- ESP_OK if the initialization is successful
- Appropriate error code from esp_err_t in case of an error

esp_err_t **esp_nimble_hci_deinit** (void)

Deinitialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

Note: This function should be called after the NimBLE host is deinitialized.

Returns
- ESP_OK if the deinitialization is successful
- Appropriate error codes from esp_err_t in case of an error

Macros

BLE_HCI_UART_H4_NONE

BLE_HCI_UART_H4_CMD

BLE_HCI_UART_H4_ACL

BLE_HCI_UART_H4_SCO

BLE_HCI_UART_H4_EVT

ESP-IDF currently supports two host stacks. The Bluedroid based stack (default) supports classic Bluetooth as well as Bluetooth Low Energy (Bluetooth LE). On the other hand, Apache NimBLE based stack is Bluetooth Low Energy only. For users to make a choice:

- For usecases involving classic Bluetooth as well as Bluetooth Low Energy, Bluedroid should be used.
- For Bluetooth Low Energy-only usecases, using NimBLE is recommended. It is less demanding in terms of code footprint and runtime memory, making it suitable for such scenarios.

For the overview of the ESP32 Bluetooth stack architecture, follow the links below:

- ESP32 Bluetooth Architecture (PDF)

Code examples for this API section are provided in the bluetooth/bluedroid directory of ESP-IDF examples.

The following examples contain detailed walkthroughs:

- GATT Client Example Walkthrough
- GATT Server Service Table Example Walkthrough
- GATT Server Example Walkthrough
- GATT Security Client Example Walkthrough
- GATT Security Server Example Walkthrough
- GATT Client Multi-connection Example Walkthrough

2.4 Error Codes Reference

This section lists various error code constants defined in ESP-IDF.

For general information about error codes in ESP-IDF, see *Error Handling*.

ESP_FAIL (-1): Generic esp_err_t code indicating failure

ESP_OK (0): esp_err_t value indicating success (no error)

ESP_ERR_NO_MEM **(0x101)**: Out of memory

ESP_ERR_INVALID_ARG **(0x102)**: Invalid argument

ESP_ERR_INVALID_STATE **(0x103)**: Invalid state

ESP_ERR_INVALID_SIZE **(0x104)**: Invalid size

ESP_ERR_NOT_FOUND **(0x105)**: Requested resource not found

ESP_ERR_NOT_SUPPORTED **(0x106)**: Operation or feature not supported

ESP_ERR_TIMEOUT **(0x107)**: Operation timed out

ESP_ERR_INVALID_RESPONSE **(0x108)**: Received response was invalid

ESP_ERR_INVALID_CRC **(0x109)**: CRC or checksum was invalid

ESP_ERR_INVALID_VERSION **(0x10a)**: Version was invalid

ESP_ERR_INVALID_MAC **(0x10b)**: MAC address was invalid

ESP_ERR_NOT_FINISHED **(0x10c)**: Operation has not fully completed

ESP_ERR_NOT_ALLOWED **(0x10d)**: Operation is not allowed

ESP_ERR_NVS_BASE **(0x1100)**: Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED **(0x1101)**: The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND **(0x1102)**: A requested entry couldn't be found or namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH **(0x1103)**: The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY **(0x1104)**: Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE **(0x1105)**: There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME **(0x1106)**: Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE **(0x1107)**: Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED **(0x1108)**: The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG **(0x1109)**: Key name is too long

ESP_ERR_NVS_PAGE_FULL **(0x110a)**: Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE **(0x110b)**: NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH **(0x110c)**: String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES **(0x110d)**: NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG **(0x110e)**: Value doesn't fit into the entry or string or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND **(0x110f)**: Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND **(0x1110)**: NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED **(0x1111)**: XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED **(0x1112)**: XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED **(0x1113)**: XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND **(0x1114)**: XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED **(0x1115)**: NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED **(0x1116)**: NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART **(0x1117)**: NVS key partition is corrupt

ESP_ERR_NVS_CONTENT_DIFFERS **(0x1118)**: Internal error; never returned by nvs API functions. NVS key is different in comparison

ESP_ERR_NVS_WRONG_ENCRYPTION **(0x1119)**: NVS partition is marked as encrypted with generic flash encryption. This is forbidden since the NVS encryption works differently.

ESP_ERR_ULP_BASE **(0x1200)**: Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG **(0x1201)**: Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR **(0x1202)**: Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL **(0x1203)**: More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL **(0x1204)**: Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE **(0x1205)**: Branch target is out of range of B instruction (try replacing with BX)

ESP_ERR_OTA_BASE **(0x1500)**: Base error code for ota_ops api

ESP_ERR_OTA_PARTITION_CONFLICT **(0x1501)**: Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID **(0x1502)**: Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED **(0x1503)**: Error if OTA app image is invalid

ESP_ERR_OTA_SMALL_SEC_VER **(0x1504)**: Error if the firmware has a secure version less than the running firmware.

ESP_ERR_OTA_ROLLBACK_FAILED **(0x1505)**: Error if flash does not have valid firmware in passive partition and hence rollback is not possible

ESP_ERR_OTA_ROLLBACK_INVALID_STATE **(0x1506)**: Error if current active firmware is still marked in pending validation state (ESP_OTA_IMG_PENDING_VERIFY), essentially first boot of firmware image post upgrade and hence firmware upgrade is not possible

ESP_ERR_EFUSE **(0x1600)**: Base error code for efuse api.

ESP_OK_EFUSE_CNT **(0x1601)**: OK the required number of bits is set.

ESP_ERR_EFUSE_CNT_IS_FULL **(0x1602)**: Error field is full.

ESP_ERR_EFUSE_REPEATED_PROG **(0x1603)**: Error repeated programming of programmed bits is strictly forbidden.

ESP_ERR_CODING **(0x1604)**: Error while a encoding operation.

ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS **(0x1605)**: Error not enough unused key blocks available

ESP_ERR_DAMAGED_READING **(0x1606)**: Error. Burn or reset was done during a reading operation leads to damage read data. This error is internal to the efuse component and not returned by any public API.

ESP_ERR_IMAGE_BASE **(0x2000)**

ESP_ERR_IMAGE_FLASH_FAIL **(0x2001)**

ESP_ERR_IMAGE_INVALID **(0x2002)**

ESP_ERR_WIFI_BASE **(0x3000)**: Starting number of WiFi error codes

ESP_ERR_WIFI_NOT_INIT **(0x3001)**: WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED **(0x3002)**: WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED **(0x3003)**: WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF **(0x3004)**: WiFi interface error

ESP_ERR_WIFI_MODE **(0x3005)**: WiFi mode error

ESP_ERR_WIFI_STATE **(0x3006)**: WiFi internal state error

ESP_ERR_WIFI_CONN **(0x3007)**: WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS **(0x3008)**: WiFi internal NVS module error

ESP_ERR_WIFI_MAC **(0x3009)**: MAC address is invalid

ESP_ERR_WIFI_SSID **(0x300a)**: SSID is invalid

ESP_ERR_WIFI_PASSWORD **(0x300b)**: Password is invalid

ESP_ERR_WIFI_TIMEOUT **(0x300c)**: Timeout error

ESP_ERR_WIFI_WAKE_FAIL **(0x300d)**: WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK **(0x300e)**: The caller would block

ESP_ERR_WIFI_NOT_CONNECT **(0x300f)**: Station still in disconnect status

ESP_ERR_WIFI_POST **(0x3012)**: Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE **(0x3013)**: Invalid WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE **(0x3014)**: Returned when WiFi is stopping

ESP_ERR_WIFI_NOT_ASSOC **(0x3015)**: The WiFi connection is not associated

ESP_ERR_WIFI_TX_DISALLOW **(0x3016)**: The WiFi TX is disallowed

ESP_ERR_WIFI_TWT_FULL **(0x3017)**: no available flow id

ESP_ERR_WIFI_TWT_SETUP_TIMEOUT **(0x3018)**: Timeout of receiving twt setup response frame, timeout times can be set during twt setup

ESP_ERR_WIFI_TWT_SETUP_TXFAIL **(0x3019)**: TWT setup frame tx failed

ESP_ERR_WIFI_TWT_SETUP_REJECT **(0x301a)**: The twt setup request was rejected by the AP

ESP_ERR_WIFI_DISCARD **(0x301b)**: Discard frame

ESP_ERR_WIFI_ROC_IN_PROGRESS **(0x301c)**: ROC op is in progress

ESP_ERR_WIFI_REGISTRAR **(0x3033)**: WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE **(0x3034)**: WPS type error

ESP_ERR_WIFI_WPS_SM **(0x3035)**: WPS state machine is not initialized

ESP_ERR_ESPNOW_BASE **(0x3064)**: ESPNOW error number base.

ESP_ERR_ESPNOW_NOT_INIT **(0x3065)**: ESPNOW is not initialized.

ESP_ERR_ESPNOW_ARG **(0x3066)**: Invalid argument

ESP_ERR_ESPNOW_NO_MEM **(0x3067)**: Out of memory

ESP_ERR_ESPNOW_FULL **(0x3068)**: ESPNOW peer list is full

ESP_ERR_ESPNOW_NOT_FOUND **(0x3069)**: ESPNOW peer is not found

ESP_ERR_ESPNOW_INTERNAL **(0x306a)**: Internal error

ESP_ERR_ESPNOW_EXIST **(0x306b)**: ESPNOW peer has existed

ESP_ERR_ESPNOW_IF **(0x306c)**: Interface error

ESP_ERR_ESPNOW_CHAN **(0x306d)**: Channel error

ESP_ERR_DPP_FAILURE **(0x3097)**: Generic failure during DPP Operation

ESP_ERR_DPP_TX_FAILURE **(0x3098)**: DPP Frame Tx failed OR not Acked

ESP_ERR_DPP_INVALID_ATTR **(0x3099)**: Encountered invalid DPP Attribute

ESP_ERR_DPP_AUTH_TIMEOUT **(0x309a)**: DPP Auth response was not recieved in time

ESP_ERR_MESH_BASE **(0x4000)**: Starting number of MESH error codes

ESP_ERR_MESH_WIFI_NOT_START **(0x4001)**

ESP_ERR_MESH_NOT_INIT **(0x4002)**

ESP_ERR_MESH_NOT_CONFIG **(0x4003)**

ESP_ERR_MESH_NOT_START **(0x4004)**

ESP_ERR_MESH_NOT_SUPPORT **(0x4005)**

ESP_ERR_MESH_NOT_ALLOWED **(0x4006)**

ESP_ERR_MESH_NO_MEMORY **(0x4007)**

ESP_ERR_MESH_ARGUMENT **(0x4008)**

ESP_ERR_MESH_EXCEED_MTU **(0x4009)**

ESP_ERR_MESH_TIMEOUT **(0x400a)**

ESP_ERR_MESH_DISCONNECTED **(0x400b)**

ESP_ERR_MESH_QUEUE_FAIL **(0x400c)**

ESP_ERR_MESH_QUEUE_FULL **(0x400d)**

ESP_ERR_MESH_NO_PARENT_FOUND **(0x400e)**

ESP_ERR_MESH_NO_ROUTE_FOUND **(0x400f)**

ESP_ERR_MESH_OPTION_NULL **(0x4010)**

ESP_ERR_MESH_OPTION_UNKNOWN **(0x4011)**

ESP_ERR_MESH_XON_NO_WINDOW **(0x4012)**

ESP_ERR_MESH_INTERFACE **(0x4013)**

ESP_ERR_MESH_DISCARD_DUPLICATE **(0x4014)**

ESP_ERR_MESH_DISCARD **(0x4015)**

ESP_ERR_MESH_VOTING **(0x4016)**

ESP_ERR_MESH_XMIT **(0x4017)**

ESP_ERR_MESH_QUEUE_READ **(0x4018)**

ESP_ERR_MESH_PS **(0x4019)**

ESP_ERR_MESH_RECV_RELEASE **(0x401a)**

ESP_ERR_ESP_NETIF_BASE **(0x5000)**

ESP_ERR_ESP_NETIF_INVALID_PARAMS **(0x5001)**

ESP_ERR_ESP_NETIF_IF_NOT_READY **(0x5002)**

ESP_ERR_ESP_NETIF_DHCPC_START_FAILED **(0x5003)**

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED **(0x5004)**

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED **(0x5005)**

ESP_ERR_ESP_NETIF_NO_MEM **(0x5006)**

ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED **(0x5007)**

ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED **(0x5008)**

ESP_ERR_ESP_NETIF_INIT_FAILED **(0x5009)**

ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED **(0x500a)**

ESP_ERR_ESP_NETIF_MLD6_FAILED **(0x500b)**

ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED **(0x500c)**

ESP_ERR_ESP_NETIF_DHCPS_START_FAILED **(0x500d)**

ESP_ERR_FLASH_BASE **(0x6000)**: Starting number of flash error codes

ESP_ERR_FLASH_OP_FAIL **(0x6001)**

ESP_ERR_FLASH_OP_TIMEOUT **(0x6002)**

ESP_ERR_FLASH_NOT_INITIALISED **(0x6003)**

ESP_ERR_FLASH_UNSUPPORTED_HOST **(0x6004)**

ESP_ERR_FLASH_UNSUPPORTED_CHIP **(0x6005)**

ESP_ERR_FLASH_PROTECTED **(0x6006)**

ESP_ERR_HTTP_BASE **(0x7000)**: Starting number of HTTP error codes

ESP_ERR_HTTP_MAX_REDIRECT **(0x7001)**: The error exceeds the number of HTTP redirects

ESP_ERR_HTTP_CONNECT **(0x7002)**: Error open the HTTP connection

ESP_ERR_HTTP_WRITE_DATA **(0x7003)**: Error write HTTP data

ESP_ERR_HTTP_FETCH_HEADER **(0x7004)**: Error read HTTP header from server

ESP_ERR_HTTP_INVALID_TRANSPORT **(0x7005)**: There are no transport support for the input scheme

ESP_ERR_HTTP_CONNECTING **(0x7006)**: HTTP connection hasn't been established yet

ESP_ERR_HTTP_EAGAIN **(0x7007)**: Mapping of errno EAGAIN to esp_err_t

ESP_ERR_HTTP_CONNECTION_CLOSED **(0x7008)**: Read FIN from peer and the connection closed

ESP_ERR_ESP_TLS_BASE **(0x8000)**: Starting number of ESP-TLS error codes

ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME **(0x8001)**: Error if hostname couldn't be resolved upon tls connection

ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET **(0x8002)**: Failed to create socket

ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY **(0x8003)**: Unsupported protocol family

ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST **(0x8004)**: Failed to connect to host

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED **(0x8005)**: failed to set/get socket option

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT **(0x8006)**: new connection in esp_tls_low_level_conn connection timeouted

ESP_ERR_ESP_TLS_SE_FAILED **(0x8007)**

ESP_ERR_ESP_TLS_TCP_CLOSED_FIN **(0x8008)**

ESP_ERR_MBEDTLS_CERT_PARTLY_OK **(0x8010)**: mbedtls parse certificates was partly successful

ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED **(0x8011)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED **(0x8012)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED **(0x8013)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED **(0x8014)**: mbedtls api returned error

ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED **(0x8015)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED **(0x8016)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SETUP_FAILED **(0x8017)**: mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_WRITE_FAILED **(0x8018)**: mbedtls api returned error

ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED **(0x8019)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED **(0x801a)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED **(0x801b)**: mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED **(0x801c)**: mbedtls api returned failed

ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED **(0x8031)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED **(0x8032)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED **(0x8033)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED **(0x8034)**: wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED **(0x8035)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_CTX_SETUP_FAILED **(0x8036)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_SETUP_FAILED **(0x8037)**: wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_WRITE_FAILED **(0x8038)**: wolfSSL api returned failed

ESP_ERR_HTTPS_OTA_BASE **(0x9000)**

ESP_ERR_HTTPS_OTA_IN_PROGRESS **(0x9001)**

ESP_ERR_PING_BASE **(0xa000)**

ESP_ERR_PING_INVALID_PARAMS **(0xa001)**

ESP_ERR_PING_NO_MEM **(0xa002)**

ESP_ERR_HTTPD_BASE **(0xb000)**: Starting number of HTTPD error codes

ESP_ERR_HTTPD_HANDLERS_FULL **(0xb001)**: All slots for registering URI handlers have been consumed

ESP_ERR_HTTPD_HANDLER_EXISTS **(0xb002)**: URI handler with same method and target URI already registered

ESP_ERR_HTTPD_INVALID_REQ **(0xb003)**: Invalid request pointer

ESP_ERR_HTTPD_RESULT_TRUNC **(0xb004)**: Result string truncated

ESP_ERR_HTTPD_RESP_HDR **(0xb005)**: Response header field larger than supported

ESP_ERR_HTTPD_RESP_SEND **(0xb006)**: Error occured while sending response packet

ESP_ERR_HTTPD_ALLOC_MEM **(0xb007)**: Failed to dynamically allocate memory for resource

ESP_ERR_HTTPD_TASK **(0xb008)**: Failed to launch server task/thread

ESP_ERR_HW_CRYPTO_BASE **(0xc000)**: Starting number of HW cryptography module error codes

ESP_ERR_HW_CRYPTO_DS_HMAC_FAIL **(0xc001)**: HMAC peripheral problem

ESP_ERR_HW_CRYPTO_DS_INVALID_KEY **(0xc002)**

ESP_ERR_HW_CRYPTO_DS_INVALID_DIGEST **(0xc004)**

ESP_ERR_HW_CRYPTO_DS_INVALID_PADDING **(0xc005)**

ESP_ERR_MEMPROT_BASE **(0xd000)**: Starting number of Memory Protection API error codes

ESP_ERR_MEMPROT_MEMORY_TYPE_INVALID **(0xd001)**

ESP_ERR_MEMPROT_SPLIT_ADDR_INVALID **(0xd002)**

ESP_ERR_MEMPROT_SPLIT_ADDR_OUT_OF_RANGE **(0xd003)**

ESP_ERR_MEMPROT_SPLIT_ADDR_UNALIGNED **(0xd004)**

ESP_ERR_MEMPROT_UNIMGMT_BLOCK_INVALID **(0xd005)**

ESP_ERR_MEMPROT_WORLD_INVALID **(0xd006)**

ESP_ERR_MEMPROT_AREA_INVALID **(0xd007)**

ESP_ERR_MEMPROT_CPUID_INVALID **(0xd008)**

ESP_ERR_TCP_TRANSPORT_BASE **(0xe000)**: Starting number of TCP Transport error codes

ESP_ERR_TCP_TRANSPORT_CONNECTION_TIMEOUT **(0xe001)**: Connection has timed out

ESP_ERR_TCP_TRANSPORT_CONNECTION_CLOSED_BY_FIN **(0xe002)**: Read FIN from peer and the connection has closed (in a clean way)

ESP_ERR_TCP_TRANSPORT_CONNECTION_FAILED **(0xe003)**: Failed to connect to the peer

ESP_ERR_TCP_TRANSPORT_NO_MEM **(0xe004)**: Memory allocation failed

ESP_ERR_NVS_SEC_BASE **(0xf000)**: Starting number of error codes

ESP_ERR_NVS_SEC_HMAC_KEY_NOT_FOUND **(0xf001)**: HMAC Key required to generate the NVS encryption keys not found

ESP_ERR_NVS_SEC_HMAC_KEY_BLK_ALREADY_USED **(0xf002)**: Provided eFuse block for HMAC key generation is already in use

ESP_ERR_NVS_SEC_HMAC_KEY_GENERATION_FAILED **(0xf003)**: Failed to generate/write the HMAC key to eFuse

ESP_ERR_NVS_SEC_HMAC_XTS_KEYS_DERIV_FAILED **(0xf004)**: Failed to derive the NVS encryption keys based on the HMAC-based scheme

2.5 Networking APIs

2.5.1 Wi-Fi

ESP-NOW

Overview ESP-NOW is a kind of connectionless Wi-Fi communication protocol that is defined by Espressif. In ESP-NOW, application data is encapsulated in a vendor-specific action frame and then transmitted from one Wi-Fi device to another without connection.

CTR with CBC-MAC Protocol (CCMP) is used to protect the action frame for security. ESP-NOW is widely used in smart light, remote controlling, sensor, etc.

Frame Format ESP-NOW uses a vendor-specific action frame to transmit ESP-NOW data. The default ESP-NOW bit rate is 1 Mbps. The format of the vendor-specific action frame is as follows:

```
------------------------------------------------------------------------------------------------------------
| MAC Header | Category Code | Organization Identifier | Random Values | Vendor Specific Content |   FCS   |
------------------------------------------------------------------------------------------------------------
  24 bytes         1 byte              3 bytes               4 bytes             7-257 bytes        4 bytes
```

- Category Code: The Category Code field is set to the value (127) indicating the vendor-specific category.

4.12.3 Example

ESP-IDF provides an example to show how to implement the deep sleep wake stub.

- system/deep_sleep_wake_stub

4.13 Error Handling

4.13.1 Overview

Identifying and handling run-time errors is important for developing robust applications. There can be multiple kinds of run-time errors:

- Recoverable errors:
  - Errors indicated by functions through return values (error codes)
  - C++ exceptions, thrown using `throw` keyword
- Unrecoverable (fatal) errors:
  - Failed assertions (using `assert` macro and equivalent methods, see *Assertions*) and `abort()` calls.
  - CPU exceptions: access to protected regions of memory, illegal instruction, etc.
  - System level checks: watchdog timeout, cache access error, stack overflow, stack smashing, heap corruption, etc.

This guide explains ESP-IDF error handling mechanisms related to recoverable errors, and provides some common error handling patterns.

For instructions on diagnosing unrecoverable errors, see *Fatal Errors*.

4.13.2 Error Codes

The majority of ESP-IDF-specific functions use `esp_err_t` type to return error codes. `esp_err_t` is a signed integer type. Success (no error) is indicated with `ESP_OK` code, which is defined as zero.

Various ESP-IDF header files define possible error codes using preprocessor defines. Usually these defines start with `ESP_ERR_` prefix. Common error codes for generic failures (out of memory, timeout, invalid argument, etc.) are defined in `esp_err.h` file. Various components in ESP-IDF may define additional error codes for specific situations.

For the complete list of error codes, see *Error Code Reference*.

4.13.3 Converting Error Codes to Error Messages

For each error code defined in ESP-IDF components, `esp_err_t` value can be converted to an error code name using `esp_err_to_name()` or `esp_err_to_name_r()` functions. For example, passing `0x101` to `esp_err_to_name()` will return a `ESP_ERR_NO_MEM` string. Such strings can be used in log output to make it easier to understand which error has happened.

Additionally, `esp_err_to_name_r()` function will attempt to interpret the error code as a standard POSIX error code, if no matching `ESP_ERR_` value is found. This is done using `strerror_r` function. POSIX error codes (such as `ENOENT`, `ENOMEM`) are defined in `errno.h` and are typically obtained from `errno` variable. In ESP-IDF this variable is thread-local: multiple FreeRTOS tasks have their own copies of `errno`. Functions which set `errno` only modify its value for the task they run in.

This feature is enabled by default, but can be disabled to reduce application binary size. See *CONFIG_ESP_ERR_TO_NAME_LOOKUP*. When this feature is disabled, `esp_err_to_name()` and `esp_err_to_name_r()` are still defined and can be called. In this case, `esp_err_to_name()` will return UNKNOWN ERROR, and *esp_err_to_name_r()* will return Unknown error 0xXXXX(YYYYY), where 0xXXXX and YYYYY are the hexadecimal and decimal representations of the error code, respectively.

4.13.4 ESP_ERROR_CHECK Macro

ESP_ERROR_CHECK macro serves similar purpose as assert, except that it checks *esp_err_t* value rather than a bool condition. If the argument of *ESP_ERROR_CHECK* is not equal *ESP_OK*, then an error message is printed on the console, and abort() is called.

Error message will typically look like this:

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf

file: "/Users/user/esp/example/main/main.c" line 20
func: app_main
expression: sdmmc_card_init(host, &card)

Backtrace: 0x40086e7c:0x3ffb4ff0 0x40087328:0x3ffb5010 0x400d1fdf:0x3ffb5030␣
↪0x400d0816:0x3ffb5050
```

Note: If *ESP-IDF monitor* is used, addresses in the backtrace will be converted to file names and line numbers.

- The first line mentions the error code as a hexadecimal value, and the identifier used for this error in source code. The latter depends on *CONFIG_ESP_ERR_TO_NAME_LOOKUP* option being set. Address in the program where error has occurred is printed as well.
- Subsequent lines show the location in the program where *ESP_ERROR_CHECK* macro was called, and the expression which was passed to the macro as an argument.
- Finally, backtrace is printed. This is part of panic handler output common to all fatal errors. See *Fatal Errors* for more information about the backtrace.

4.13.5 ESP_ERROR_CHECK_WITHOUT_ABORT Macro

ESP_ERROR_CHECK_WITHOUT_ABORT macro serves similar purpose as ESP_ERROR_CHECK, except that it will not call abort().

4.13.6 ESP_RETURN_ON_ERROR Macro

ESP_RETURN_ON_ERROR macro checks the error code, if the error code is not equal *ESP_OK*, it prints the message and returns the error code.

4.13.7 ESP_GOTO_ON_ERROR Macro

ESP_GOTO_ON_ERROR macro checks the error code, if the error code is not equal *ESP_OK*, it prints the message, sets the local variable ret to the code, and then exits by jumping to goto_tag.

4.13.8 ESP_RETURN_ON_FALSE Macro

ESP_RETURN_ON_FALSE macro checks the condition, if the condition is not equal true, it prints the message and returns with the supplied err_code.

4.13.9 ESP_GOTO_ON_FALSE Macro

ESP_GOTO_ON_FALSE macro checks the condition, if the condition is not equal `true`, it prints the message, sets the local variable `ret` to the supplied `err_code`, and then exits by jumping to `goto_tag`.

4.13.10 CHECK MACROS Examples

Some examples

```
static const char* TAG = "Test";

esp_err_t test_func(void)
{
    esp_err_t ret = ESP_OK;

    ESP_ERROR_CHECK(x);                                         // err message␣
↪printed if `x` is not `ESP_OK`, and then `abort()`.
    ESP_ERROR_CHECK_WITHOUT_ABORT(x);                           // err message␣
↪printed if `x` is not `ESP_OK`, without `abort()`.
    ESP_RETURN_ON_ERROR(x, TAG, "fail reason 1");               // err message␣
↪printed if `x` is not `ESP_OK`, and then function returns with code `x`.
    ESP_GOTO_ON_ERROR(x, err, TAG, "fail reason 2");            // err message␣
↪printed if `x` is not `ESP_OK`, `ret` is set to `x`, and then jumps to `err`.
    ESP_RETURN_ON_FALSE(a, err_code, TAG, "fail reason 3");     // err message␣
↪printed if `a` is not `true`, and then function returns with code `err_code`.
    ESP_GOTO_ON_FALSE(a, err_code, err, TAG, "fail reason 4"); // err message␣
↪printed if `a` is not `true`, `ret` is set to `err_code`, and then jumps to␣
↪`err`.

err:
    // clean up
    return ret;
}
```

Note: If the option *CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT* in Kconfig is enabled, the error message will be discarded, while the other action works as is.

The `ESP_RETURN_XX` and `ESP_GOTO_xx` macros cannot be called from ISR. While there are `xx_ISR` versions for each of them, e.g., `ESP_RETURN_ON_ERROR_ISR`, these macros could be used in ISR.

4.13.11 Error Handling Patterns

1. Attempt to recover. Depending on the situation, we may try the following methods:
   - retry the call after some time;
   - attempt to de-initialize the driver and re-initialize it again;
   - fix the error condition using an out-of-band mechanism (e.g reset an external peripheral which is not responding).

   Example:

   ```
   esp_err_t err;
   do {
       err = sdio_slave_send_queue(addr, len, arg, timeout);
       // keep retrying while the sending queue is full
   } while (err == ESP_ERR_TIMEOUT);
   if (err != ESP_OK) {
       // handle other errors
   }
   ```

2. Propagate the error to the caller. In some middleware components this means that a function must exit with the same error code, making sure any resource allocations are rolled back.

   Example:

```
sdmmc_card_t* card = calloc(1, sizeof(sdmmc_card_t));
if (card == NULL) {
    return ESP_ERR_NO_MEM;
}
esp_err_t err = sdmmc_card_init(host, &card);
if (err != ESP_OK) {
    // Clean up
    free(card);
    // Propagate the error to the upper layer (e.g., to notify the␣
↪user).
    // Alternatively, application can define and return custom error␣
↪code.
    return err;
}
```

3. Convert into unrecoverable error, for example using `ESP_ERROR_CHECK`. See *ESP_ERROR_CHECK macro* section for details.

   Terminating the application in case of an error is usually undesirable behavior for middleware components, but is sometimes acceptable at application level.

   Many ESP-IDF examples use `ESP_ERROR_CHECK` to handle errors from various APIs. This is not the best practice for applications, and is done to make example code more concise.

   Example:

```
ESP_ERROR_CHECK(spi_bus_initialize(host, bus_config, dma_chan));
```

4.13.12 C++ Exceptions

See *Exception Handling*.

4.14 ESP-BLE-MESH

Bluetooth® mesh networking enables many-to-many (m:m) device communications and is optimized for creating large-scale device networks.

Devices may relay data to other devices not in direct radio range of the originating device. In this way, mesh networks can span very large physical areas and contain large numbers of devices. It is ideally suited for building automation, sensor networks, and other IoT solutions where tens, hundreds, or thousands of devices need to reliably and securely communicate with one another.

Bluetooth mesh is not a wireless communications technology, but a networking technology. This technology is dependent upon Bluetooth Low Energy (BLE) - a wireless communications protocol stack.

Built on top of Zephyr Bluetooth Mesh stack, the ESP-BLE-MESH implementation supports device provisioning and node control. It also supports such node features as Proxy, Relay, Low power and Friend.

Please see the *ESP-BLE-MESH Architecture* for information about the implementation of ESP-BLE-MESH architecture and *ESP-BLE-MESH API Reference* for information about respective API.

ESP-BLE-MESH is implemented and certified based on the latest Mesh Profile v1.0.1, users can refer here for the certification details of ESP-BLE-MESH.

Note: If you are looking for Wi-Fi based implementation of mesh for ESP32, please check another product by Espressif called ESP-WIFI-MESH. For more information and documentation see *ESP-WIFI-MESH*.

4.16.6 Chip Revisions

There are some issues with certain revisions of ESP32 that have repercussions for use with external RAM. The issues are documented in the ESP32 Series SoC Errata document. In particular, ESP-IDF handles the bugs mentioned in the following ways:

ESP32 Rev v0.0

ESP-IDF has no workaround for the bugs in this revision of silicon, and it cannot be used to map external PSRAM into ESP32's main memory map.

ESP32 Rev v1.0

The bugs in this revision of silicon cause issues if certain sequences of machine instructions operate on external memory. (ESP32 Series SoC Errata 3.2). As a workaround, the `-mfix-esp32-psram-cache-issue` flag has been added to the ESP32 GCC compiler such that these sequences are filtered out. As a result, the compiler only outputs code that can safely be executed. The *CONFIG_SPIRAM_CACHE_WORKAROUND* option can be used to enable this workaround.

Aside from linking to a recompiled version of Newlib with the additional flag, ESP-IDF also does the following:

- Avoids using some ROM functions
- Allocates static memory for the Wi-Fi stack

ESP32 Rev v3.0

ESP32 rev v3.0 fixes the PSRAM cache issue found in rev v1.0. When *CONFIG_ESP32_REV_MIN* option is set to `rev v3.0`, compiler workarounds related to PSRAM will be disabled. For more information about ESP32 v3.0, see ESP32 Chip Revision v3.0 User Guide.

4.17 Fatal Errors

4.17.1 Overview

In certain situations, the execution of the program can not be continued in a well-defined way. In ESP-IDF, these situations include:

- CPU Exceptions: Illegal Instruction, Load/Store Alignment Error, Load/Store Prohibited error, Double Exception.
- System level checks and safeguards:
  - *Interrupt watchdog* timeout
  - *Task watchdog* timeout (only fatal if *CONFIG_ESP_TASK_WDT_PANIC* is set)
  - Cache access error
  - Brownout detection event
  - Stack overflow
  - Stack smashing protection check
  - Heap integrity check
  - Undefined behavior sanitizer (UBSAN) checks
- Failed assertions, via `assert`, `configASSERT` and similar macros.

This guide explains the procedure used in ESP-IDF for handling these errors, and provides suggestions on troubleshooting the errors.

4.17.2 Panic Handler

Every error cause listed in the *Overview* will be handled by the *panic handler*.

The panic handler will start by printing the cause of the error to the console. For CPU exceptions, the message will be similar to

```
Guru Meditation Error: Core 0 panic'ed (IllegalInstruction). Exception was␣
↪unhandled.
```

For some of the system level checks (interrupt watchdog, cache access error), the message will be similar to

```
Guru Meditation Error: Core 0 panic'ed (Cache disabled but cached memory␣
↪region accessed). Exception was unhandled.
```

In all cases, the error cause will be printed in parentheses. See *Guru Meditation Errors* for a list of possible error causes.

Subsequent behavior of the panic handler can be set using *CONFIG_ESP_SYSTEM_PANIC* configuration choice. The available options are:

- Print registers and reboot (`CONFIG_ESP_SYSTEM_PANIC_PRINT_REBOOT`) —default option.
  This will print register values at the point of the exception, print the backtrace, and restart the chip.
- Print registers and halt (`CONFIG_ESP_SYSTEM_PANIC_PRINT_HALT`)
  Similar to the above option, but halt instead of rebooting. External reset is required to restart the program.
- Silent reboot (`CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT`)
  Do not print registers or backtrace, restart the chip immediately.
- Invoke GDB Stub (`CONFIG_ESP_SYSTEM_PANIC_GDBSTUB`)
  Start GDB server which can communicate with GDB over console UART port. This option will only provide read-only debugging or post-mortem debugging. See *GDB Stub* for more details.

Note: The `CONFIG_ESP_SYSTEM_PANIC_GDBSTUB` choice in the configuration option *CONFIG_ESP_SYSTEM_PANIC* is only available when the component `esp_gdbstub` is included in the build.

The behavior of the panic handler is affected by three other configuration options.

- If *CONFIG_ESP_DEBUG_OCDAWARE* is enabled (which is the default), the panic handler will detect whether a JTAG debugger is connected. If it is, execution will be halted and control will be passed to the debugger. In this case, registers and backtrace are not dumped to the console, and GDBStub / Core Dump functions are not used.
- If the *Core Dump* feature is enabled, then the system state (task stacks and registers) will be dumped to either Flash or UART, for later analysis.
- If *CONFIG_ESP_PANIC_HANDLER_IRAM* is disabled (disabled by default), the panic handler code is placed in flash memory, not IRAM. This means that if ESP-IDF crashes while flash cache is disabled, the panic handler will automatically re-enable flash cache before running GDB Stub or Core Dump. This adds some minor risk, if the flash cache status is also corrupted during the crash.
  If this option is enabled, the panic handler code (including required UART functions) is placed in IRAM, and hence will decrease the usable memory space in SRAM. But this may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash) or when flash cache is corrupted when an exception is triggered.
- If *CONFIG_ESP_SYSTEM_PANIC_REBOOT_DELAY_SECONDS* is enabled (disabled by default) and set to a number higher than 0, the panic handler will delay the reboot for that amount of time in seconds. This can help if the tool used to monitor serial output does not provide a possibility to stop and examine the serial output. In that case, delaying the reboot will allow users to examine and debug the panic handler output (backtrace, etc.) for the duration of the delay. After the delay, the device will reboot. The reset reason is preserved.

The following diagram illustrates the panic handler behavior:

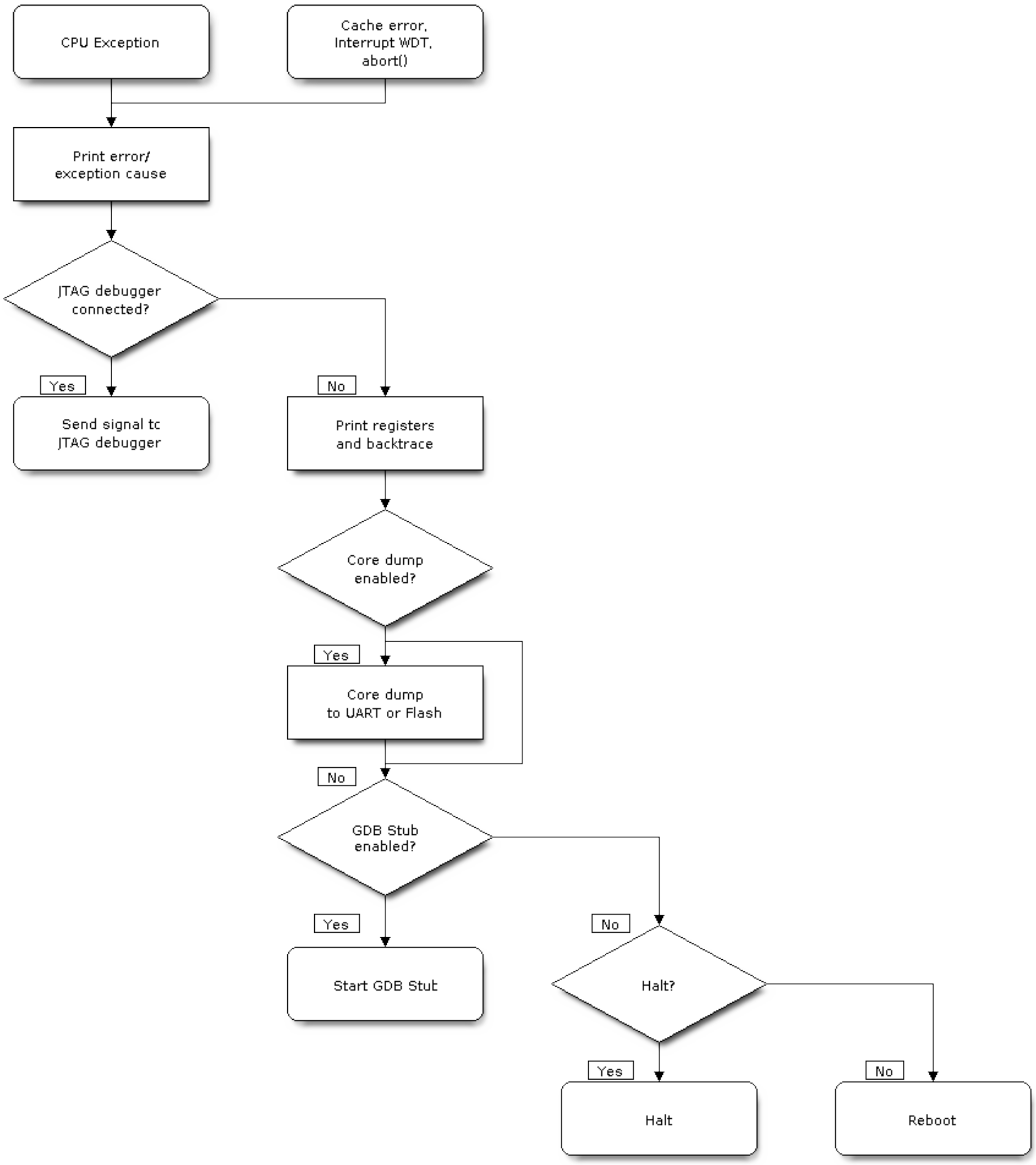

Fig. 41: Panic Handler Flowchart (click to enlarge)

4.17.3 Register Dump and Backtrace

Unless the `CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT` option is enabled, the panic handler prints some of the CPU registers, and the backtrace, to the console

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      :␣
↪0x3ffb5030
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      :␣
↪0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      :␣
↪0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     :␣
↪0x06ff1ff8
A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE:␣
↪0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
```

The register values printed are the register values in the exception frame, i.e., values at the moment when the CPU exception or another fatal error has occurred.

A Register dump is not printed if the panic handler has been executed as a result of an `abort()` call.

In some cases, such as interrupt watchdog timeout, the panic handler may print additional CPU registers (EPC1-EPC4) and the registers/backtrace of the code running on the other CPU.

The backtrace line contains PC:SP pairs, where PC is the Program Counter and SP is Stack Pointer, for each stack frame of the current task. If a fatal error happens inside an ISR, the backtrace may include PC:SP pairs both from the task which was interrupted, and from the ISR.

If *IDF Monitor* is used, Program Counter values will be converted to code locations (function name, file name, and line number), and the output will be annotated with additional lines:

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      :␣
↪0x3ffb5030
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      :␣
↪0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      :␣
↪0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     :␣
↪0x06ff1ff8
0x40082d1c: _calloc_r at /Users/user/esp/esp-idf/components/newlib/syscalls.c:51

A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE:␣
↪0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  :␣
↪0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

0x400d0802: main_task at /Users/user/esp/esp-idf/components/esp32/cpu_start.c:470
```

To find the location where a fatal error has happened, look at the lines which follow the "Backtrace" line. Fatal error location is the top line, and subsequent lines show the call stack.

4.17.4 GDB Stub

If the `CONFIG_ESP_SYSTEM_PANIC_GDBSTUB` option is enabled, the panic handler will not reset the chip when a fatal error happens. Instead, it will start a GDB remote protocol server, commonly referred to as GDB Stub. When this happens, a GDB instance running on the host computer can be instructed to connect to the ESP32 UART port.

If *IDF Monitor* is used, GDB is started automatically when a GDB Stub prompt is detected on the UART. The output looks like this:

```
Entering gdb stub now.
$T0b#e6GNU gdb (crosstool-NG crosstool-ng-1.22.0-80-gff1f415) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_apple-darwin16.3.0 --target=xtensa-
↪esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /Users/user/esp/example/build/example.elf...done.
Remote debugging using /dev/cu.usbserial-31301
0x400e1b41 in app_main ()
    at /Users/user/esp/example/main/main.cpp:36
36      *((int*) 0) = 0;
(gdb)
```

The GDB prompt can be used to inspect CPU registers, local and static variables, and arbitrary locations in memory. It is not possible to set breakpoints, change the PC, or continue execution. To reset the program, exit GDB and perform an external reset: Ctrl-T Ctrl-R in IDF Monitor, or using the external reset button on the development board.

4.17.5 RTC Watchdog Timeout

The RTC watchdog is used in the startup code to keep track of execution time and it also helps to prevent a lock-up caused by an unstable power source. It is enabled by default (see *CONFIG_BOOTLOADER_WDT_ENABLE*). If the execution time is exceeded, the RTC watchdog will restart the system. In this case, the ROM bootloader will print a message with the `RTC Watchdog Timeout` reason for the reboot.

```
rst:0x10 (RTCWDT_RTC_RESET)
```

The RTC watchdog covers the execution time from the first stage bootloader (ROM bootloader) to application startup. It is initially set in the ROM bootloader, then configured in the bootloader with the *CONFIG_BOOTLOADER_WDT_TIME_MS* option (9000 ms by default). During the application initialization stage, it is reconfigured because the source of the slow clock may have changed, and finally disabled right before the `app_main()` call. There is an option *CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE* which prevents the RTC watchdog from being disabled before `app_main`. Instead, the RTC watchdog remains active and must be fed periodically in your application's code.

4.17.6 Guru Meditation Errors

This section explains the meaning of different error causes, printed in parens after the `Guru Meditation Error: Core panic'ed` message.

Note: See the Guru Meditation Wikipedia article for historical origins of "Guru Meditation".

IllegalInstruction

This CPU exception indicates that the instruction which was executed was not a valid instruction. The most common reasons for this error include:

- FreeRTOS task function has returned. In FreeRTOS, if a task function needs to terminate, it should call `vTaskDelete()` and delete itself, instead of returning.
- Failure to read next instruction from SPI flash. This usually happens if:
  - Application has reconfigured the SPI flash pins as some other function (GPIO, UART, etc.). Consult the Hardware Design Guidelines and the datasheet for the chip or module for details about the SPI flash pins.
  - Some external device has accidentally been connected to the SPI flash pins, and has interfered with communication between ESP32 and SPI flash.
- In C++ code, exiting from a non-void function without returning a value is considered to be an undefined behavior. When optimizations are enabled, the compiler will often omit the epilogue in such functions. This most often results in an IllegalInstruction exception. By default, ESP-IDF build system enables `-Werror=return-type` which means that missing return statements are treated as compile time errors. However if the application project disables compiler warnings, this issue might go undetected and the IllegalInstruction exception will occur at run time.

InstrFetchProhibited

This CPU exception indicates that the CPU could not read an instruction because the address of the instruction does not belong to a valid region in instruction RAM or ROM.

Usually, this means an attempt to call a function pointer, which does not point to valid code. `PC` (Program Counter) register can be used as an indicator: it will be zero or will contain a garbage value (not `0x4xxxxxxx`).

LoadProhibited, StoreProhibited

These CPU exceptions happen when an application attempts to read from or write to an invalid memory location. The address which has been written/read is found in the `EXCVADDR` register in the register dump. If this address is zero, it usually means that the application has attempted to dereference a NULL pointer. If this address is close to zero, it usually means that the application has attempted to access a member of a structure, but the pointer to the structure is NULL. If this address is something else (garbage value, not in `0x3fxxxxxx` - `0x6xxxxxxx` range), it likely means that the pointer used to access the data is either not initialized or has been corrupted.

IntegerDivideByZero

Application has attempted to do an integer division by zero.

LoadStoreAlignment

Application has attempted to read or write a memory location, and the address alignment does not match the load/store size. For example, a 32-bit read can only be done from a 4-byte aligned address, and a 16-bit write can only be done to a 2-byte aligned address.

LoadStoreError

This exception may happen in the following cases:

- If the application has attempted to do an 8- or 16- bit read to, or write from, a memory region which only supports 32-bit reads/writes. For example, dereferencing a `char*` pointer to instruction memory (IRAM, IROM) will result in such an error.
- If the application has attempted to write to a read-only memory region, such as IROM or DROM.

Unhandled Debug Exception

This CPU exception happens when the instruction `BREAK` is executed.

Interrupt Watchdog Timeout on CPU0/CPU1

Indicates that an interrupt watchdog timeout has occurred. See *Watchdogs* for more information.

Cache disabled but cached memory region accessed

In some situations, ESP-IDF will temporarily disable access to external SPI Flash and SPI RAM via caches. For example, this happens when spi_flash APIs are used to read/write/erase/mmap regions of SPI Flash. In these situations, tasks are suspended, and interrupt handlers not registered with `ESP_INTR_FLAG_IRAM` are disabled. Make sure that any interrupt handlers registered with this flag have all the code and data in IRAM/DRAM. Refer to the *SPI flash API documentation* for more details.

4.17.7 Other Fatal Errors

Brownout

ESP32 has a built-in brownout detector, which is enabled by default. The brownout detector can trigger a system reset if the supply voltage goes below a safe level. The brownout detector can be configured using *CONFIG_ESP_BROWNOUT_DET* and *CONFIG_ESP_BROWNOUT_DET_LVL_SEL* options.

When the brownout detector triggers, the following message is printed:

```
Brownout detector was triggered
```

The chip is reset after the message is printed.

Note that if the supply voltage is dropping at a fast rate, only part of the message may be seen on the console.

Corrupt Heap

ESP-IDF's heap implementation contains a number of run-time checks of the heap structure. Additional checks ("Heap Poisoning") can be enabled in menuconfig. If one of the checks fails, a message similar to the following will be printed:

```
CORRUPT HEAP: Bad tail at 0x3ffe270a. Expected 0xbaad5678 got 0xbaac5678
assertion "head != NULL" failed: file "/Users/user/esp/esp-idf/components/heap/
↪multi_heap_poisoning.c", line 201, function: multi_heap_free
abort() was called at PC 0x400dca43 on core 0
```

Consult *Heap Memory Debugging* documentation for further information.