave_port: I2C slave port number.

running_status:

#HAL_I2C_SLAVE_STATUS_BUS_BUSY, the I2C slave is in busy status;

#HAL_I2C_SLAVE_RUNNING_STATUS_READY, the I2C slave is ready to use;

#HAL_I2C_SLAVE_RUNNING_STATUS_TX_USING, the I2C slave is in sending mode;

#HAL_I2C_SLAVE_RUNNING_STATUS_RX_USING, the I2C slave is in receiving mode;

#HAL_I2C_SLAVE_STATUS_IDLE, the I2C slave is in idle status; User needs to call #hal_i2c_slave_init() now.

Return:

#HAL_I2C_SLAVE_STATUS_INVALID_PORT_NUMBER, an invalid port number is given;

#HAL_I2C_SLAVE_STATUS_OK, the operation completed successfully.

2.8.6. hal_spi_master

Header file

Examples/driver/chip/inc/hal_spi_master.h

Function

hal_spi_master_status_t hal_spi_master_init(hal_spi_master_port_t master_port,

hal_spi_master_config_t *spi_config);

This function is mainly used to initialize the SPI master and set user defined common parameters like clock frequency, bit order, clock polarity, clock phase and default settings.

Parameter:

master_port: SPI master port number.

spi_config: SPI master configures parameters.

Return:

#HAL_SPI_MASTER_STATUS_ERROR means function error

#HAL_SPI_MASTER_STATUS_ERROR_BUSY means this port of SPI master is busy and not available for use

#HAL_SPI_MASTER_STATUS_ERROR_PORT means master_port parameter is an invalid port number

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER means an invalid parameter is given by user

#HAL_SPI_MASTER_STATUS_OK means this function return successfully.

#hal_spi_master_deinit() must be called when the SPI master is no longer in use, release the SPI master resource for the other users to use this SPI master.

Function

hal_spi_master_status_t hal_spi_master_deinit(hal_spi_master_port_t master_port);

This function will reset the SPI master, gates its clock and disables interrupts.

Parameter:

master_port: SPI master port number.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT means master_port parameter is an invalid port number.

#HAL_SPI_MASTER_STATUS_OK means this function return successfully.

Function

hal_spi_master_status_t hal_spi_master_set_advanced_config(hal_spi_master_port_t master_port,hal_spi_master_advanced_config_t *advanced_config);

SPI master advanced configuration function. User can call this function to customize more settings for the SPI device operation.

Parameter:

master_port: SPI master port number.

advanced_config: Provides advanced configuration parameters for the SPI master.

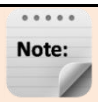
Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, this port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.



This function must be called after #hal_spi_master_init().

Function

hal_spi_master_status_t hal_spi_master_send_polling(hal_spi_master_port_t master_port, uint8_t *data,uint32_t size);

This function is used to send data synchronously with FIFO mode. This function doesn't return until the transfer is complete.

Parameter:

master_port: SPI master port number.

Data: Data buffer to send, this parameter cannot be NULL.

size: Number of bytes to send. Note the user cannot send data size larger than #HAL_SPI_MAXIMUM_POLLING_TRANSACTION_SIZE bytes.

Return:

#HAL_SPI_MASTER_STATUS_ERROR, the byte_order parameter is not properly configured for the FIFO mode;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_send_dma(hal_spi_master_port_t master_port, uint8_t *data,uint32_t size);

This function is used to send data asynchronously with DMA mode. This function returns immediately.

Before calling this function, user should call #hal_spi_master_register_callback() to register a callback, then the callback will be called in the SPI ISR.

Parameter:

master_port: SPI master port number.

data: Data buffer to send, this parameter cannot be NULL, also the address must be a non-cacheable and 4 bytes align address.

size: Number of bytes to send.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_send_dma_blocking(hal_spi_master_port_t master_port,uint8_t *data,uint32_t size);

This function is used to send data synchronously with DMA mode. This function doesn't return until the transfer is complete. Normally this function is used in the scenario where the IRQ must be disabled.

Parameter:

master_port: SPI master port number.

data: Data buffer to send, this parameter cannot be NULL, also the address must be a non-cacheable and 4 bytes align address.

size: Number of bytes to send.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_send_and_receive_polling(hal_spi_master_port_t master_port,hal_spi_master_send_and_receive_config_t *spi_send_and_receive_config);

This function simultaneously sends and receives data in the FIFO mode. This function doesn't return until the transfer is complete.

Parameter:

master_port: SPI master port number.

spi_send_and_receive_config: Structure that contains data buffer and data size

Return:

#HAL_SPI_MASTER_STATUS_ERROR, the byte_order parameter is not properly configured for the FIFO mode;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter in spi_send_and_receive_config is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_send_and_receive_dma(hal_spi_master_port_t master_port,hal_spi_master_send_and_receive_config_t *spi_send_and_receive_config);

This function is used to send and receive data asynchronously with DMA mode. This function returns immediately.

Before calling this function, call #hal_spi_master_register_callback() to register a callback, once the transaction is complete, the callback will be called in the SPI ISR.

Parameter:

master_port: SPI master port number.

spi_send_and_receive_config: Structure that contains data buffer and data size.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter in spi_send_and_receive_config is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_thal_spi_master_send_and_receive_dma_blocking(hal_spi_master_port_t master_port,hal_spi_master_send_and_receive_config_t *spi_send_and_receive_config);

This function simultaneously sends and receives data in the DMA mode. This function doesn't return until the transfer is complete. Normally this function is used in the scenario where the IRQ must be disabled.

Parameter:

master_port: SPI master port number.

spi_send_and_receive_config: Structure that contains data buffer and data size.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter in spi_send_and_receive_config is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_get_running_status(hal_spi_master_port_t master_port,hal_spi_master_running_status_t *running_status);

This function gets current running status of the SPI master. Note this API can only be called after #hal_spi_master_init().

Parameter:

master_port: SPI master port number.

running_status: Current running status.

#HAL_SPI_MASTER_BUSY, the SPI master is in busy status;

#HAL_SPI_MASTER_IDLE, the SPI master is in idle status, user can use it to transfer data now.

Return:

#HAL_SPI_MASTER_STATUS_ERROR, this API is called before #hal_spi_master_init() is called;

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, running_status parameter is NULL;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_set_chip_select_timing(hal_spi_master_port_t master_port,hal_spi_master_chip_select_timing_t chip_select_timing);

This function is used to configure SPI master chip select timing parameter. User can call this function to customize chip select signal timing if the default SPI master chip select signal timing doesn't match SPI device's requirement.

Parameter:

master_port: SPI master port number.

chip_select_timing: Parameter settings for chip select timing.

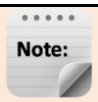
Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid chip select timing parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.



This function must be called after #hal_spi_master_init().

Function

hal_spi_master_status_t hal_spi_master_set_deassert(hal_spi_master_port_t master_port,

hal_spi_master_deassert_t deassert);

SPI master chip selects de-assertion mode configuration. User can call this function to enable the de-assertion feature if the SPI device requires switching the chip select signal from invalid to valid after each byte transfer.

Parameter:

master_port: SPI master port number.

Deassert: Parameter to set SPI master chip select signal behavior after sending one byte.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid deassert parameter is given;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

This function must be called after #hal_spi_master_init().

Function

hal_spi_master_status_t hal_spi_master_set_macro_selection(hal_spi_master_port_t master_port,hal_spi_master_macro_select_t macro_select);

SPI master macro group configuration. If the GPIO selected doesn't belong to SPI pad macro group A, user should call this function to config it to the right pad macro.

Parameter:

master_port: SPI master port number.

macro_select: Parameter for macro group selection.

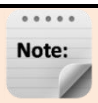
Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid macro selection parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.



This function must be called after #hal_spi_master_init().

Function

hal_spi_master_status_t hal_spi_master_register_callback(hal_spi_master_port_t master_port,hal_spi_master_callback_t callback,void *user_data);

This function is used to register user's callback to SPI master driver. This function should be called when user wants to use the DMA mode, the callback will be called in SPI interrupt service routine.

Parameter:

master_port: SPI master port number.

callback: Callback function given by user, which will be called at SPI master interrupt service routine.

user_data: Parameter given by user and will pass to user while the callback function is called.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_set_mode(hal_spi_master_port_t master_port, hal_spi_master_mode_t mode);

SPI master mode configuration can be configured to single mode, dual mode or quad mode.

Parameter:

master_port: SPI master port number.

mode: Parameter for mode configuration.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid macro selection parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_set_dummy_bits(hal_spi_master_port_t master_port, uint8_t dummy_bits);

This function is used to provide dummy cycle configuration for SPI master.

Parameter:

master_port: SPI master port number.

dummy_bits: Parameter for dummy cycle configuration.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid macro selection parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

Function

hal_spi_master_status_t hal_spi_master_set_command_bytes(hal_spi_master_port_t master_port,uint8_t command_bytes);

This function is used to check SPI master command cycle configuration, only valid when SPI master is configured in dual mode or quad mode.

Parameter:

master_port: SPI master port number.

command_bytes: Parameter for command cycle configuration.

Return:

#HAL_SPI_MASTER_STATUS_ERROR_PORT, master_port parameter is an invalid port number;

#HAL_SPI_MASTER_STATUS_ERROR_BUSY, the port of SPI master is busy and not available for use;

#HAL_SPI_MASTER_STATUS_INVALID_PARAMETER, an invalid macro selection parameter is given by the user;

#HAL_SPI_MASTER_STATUS_OK, the operation completed successfully.

2.8.7. hal_spi_slave

Header file

Examples/driver/chip/inc/hal_spi_slave.h

Function

hal_spi_slave_status_t hal_spi_slave_init(hal_spi_slave_port_t spi_port, hal_spi_slave_config_t *spi_configure);

This function initializes the SPI slave and sets user defined common parameters such as timeout threshold, bit order, clock polarity and clock phase.

Parameter:

spi_port: SPI slave port number

spi_configure: SPI slave configure parameters

Return:

#HAL_SPI_SLAVE_STATUS_ERROR, if the SPI slave port is in use.

#HAL_SPI_SLAVE_STATUS_ERROR_BUSY, if the SPI slave port is busy and not available for use;

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_deinit(hal_spi_slave_port_t spi_port);

This function resets the SPI slave, gates its clock and disables interrupts.

Parameter:

spi_port: SPI slave port number

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_register_callback(hal_spi_slave_port_t spi_port, hal_spi_slave_callback_t callback_function, void *user_data);

This function registers user's callback in the SPI slave driver. This function should always be called when using the SPI slave driver, the callback will be called in SPI ISR.

Parameter:

spi_port: SPI slave port number.

callback_function: Callback function given by user, which will be called at SPI slave interrupt service routine.

user_data: Parameter given by user and will pass to user while the callback function is called.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_send(hal_spi_slave_port_t spi_port, const uint8_t *data, uint32_t size);

This function sends data asynchronously with DMA mode, this function should be called from user's callback function.

Parameter:

spi_port: SPI slave port number.

data: Buffer of data to be sent, this parameter cannot be NULL, also the address must be as non-cacheable address.

size: Number of bytes to send.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_get_master_read_config(hal_spi_slave_port_t spi_port, uint32_t *address, uint32_t *length);

This function queries the address and length from the config read command sent by the SPI master.

Parameter:

spi_port: SPI slave port number.

address: Address of data to read from.

length: Data length to read.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_ERROR, if the address and length of data cannot be queried.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_receive(hal_spi_slave_port_t spi_port, uint8_t *buffer, uint32_t size);

This function receives data asynchronously with DMA mode. This function should be called from user's callback function.

Parameter:

spi_port: SPI slave port number.

buffer: Data buffer where the received data are stored, this parameter cannot be NULL, also the address must be as non-cacheable address.

size: Number of bytes to receive.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_get_master_write_config(hal_spi_slave_port_t spi_port, uint32_t *address, uint32_t *length);

This function queries the address and the length from the config write command sent by the SPI master.

Parameter:

spi_port: SPI slave port number.

address: Address of data the master wants to write to.

length: Data length to write.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_ERROR, if the address and length of data cannot be queried.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_set_early_miso(hal_spi_slave_port_t spi_port, hal_spi_slave_early_miso_t early_miso);

This function configures the SPI slave send MISO early half SCLK cycle parameter.

Parameter:

spi_port: SPI slave port number.

early_miso: Parameter to enable the send MISO data half SCLK cycle early feature or disable it.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_set_command(hal_spi_slave_port_t spi_port, hal_spi_slave_command_type_t command, uint8_t value);

This function configures the SPI slave command value.

Parameter:

spi_port: SPI slave port number.

command: SPI slave command type.

value: Command value that needs to be configured.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_init(hal_spi_slave_port_t spi_port, hal_spi_slave_config_t *spi_configure);

This function initializes the SPI slave and sets user defined common parameters including clock polarity and clock phase, always setup internal configuration for 20MHz SPI clock for better throughput. Note that the SPI slave supports only MSB bit order and cannot be config to LSB bit order.

Parameter:

spi_port: SPI slave port number.

spi_configure: SPI slave configure parameters.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

Function

hal_spi_slave_status_t hal_spi_slave_register_callback(hal_spi_slave_port_t spi_port, hal_spi_slave_callback_t callback_function, void *user_data);

This function registers user's callback in the SPI slave driver. This function may be called when using the SPI slave driver, the callback will be called in SPI interrupt service routine.

Parameter:

spi_port: SPI slave port number.

callback_function: Callback function given by user, which will be called at SPI slave interrupt service routine.

user_data: Parameter given by user and will pass to user while the callback function is called.

Return:

#HAL_SPI_SLAVE_STATUS_ERROR_PORT, if the SPI slave port is invalid.

#HAL_SPI_SLAVE_STATUS_INVALID_PARAMETER, if an invalid parameter is given by user.

#HAL_SPI_SLAVE_STATUS_OK, if this function returned successfully.

2.8.8. hal_uart

Header file

Examples/driver/chip/inc/hal_uart.h

Function

hal_uart_status_t hal_uart_init(hal_uart_port_t uart_port, hal_uart_config_t *uart_config);

This function initializes the UART hardware with basic functionality.

Parameter:

uart_port: Initializes the specified UART port number.

uart_config: Specifies configure parameters for UART hardware initialization.

Return:

#HAL_UART_STATUS_OK if OK.

#HAL_UART_STATUS_ERROR_BUSY if the UART hardware has been initialized before.

#HAL_UART_STATUS_ERROR_PARAMETER if parameter is invalid.

Function

hal_uart_status_t hal_uart_deinit(hal_uart_port_t uart_port);

This function deinitializes the UART hardware to its default status.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

#HAL_UART_STATUS_OK if OK.

#HAL_UART_STATUS_ERROR_PARAMETER if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED if UART has not been initialized.

User must call this function when they don't use this UART port.

Function

void hal_uart_put_char(hal_uart_port_t uart_port, char byte);

This function places one character at a time to the UART port in a polling mode. In this function, driver first poll the status of UART transmit in a loop until it's ready. Then driver places the character in the UART transmit register and returns from this function.

Parameter:

uart_port: Initializes the specified UART port number.

byte: Specifies the data to send.

Function

char hal_uart_get_char(hal_uart_port_t uart_port);

This function gets one character from UART port in a polling mode.

In this function, driver first polls the status of a UART receive operation in a loop until it's ready. Then driver get the character from UART receive register and return it.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

The characters received from the UART port.

Function

uint32_t hal_uart_get_char_unblocking(hal_uart_port_t uart_port);

This function gets one character from UART port in the unblocking polling mode. In this function, driver directly polls the status of a UART receive operation. If data is not ready, return error data (-1). If data is ready, get the character from the UART receive register and return it.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

The character received from the UART port or error code to indicate that there is no valid character in the UART port.

Function

uint32_t hal_uart_send_polling(hal_uart_port_t uart_port, const uint8_t *data, uint32_t size);

This function sends user data byte by byte in a polling mode. In this function, driver first poll status of UART transmit in a loop until it's ready. Then driver put the character into UART transmit register. Driver continues this operation until all user data is sent out, then it returns the transmitted bytes.

Parameter:

uart_port: Initializes the specified UART port number.

data: Specifies pointer to the user's data buffer.

size: Specifies size of the user's data buffer.

Return:

Bytes of transmitted data.

Function

uint32_t hal_uart_send_dma(hal_uart_port_t uart_port, const uint8_t *data, uint32_t size);

This function sends user data in a VFIFO DMA mode.

Parameter:

uart_port: Initializes the specified UART port number.

data: Specifies pointer to the user's data buffer.

size: Specifies size of the user's data buffer.

Return:

Bytes of transmitted data.

User must call hal_uart_init() before calling this function.

Function

uint32_t hal_uart_receive_polling(hal_uart_port_t uart_port, uint8_t *buffer, uint32_t size);

This function receives data byte by byte in a polling mode. In this function, driver first polls the status of a UART receive operation in a loop until it's ready. Then the driver gets the character from the UART transmit register and continues this operation until all user data is received. The final output is to get the character from the UART receive register and return it.

Parameter:

uart_port: Initializes the specified UART port number.

buffer: Specifies pointer to the user's data buffer.

size: Specifies size of the user's data buffer.

Return:

The characters received from the UART port.

Function

uint32_t hal_uart_receive_dma(hal_uart_port_t uart_port, uint8_t *buffer, uint32_t size);

This function receives user data in a VFIFO DMA mode.

Parameter:

uart_port: Initializes the specified UART port number.

buffer: Specifies pointer to the user's data buffer.

size: Specifies size of the user's data buffer.

Return:

The characters received from the UART port.

Function

uint32_t hal_uart_get_available_send_space(hal_uart_port_t uart_port);

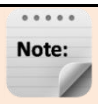
This function queries the available space in VFIFO TX buffer.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

Available space of the VFIFO TX buffer.



This function is used only in a DMA mode.

Function

uint32_t hal_uart_get_available_receive_bytes(hal_uart_port_t uart_port);

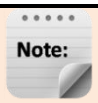
This function queries the available data in VFIFO RX buffer.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

Available data of the VFIFO RX buffer.



This function is used only in DMA mode.

Function

hal_uart_status_t hal_uart_register_callback(hal_uart_port_t uart_port,hal_uart_callback_t user_callback,void *user_data);

This function registers user's callback in the UART driver.

Parameter:

uart_port: Initializes the specified UART port number.

user_callback: Specifies user's callback

user_data: Specifies user's data for this callback

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_BUSY, if the UART port is already registered.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

Function

hal_uart_status_t hal_uart_set_hardware_flowcontrol(hal_uart_port_t uart_port);

This function sets and enables hardware flow control of the UART.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_set_software_flowcontrol(hal_uart_port_t uart_port,uint8_t xon, uint8_t xoff,uint8_t escape_character);

This function sets and enables software flow control of the UART.

Parameter:

uart_port: Initializes the specified UART port number.

xon: Specifies user defined XON character that indicates if the data transmission is turned on.

xoff: Specifies user defined XOFF character that indicates if the data transmission is turned off.

escape_character: Specifies a user defined escape character.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_disable_flowcontrol(hal_uart_port_t uart_port);

This function disables flow control of the UART.

Parameter:

uart_port: Initializes the specified UART port number.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_set_baudrate(hal_uart_port_t uart_port, hal_uart_baudrate_t baudrate);

This function sets a baud rate for the UART frame.

Parameter:

uart_port: Initializes the specified UART port number.

baudrate: Specifies baud rate for the UART frame.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_set_format(hal_uart_port_t uart_port, const hal_uart_config_t *config);

This function sets the UART's frame parameter.

Parameter:

uart_port: Initializes the specified UART port number.

config: Specifies a user defined frame parameter.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_set_dma(hal_uart_port_t uart_port, const hal_uart_dma_config_t *dma_config);

This function sets the VFIFO DMA hardware relative to the UART.

Parameter:

uart_port: Initializes the specified UART port number.

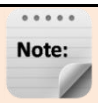
dma_config: Specifies a user defined VFIFO DMA configuration parameter.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if UART has not been initialized.



This function must be called after hal_uart_init().

Function

hal_uart_status_t hal_uart_set_dma_timeout(hal_uart_port_t uart_port, uint32_t timeout);

This function sets timeout value for the UART VFIFO DMA.

Parameter:

uart_port: Initializes the specified UART port number.

timeout: Specifies time out value for UART VFIFO DMA. The unit is ms. This value must not larger than #HAL_UART_TIMEOUT_VALUE_MAX.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.

#HAL_UART_STATUS_ERROR_UNINITIALIZED, if the UART has not been initialized.

Function

hal_uart_status_t hal_uart_set_auto_baudrate(hal_uart_port_t uart_port, bool is_enable);

This function sets auto baud rate for the specific UART port.

Parameter:

uart_port: Initializes the specified UART port number.

is_enable: Specifies that if auto baud rate feature is enabled or not.

Return:

#HAL_UART_STATUS_OK, if the operation completed successfully.

#HAL_UART_STATUS_ERROR_PARAMETER, if parameter is invalid.