

# Avnet IoT API Reference

Contents

[Avnet IoT API Reference 1](file:///C:\cygwin64\home\043674\AvnetWNCSDK\Avnet_IoT_API_Reference.docx#_Toc494380873)

[Overview 3](#_Toc494380874)

[Hardware Application Programming Interfaces (APIs) 4](#_Toc494380875)

[i2c Operation 4](#_Toc494380876)

[I2C Variable Types 4](#_Toc494380877)

[i2c\_bus\_t 4](#_Toc494380878)

[i2c\_handle\_t 4](#_Toc494380879)

[i2c\_flag\_t 4](#_Toc494380880)

[I2C API(s) 5](#_Toc494380881)

[i2c\_bus\_init 5](#_Toc494380882)

[i2c\_bus\_deinit 5](#_Toc494380883)

[i2c\_read 6](#_Toc494380884)

[i2c\_write 8](#_Toc494380885)

[SPI Interface 9](#_Toc494380886)

[SPI Typedefs 9](#_Toc494380887)

[spi\_bus\_t 9](#_Toc494380888)

[spi\_mode\_t 10](#_Toc494380889)

[spi\_bpw\_t 11](#_Toc494380890)

[spi\_handle\_t 11](#_Toc494380891)

[SPI API(s) 12](#_Toc494380892)

[spi\_bus\_init 12](#_Toc494380893)

[spi\_bus\_deinit 12](#_Toc494380894)

[spi\_format 13](#_Toc494380895)

[spi\_frequency 13](#_Toc494380896)

[spi\_transfer 14](#_Toc494380897)

[GPIO Interface 16](#_Toc494380898)

[gpio typdefs 16](#_Toc494380899)

[gpio\_pin\_t 16](#_Toc494380900)

[gpio\_direction\_t 17](#_Toc494380901)

[gpio\_level\_t 17](#_Toc494380902)

[gpio\_handle\_t 17](#_Toc494380903)

[gpio\_irq\_trig\_t 18](#_Toc494380904)

[gpio\_irq\_callback\_fn\_t 18](#_Toc494380905)

[GPIO API(s) 18](#_Toc494380906)

[gpio\_init 18](#_Toc494380907)

[gpio\_deinit 19](#_Toc494380908)

[gpio\_is\_inited 19](#_Toc494380909)

[gpio\_dir 20](#_Toc494380910)

[gpio\_write 21](#_Toc494380911)

[gpio\_read 21](#_Toc494380912)

[gpio\_irq\_request 23](#_Toc494380913)

[gpio\_irq\_free 23](#_Toc494380914)

[ADC Interface 25](#_Toc494380915)

[ADC typedefs 25](#_Toc494380916)

[adc\_handle\_t 25](#_Toc494380917)

[ADC API(s) 25](#_Toc494380918)

[adc\_init 25](#_Toc494380919)

[adc\_deinit 26](#_Toc494380920)

[adc\_read 26](#_Toc494380921)

[Revision History 26](#_Toc494380922)

# Overview

The Avnet M18Qx LTE IoT module is built using the WNC M18Qx module. At the core of this module is a QUALCOMM MDM9207 (a quad-core ARM Cortex-A7 running at 1.2 GHz). Three of the four cores are dedicated to Radio/Network operation, but one of the cores is dedicated to running OpenEmbedded Linux to allow user applications to be run. Running Applications on the OpenEmbedded Linux core, is referred to as operating “Host Mode” applications. This document provides information on the WNC IoT APIs that are supported for the M18Qx when running in **Host-mode**.

Host-mode APIs provide allow a user to control the hardware interfaces on the WNC module. These APIs augment functionality already present in an OpenEmbedded Linux platform.

In addition to the hardware APIs outlined in this document, additional capabilities/APIs are available for controlling the Cellular operation of the module, please refer to the **Avnet\_JSON\_MAL\_APIs.docx**.

# Hardware Application Programming Interfaces (APIs)

# i2c Operation

I2C operation is fixed to 400 khz and cannot be changed.

## I2C Variable Types

### i2c\_bus\_t

|  |
| --- |
| ***Description***  Define the enumeration of supported I2C bus. |
| ***Declaration***  typedef enum \_i2c\_bus\_e {  I2C\_BUS\_I = 0,  I2C\_BUS\_MAX = I2C\_BUS\_I,  } i2c\_bus\_t; |
| ***Notes***  None |

### i2c\_handle\_t

|  |
| --- |
| ***Description***  Defines an I2C object for use in I2C API calls |
| ***Declaration***  typedef u\_int32\_t i2c\_handle\_t; |
| ***Notes***  None***.*** |

### i2c\_flag\_t

|  |
| --- |
| ***Description***  Enumeration for I2C bus behaviour |
| ***Declaration***  typedef enum \_i2c\_flag\_e {  I2C\_NO\_STOP = 0,  I2C\_STOP  } i2c\_flag\_t; |
| ***Notes***  None. |

## 

## I2C API(s)

### i2c\_bus\_init

|  |
| --- |
| ***Description***  Initialize I2C bus operation and obtain a handle to it for later use.  int i2c\_bus\_init( i2c\_bus\_t bus\_type, i2c\_handle\_t &handle\_ptr ); |
| ***Parameters***  bus\_type – i2c\_bus\_t variable indicates what type i2c bus to create  handle\_ptr – a pointer to the i2c object to be initialized  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

**Example:**

|  |
| --- |
| int ret = 0;  i2c\_handle\_t my\_handle = NULL;  if ((ret = i2c\_bus\_init(I2C\_BUS\_I, &my\_handle)) < 0){  printf(“Fail to initial I2C bus %d, ret:%d”, I2C\_BUS\_I, ret);  } |

### i2c\_bus\_deinit

|  |
| --- |
| ***Description***  Deinitialize the I2C object  int i2c\_bus\_deinit(i2c\_handle\_t &handle\_ptr ); |
| ***Parameters***  handle\_ptr – pointer to the I2C object to deinitialize  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

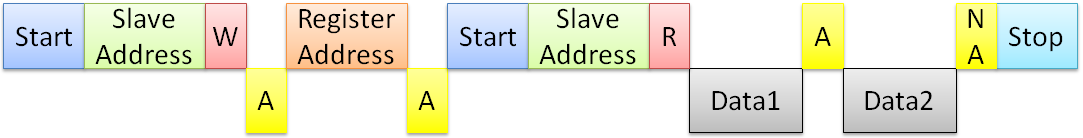
Only valid when invoked on an initialized i2c\_handle\_t object.

### i2c\_read

|  |
| --- |
| ***Description***  Perform an I2C read operation.  int i2c\_read( i2c\_handle\_t handle, uint16\_t slave\_address, unsigned char \*buffer, uint16\_t length) |
| ***Parameters***  handle – a previously initialized i2c\_handle\_t object  slave\_address – an 7-bit I2C slave address  buffer – a buffer pointer which is pointing at the destination buffer  length – the amount of data expected to be received (buffer length must be of sufficient size)  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes***  Only valid on an initialized i2c\_handle\_t object  Slave address is 7-bit and must be passed as x<<1 so the LSB is 0 as it is used for R/W indication.  Buffer length must be sufficient to contain received data length |

**Example:**

This example shows how to perform an I2C read sequence as the below depict:



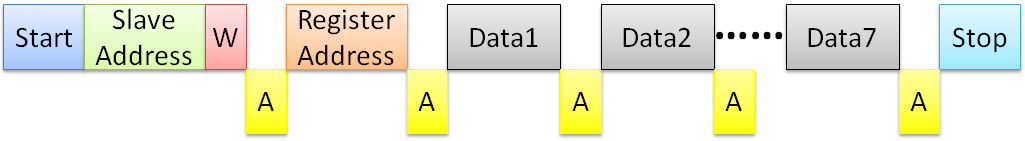
|  |
| --- |
| int ret = 0;  i2c\_handle\_t my\_handle = NULL;  unsigned char reg\_address[1] = { 0x16 };  unsigned char slave\_address[1] = { 0xA2 };  unsigned char buff\_read[2];  if ((ret = i2c\_bus\_init(I2C\_BUS\_I, &my\_handle)) < 0){  printf(“Fail to initial I2C bus %d, ret:%d”, I2C\_BUS\_I, ret);  exit (-1);  }  I2c\_bus\_write(my\_handle, slave\_address[0], reg\_address, 1, I2C\_NO\_STOP);  if ((ret = I2c\_bus\_read(my\_handle, slave\_address[0], buff\_read, 2)) < 0){  printf(“Fail to read register:%x from the I2C device, ret:%d”, reg\_address[0], ret);  i2c\_bus\_deinit(&my\_handle);  exit (-1);  } |
| printf(“Read buff\_read[0]:%x and buff\_read[1]:%x from register:%x”, buff\_read[0], buff\_read[1], reg\_address[0]);  i2c\_bus\_deinit(&my\_handle); |

### i2c\_write

|  |
| --- |
| ***Description***  Perform an I2C write sequence  int i2c\_write(i2c\_handle\_t handle, uint16\_t slaveAdd, unsigned char \*buff, uint16\_t len, i2c\_flag\_t stop) |
| ***Parameters***  handle – a previously initialized i2c\_handle\_t object  slaveAdd – an 7-bit I2C slave address  buff – a buffer containing the data to be sent  len – the amount of data to be sent  stop – the type of stop bit used after data is sent  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes***  Only valid on an initialized i2c\_handle\_t object  Slave address is 7-bit and must be passed as x<<1 so the LSB is 0 as it is used for R/W indication. |

**Example:**

This example shows how to perform an I2C write sequence:



|  |
| --- |
| int ret = 0;  i2c\_handle\_t my\_handle = NULL;  unsigned char reg\_address[1] = { 0x16 };  unsigned char buff\_write[8];  if ((ret = i2c\_bus\_init(I2C\_BUS\_I, &my\_handle)) < 0){  printf(“Fail to initial I2C bus %d, ret:%d”, I2C\_BUS\_I, ret);  exit (-1);  }  buff\_write[0] = reg\_address[0]; //the first byte is for a register address  buff\_write[1] = 0x01; //the following seven bytes are data1 ~ data7  buff\_write[2] = 0x02;  buff\_write[3] = 0x03;  buff\_write[4] = 0x04;  buff\_write[5] = 0x05;  buff\_write[6] = 0x06;  buff\_write[7] = 0x07;  if ((ret = I2c\_bus\_write(my\_handle, 0xA2, buff\_write, 8, I2C\_ STOP)) < 0) {  printf(“Fail to write data to register:%x from the I2C device, ret:%d”, reg\_address[0], ret);  i2c\_bus\_deinit(&my\_handle);  exit (-1);  } |

# SPI Interface

## SPI Typedefs

### spi\_bus\_t

|  |
| --- |
| ***Description***  Defines the enumeration for SPI bus devices.  ***spi\_bus\_t*** |
| ***Declaration***  typedef enum \_spi\_bus\_e {  SPI\_BUS\_I = 0,  SPI\_BUS\_II,  SPI\_BUS\_MAX = SPI\_BUS\_II,  } spi\_bus\_t; |
| ***Notes***  None |

### spi\_mode\_t

|  |
| --- |
| ***Description***  Provides defines for SPI bus supported modes  ***spi\_mode\_t*** |
| ***Declaration***  typedef enum \_spi\_mode\_e {  SPIMODE\_CPOL\_0\_CPHA\_0 = 0,  SPIMODE\_CPOL\_0\_CPHA\_1,  SPIMODE\_CPOL\_1\_CPHA\_0,  SPIMODE\_CPOL\_1\_CPHA\_1,  SPIMODE\_CPOL\_CPHA\_MAX=PIMODE\_CPOL\_1\_CPHA\_1  } spi\_mode\_t; |
| ***Notes***  None |

### 

### spi\_bpw\_t

|  |
| --- |
| ***Description***  Provides defines for SPI bandwidth  ***spi\_bpw\_t*** |
| ***Declaration***  typedef enum \_spi\_bpw\_e {  SPI\_BPW\_4 = 0,  SPI\_BPW\_5,  SPI\_BPW\_6,  SPI\_BPW\_7,  SPI\_BPW\_8,  SPI\_BPW\_9,  SPI\_BPW\_10,  SPI\_BPW\_11,  SPI\_BPW\_12,  SPI\_BPW\_13,  SPI\_BPW\_14,  SPI\_BPW\_15,  SPI\_BPW\_16,  SPI\_BPW\_17,  SPI\_BPW\_18,  SPI\_BPW\_19,  SPI\_BPW\_20,  SPI\_BPW\_21,  SPI\_BPW\_22,  SPI\_BPW\_23,  SPI\_BPW\_24,  SPI\_BPW\_25,  SPI\_BPW\_26,  SPI\_BPW\_27,  SPI\_BPW\_28,  SPI\_BPW\_29,  SPI\_BPW\_30,  SPI\_BPW\_31,  SPI\_BPW\_32,  SPI\_BPW\_MAX = SPI\_BPW\_32,  } spi\_bpw\_t; |
| ***Notes***  None |

### spi\_handle\_t

|  |
| --- |
| ***Description***  Defines a SPI object  ***spi\_handle\_t*** |
| ***Declaration***  typedef uint32\_t spi\_handle\_t; |
| ***Notes***  None |

## SPI API(s)

### spi\_bus\_init

|  |
| --- |
| ***Description***  Initialize a SPI bus object  ***int spi\_bus\_init(spi\_bus\_t which, spi\_handle\_t \*handle\_ptr)*** |
| ***Parameters***  ***which – define what type of SPI bus this is***  ***handle\_ptr – a pointer to an initialized SPI bus***  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

### spi\_bus\_deinit

|  |
| --- |
| ***Description***  Deinitialize a currently active SPI bus object  ***int spi\_bus\_deinit(spi\_handle\_t \*handle\_ptr)*** |
| ***Parameters***  ***handle\_ptr – a pointer to an initialized SPI bus***  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

### 

### spi\_format

|  |
| --- |
| ***Description***  Set the format of bus protocol of the SPI bus  ***int spi\_format( spi\_handle\_t handle, spi\_mode\_t mode, spi\_bpw\_t bits)*** |
| ***Parameters***  handle\_ptr – a pointer to the SPI bus to be configured  mode – Set the mode of CPHA and CPOL of the SPI bus.  Bits – Set the bit-per-word configure of the SPI bus.  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes***  This function can only be invoked after [spi\_bus\_init](#_Toc467159268)() initializes the SPI bus. |

### spi\_frequency

|  |
| --- |
| ***Description***  Configure the speed of the SPI bus  int spi\_frequency(spi\_handle\_t handle, uint32\_t Hz); |
| ***Parameters***  Handle – the SPI bus object to set the speed of  Hz -- the speed of the SPI bus  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

### 

### spi\_transfer

|  |
| --- |
| ***Description***  Perform a SPI transfer  int spi\_transfer( spi\_handle\_t handle, uint8\_t \*txBuff, uint32\_t txLen, uint8\_t \*rxBuff, uint32\_t rxLen); |
| ***Parameters***  Handle – the SPI bus object to send/receive data to  txBuff – a pointer to the transmit buffer  TxLen – number of bytes to TX  rxBuff – a pointer to the receive buffer  rxLen – the number of bytes to receive  ***Returns***  0 if successful, !0 if an error occurred |
| ***Notes*** |

**Example:**

|  |
| --- |
| int i;  int ret = 0;  spi\_handle\_t my\_spiII = 0;  ret = spi\_bus\_init(SPI\_BUS\_II, &my\_spiII);  if (ret != 0)  printf("Failed to initial SPI bus: %d, ret: %d\n", SPI\_BUS\_II, ret);  else {  uint8\_t spi\_mode = SPIMODE\_CPOL\_0\_CPHA\_0;  uint8\_t spi\_bits = SPI\_BPW\_8;  ret = spi\_format(my\_spiII, spi\_mode, spi\_bits);  if (ret != 0)  printf("Failed to set SPI mode: %d bits: %d, ret: %d\n", spi\_mode, spi\_bits, ret);  else {  uint32\_t frequency\_hz = 960000;  ret = spi\_frequency(my\_spiII, frequency\_hz);  if (ret != 0)  printf("Failed to set SPI speed: %d, ret: %d\n", frequency\_hz, ret);  else {  uint8\_t tx\_buffer[] = {  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  0x40, 0x00, 0x00, 0x00, 0x00, 0x95,  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,  };  uint32\_t tx\_length = ARRAY\_SIZE(tx\_buffer);  uint8\_t rx\_buffer[ARRAY\_SIZE(tx\_buffer)] = {0, };  uint32\_t rx\_length = ARRAY\_SIZE(rx\_buffer);  ret = spi\_transfer(my\_spiII, tx\_buffer, tx\_length, rx\_buffer, rx\_length);  if (ret != 0)  printf("Failed to transfer SPI data, ret: %d\n", ret);  else {  printf("spi mode: %d\n", spi\_mode);  printf("bits per word: %d\n", (spi\_bits+4));  printf("max speed: %d Hz (%d KHz)\n", frequency\_hz, frequency\_hz/1000);  for (i = 0; i < tx\_length; i++) {  if (!(i % 6))  puts("");  printf("%.2X ", rx\_buffer[i]);  }  puts("");  }  }  }  }  spi\_bus\_deinit(&my\_spiII); |

# 

# GPIO Interface

## gpio typdefs

### gpio\_pin\_t

|  |
| --- |
| ***Description***  Defines the enumeration of a GPIO pin.  ***gpio\_pin\_t*** |
| ***Declaration***  typedef enum \_gpio\_pin\_e {  GPIO\_PIN\_1 = 0,  GPIO\_PIN\_2,  GPIO\_PIN\_3,  GPIO\_PIN\_4,  GPIO\_PIN\_5,  GPIO\_PIN\_6,  GPIO\_PIN\_7,  GPIO\_PIN\_8,  GPIO\_PIN\_46,  GPIO\_PIN\_47,  GPIO\_PIN\_48,  GPIO\_PIN\_49,  GPIO\_PIN\_92,  GPIO\_PIN\_93,  GPIO\_PIN\_94,  GPIO\_PIN\_95,  GPIO\_PIN\_96,  GPIO\_PIN\_97,  GPIO\_PIN\_98,  GPIO\_PIN\_101,  GPIO\_PIN\_102,  GPIO\_PIN\_103,  GPIO\_PIN\_MAX = GPIO\_PIN\_103  } gpio\_pin\_t; |
| ***Notes***  Only the listed GPIO pins are supported. |

### 

### gpio\_direction\_t

|  |
| --- |
| ***Description***  Specify the direction of the GPIO  ***gpio\_direction\_t*** |
| ***Declaration***  typedef enum \_gpio\_direction\_e {  GPIO\_DIR\_OUTPUT = 0,  GPIO\_DIR\_INPUT,  GPIO\_DIR\_MAX = GPIO\_DIR\_INPUT,  } gpio\_direction\_t; |
| ***Notes***  None. |

### gpio\_level\_t

|  |
| --- |
| ***Description***  Used to specify the binary I/O level  ***gpio\_level\_t*** |
| ***Declaration***  typedef enum \_gpio\_level\_e {  GPIO\_LEVEL\_LOW = 0,  GPIO\_LEVEL\_HIGH,  GPIO\_LEVEL\_MAX = GPIO\_LEVEL\_HIGH,  } gpio\_level\_t; |
| ***Notes***  None. |

### gpio\_handle\_t

|  |
| --- |
| ***Description***  Used when defining a Binary I/O  ***gpio\_handle\_t*** |
| ***Declaration***  typedef uint32\_t gpio\_handle\_t; |
| ***Notes***  None. |

### 

### gpio\_irq\_trig\_t

|  |
| --- |
| ***Description***  Used to describe conditions that will cause an interrupt on a gpio |
| ***Declaration***  typedef enum \_gpio\_irq\_trig\_e {  GPIO\_IRQ\_TRIG\_RISING = 0,  GPIO\_IRQ\_TRIG\_FALLING,  GPIO\_IRQ\_TRIG\_BOTH,  GPIO\_IRQ\_TRIG\_MAX = GPIO\_IRQ\_TRIG\_BOTH,  } gpio\_irq\_trig\_t; |
| ***Notes*** |

### gpio\_irq\_callback\_fn\_t

|  |
| --- |
| ***Description***  Used when defining the function to call back a gpio interrupt |
| ***Declaration***  typedef int (\*gpio\_irq\_callback\_fn\_t)(gpio\_pin\_t pin\_name, gpio\_irq\_trig\_t direction); |
| ***Notes*** |

## GPIO API(s)

### gpio\_init

***Description***

Initialize a GPIO pin

|  |
| --- |
| ***int gpio\_init(gpio\_pin\_t pin\_name, gpio\_handle\_t \*handle\_ptr);*** |
| ***Parameters***  pin\_name – the gpio\_pin\_t name that is to be initialized  handle\_ptr – a pointer to a gpio\_handle\_t object  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes***  None. |

### gpio\_deinit

|  |
| --- |
| ***Description***  Deinitialize a previously initialized gpio  ***int gpio\_deinit(gpio\_handle\_t \*handle\_ptr);*** |
| ***Parameters***  handle\_ptr – a pointer to a gpio\_handle\_t object  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

### gpio\_is\_inited

|  |
| --- |
| ***Description***  Checks to see if the GPIO pin has been initialized  ***int gpio\_is\_inited(const gpio\_handle\_t handle);*** |
| ***Parameters***  handle\_ptr – the gpio\_handle\_t object to check  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

### 

### gpio\_dir

|  |
| --- |
| ***Description***  Configure the direction of the GPIO pin  ***int gpio\_dir(gpio\_handle\_t handle, gpio\_direction\_t direction);*** |
| ***Parameters***  handle – the gpio\_handle\_t object to the pin to set  direction – the direction to set the pin to  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes***  This function can only be invoked after [gpio\_init](#_Toc467159280)() initializes the the GPIO pin. |

### gpio\_write

|  |
| --- |
| ***Description***  Sets to logic level of the specified GPIO pin  ***int gpio\_write(gpio\_handle\_t handle, gpio\_level\_t value);*** |
| ***Parameters***  handle – the gpio\_handle\_t object to write to  direction – the value to set the pin to  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

### gpio\_read

|  |
| --- |
| ***Description***  Read the current logic level of a gpio pin  ***int gpio\_read(gpio\_handle\_t handle, gpio\_level\_t \*value);*** |
| ***Parameters***  handle – the gpio\_handle\_t object to write to  value – the variable to hold the current logic level  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

**Example:**

|  |
| --- |
| gpio\_handle\_t my\_gpio = NULL;  int ret;  gpio\_level\_t value;  while ( (ret = gpio\_init(GPIO\_PIN\_7, &my\_gpio)) != 0 )  /\* try again \*/;  if ( gpio\_is\_inited(my\_gpio) ) {  gpio\_dir(my\_gpio, GPIO\_DIR\_OUTPUT); // set it to be an output pin  gpio\_write(my\_gpio, GPIO\_LEVEL\_HIGH);  gpio\_dir(my\_gpio, GPIO\_DIR\_INPUT); // set it to be an input pin  gpio\_read(my\_gpio, &value);  gpio\_deinit(&my\_gpio); // release the instance  } |

### gpio\_irq\_request

|  |
| --- |
| ***Description***  Register an interrupt function for a GPIO and set up the callback handler  ***int gpio\_irq\_request(gpio\_handle\_t handle, gpio\_irq\_trig\_t trigger, gpio\_irq\_callback\_fn irq\_callback)*** |
| ***Parameters***  handle – the gpio\_handle\_t object that contains the gpio to attach this too  trigger – the type of event that will generate an interrupt  callback – the function to call when the interrupt occurs  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

### gpio\_irq\_free

|  |
| --- |
| ***Description***  De-register the interrupt function for the GPIO.  ***int gpio\_irq\_free(gpio\_handle\_t handle);*** |
| ***Parameters***  handle – the gpio\_handle\_t object containing the interrupt gpio  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes*** |

**Example:**

|  |
| --- |
| static int gpio07\_irq\_callback(gpio\_pin\_t pin\_name, gpio\_irq\_trig\_t direction) {  if (pin\_name != GPIO\_PIN\_7)  return 0;  // Please implement what is supposed to be handled here.  }  int main(int argc, char \*argv[])  {  gpio\_handle\_t my\_interrupt = NULL;  int ret;  gpio\_level\_t value;  while ( (ret = gpio\_init(GPIO\_PIN\_7, &my\_interrupt)) != 0 )  /\* try again \*/  if ( gpio\_is\_inited(my\_interrupt) ) {  // register it to be an interrupt with falling edge triggering  gpio\_irq\_request(my\_interrupt, GPIO\_IRQ\_TRIG\_FALLING, gpio07\_irq\_callback);  /\*  Here is implementation of main function, such as dispatch of events,  event handlings...  \*/  gpio\_irq\_free(my\_interrupt); // free the interrupt  gpio\_deinit(&my\_interrupt); // release the instance  }  } |

# ADC Interface

## ADC typedefs

### adc\_handle\_t

|  |
| --- |
| ***Description***  Type for an ADC object that is used in subsequent ADC API calls |
| ***Declaration***  typedef uint32\_t adc\_handle\_t; |
| ***Notes***  None. |

## ADC API(s)

### adc\_init

|  |
| --- |
| ***Description***  Initialize an ADC object  ***adc\_init( &handle\_ptr )*** |
| ***Parameters***  handle\_ptr – pointer to a adc\_handle\_t object |
| ***Return Value***  0 if successful, !0 if an error occurred  ***Notes***  None. |

### adc\_deinit

|  |
| --- |
| ***Description***  Deinitialize the ADC functionality and destroy ADC object.  ***adc\_deinit( adc\_handle\_t \*handle\_ptr )*** |
| ***Parameters***  handle\_ptr – pointer to the adc\_handle\_t to deinitialize  ***Return Value***  0 if successful, !0 if an error occurred |
| ***Notes***  This function can only be invoked on a currently initialized adc\_handle\_t object |

### adc\_read

|  |
| --- |
| Perform an ADC measurement and get the result.  ***adc\_read( handle, &adc\_value)*** |
| ***Parameters***  handle – The [adc\_handle\_t](#adc_handle_t) to perform a measurement from.  adc\_value -- pointer to a float variable to store reading in  **Return Value**  If successfully, return 0; else a none-zero negative integer. |
| ***Notes***  This function can only be invoked after a call to [adc\_init](#adc_init)(). |

# Revision History

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Revision** |
| 28 Sept 17 | 00 | Initial Draft |