Decawave Is Now qorvo

DW3xxx DEVICE DRIVER APPLICATION PROGRAMMING INTERFACE (API) GUIDE

USING API FUNCTIONS TO CONFIGURE AND PROGRAM THE DW3000 and QM33120 UWB TRANSCEIVER

This document is subject to change without notice

DOCUMENT INFORMATION

Disclaimer

Decawave reserves the right to change product specifications without notice. As far as possible changes to functionality and specifications will be issued in product specific errata sheets or in new versions of this document. Customers are advised to check the Decawave website for the most recent updates on this product

LIFE SUPPORT POLICY

Decawave products are not authorized for use in safety-critical applications (such as life support) where a failure of the Decawave product would reasonably be expected to cause severe personal injury or death. Decawave customers using or selling Decawave products in such a manner do so entirely at their own risk and agree to fully indemnify Decawave and its representatives against any damages arising out of the use of Decawave products in such safety-critical applications.

Caution! ESD sensitive device.

Precaution should be used when handling the device in order to prevent permanent damage

DISCLAIMER

This Disclaimer applies to the DW3xxx API source code (collectively “Decawave Software”) provided by Decawave Ltd. (“Decawave”).

Downloading, accepting delivery of or using the Decawave Software indicates your agreement to the terms of this Disclaimer. If you do not agree with the terms of this Disclaimer do not download, accept delivery of or use the Decawave Software.

Decawave Software is solely intended to assist you in developing systems that incorporate Decawave semiconductor products. You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your systems and products. THE DECISION TO USE DECAWAVE SOFTWARE IN WHOLE OR IN PART IN YOUR SYSTEMS AND PRODUCTS RESTS ENTIRELY WITH YOU.

DECAWAVE SOFTWARE IS PROVIDED "AS IS". DECAWAVE MAKES NO WARRANTIES OR REPRESENTATIONS WITH REGARD TO THE DECAWAVE SOFTWARE OR USE OF THE DECAWAVE SOFTWARE, EXPRESS, IMPLIED OR STATUTORY, INCLUDING ACCURACY OR COMPLETENESS. DECAWAVE DISCLAIMS ANY WARRANTY OF TITLE AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO DECAWAVE SOFTWARE OR THE USE THEREOF.

DECAWAVE SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY THIRD PARTY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON THE DECAWAVE SOFTWARE OR THE USE OF THE DECAWAVE SOFTWARE WITH DECAWAVE SEMICONDUCTOR TECHNOLOGY. IN NO EVENT SHALL DECAWAVE BE LIABLE FOR ANY ACTUAL, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, INCLUDING WITHOUT LIMITATION TO THE GENERALITY OF THE FOREGOING, LOSS OF ANTICIPATED PROFITS, GOODWILL, REPUTATION, BUSINESS RECEIPTS OR CONTRACTS, COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION), LOSSES OR EXPENSES RESULTING FROM THIRD PARTY CLAIMS. THESE LIMITATIONS WILL APPLY REGARDLESS OF THE FORM OF ACTION, WHETHER UNDER STATUTE, IN CONTRACT OR TORT INCLUDING NEGLIGENCE OR ANY OTHER FORM OF ACTION AND WHETHER OR NOT DECAWAVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF DECAWAVE SOFTWARE OR THE USE OF DECAWAVE SOFTWARE.

You are authorized to use Decawave Software in your end products and to modify the Decawave Software in the development of your end products. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER DECAWAVE INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY THIRD PARTY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT, IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which Decawave semiconductor products or Decawave Software are used.

You acknowledge and agree that you are solely responsible for compliance with all legal, regulatory and safety-related requirements concerning your products, and any use of Decawave Software in

your applications, notwithstanding any applications-related information or support that may be provided by Decawave.

Decawave reserves the right to make corrections, enhancements, improvements and other changes to its software at any time.

Mailing address: -
Decawave Ltd.,
Adelaide Chambers,
Peter Street,
Dublin D08 T6YA

TABLE OF CONTENTS

List of Tables

List of Figures

1 INTRODUCTION AND OVERVIEW

The DW3xxx IC is a radio transceiver IC implementing the UWB HRP physical layer defined in IEEE 802.15.4 standard [3]. For more details of this device the reader is referred to:

- The Data Sheet [1]
- The User Manual [2]

This document, "*DW3xxx Device Driver - Application Programming Interface (API) Guide*" is a guide to the device driver software developed by Decawave to drive Decawave's family of UWB radio transceiver ICs: DW3000 and QM33120.

The device driver is essentially a set of low-level functions providing a means to exercise the main features of the transceiver without having to deal with the details of accessing the device directly through its SPI interface register set.

The device driver is provided as source code to allow it to be ported to any target microprocessor system with an SPI interface[1]. The source code employs the C programming language.

The device driver is controlled through its Application Programming Interface (API) which is comprised of a set of functions. This document is predominately a guide to the device driver API describing each of the API functions in detail in terms of its parameters, functionality and utility.

This document relates to: `"DW3xxx Device Driver Version 06.00.xx"`

The device driver version information may be found in source code file "deca_version.h".

[1] Since the DW3xxx IC is controlled through its SPI interface, an SPI interface is a mandatory requirement for the system.

2 GENERAL FRAMEWORK

Figure 1 shows the general framework of the software system encompassing the DW3xxx device driver. The device driver controls the IC through its SPI interface. The device driver abstracts the target SPI device by calling it through generic functions *writetospi()* and *readfromspi ()*. In porting the IC device driver to different target hardware, the body of these SPI functions are written/re-written/provided to drive the target microcontroller device's physical SPI hardware. The initialisation of the physical SPI interface mode and data rate is considered to be part of the target system outside the IC device driver.

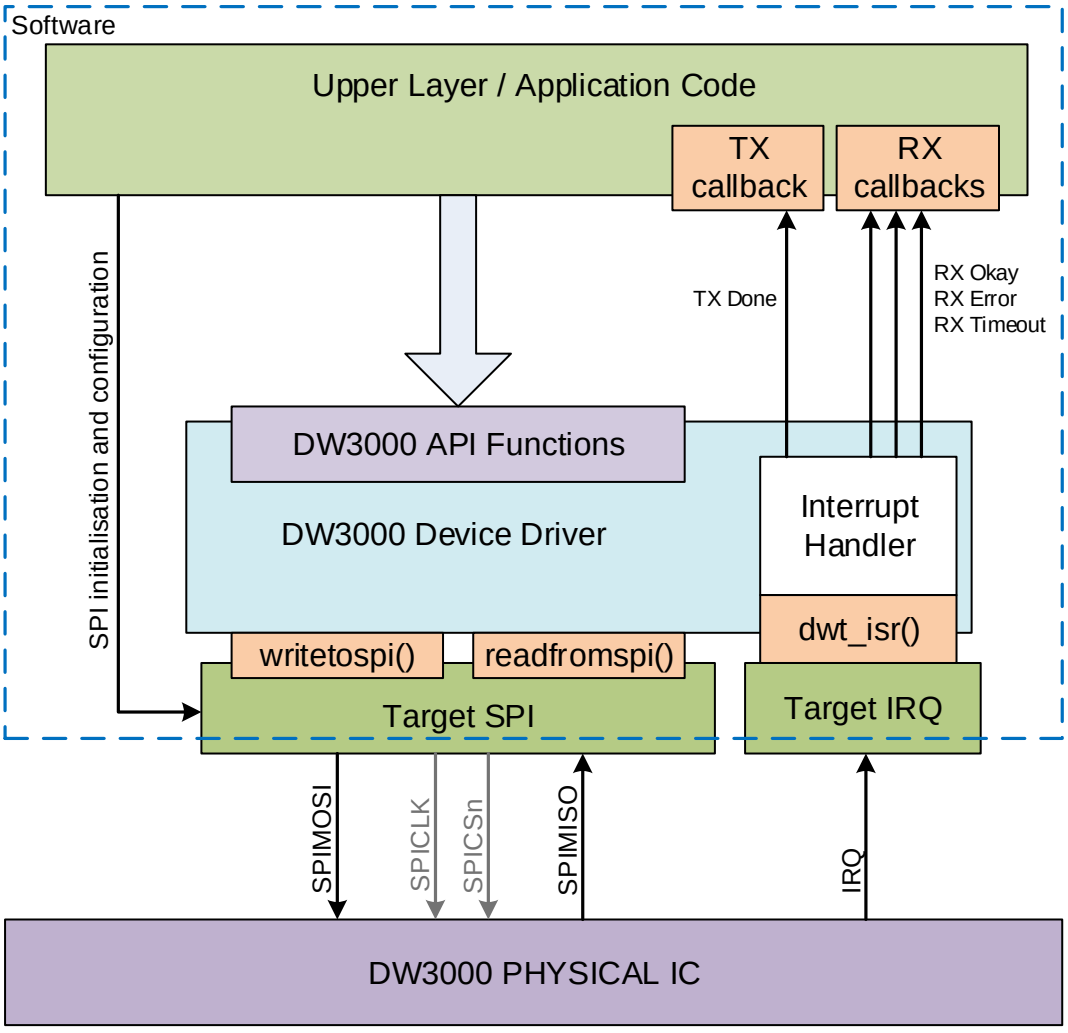


Figure 1: General software framework of the device driver

The control of the IC through the device driver software is achieved via a set of API functions, documented in section 5 *– API function descriptions* below, and called from the upper layer application code.

The IRQ interrupt line output from the IC (assuming interrupts are being employed) is connected to the target microcontroller system's interrupt handling logic. Again, this is considered to be outside the device driver. It is assumed that the target systems interrupt handling logic and its associated

target specific interrupt handling software will correctly identify the assertion of the IC's IRQ and will as a result call the device driver's interrupt handling function *dwt_isr()* to process the interrupt.

The device driver's *dwt_isr ()* function processes the IC interrupts and calls TX, RX, RX error, RX timeout, SPI error or SPI ready call-back functions in the upper layer application code. This is done via function pointers **cbTxDone(), *cbRxOk(), *cbRxTo, *cbRxErr(), *cbSPIErr()* and **cbSPIRdy()* or **dualSPIavailable()* which are configured to call the upper layer application code's own call-back functions via the *dwt_setcallbacks ()* API function.

Using interrupts is recommended, but it is possible to drive the IC without employing interrupts. In this case the background loop can periodically call the device driver's *dwt_isr ()* function, which will poll the IC status register and process any events that are active.

The following is IMPORTANT:

Note *background* application activity invoking API functions employing the SPI interface can conflict with *foreground* interrupt activity also needing to employ the SPI interface.

The device driver's interrupt handler accesses the IC through the *writetospi()* and *readfromspi ()* functions, and, it is generally expected that the call-back functions will also access the IC through the device driver's API functions which ultimately also call the *writetospi()* and *readfromspi ()* functions.

This means that the *writetospi()* and *readfromspi ()* functions need to incorporate protection against *foreground* activity occurring when they are being used in the *background*. This is achieved by incorporating calls to *decamutexon ()* and *decamutexoff()* within the *writetospi()* and *readfromspi ()* functions to disable interrupts from the IC from being recognised while the *background* SPI access is in progress.

Examples of be *decamutexon()*and *decamutexoff()* within the *writetospi()* and *readfromspi()* functions found in source code file "deca_irq.c" and the definitions of the *writetospi()* and *readfromspi()* functions in "deca_spi.c" source file.

Other than the provisions for interrupt handling, the device driver and its API functions are not written to be re-entrant or for simultaneous use by multiple threads. The design in general assumes a single caller that allows each function to complete before it is called again.

2.1 Compatibility Layer

The driver also includes a "compatibility layer" that sits within the device driver. Its purpose is to "route" the API calls to the correct function for the calling device. For example, if device "A" wants to check the version of the API, the compatibility layer will route it to the correct code for device "A". If device "B" wants to do the same, it will route it to the correct code for device "B". However, the upper layer / application code will only need to call one API and the device driver will route to the correct device by itself.

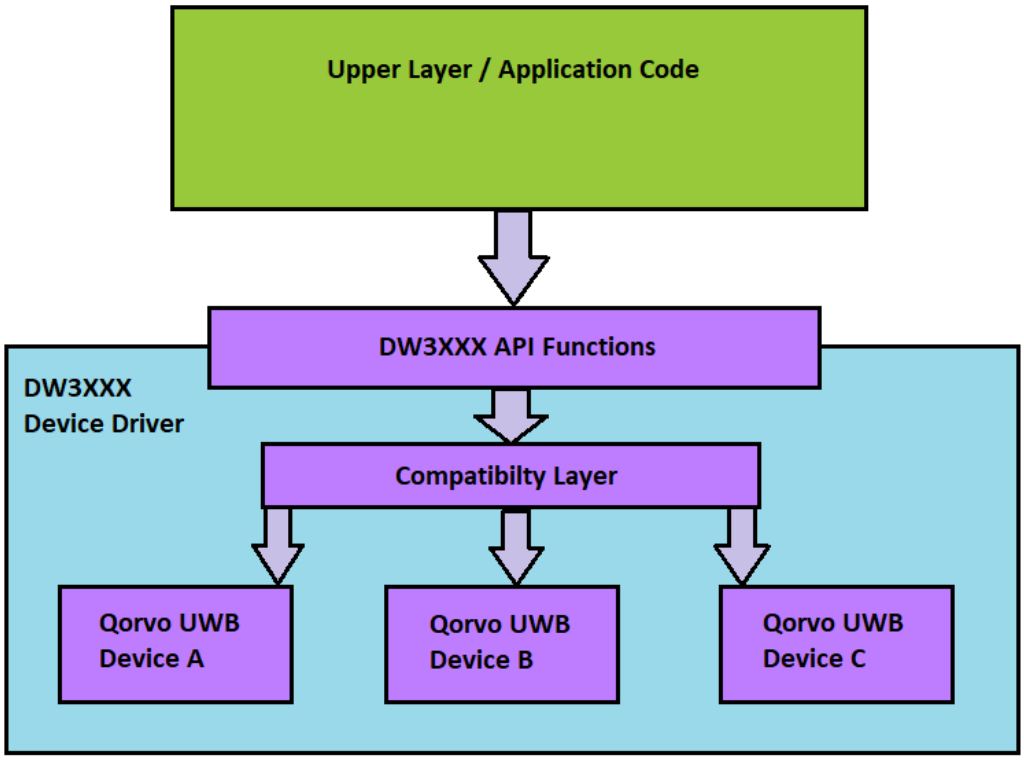


Figure 2: Device Driver Compatibility Layer

The main purpose of this compatibility layer is to allow for inter-operability between Qorvo/Decawave UWB devices of the same family. However, it is only implemented for one device at present. Future releases will allow for compatibility with other UWB devices of the same family.

2.1.1 Device Descriptor Structures Stored in Memory

Each of the Qorvo/Decawave UWB devices that are supported by the device driver will need to populate the device descriptor structure and store it into flash memory (or equivalent, presuming an embedded platform). For example, the DW3000 device descriptor is declared like so:

```
const struct dwt_driver_s dw3000_driver __attribute__((section(".dw_drivers"))) = {
    .devid = DWT_DW3000_PDOA_DEV_ID,
    .devmatch = 0xffffff0f,
    .name = DRIVER_NAME,
    .version = DRIVER_VERSION_STR,
    .dwt_ops = &dw3000_ops,
    .dwt_mcps_ops = &dw3000_mcps_ops,
    .vernum = DRIVER_VERSION_HEX
};
```

This package contains support for both the STM Nucleo F429 and Nordic nRF52840-DK evaluation boards. For the STM board, the linker script (STM32F429ZITx_FLASH.ld) is changed to include this structure in the ".text" segment of memory like so:

```
    . = ALIGN(4);
  __dw_drivers_start = .;
  KEEP(*(.dw_drivers))
  __dw_drivers_end = .;
```

For the Nordic board, the linker script (flash_placement.xml) is changed to include this structure in the ".text" segment of memory like so:

```
<ProgramSection alignment="4" keep="Yes" load="Yes"
name=".dw_drivers" inputsections="*(SORT(.dw_drivers*))"
address_symbol="__dw_drivers_start" end_symbol="__dw_drivers_end" />
```

Any additional device descriptor structures that are declared must be stored between "__dw_drivers_start" and "__dw_driver_end" addresses in memory.

3 TYPICAL SYSTEM START-UP

Figure 3 shows the typical flow of initialisation of the DW3xxx in a microprocessor system.

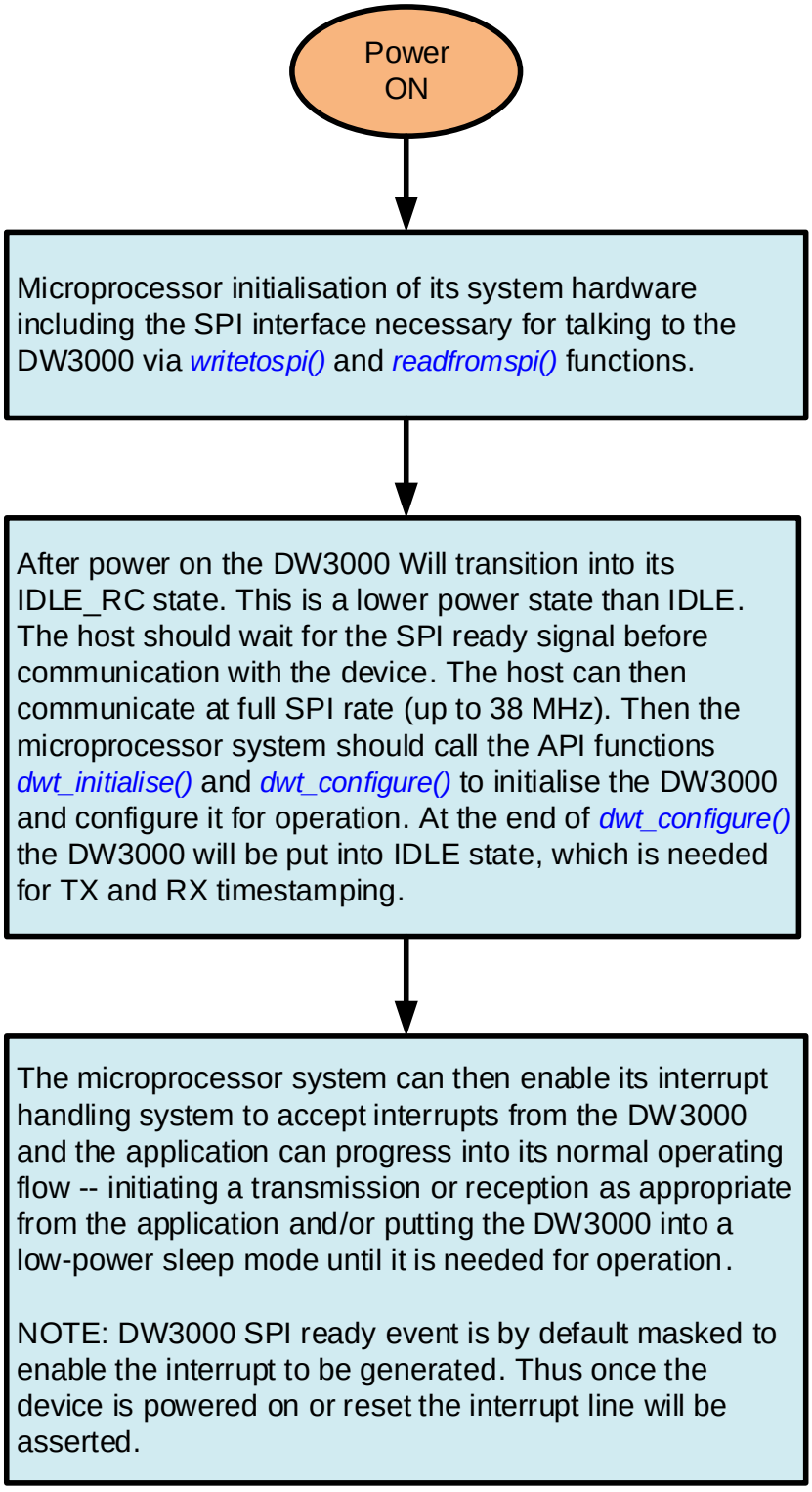


Figure 3: Typical flow of initialisation

4 INTERRUPT HANDLING

Figure 4 shows how the DW3xxx interrupts should be processed by the microcontroller system. Once the interrupt is active, the microcontroller's target specific interrupt handler for that interrupt line should get called. This in turn calls the device driver's interrupt handler service routine, the *dwt_isr()* API function, which processes the event that triggered the interrupt.

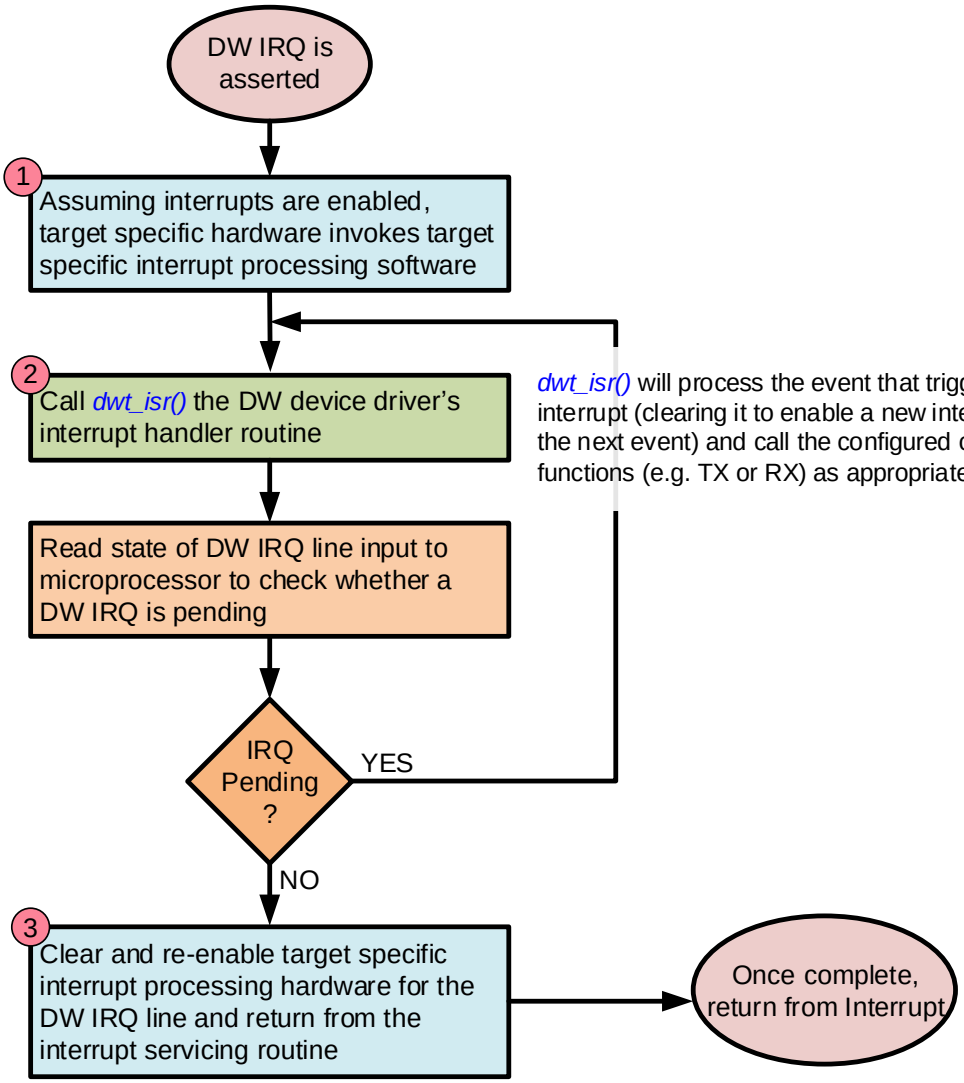


Figure 4: Interrupt handling

The flow shown above, with the rechecking of continued IRQ line activation and calling the *dwt_isr()* API function again, is only required for edge sensitive interrupts. This is done in case another interrupt becomes pending during the processing of the first interrupt, in this case if all interrupt sources are not cleared the IRQ line will not be de-asserted and edge sensitive interrupt processing hardware will not see another edge. For proper level sensitive interrupts only steps numbered 1, 2, and 3 are required – any still pending interrupt should cause the interrupt handler to be re-invoked as soon as it finishes processing the first interrupt.

More information about individual interrupt events and associated processing is shown in Figure 7: Interrupt handling.

5 API FUNCTION DESCRIPTIONS

This section describes device driver's API function calls. The API functions are provided to aid developers in driving the DW3xxx (Decawave's IEEE 802.15.4 [3] UWB transceiver IC).

These functions are implemented in the device driver source code file "deca_device.c", written in the 'C' programming language.

The device driver code interacts with the IC using simple SPI read and write functions. These are abstracted from the physical hardware, and are easily ported to any specific SPI implementation of the target system. There are two SPI functions: *writetospi()* and *readfromspi()* these prototypes are defined in the source code file "deca_spi.c".

The functions of the device driver are covered below in individual sub-sections.

5.1 Initialise APIs

5.1.1 dwt_probe

int dwt_probe(void);

This function will read the device identifier (DEV_ID) from the device and initialise all the required pointers for the API calls to pass through the Compatibility Layer based on the device identifier. This function must be called first in any application. Otherwise, all other subsequent API calls will fail as they will not be able to pass through the compatibility layer correctly.

On embedded platforms, a section of the flash memory must be reserved to hold the device structure (struct dwt_driver_s) that contains information such as device identifiers, device names, device API versions, etc. Please see Device Descriptor Structures Stored in Memory for more information.

This section of flash memory will have a start address and end address. Data structures containing information on the available devices will be stored in between the start and end address of this memory block.

If the DEV_ID that was read from the device matches the DEV_ID that is stored in the device descriptor structure in memory, then this function will set the compatibility layer to 'point' to the correct code.

Parameters:

none

Return Parameters:

Type	Description
int	DWT_SUCCESS if the DEV_ID read from the device matches a device descriptor structure stored in memory

	DWT_ERROR if no DEV_ID can be read from device, or if the DEV_ID does not match what is stored in memory

Notes:

5.1.2 dwt_apiversion

int32_t dwt_apiversion(void);

This function returns the version of the API as defined by DRIVER_VERSION_HEX.

Parameters:

none

Return Parameters:

Type	Description
int32_t	Driver version e.g. 0x050600

Notes:

5.1.3 dwt_version_string

char *dwt_version_string(void);

This function returns the version string of the API as defined by device descriptor structure (struct dwt_driver_s). It will return a char pointer to the string: DRIVER_VERSION_STR.

Parameters:

none

Return Parameters:

Type	Description
char*	char pointer to device descriptor version name (e.g. DW3000 Device Driver Version 05.00.00").

Notes:

5.1.4 dwt_readdevid

uint32_t dwt_readdevid(void);

This function returns the device identifier (DEV_ID) register value (32-bit value). It reads the DEV_ID register (0x00) and returns the result to the caller. This may be used for instance by the application to verify the DW IC is connected properly over the SPI bus and is running.

Parameters:

none

Return Parameters:

Type	Description
uint32_t	32-bit device ID value, e.g. for QM33120 the device ID is 0xDECA0314.

Notes:

This function can be called any time to read the device ID value. A return value of 0xFFFFFFFF or 0x0 indicates an error unless the device is in DEEP_SLEEP or SLEEP mode.

Example code:

```
uint32_t devID = dwt_readdevid();
```

5.1.5 dwt_check_dev_id

int dwt_check_dev_id(void);

This function checks if the device ID that is being read back from the DEV_ID register is a correct value. If the device ID is incorrect, it can mean that either something is wrong with the SPI reads to the DW3XXX, or the driver is reading a device ID from an incompatible version of hardware. The latter can occur if a newer version of the software is used to try read from an older version of hardware. It is useful to run this function upon initialisation.

Parameters:

none

Return Parameters:

Type	Description
int	DWT_SUCCESS is returned if the device ID read back from the device is correct. DWT_ERROR is returned if the device ID does not conform with this version of software.

Notes:

Calling this function upon initialisation can be beneficial in cases where there is potential for versions of software and hardware to be out of sync.

Example code:

```
	/* Reads and validate device ID returns DWT_ERROR if it does not match
expected else DWT_SUCCESS */
	int err;

	if ((err=dwt_check_dev_id())==DWT_SUCCESS)
	{
	    printf("DEV ID OK\n");
	}
	else
	{
	    printf("DEV ID FAILED\n");
	}
```

5.1.6 dwt_getpartid

uint32_t dwt_getpartid(void);

This function returns the part identifier as programmed in the factory during device test and qualification. (*dwt_initialise()* must be called prior to this)

Parameters:

none

Return Parameters:

Type	Description
uint32_t	32-bit part ID value.

Notes:

This function can be called any time to read the locally stored value which will be valid after device initialisation is done by a call to the *dwt_initialise ()* function.

Example code:

```
	uint32_t partID = dwt_getpartid();
```

5.1.7 dwt_getlotid

uint32_t dwt_getlotid(void);

This function returns the lot identifier as programmed in the factory during device test and qualification.(*dwt_initialise()* must be called prior to this)

Parameters:

none

Return Parameters:

Type	Description
uint32_t	32-bit lot ID value.

Notes:

This function can be called any time to read the locally stored value which will be valid after device initialisation is done by a call to the *dwt_initialise()* function .

Example code:

```
uint32_t lotID = dwt_getlotid();
```

5.1.8 dwt_geticrefvolt

uint8_t dwt_geticrefvolt(void);

During the IC manufacturing test, a 3.0 volt reference level is applied to the power the device and the battery voltage reported by the battery voltage monitor SAR A/D convertor is sampled and programmed into OTP address 0x8 (VBAT_ADDRESS). This reference value may be used to calibrate/interpret battery voltage monitor values during IC use. The *dwt_geticrefvolt()* function returns this factory reference voltage value.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	8-bit V measured value at 3.0 V.

Notes:

This function can be called any time to read the locally stored value which will be valid after device initialisation is done by a call to the *dwt_initialise() API* function.

5.1.9 dwt_geticreftemp

uint8_t dwt_geticreftemp(void);

During the IC manufacturing test, in a controlled environment with approximately 23 °C ambient temperature the temperature monitor SAR A/D convertor is sampled and programmed into OTP address 0x9 (VTEMP_ADDRESS). This reference value may be used to calibrate/interpret temperature monitor values during IC use. The *dwt_geticreftemp()* API function returns this factory reference temperature value.

Parameters:

none

Return Parameters:

Type	Description

uint8_t	8-bit Temperature measured value at 23 ˚C.

Notes:

This function can be called any time to read the locally stored value which will be valid after device initialisation is done by a call to the *dwt_initialise() API* function.

5.1.10 dwt_getxtaltrim

uint8_t dwt_getxtaltrim(void);

This function returns the current value of XTAL trim. If called after *dwt_initialise() API* on power up, it will either contain crystal trim value loaded from OTP memory or a default value.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	Current crystal trim value.

Notes:

5.1.11 dwt_setlocaldataptr

int dwt_setlocaldataptr(unsigned int index);

The DW3xxx API uses an internal data structure to hold some local state data. The device driver is able to handle multiple DW3xxx devices by using an array of those structures, as set by the #define of the DWT_NUM_DW_DEV pre-processor symbol. This *dwt_setlocaldataptr()* API function sets the local data structure pointer to point to the element in the local array as given by the index.

Parameters:

Type	Name	Description
unsigned int	index	This selects the array element to point to. Must be within the array bounds, i.e. < DWT_NUM_DW_DEV.

Return Parameters:

Type	Description

int	Return value can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

The local device static data is an array to support multiple devices, e.g. in testing applications and platforms. This function selects which element of the array is being accessed. For example, if two QM33120 devices are controlled in your application then this function should be called before accessing either of the devices to configure the local structure pointer. To handle multiple devices the low-level SPI access function also needs to be set to talk to the correct device.

5.1.12 dwt_otprevision

uint8_t dwt_otprevision(void);

This function returns OTP revision as read during the call to *dwt_initialise()*. This location is suggested for customer programming, (and is used in Decawave's evaluation board products to identify different/changes in usage of the OTP area).

Parameters:

none

Return Parameters:

Type	Description
uint8_t	8-bit OTP revision value.

5.1.13 dwt_softreset

void dwt_softreset(int reset_semaphore);

This function performs a software-controlled reset of the transceiver IC. All of the IC configurations will be reset back to default. Please refer to the User Manual [2] for details of IC default configuration register values. **The SPI rate must be set to <= 7 MHz before a calling this API.**

Parameters:

Type	Name	Description
int	reset_semaphore	The API 3000 does not support this parameter. This parameter should be 0.

Return Parameters:

none

Notes:

This function is used to reset the IC, e.g. before applying new configuration to clear all of the previously set values. After reset the IC will be in the INIT state, and all of the registers will have default values. Any values programmed into the always on (AON) low-power configuration array store will also be cleared. Then it will progress to IDLE_RC state. Once in IDLE_RC the SPI_RDY event will be set.

Note: The RSTn pin can also be used to reset the device. Host microprocessor can use this pin to reset the device instead of calling *dwt_softreset()* function. The pin should be driven low (for 10 ns) and then left in open-drain mode. **RSTn pin should never be driven high.**

5.1.14 dwt_checkidlerc

uint8_t dwt_checkidlerc(void);

The DW3XXX states are described in the User Manual. On power up, or following a reset the device will progress from INIT_RC to IDLE_RC. Once the device is in IDLE_RC SPI rate ca be increased to more than 7 MHz. The device will automatically proceed from INIT_RC to IDLE_RC and both INIT_RC and SPI_RDY event flags will be set, once device is in IDLE_RC. It is recommended that host waits for SPI_RDY event, which will also generate interrupt once device is ready after reset/power on. If the host cannot use interrupt as a way to check device is ready for SPI comms, then we recommend the host waits for 2 ms and reads this function, which checks if the device is in IDLE_RC state by reading the SYS_STATUS register and checking for the IDLE_RC event to be set. If host initiates SPI transaction with the device prior to it being ready, the SPI transaction may be incorrectly decoded by the device and device may be misconfigured. Reading registers over SPI prior to device being ready may return garbage on the MISO, which may confuse the host application.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	Return value is 1 if device is in IDLE_RC state and 0 otherwise.

Notes:

It is advised to call this function before calling *dwt_initialise()* in order to check that the device is in the correct state before initializing.

Example code:

```
	/* Reads and validate device ID returns DWT_ERROR if it does not match
expected else DWT_SUCCESS */
	int err;

	if ((err=dwt_check_dev_id())==DWT_SUCCESS)
	{
	    printf("DEV ID OK\n");
	}
```

```
else
{
    printf("DEV ID FAILED\n");
}
```

5.1.15 dwt_initialise

int dwt_initialise(int mode);

The *dwt_initialise()* is function which initialises the DW3xxx transceiver and sets up values in an internal static data structure used within the device driver functions. This static data is private data used within the device driver implementation.

Parameters:

Type	Name	Description
int	mode	This is a bitmask which specifies which configuration to load from OTP as part of initialisation Table 1 shows the values of individual bit fields

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

This *dwt_initialise()* function is the first function that should be called to initialise the device, e.g. after the power has been applied. It reads the device ID to verify the IC is one supported by this software (e.g. for QM33120 the 32-bit device ID value is 0xDECA0314).

This *dwt_initialise()* function also reads some data from OTP:

- LDO tune and crystal trim values, which are applied directly if they are valid.
- Device's Part ID and Lot ID which are stored in driver's local structure for future access.

If the DWT_ERROR is returned by *dwt_initialise()* then further configuration and operation of the IC is not advised, as the IC will not be functioning properly.

Table 1: Config parameter to dwt_initialise() function

Mode	Mask Value	Description
DWT_DW_INIT	0x0	Do not load any OTP values, also leave the device in INIT state.
DWT_READ_OTP_PID	0x10	Reads part ID from OTP, and stores it in internal structure. The *dwt_getpartid()* API can then be used to access it.
DWT_READ_OTP_LID	0x20	Reads lot ID from OTP, and stores it in internal structure. The *dwt_getlotid()* API can then be used to access it.

Mode	Mask Value	Description
DWT_READ_OTP_BAT	0x40	Reads reference (measured @ 3.0 V) raw Voltage value from OTP, and stores it in internal structure. The *dwt_geticrefvolt()* API can then be used to access it.
DWT_READ_OTP_TMP	0x80	Reads reference (measured @ 23 ˚C) raw Temperature value from OTP, and stores it in internal structure. The *dwt_geticreftemp()* API can then be used to access it.

Notes:

For more details of the OTP memory programming please refer to section 5.9 OTP and AON access APIs. **Programming OTP memory is a one-time only activity, any values programmed in error cannot be corrected. Also, please take care when programming OTP memory to only write to the designated areas – programming elsewhere may permanently damage the IC's ability to function normally.**

Example: - Following power up

```
//Initialise QM33120 device, load OTP values and
dwt_initialise(DWT_READ_OTP_PID | DWT_READ_OTP_LID | DWT_READ_OTP_BAT |
DWT_READ_OTP_TMP)
```

5.2 Configure APIs

5.2.1 dwt_configure

int dwt_configure(dwt_config_t *config);

This function is responsible for setting up the channel configuration parameters for use by both the transmitter and the receiver. The settings are specified by the *dwt_config_t* structure passed into the function, see notes below. (Note also there is a separate function *dwt_setplenfine()* for setting preamble length in blocks of 8. This is described in section 5.2.3 below). The device will be put into IDLE sate after the configuration of the PLL.

Parameters:

Type	Name	Description
dwt_config_t*	config	This is a pointer to the configuration structure, which contains the device configuration data. Individual fields are described in detail in the notes below.

```
typedef struct
{
	uint8_t		chan ;			//!< channel number {5, 9}
	dwt_tx_plen_e txPreambLength;		//!< DWT_PLEN_64..DWT_PLEN_4096
	dwt_pac_size_e rxPAC ;			//!< Acquisition Chunk Size (Relates to RX
						// preamble length)
	uint8_t txCode ;			//!< TX preamble code
	uint8_t rxCode ;			//!< RX preamble code
	dwt_sfd_type_e sfdType ;		//!< SFD type (0 - short IEEE 8 standard
				// 1 - DW 8, 2 - DW 16, 3 - 4z
```

```
                                // use non-std SFD for better performance
        dwt_uwb_bit_rate_e dataRate ;     //!< Data Rate {DWT_BR_850K or DWT_BR_6M8}
        dwt_phr_mode_e phrMode ;          //!< PHR mode:
                                // 0x0 - standard DWT_PHRMODE_STD
                                // 0x3 - extended frames DWT_PHRMODE_EXT
        dwt_phr_rate_e phrRate ;  //!< PHR rate {0x0 - standard DWT_PHRRATE_STD,
                                  //   0x1 - at datarate DWT_PHRRATE_DTA}
        uint16_t sfdTO ;          //!< SFD timeout value (in symbols)
        dwt_sts_mode_e    stsMode ;//!< STS mode (no STS, before PHR or after data)
        dwt_sts_lengths_e stsLength ;     //!< STS length
        dwt_pdoa_mode_e   pdoaMode ;      //!< PDOA mode

} dwt_config_t ;

        typedef enum
        {
            DWT_PLEN_4096 = 0x03, //! Standard preamble length 4096 symbols
            DWT_PLEN_2048 = 0x0A, //! Non-standard preamble length 2048 symbols
            DWT_PLEN_1536 = 0x06, //! Non-standard preamble length 1536 symbols
            DWT_PLEN_1024 = 0x02, //! Standard preamble length 1024 symbols
            DWT_PLEN_512 = 0x0d,  //! Non-standard preamble length 512 symbols
            DWT_PLEN_256 = 0x09,  //! Non-standard preamble length 256 symbols
            DWT_PLEN_128 = 0x05,  //! Non-standard preamble length 128 symbols
            DWT_PLEN_64 = 0x01,   //! Standard preamble length 64 symbols
            DWT_PLEN_32 = 0x04,   //! Non-standard length 32
            DWT_PLEN_72 = 0x07,   //! Non-standard length 72
        } dwt_tx_plen_e;

        typedef enum
        {
            DWT_BR_850K = 0,    //!< UWB bit rate 850 kbits/s
            DWT_BR_6M8 = 1,     //!< UWB bit rate 6.8 Mbits/s
            DWT_BR_NODATA = 2, //!< No data (SP3 packet format)
        } dwt_uwb_bit_rate_e;

        typedef enum
        {
            DWT_PRF_16M = 1, //!< UWB PRF 16 MHz
            DWT_PRF_64M = 2, //!< UWB PRF 64 MHz
            DWT_PRF_SCP = 3, //!< SCP UWB PRF ~100 MHz
        } dwt_prf_e;

        typedef enum
        {
            DWT_PAC8 = 0,  //!< PAC  8 (recommended for RX of preamble length  <=128
            DWT_PAC16 = 1, //!< PAC 16 (recommended for RX of preamble length  256
            DWT_PAC32 = 2, //!< PAC 32 (recommended for RX of preamble length  512
            DWT_PAC4 = 3,  //!< PAC  4 (recommended for RX of preamble length  < 127
        } dwt_pac_size_e;

        typedef enum
        {
            DWT_SFD_IEEE_4A = 0, //!< IEEE 8-bit ternary
            DWT_SFD_DW_8 = 1,    //!< DW 8-bit
            DWT_SFD_DW_16 = 2,   //!< DW 16-bit
            DWT_SFD_IEEE_4Z = 3, //!< IEEE 8-bit binary (4z)
            DWT_SFD_LEN8 = 8,    //!< IEEE, and DW 8-bit are length 8
            DWT_SFD_LEN16 = 16,  //!< DW 16-bit is length 16
        } dwt_sfd_type_e;

        typedef enum
        {
            DWT_STS_LEN_32  =0,
            DWT_STS_LEN_64  =1,
            DWT_STS_LEN_128 =2,
            DWT_STS_LEN_256 =3,
            DWT_STS_LEN_512 =4,
```

```
    DWT_STS_LEN_1024=5,
    DWT_STS_LEN_2048=6
}dwt_sts_lengths_e;

typedef enum
{
    DWT_PHRMODE_STD = 0x0, // standard PHR mode
    DWT_PHRMODE_EXT = 0x1, // DW proprietary extended frames PHR mode
} dwt_phr_mode_e;


typedef enum
{
    DWT_PHRRATE_STD = 0x0, // standard PHR rate
    DWT_PHRRATE_DTA = 0x1, // PHR at data rate (6M81)
} dwt_phr_rate_e;


typedef enum
{
    DWT_PDOA_M0 = 0x0, // DW PDOA mode is off
    DWT_PDOA_M1 = 0x1, // DW PDOA mode 1
    DWT_PDOA_M3 = 0x3, // DW PDOA mode 3
} dwt_pdoa_mode_e;

typedef enum
{
    DWT_STS_MODE_OFF = 0x0, // STS is off
    DWT_STS_MODE_1 = 0x1,   // STS mode 1
    DWT_STS_MODE_2 = 0x2,   // STS mode 2
    DWT_STS_MODE_ND = 0x3,  // STS with no data
    DWT_STS_MODE_SDC = 0x8, // Enable Super Deterministic Codes
    DWT_STS_CONFIG_MASK = 0xB,
    DWT_STS_CONFIG_MASK_NO_SDC = 0x3,
} dwt_sts_mode_e;
```

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or an error: DWT_ERR_PLL_LOCK (-2), DWT_ERR_RX_CAL_PGF (-3), DWT_ERR_RX_CAL_RESI (-4), DWT_ERR_RX_CAL_RESQ (-5) or DWT_ERR_RX_ADC_CAL(-6).

Notes:

The *dwt_configure()* function should be used to configure the IC's channel (TX/RX) parameters before receiver enable or before issuing a start transmission command. It can be called again to change configurations as needed, however before using *dwt_configure()* the IC should be returned to idle mode using the *dwt_forcetrxoff()* API call.

The *config* parameter points to a *dwt_config_t* structure that has various fields to select and configure different parameters within the IC. The fields of the *dwt_config_t* structure are identified are individually described below:

Fields	Description of fields within the *dwt_config_t* structure
chan	The *chan* parameter sets the UWB channel number, (defining the centre frequency and bandwidth). The supported channels are 5, and 9.
txCode and *rxCode*	The *txCode* and *rxCode* parameters select the preamble codes to use in the transmitter and the receiver – these are generally both set to the same values. For correct operation of the device, the selected preamble code should follow the rules of IEEE 802.15.4-2015 [3] UWB with respect to which codes are allowed in the particular channel and PRF configuration, this is shown in Table 2 below. The code will also define the PRF (pulse repetition frequency) used.
sfdType	The *sfdType* parameter enables the use of one of 4 possible SFD (Start Frame Delimiter) sequences. The supported values are: 0 - short IEEE 8-bit standard, 1 - DW 8-bit, 2 - DW 16-bit, 3 - 4z BPRF) DW 8 and 16 bit sequences, which Decawave has found to be more robust than that specified in the IEEE 802.15.4 standard, give improved performance.
dataRate	The *dataRate* parameter specifies the data rate to be one of 850kbps or 6800kbps, via symbolic definitions DWT_BR_850K and DWT_BR_6M8.
txPreambLength	The *txPreambLength* parameter specifies preamble length which has a range of values given by symbolic definitions: DWT_PLEN_4096, DWT_PLEN_2048, DWT_PLEN_1536, DWT_PLEN_1024, DWT_PLEN_512, DWT_PLEN_256, DWT_PLEN_128, DWT_PLEN_64, DWT_PLEN_32. Table 3 gives recommended preamble sequence lengths to use depending on the data rate.
rxPAC	The *rxPAC* parameter specifies the Preamble Acquisition Chunk size to use. Allowed values are DWT_PAC8, DWT_PAC16, DWT_PAC32 or DWT_PAC4. Table 4 below gives the recommended PAC size to use in the receiver depending on the preamble length being used in the transmitter. PAC size is specified in preamble symbols, which are approximately 1 µs each. Note: The *dwt_setsniffmode()* and *dwt_setpreambledetecttimeout()* API functions use PACs as the unit to specify the time the receiver is on looking for preamble.
phrMode	The *phrMode* parameter selects between either the standard or extended PHR mode is set, either DWT_PHRMODE_STD for standard length frames 5 to 127 octets long or non-standard DWT_PHRMODE_EXT allowing frames of length 5 to 1023 octets long.
phrRate	The *phrRate* parameter selects between either using the standard 850 kbps rate for PHR symbols or using a higher rate – same as the data rate

Fields	Description of fields within the *dwt_config_t* structure
sfdTO	The *sfdTO* parameter sets the SFD timeout value. The purpose of the SFD detection timeout is to recover from the occasional false preamble detection events that may occur. By default, this value is 4096 + 64 + 1 symbols, which is just longer the longest possible preamble and SFD sequence. This is the maximum value that is sensible. When it is known that a shorter preamble is being used then the value can be reduced appropriately. The function does not allow a value of zero. (If a 0 value is selected the default value of 4161 symbols (*DWT_SFDTOC_DEF*) will be used). The recommended value is preamble length + 1 + SFD length – PAC size.
stsMode	The *stsMode* parameter sets one of four possible STS modes: no STS, STS before PHR, STS after data, or no data mode.
stsLength	The *stsLength* parameter specifies STS length. The API supports the lengths defined by the *dwt_sts_lengths_e*.
pdoaMode	The *pdoaMode* parameter configures one of three possible PDOA modes: no PDOA, mode 1 (PDOA is calculated between Ipatov POA and STS POA), mode 3 (PDOA is calculated between two STS blocks)

The *dwt_configure()* function does not error check the input parameters unless the DWT_API_ERROR_CHECK code switch is defined. If this is defined, it will assert in case an error is detected. It is up to the developer to ensure that the assert macro is correctly enabled in order to trap any error conditions that arise. If DWT_API_ERROR_CHECK switch is not defined, error checks are not performed.

NOTE: SFD timeout cannot be set to 0; if a zero value is passed into the function the default value will be programmed. To minimise power consumption in the receiver, the SFD timeout of the receiving device, *sfdTO* parameter, should be set according to the TX preamble length of the transmitting device. As an example, if the transmitting device is using 128 preamble length, an SFD length of 8 and a PAC size of 8, the corresponding receiver should have *sfdTO* parameter set to 129 (128 + 1 + 8 - 8).

Table 2: supported UWB channels and recommended preamble codes

Channel number	Preamble Codes (16 MHz PRF)	Preamble Codes (64 MHz PRF)	Preamble Codes (SCP)
5, 9	3, 4	9, 10, 11, 12	25, 26, 27, 28, 29

In addition to the preamble codes in shown in Table 2 above, for 64 MHz PRF there are eight additional preamble codes, (13 to 16, and 21 to 24), available for use on all channels. These should only be selected as part of implementing dynamic preamble selection (DPS). Please refer to the IEEE 802.15.4 standard [3] for more details of the dynamic preamble selection technique.

The preamble sequence used on a particular channel is the same at all data rates, however its length, (i.e. the number of symbol times for which it is repeated), has a significant effect on the operational range. Table 3 gives some recommended preamble sequence lengths to use depending on the data rate. In general, a longer preamble gives improved range performance and better first path time of arrival information while a shorter preamble gives a shorter air time and saves power. When operating a low data rate for long range, then a long preamble is needed to achieve that range. At higher data rates the operating range is naturally shorter so there is no point in sending an overly long preamble as it wastes time and power for no added range advantage.

Table 3: Recommended preamble lengths

Data Rate	Recommended preamble sequence length
6.8Mbps	32, 64, 128 or 256
850kbps	256, or higher

The preamble sequence is detected by cross-correlating in chunks which are a number of preamble symbols long. The size of chunk used is selected by the PAC size configuration, which should be selected depending on the expected preamble size. A larger PAC size gives better performance when the preamble is long enough to allow it. But if the PAC size is too large for the preamble length then receiver performance will reduce or fail to work at the extremes – (e.g. a PAC of 32 will never receive frames with just 32 preamble symbols). Table 4 below gives the recommended PAC size configuration to use in the receiver depending on the preamble length being used in the transmitter.

Table 4: Recommended PAC size

Expected preamble length of frames being received	Data Rate	Recommended PAC size
32	6.81 Mb/s	4
≥ 64	6.81 Mb/s	8
≥ 128	850 kb/s	16

Notes of STS modes:

This function configures the STS mode (defined in Table 5). When STS modes 1 or 2 are configured, the IC will transmit a STS after the SFD (in Mode 1) or after data (in Mode 2). The STS PRF will match the Ipatov. The API supports a number of STS length configurations as specified by dwt_sts_lengths_e. Prior to transmission of STS, a 128-bit IV (initial value), via counter register, generates a 128-bit "NONCE" value which with the 128-bit key value feeds into the CPRNG block (pseudo-random number generator block) to generate codes for each pair of preamble symbols, see Figure 6. The supported PRF is 64 MHz.

A user is expected to either set a unique key and IV e.g. per ranging/communication session is started between a number of devices, or keep the same key for some time but update the IV (e.g. the low x

bits) to keep session secure. Optionally Super Deterministic Codes can be used, this means that the programming of STS key and IV is not necessary.

Table 5: *stsMode* parameter to dwt_configure() function

Mode	Value	Description
DWT_STS_MODE_OFF	0x0	The IC is not configured to use STS.
DWT_STS_MODE_1	0x1	The IC is configured to use STS mode 1, i.e. the STS comes before data
DWT_STS_MODE_2	0x2	The IC is configured to use STS mode 2, i.e. the STS comes after data
DWT_STS_MODE_SDC	0x8	The Super Deterministic Codes are used in the STS
DWT_STS_MODE_ND	0x3	Configured to use no data STS mode

The QM33120 supports four STS modes of operation, the packet format of each of these is shown in Figure 5. When the STS modes are used then the IC will transmit a STS preamble, which is a pseudo-random sequence of pulses. This pseudo-random sequence is based on AES128 in counter mode.

The host controller needs to provide the IC with a seed and Initial Value, which can be (re)programmed individually (for this *dwt_configurestskey()* and *dwt_configurestsiv()* APIs are used).

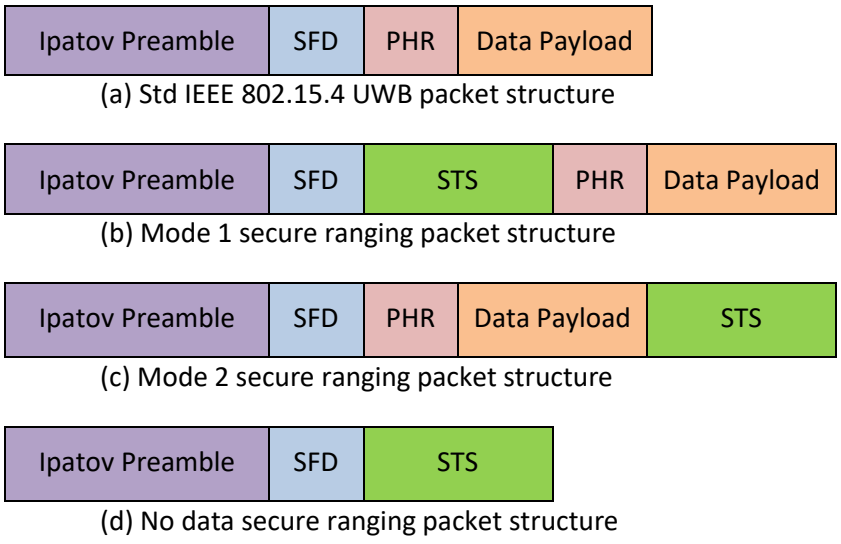


Figure 5: Standard compliant versus secure ranging packet

Assuming the seed is not changed, and the counter is not reset between packets (i.e. by loading a new Initial Value), then a different set of STS symbols will be generated as a result of the natural advancement of the counter. These will be applied to generate the STS that is either used for the transmitted frames, or used in the receiver to correlate with the symbols of the STS in the received frame. Over time the counter will generate 2^{31} - separate 128-bit values before it repeats.

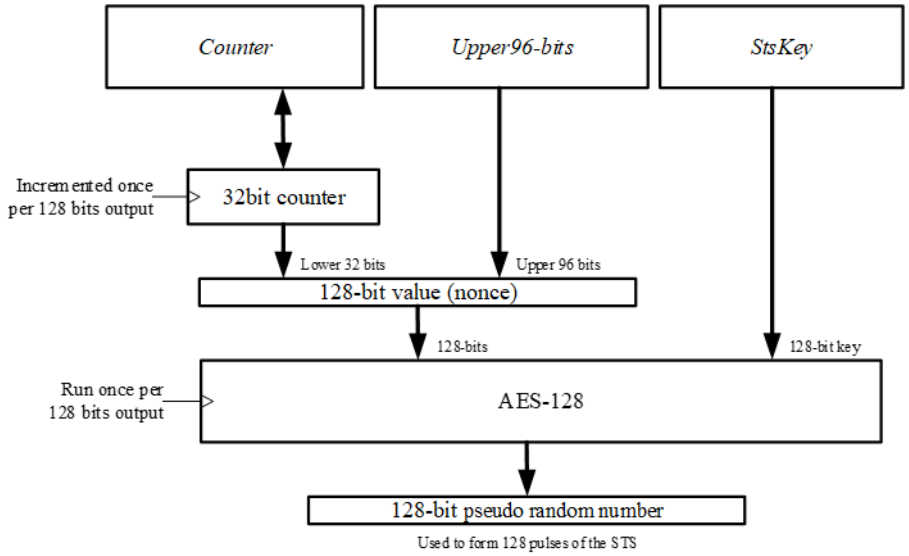


Figure 6: AES in counter mode based CPRNG

See also: *dwt_configuretxrf()* for setting certain TX parameters and

dwt_setsniffmode() for setting certain RX (preamble hunt) operating mode.

5.2.2 dwt_restoreconfig

void dwt_restoreconfig(void);

This function needs to be called after device is woken up from DEEPSLEEP/SLEEP state, to restore the configuration and calibration data which has not been automatically restored from AON.

Parameters:

none

Return Parameters:

none

Notes:

After a device has woken from a sleep state, this function must be called in order to restore configurations such as LDO configuration, bias tuning and gear table configuration.

5.2.3 dwt_setplenfine

void dwt_setplenfine (uint8_t preambleLength);

This API function is used to configure frame preamble length, the frame preamble length can be configured in steps of 8, from 16 to 2048 symbols. If a non-zero value is configured, then the TXPSR_PE setting is ignored.

Parameters:

Type	Name	Description
uint8_t	preambleLength	Units are 8-symbol blocks, value of 1 is a minimum i.e. 16 symbols, value of 255 is max i.e. 2048 symbols. Value of 0 disables this setting.

Return Parameters:

none

Notes:

5.2.4 dwt_configuretxrf

void dwt_configuretxrf(dwt_txconfig_t *config);

The *dwt_configuretxrf()* function is responsible for setting up the transmit RF configuration parameters. The values present in the input structure are pulse generator delay (PGdly), transmit output power (power) and pulse generator count (PGcount). The pulse generator delay value sets the width of transmitted pulses effectively setting the output bandwidth. Transmit output power setting assigns a power level to the outputted transmissions. The pulse generator count is a measurement of the pulse generator delay lines – it allows for the calibration of pulse generator delay.

The pulse generator uses delay lines to generate the pulses for the UWB signal. The length of these delay lines determines the length of the pulse (and thus, inversely, the bandwidth of the spectrum). These delay lines are analog (in the sense that they vary across device and temperature). The PGDelay sets the length of the lines to explicitly set the delay.

Parameters:

Type	Name	Description
dwt_txconfig_t*	config	This is a pointer to the TX parameters configuration structure, which contains the device configuration data. Individual fields are described in detail below.

```
typedef struct
{
            uint8_t      PGdly;       //Pulse generator delay value
            uint32_t     power;       //the TX power – 4 bytes
            uint16_t     PGcount;     //Pulse generator count value
} dwt_txconfig_t ;
```

Return Parameters:

none

Notes:

This function can be called any time and it will configure the IC's TX spectrum parameters. The *config* parameter points to a *dwt_txconfig_t* structure (shown above) with fields to configure the pulse generator delay (*PGdly*), TX power (*power*), and pulse generator count (*PGCount*). Recommended

values for *PGdly* are given in Table 6 below. (this may be called to adjust power and bandwidth as temperature fluctuates)

Table 6: PGdly recommended values

TX Channel	recommended PGdly value
5	0x34
9	0x34

Table 7: TX power recommended values

TX Channel	recommended TX power value
5	0xfdfdfdfd
9	0xfefefefe

Table 8: dwt_txconfig_t parameter: *power* function

`power` byte index	Byte name	Function
0	DATA_PWR	Power level applied during the transmission of the data portion of the frame
1	PHR_PWR	Power level applied during the transmission of the PHR portion of the frame
2	SHR_PWR	Power level applied during the transmission of the preamble and SFD portions of the frame
3	STS_PWR	Power level applied during the transmission of the STS preamble portion of the frame

Table 7 above gives the recommended TX power settings. The use of the individual octet values of the 4-byte TX power array are specified in Table 8, for further details of the power values please refer to the User Manual [2].

NB: The values in Table 7 have been chosen to suit Decawave's evaluation boards. For other hardware designs the values here may need to be changed as part of the transmit power calibration activity, and there is a location in OTP memory where the calibrated values can be stored and then read and programmed from the application code. Please consult with Decawave's applications support team for details of transmit power calibration procedures and considerations.

5.2.5 dwt_adjust_tx_power

int dwt_adjust_tx_power(uint16_t boost, uint32_t ref_tx_power, uint8_t channel, uint32_t* adj_tx_power, uint16_t* applied_boost);

This function calculates a new TX power setting (*adj_tx_power*) relative to a reference TX power setting (*ref_tx_power*) and a *boost* to apply on top on this reference setting. The function finds the best possible adjustment that can be made to give the closest solution (power value) corresponding to the required boost. The boost which is applied is provided through *applied_boost* parameter. The setting calculation depends on the operating channel (*channel*).

This function is typically used to give a benefit of the additional TX power than can be transmitted for shorter frames.

Parameters:

Type	Name	Description
uint16_t	boost	The amount of boost in 0.1dB step to be applied on top of the reference TX power setting.
uint32_t	ref_tx_power	The reference TX power setting.
uint8_t	channel	The channel for which the calculation must be performed.
uint32_t	adj_tx_power	The adjusted TX power setting.
uint16_t	applied_boost	The amount of boost in 0.1dB step that was effectively applied by the API on top of the reference TX power setting.

Return Parameters:

Type	Description
int	DWT_SUCCESS = 0: if an adjusted TX power setting could be calculated. DWT_ERROR = -1: if the API could not calculate a valid adjusted TX power setting.

Notes:

Example:
Theoretical boost between 1000us and 250us is 6dB (Boost = 10*log10(ref_duration/fr_duration))

Input:
reference_tx_power: 0x4a4a4a4a // Setting for 1000s frame to comply with regulation
boost = 60 // 6dB in 0.1dB steps
channel = 9

Output:
adj_tx_power: 0x9a9a9a9a// Setting for 250s frame to comply with regulation
applied_boost: 58 //Actual boost that was applied is 5.8dB

The setting 0x9a9a9a9a should be use to transmit a 250us frame in channel 9 and comply with regulations.

5.2.6 dwt_setrxantennadelay

void dwt_setrxantennadelay(uint16_t antennaDly);

This function sets the RX antenna delay. The *antennaDelay* value passed is programmed into the RX antenna delay register. This needs to be set so that the RX timestamp is correctly adjusted to account for the time delay between the antenna and the internal digital RX timestamp event. This is determined by a calibration activity. Please consult with Decawave applications support team for details of antenna delay calibration procedures and considerations.

Parameters:

Type	Name	Description
uint16_t	antennaDly	The delay value is in DWT_TIME_UNITS (15.65 picoseconds ticks)

Return Parameters:

none

Notes:

This function is used to program the RX antenna delay.

5.2.7 dwt_getrxantennadelay

uint16_t dwt_getrxantennadelay(void);

This function returns the RX antenna delay, the value programmed into the RX antenna delay register.

Parameters:

none

Return Parameters:

Type	Description
uint16_t	RX antenna delay value as currently set in RX antenna delay register.

Notes:

This function is used to read the RX antenna delay programmed in the device.

5.2.8 dwt_settxantennadelay

void dwt_gettxantennadelay(uint16_t antennaDly);

This function returns the TX antenna delay, the value programmed into the TX antenna delay register.

Parameters:

Type	Name	Description
uint16_t	antennaDly	The delay value is in DWT_TIME_UNITS (15.65 picoseconds ticks)

Return Parameters:

none

Notes:

This function is used to program the TX antenna delay.

5.2.9 dwt_gettxantennadelay

uint16_t dwt_gettxantennadelay(void);

This function sets the TX antenna delay. The *antennaDelay* value passed is programmed into the TX antenna delay register. This needs to be set so that the TX timestamp is correctly adjusted to account for the time delay between internal digital TX timestamp event and the signal actually leaving the antenna. This is determined by a calibration activity. Please consult with Decawave applications support team for details of antenna delay calibration procedures and considerations.

Parameters:

none

Return Parameters:

Type	Description
uint16_t	TX antenna delay value as currently set in TX antenna delay register.

Notes:

This function is used to read the TX antenna delay programmed in the device .

5.2.10 dwt_setpdoaoffset

void dwt_setpdoaoffset(uint16_t offset);

This function sets PDoA offset, the PDoA result (*dwt_readpdoa()*) will have this offset applied.

Parameters:

Type	Name	Description
uint16_t	offset	The delay value is in DWT_TIME_UNITS (15.65 picoseconds ticks)

Return Parameters:

none

Notes:

5.2.11 dwt_readpdoaoffset

uint16_t dwt_readpdoaoffset(void);

This function reads PDoA offset as is currently set in the device. The values is in DWT_TIME_UNITS (15.65 picoseconds ticks).

Parameters:

none

Return Parameters:

Type	Description
uint16_t	PDoA offset value.

Notes:

5.2.12 dwt_configurestskey

void dwt_configurestskey (dwt_sts_cp_key_t* pStsKey);

This function can be used to configure the STS 128-bit AES key value. The default value is [31:0] C9A375FA, [63:32] 8DF43A20, [95:64] B5E5A4ED, [127:96] 0738123B

Parameters:

Type	Name	Description
dwt_sts_cp_key_t*	pStsKey	Pointer to the structure of dwt_sts_cp_iv_t type containing the 128-bit AES key for STS, (i.e. 16 bytes, LSB comes first – to write default value in: pStsKey [0] would be 0xFA, pStsKey [1] would be 0x75, etc.)

```
typedef struct
{
	uint32_t	key0;
	uint32_t	key1;
	uint32_t	key2;
	uint32_t	key3;
} dwt_ata_cp_key_t;
```

Return Parameters:

none

Notes:

5.2.13 dwt_configurestsiv

void dwt_configurestsiv (dwt_sts_cp_iv_t* pStsIv);

This function is used to configure the STS 128-bit initial value. The default value is 1, i.e. the IC reset value is 1. A value of all 0 is invalid and the IC will automatically detect this condition and set it to 1.

Parameters:

Type	Name	Description
dwt_sts_cp_iv_t*	pStsIv	Pointer to the structure of dwt_sts_cp_iv_t type containing the 128-bit initial value for STS. (i.e. 16 bytes, LSB comes first)

```
typedef struct
{
	uint32_t	iv0;
	uint32_t	iv1;
	uint32_t	iv2;
	uint32_t	iv3;
} dwt_sts_cp_iv_t;
```

Return Parameters:

none

Notes:

In order to make the counter more random its initial state (IV) is seeded from time to time (updating frequency should be less than its period). For security reasons the key should also be updated at the same time (with a different value).

The values it generates act as a nonce which together with the supplied key are used to generate the STS.

5.2.14 dwt_configurestsloadiv

void dwt_configurestsloadiv (void);

This function is used to load the IV and KEY initial value into the STS AES block.

Parameters:

none

Return Parameters:

none

Notes:

If this function is called when reloading the counter with a previously set IV, without changing the key, then the generated STS sequence will potentially be repeating in a predictable way. While this may be useful in some testing scenarios, it could be a security risk if it were done in normal operation, where any reloading of the counter should be accompanied by a new key load also.

5.2.15 dwt_configurestsmode

void dwt_configurestsmode(uint8_t stsMode);

This function configures STS mode (e.g. DWT_STS_MODE_OFF, DWT_STS_MODE_1, etc.) The dwt_configure() should be called prior to this to configure other parameters.

Parameters:

Type	Name	Description
uint8_t	stsMode	The STS mode that the device should be configured for.

Return Parameters:

none

Notes:

For more information on the input parameter, please see Table 5: *stsMode* parameter to dwt_configure() function.

5.2.16 dwt_configuresfdtype

void dwt_configuresfdtype(uint8_t sfdType);

This function configures SFD type only: e.g. IEEE 4a - 8, DW-8, DW-16, or IEEE 4z -8 (binary). The dwt_configure() should be called prior to this to configure other parameters.

Parameters:

Type	Name	Description
uint8_t	sfdType	This value is used to assign the SFD type used when configuring the device. Please see Table 9 for more information.

Table 9: *sfdType* parameter to dwt_configuresfdtype() function

Mode	Value	Description
DWT_SFD_IEEE_4A	0x0	IEEE 8-bit ternary

Mode	Value	Description
DWT_SFD_DW_8	0x1	Decawave / Qorvo 8-bit SFD
DWT_SFD_DW_16	0x2	Decawave / Qorvo 16-bit SFD
DWT_SFD_IEEE_4Z	0x3	IEEE 8-bit binary (4z)

Return Parameters:

none

5.2.17 dwt_setleds

void dwt_setleds(uint8_t mode);

This is used to set up Tx/Rx GPIOs which are then used to control (for example) LEDs. This is not completely IC dependent; it requires that LEDs are connected to the GPIO lines.

Parameters:

Type	Name	Description
uint8_t	mode	This is a bit field value interpreted as follows: - bit 0: set to 1 to enable LEDs, 0 to disable them. - bit 1: set to 1 to make LEDs blink once on init. This is only valid if bit 0 is set (enable LEDs). - Bits 2 to 7: Reserved.

Return Parameters:

none

Notes:

For more information on GPIO control and configuration please consult the User Manual [2] and Data Sheet [1].

5.2.18 dwt_setlnapamode

void dwt_setlnapamode(int lna_pa)

This is used to enable GPIO for external LNA or PA functionality – HW dependent, consult the User Manual [2]. This can also be used for debug as enabling TX and RX GPIOs is can help monitoring the IC's activity.

Parameters:

Type	Name	Description
int	lna_pa	This parameter is treated as a bit field.

		If bit 0 is set (DWT_LNA_ENABLE), it will enable LNA functionality. If bit 1 is set (DWT_PA_ENABLE), it will enable PA functionality. If bit 2 is set (DWT_TXRX_EN), it will enable RX/TX sampling on GPIOs 0 & 1. To disable LNA/PA functionality, set bits 0 and 1 to 0.

Return Parameters:

none

Notes:

Enabling PA functionality requires that fine grain TX sequencing is deactivated. This can be done using the *dwt_setfinegraintxseq()* API function.

For more information on GPIO control and configuration please consult the User Manual [2] and Data Sheet [1].

5.2.19 dwt_generatecrc8

uint8_t dwt_generatecrc8 (const uint8_t* byteArray, int len, uint8_t crcRemainderInit);

This function is used to generate 8-bit CRC to send as part of SPI write transaction when the IC is configured for SPI CRC check mode.

Parameters:

Type	Name	Description
uint8_t*	byteArray	This array supplies the bytes for which to calculate the CRC.
int	len	Length of the *byteArray* parameter in bytes
uint8_t	crcRemainderInit	The remainder is the CRC, also it is initially set to the initialisation value for CRC calculation.

Return Parameters:

Type	Description
uint8_t	The calculated 8-bit CRC function.

Notes:

It is possible to calculate CRC for two separate blocks (that will be sent as one contiguous SPI transaction) by calling the *dwt_generatecrc8()* function twice, once for each; setting the first crcRemainderInit parameter to the configured CRC seed (zero by default), and the second crcRemainderInit parameter to the CRC value returned from the first call to the *dwt_generatecrc8()* function.

5.2.20 dwt_enablespicrccheck

void dwt_enablespicrccheck (dwt_spi_crc_mode_e crc_mode, dwt_spierrcb_t spireaderr_cb);

This function can be used to enable or disable the SPI CRC check in the IC.

Parameters:

Type	Name	Description
dwt_spi_crc_mode_e	crc_mode	If set to DWT_SPI_CRC_MODE_WR then SPI CRC checking will be performed in QM33120 on each SPI write. The last byte of the SPI write transaction needs to be the 8-bit CRC, if it does not match the one calculated by QM33120 SPI, a CRC ERROR event will be set in the status register (SPICRC bit of SYS_STATUS_LO register).
dwt_spierrcb_t	spireaderr_cb	This parameter needs to contain the callback function pointer which will be called when a SPI read error is detected (when the QM33120 generated CRC does not match the one calculated by dwt_generatecrc8 following the SPI read transaction).

Return Parameters:

none

Notes:

The *crc_mode* parameter is used to select the CRC mode to be used. The following table shows all the available options in the *dwt_spi_crc_mode_e* enum that is used to contain the argument:

Table 10: Valid crc_mode options

Parameter	Value	Description
DWT_SPI_CRC_MODE_NO	0	No CRC
DWT_SPI_CRC_MODE_WR	1	Enable SPI CRC check on write functions.
DWT_SPI_CRC_MODE_WRRD	2	Enable SPI CRC on both write and read operations.

During normal SPI mode the SPICRC bit of the SYS_STATUS_LO register will be asserted. When using SPI CRC check mode this will only be asserted if there is a mismatch between the CRC byte calculated by the IC and the one sent in the SPI write transaction.

5.2.21 dwt_configmrxlut

void dwt_configmrxlut(int channel);

This function sets the default values of the lookup tables depending on the channel selected.

Parameters:

Type	Name	Description
int	channel	This is the channel: 5 or 9.

Return Parameters:

none

Notes:

The lookup table contains the desired front-end settings for the chosen level of noise dithering. Different settings will apply for channel 5 or channel 9. The seven registers starting at DGC_DGC_LUT_0_CFG and ending at DGC_DGC_LUT_6_CFG are populated as follows depending on the channel given:

Table 11: Channel 5 Lookup Table Configuration for DW3xxx devices

LUT #	DW3000 – LUT register values	QM33120 – LUT register values
CH5_DGC_LUT_0	0x1C0FD	0x3803E
CH5_DGC_LUT_1	0x1C43E	0x3876E
CH5_DGC_LUT_2	0x1C6BE	0x397FE
CH5_DGC_LUT_3	0x1C77E	0x38E6E
CH5_DGC_LUT_4	0x1CF36	0x39C7E
CH5_DGC_LUT_5	0x1CFB5	0x39DFE
CH5_DGC_LUT_6	0x1CFF5	0x39FF6

Table 12: Channel 9 Lookup Table Configuration for DW3xxx devices

LUT #	DW3000 – LUT register values	QM33120 – LUT register values
CH9_DGC_LUT_0	0x2A8FE	0x5407E
CH9_DGC_LUT_1	0x2AC36	0x547BE
CH9_DGC_LUT_2	0x2A5FE	0x54D36
CH9_DGC_LUT_3	0x2AF3E	0x55E36
CH9_DGC_LUT_4	0x2AF7D	0x55F36
CH9_DGC_LUT_5	0x2AFF5	0x55DF6

LUT #	DW3000 – LUT register values	QM33120 – LUT register values
CH9_DGC_LUT_6	0x2AFB5	0x55FFE

5.2.22 dwt_enablegpioclocks

void dwt_enablegpioclocks(void);

This function enables the GPIO clocks on the IC. GPIO clocks are required to ensure correct GPIO operation.

Parameters:

none

Return Parameters:

none

Notes:

An example of how this API can be used is included the API package [5]. Also, please see *Example 13a: Use of DW3XXX GPIO lines.*

5.2.23 dwt_setgpiomode

void dwt_setgpiomode(uint32_t gpio_mask, uint32_t gpio_modes);

This function is used to configure the GPIO mode of one or more GPIO pins. The gpio_mask parameter allows to only configure the mode of specific pins, without altering the mode of other pins.

Parameters:

Type	Name	Description
uint32_t	gpio_mask	The mask of the GPIOs to change the mode of. Typically built from dwt_gpio_mask_e values.
uint32_t	gpio_modes	The GPIO modes to set. Typically built from dwt_gpio_pin_e values.

Return Parameters:

none

Notes:

An example of how this API can be used is included the API package [5]. Also, please see *Example 13a: Use of DW3XXX GPIO lines.*

5.2.24 dwt_setgpiodir

void dwt_setgpiodir(uint16_t in_out);

This is used to configure the GPIOs as inputs or outputs, default is input == 1.

Parameters:

Type	Name	Description
uint16_t	in_out	if corresponding GPIO bit is set to 1 then it is input, otherwise it is output GPIO 0 = bit 0, GPIO 1 = bit 1 etc...

Return Parameters:

none

Notes:

An example of how this API can be used is included the API package [5]. Also, please see *Example 13a: Use of DW3XXX GPIO lines.*

5.2.25 dwt_setgpiovalue

void dwt_setgpiovalue(uint16_t value);

This is used to set output value on GPIOs that have been configured for output via *dwt_setgpiodir()* API

Parameters:

Type	Name	Description
uint16_t	value	This parameter is used as a mask for setting the signal level for any GPIO pins that have been previously set as outputs.

Return Parameters:

none

Notes:

5.2.26 dwt_pgf_cal

int dwt_pgf_cal(int ldoen);

This function sets up the PGF calibration and then calls *dwt_run_pgfcal()* to run the calibration. This is needed prior to reception of any frames/packets.

Parameters:

Type	Name	Description
int	ldoen	This signifies whether the PGF LDOs should be enabled when running the function. A '1' signifies that PGF LDOs will be turned on and a '0' signifies that they will not be turned on.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or an error: DWT_ERR_RX_CAL_PGF (-3), DWT_ERR_RX_CAL_RESI (-4) or DWT_ERR_RX_CAL_RESQ (-5). Depending what is returned from *dwt_run_pgfcal().*

Notes:

This function is run as part of the final step of *dwt_configure()*.

5.2.27 dwt_run_pgfcal

int dwt_run_pgfcal(void);

This function runs the PGF calibration. It is usually called by *dwt_pgf_cal()*. This is needed prior to reception of any frames/packets.

Parameters:

none

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or an error: DWT_ERR_RX_CAL_PGF (-3), DWT_ERR_RX_CAL_RESI (-4) or DWT_ERR_RX_CAL_RESQ (-5).

Notes:

5.2.28 dwt_pll_cal

int dwt_pll_cal(void);

This function will be used to recalibrate and relock the PLL. If the cal/lock is successful DWT_SUCCESS will be returned otherwise DWT_ERROR will be returned.

Parameters:

none

Return Parameters:

Type	Description
int	An int value will be returned to indicate is the recalibration and relocking of the PLL was successful or not. DWT_SUCCESS indicates a success DWT_ERROR indicates a failure.

5.2.29 dwt_setdwstate

void dwt_setdwstate(int state);

This function can place DW3xxx into IDLE/IDLE_PLL or IDLE_RC mode when it is not actively in TX or RX.

Parameters:

Type	Name	Description
int	state	DWT_DW_IDLE (1) to put QM33120 into IDLE/IDLE_PLL state. DWT_DW_INIT (0) to put QM33120 into INIT_RC state. DWT_DE_IDLE_RC (2) to put QM33120 into IDLE_RC state

Return Parameters:

none

Notes:

5.2.30 dwt_enable_disable_eq

void dwt_enable_disable_eq(uint8_t en);

This function enables or disables the equaliser block within in the CIA block within the QM331XX. The equaliser should be used when receiving from devices which transmit using a Symmetric Root Raised Cosine pulse shape. The equaliser will adjust the CIR to give improved receive timestamp results. Normally, this is left disabled (the default value), which gives the best receive timestamp

performance when interworking with devices (like this IC) that use the IEEE 802.15.4z recommended minimum precursor pulse shape.

Parameters:

Type	Name	Description
uint8_t	en	DWT_EQ_ENABLED (1): Enable the equalizer block. DWT_EQ_DISABLED (0): Disable the equalizer block.

Return Parameters:

none

Notes:

5.2.31 dwt_configure_rf_port

void dwt_configure_rf_port(dwt_rf_port_selection_e rfPort, dwt_rf_port_ctrl_e enable);

This function is used to control which RF port is currently enabled.

Parameters:

Type	Name	Description
dwt_rf_port_selection_e	rfPort	This argument specifies the RF port to use: • '0' – Antenna 1 • '1' – Antenna 2
dwt_rf_port_ctrl_e	enable	This argument specifies whether the user wishes to have manual control over the RF port: • '0' – No manual control. • '1' – Enable manual control.

Return Parameters:

none

Notes:

5.2.32 dwt_configure_and_set_antenna_selection_gpio

void dwt_configure_and_set_antenna_selection_gpio(uint8_t antenna_config);

This function is used for specific customer hardware/modules where antenna selection switch is connected to GPIO6 and GPIO7. This function configures the GPIOs to give particular antenna selecton.

Parameters:

Type	Name	Description
uint8_t	antenna_config	Configure GPIO 6 and or 7 to use for antenna selection with expected value. The parameter is a bit mask: Bit 0: Use GPIO 6 Bit 1: Value to apply (0/1) Bit 2: Use GPIO 7 Bit 3: Value to apply (0/1)

Return Parameters:

none

Notes:

5.2.33 dwt_wifi_coex_set

void dwt_wifi_coex_set(dwt_wifi_coex_e enable, int coex_io_swap);

This function can set GPIO output to high (1) or low (0) which can then be used to signal e.g. WiFi chip to turn off or on. This can be used in devices with multiple radios to minimise co-existence interference.

Parameters:

Type	Name	Description
dwt_wifi_coex_e	enable	This argument will specify if the user wishes to enable or disable WiFi co-existence functionality on GPIO5 or GPIO 4 (depending on how the coex_io_swap argument is set). • "DWT_EN_WIFI_COEX" - Configure GPIO for WiFi co-ex - GPIO high. • "DWT_DIS_WIFI_COEX" - Configure GPIO for WiFi co-ex - GPIO low.
int	coex_io_swap	This argument specifies which GPIO to use for WiFi co-ex: • '0' – GPIO5. • '1' – GPIO4.

```
typedef enum
{
    DWT_EN_WIFI_COEX=0, /* Configure GPIO for WiFi co-ex - GPIO high*/
    DWT_DIS_WIFI_COEX   /* Configure GPIO for WiFi co-ex - GPIO low */
}dwt_wifi_coex_e;
```

Return Parameters:

none

Notes:

5.2.34 dwt_set_fixedsts

void dwt_set_fixedsts(uint8_t set)

This API enables "Fixed STS" function. The fixed STS function means that the same STS will be sent in each packet. And also in the receiver, each time the receiver is enabled, the STS will be reset. Thus, transmitter and the receiver will be in sync.

Parameters:

Type	Name	Description
uint8_t	set	Set to 1 to set FIXED STS mode and 0 to disable (normal mode, STS counter will increment)

Return Parameters:

none

5.2.35 dwt_set_alternative_pulse_shape

void dwt_set_alternative_pulse_shape(uint8_t set_alternate)

This API sets the Alternative Pulse Shape according to Japanese Association of Radio Industries and Businesses Standard-T91. [7] This is only supported in QM33120, which allows the use of 2 different TX pulse shapes. The new pulse shape was specifically designed to meet the Japanese ARIB on channel 9.

Parameters:

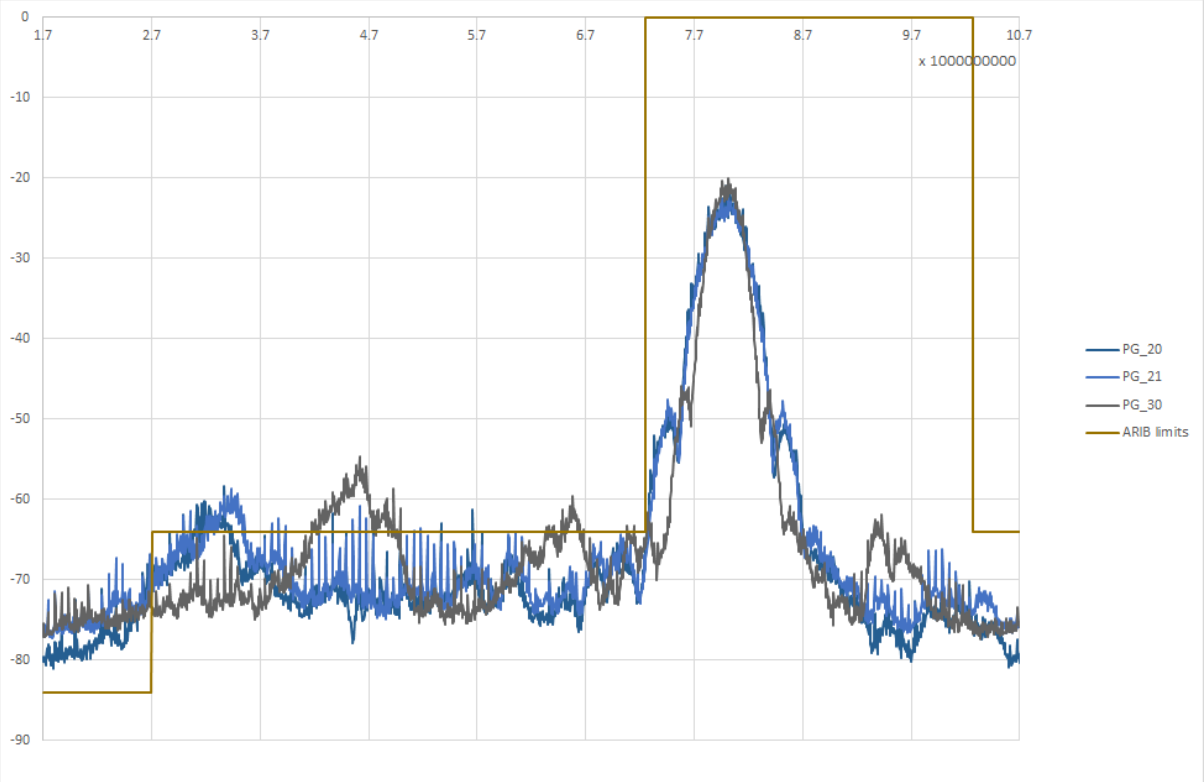
Type	Name	Description
uint8_t	set_alternate	Set to 1 to enable the alternate pulse shape and 0 to restore default shape.

Return Parameters:

none

Notes:

Below is the image for the usual pulse shape grey colour and alternate pulse shape according to ARIB STD-T91 in blue colour, fitting inside the limits of standard.



5.2.36 dwt_config_ostr_mode

void dwt_config_ostr_mode (uint8_t enable, uint16_t wait_time)

This function is used to configure the device for OSTR mode (One Shot Timebase Reset mode), this will prime the device to reset the internal system time counter on SYNC pulse / SYNC pin toggle. For more information on this operation please consult the device User Manual [2].

Parameters:

Type	Name	Description
uint8_t	set_alternate	Set to 1 to enable OSTR mode and 0 to disable.
uint16_t	wait_time	When a counter running on the 38.4 MHz external clock and initiated on the rising edge of the SYNC signal equals the WAIT programmed value, the DW3xxx timebase counter will be reset.

Return Parameters:

none

Notes:

At the time the SYNC signal is asserted, the clock PLL dividers generating the 125 MHz system clock are reset, to ensure that a deterministic phase relationship exists between the system clock and the asynchronous 38.4 MHz external clock. For this reason, the WAIT value programmed will dictate the phase relationship and should be chosen to give the desired phase relationship, as given by WAIT modulo 4. A WAIT value of 33 decimal is recommended, but if a different value is chosen it should be chosen so that WAIT modulo 4 is equal to 1, i.e. 29, 37, and so on.

5.3 TX/RX and Timestamp APIs

5.3.1 dwt_writetxdata

int dwt_writetxdata(uint16_t txBufferLength, uint8_t *txBuffer, uint16_t txBufferOffset);

This function is used to write the TX message data into the device's internal TX buffer (max size is 1024 bytes).

Parameters:

Type	Name	Description
uint16_t	txBufferLength	This is the total length to write into the TX buffer, it can include the space for the two-byte CRC but it does not have to. The device will automatically put a two-byte CRC in the last two bytes of the TX data it is sending. The length of data to transmit is specified by the *txFrameLength* parameter in *dwt_writetxfctrl()*
uint8_t*	txBuffer	Pointer to the buffer containing the data to send.
uint16_t	txBufferOffset	This specifies an offset in the device's internal TX buffer at which to start writing data.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

This function writes the specified size of *txBuffLength* bytes from the memory, pointed to by the *txBuffBytes* parameter, into the device's internal transmit data buffer, starting at the specified offset (*txBufferOffset*). During the transmission, the device will automatically add the two CRC bytes to complete the TX frame to its *txFrameLength* as specified by the *txFrameLength* parameter in *dwt_writetxfctrl ().*

NOTE: standard PHR mode allows frames of up to 127 bytes. For longer lengths non-standard PHR mode DWT_PHRMODE_EXT needs to be set in the *phrMode* configuration passed into the *dwt_configure()* function.

The *dwt_writetxfctrl()* function checks that the sum of *txBufferLength* and *txBufferOffset* is less than the max (1024 bytes) internal TX buffer length to avoid messing with IC's other registers and memory. If such an error occurs, the write is not performed and the function returns DWT_ERROR. Otherwise, the functions returns DWT_SUCCESS.

If DWT_API_ERROR_CHECK code switch is defined, the function will perform additional checks on input parameters. If an error is detected, the function will assert. It is up to the developer to ensure that the assert macro is correctly enabled in order to trap any error conditions that arise.

Example code:

Typical usage is to write the data, configure the frame control with starting buffer offset and frame length and then enable transmission as follows:

```
dwt_writetxdata(fLength,dataBuffPtr,0); // write the frame data at offset 0
dwt_writetxfctrl(fLength+2,0,0);          // set the frame control register
dwt_starttx(DWT_START_TX_IMMEDIATE);    // send the frame
```

5.3.2 dwt_writetxfctrl

void dwt_writetxfctrl(uint16_t txFrameLength, uint16_t txBufferOffset, uint8_t ranging);

This function is used to configure the TX frame control register.

Parameters:

Type	Name	Description
uint16_t	txFrameLength	This is the total frame length, including the two-byte CRC.
uint16_t	txBufferOffset	This specifies an offset in the device's internal TX Buffer at which to start writing data.
uint8_t	ranging	This specifies whether the TX frame is a ranging frame or not, i.e. whether the ranging bit is set in the PHY header (PHR) of the frame. A value of 1 sets the ranging bit the PHR of the outgoing frame, while a value of 0 will clear it.

Return Parameters:

none

Notes:

This function configures the TX frame control register parameters, namely the length of the frame and the offset in the IC's transmit data buffer where the data starts. It also controls whether the ranging bit is set in the frame's PHR.

The ranging bit identifies a frame as a ranging frame. This has no operational effect on the IC, but in some receiver implementations, it might be used to enable hardware or software associated with time stamping the frame. In the IC's receiver, the time stamping does not depend or use the ranging bit in the received PHR. The status of the ranging bit in received frames is reported by the cbRxOk function (if enabled) in the *rx_flags* element of its dwt_cb_data_t structure parameter. See the *dwt_isr()* and the *dwt_setcallbacks()* functions.

The *dwt_writetxfctrl()* function does not error check the *txFrameLength* and the *txBufferOffset* input parameter unless the DWT_API_ERROR_CHECK code switch is defined. If this is defined it will assert if an error is detected. It is up to the developer to ensure that the assert macro is correctly enabled in order to trap any error conditions that arise.

Example code:

Typical usage is to write the data, configure the frame control with starting buffer offset and frame length and then enable transmission as follows:

```
dwt_writetxdata(fLength,dataBuffPtr,0); // write the frame data at offset 0
dwt_writetxfctrl(fLength+2,0,0);          // set the frame control register
dwt_starttx(DWT_START_TX_IMMEDIATE);    // send the frame
```

5.3.3 dwt_starttx

int dwt_starttx(uint8_t mode);

This function initiates transmission of the frame. The *mode* parameter is described below.

Parameters:

Type	Name	Description
uint8_t	mode	This is a bit mask defining the operation of the function, see notes and Table 13.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

This function is called to start the transmission of a frame.

Transmission begins immediately if the *mode* parameter is DWT_START_TX_IMMEDIATE (0). When the *mode* parameter is DWT_START_TX_DELAYED (1) transmission begins when the system time reaches the *starttime* specified in the call to the *dwt_setdelayedtrxtime()* function described below. The *mode* parameter, with DWT_RESPONSE_EXPECTED set (i.e. 0x2, 0x3, 0x6, 0xA, 0x12 or 0x22), is used to turn the receiver on automatically after the TX event is complete (see table below). This is used to make sure that there are no delays in turning on the receiver and that the IC can start receiving data (e.g. ACK/response) which might come within 12 symbol times from the end of transmission. This function returns 0 for success, or -1 for error. Error means that the transmission has been aborted.

In performing a delayed transmission, if the host microprocessor is late in invoking the *dwt_starttx()* function, (i.e. so that the IC's system clock has passed the specified *starttime* and would have to complete almost a whole clock count period before the start time is reached), then the transmission is aborted (transceiver off) and the *dwt_starttx()* function returns the -1 error indication.

Table 13: Mode parameter to dwt_starttx() function

Mode	Mask Value	Description
DWT_START_TX_IMMEDIATE	0x0	The transmitter starts sending frame immediately.
DWT_START_TX_DELAYED	0x1	The transmitter will start sending a frame once the programmed *starttime* is reached. See *dwt_setdelayedtrxtime().*
DWT_RESPONSE_EXPECTED	0x2	Response is expected, once the frame is sent the transceiver will enter receive mode to wait for response message. See *dwt_setrxaftertxdelay().* **NOTE all of the START_TX commands can include DWT_RESPONSE_EXPECTED to enable receiver after completed transmission event.**
DWT_START_TX_DLY_REF	0x4	The transmitter will start sending a frame at specified time (which is the sum of time in DREF_TIME register + any time in DX_TIME register, i.e. programmed *refttime* and *starttime*) See *dwt_setreferencerxtime()* and *dwt_setdelayedtrxtime().*
DWT_START_TX_DLY_RS	0x8	The transmitter will start sending a frame at specified time (which is the sum of RX timestamp + any time in DX_TIME register, i.e. last received timestamp and *starttime*) See *dwt_readrxtimestamphi32()*and *dwt_setdelayedtrxtime().*
DWT_START_TX_DLY_TS	0x10	The transmitter will start sending a frame at specified time (which is the sum of TX timestamp + any time in DX_TIME register, i.e. last transmitted timestamp and *starttime*) See *dwt_readtxtimestamphi32()* and *dwt_setdelayedtrxtime().*
DWT_START_TX_CCA	0x20	The transmitter will start sending a frame if no preamble is detected within PTO time (as configured in preamble timeout register see *dwt_setpreambledetecttimeout()*)

Example code:

Typical usage is to write the data, configure the frame control with starting buffer offset and frame length and then enable transmission as follows:

```
dwt_writetxdata(fLength,dataBuffPtr,0); // write the frame data at offset 0
dwt_writetxfctrl(fLength,0,0);          // set the frame control register
dwt_starttx(DWT_START_TX_IMMEDIATE);    // send the frame
```

5.3.4 dwt_setdelayedtrxtime

void dwt_setdelayedtrxtime (uint32_t starttime);

This function sets a send time to use in delayed send or the time at which the receiver will turn on (a delayed receive). This function should be called to set the required send time before invoking the *dwt_starttx()* function (above) to initiate the transmission (in *DELAYED_TX* mode), or *dwt_rxenable()* (below) with *delayed* parameter set to 1.

Parameters:

Type	Name	Description
uint32_t	starttime	The TX or RX start time. The 32-bit value is the high 32-bits of the system time value at which to send the message, or at which to

		turn on the receiver. The low order bit of this is ignored. This essentially sets the TX or RX time in units of approximately 8 ns. (or more precisely 512/(499.2e6*128) seconds) For transmission this is the raw transmit timestamp not including the antenna delay, which will be added. For reception it specifies the time to turn the receiver on.

Return Parameters:

none

Notes:

This function is called to program the delayed transmit or receive start time. The *starttime* parameter specifies the time at which to send/start receiving, when the system time reaches this time (minus the times it needs to send preamble etc.) then the sending of the frame begins. The actual time at which the frame's RMARKER transits the antenna (the standard TX timestamp event) is given by the *starttime* + the transmit antenna delay. If the application wants to embed this time into the message being sent it must do this calculation itself.

The system time counter is 40 bits wide, giving a wrap period of 17.20 seconds.

NOTE: Typically, delayed sending might be used to give a fixed response delay with respect to an incoming message arrival time, or, because the application wants to embed the message send time into the message itself. The delayed receive might be used to save power and turn the receiver on only when response message is expected.

Example code:

Typical usage is to write the data, configure the frame control with starting buffer offset and frame length and then enable transmission as follows:

In this example the previous frame's TX timestamp time is used, and a new frame is sent with 100 ms delay from the previous TX time. The full 40-bit representation of 100 ms would be 0x17CDC0000, however as the delay register uses high 32 bits only a value of 0x17CDC00 is used.

```
dwt_writetxdata(frameLength,dataBufferPtr,0);  // write the frame data at
                                               // offset 0
dwt_writetxfctrl(frameLength,0,0);             // set the frame control
                                               // register
dwt_setdelayedtrxtime(0x17CDC00);              // set the 100ms delay
r = dwt_starttx(DWT_START_TX_DLY_TS);          // send the frame at
                                               // appropriate time w.r.t TX
                                               // timestamp
if (r != DWT_SUCCESS)
{
    // start TX was late, TX has been aborted.
    // Application should take appropriate recovery activity
}
```

5.3.5 dwt_setreferencerxtime

void dwt_setreferencetrxtime (uint32_t reftime);

This function sets a reference time to which any delay time is added (as specified by *starttime* in *dwt_setdelayedtrxtime()).* The sum of both will be used as a delayed send time or the time at which the receiver will turn on (a delayed receive). This function should be called to set the required reference time before invoking the *dwt_starttx()* function (above) to initiate the transmission (in *DELAYED_TX* mode), or *dwt_rxenable()* (below) with *delayed* parameter set to 1.

Parameters:

Type	Name	Description
uint32_t	reftime	The TX or RX reference time. The 32-bit value is the high 32-bits of the system time. The low order bit of this is ignored. This essentially sets the TX or RX time in units of approximately 8 ns. (or more precisely 512/(499.2e6*128) seconds)

Return Parameters:

none

Notes:

This function is called to save the delayed transmit or receive reference time. The *reftime* parameter specifies a time at which for example a "sync or beacon" frame was sent, and which will be used as a reference w.r.t. which other frames will be sent/received.

The system time counter is 40 bits wide, giving a wrap period of 17.20 seconds.

NOTE: Typically, delayed sending might be used to give a fixed response delay with respect to an incoming message arrival time, or, because the application wants to embed the message send time into the message itself. The delayed receive might be used to save power and turn the receiver on only when response message is expected.

Example code:

Typical usage is to write the data, configure the frame control with starting buffer offset and frame length and then enable transmission as follows:

In this example consider a reference "sync or beacon" frame is sent at reference time. This time is saved in the reference register. Then, sometime later, we wish to transmit a frame 100 ms after this time, perhaps a next beacon frame. The full 40-bit representation of 100ms would be 0x17CDC0000, however as the reference time register contains only the high 32 bits a value of 0x17CDC00 is used. (The TX timestamp value should be read after a TX done interrupt triggers.)

```
dwt_setreferencetrxtime(dwt_readtxtimestamphi32()) ; // read TX time e.g. of
                                                     // sync frame

dwt_writetxdata(frameLength,dataBufferPtr,0);  // write the new frame data at
                                               // offset 0
dwt_writetxfctrl(frameLength,0,0);             // set the frame control
                                               // register
dwt_setdelayedtrxtime(0x17CDC00);              // set the delay of 100 ms
r = dwt_starttx(DWT_START_TX_DLY_REF);         // send the frame at
                                               // appropriate time w.r.t. REF
                                               // timestamp

if (r != DWT_SUCCESS)
```

```
{
    // start TX was late, TX has been aborted.
    // Application should take appropriate recovery activity
}
```

5.3.6 dwt_readtxtimestamp

void dwt_readtxtimestamp(uint8_t* timestamp);

This function reads the actual time at which the frame's RMARKER transits the antenna (the standard TX timestamp event). This time will include any TX antenna delay if programmed via the *dwt_gettxantennadelay()* API function. The function returns a 40-bit timestamp value in the buffer passed in as the input parameter.

Parameters:

Type	Name	Description
uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 5 bytes long.) The low order byte is the first element.

Return Parameters:

none

Notes:

This function can be called after the transmission complete event, DWT_INT_TFRS (see *dwt_isr()* function).

5.3.7 dwt_readtxtimestamplo32

uint32_t dwt_readtxtimestamplo32(void);

This function returns the low 32-bits of the 40-bit transmit timestamp (*dwt_readtxtimestamp()).*

Parameters:

none

Return Parameters:

Type	Description
uint32_t	Low 32-bits of the 40-bit transmit timestamp.

Notes:

This function can be called after the transmission complete event, DWT_INT_TFRS (see *dwt_isr()* function).

5.3.8 dwt_readtxtimestamphi32

uint32_t dwt_readtxtimestamphi32(void);

This function returns the high 32-bits of the 40-bit transmit timestamp.

Parameters:

none

Return Parameters:

Type	Description
uint32_t	High 32-bits of the 40-bit transmit timestamp.

Notes:

This function can be called after the transmission complete event, DWT_INT_TFRS (see *dwt_isr()* function).

5.3.9 dwt_readrxtimestamp

void dwt_readrxtimestamp(uint8_t* timestamp);

This function returns the time at which the frame's RMARKER is received, including the antenna delay adjustments if this is programmed via the *dwt_setrxantennadelay()* API function. The function returns a 40-bit value.

Parameters:

Type	Name	Description
uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 5 bytes long.) The low order byte is the first element.

Return Parameters:

none

Notes:

This function can be called after the frame received event, DWT_INT_RFCG (see *dwt_isr()* function). Which means we have a good FCS and a valid timestamp.

5.3.10 dwt_readrxtimestamp_ipatov

void dwt_readrxtimestamp_ipatov(uint8_t* timestamp);

This function returns the time at which the frame's RMARKER is received, including the antenna delay adjustments if this is programmed via the *dwt_setrxantennadelay()* API function, w.r.t. Ipatov CIR. The function returns a 40-bit value.

Parameters:

Type	Name	Description
uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 5 bytes long.) The low order byte is the first element.

Return Parameters:

none

Notes:

This function can be called after the frame received event, DWT_INT_RFCG (see *dwt_isr()* function). Which means we have a good FCS and a valid timestamp.

5.3.11 dwt_readrxtimestamp_sts

void dwt_readrxtimestamp_sts(uint8_t* timestamp);

This function returns the time at which the frame's RMARKER is received, including the antenna delay adjustments if this is programmed via the *dwt_setrxantennadelay()* API function, w.r.t. STS CIR. It is only valid if packet with STS is used, see *dwt_configure()*. The function returns a 40-bit value.

Parameters:

Type	Name	Description
uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 5 bytes long.) The low order byte is the first element.

Return Parameters:

none

Notes:

This function can be called after the frame received event, DWT_INT_RFCG (see *dwt_isr()* function). Which means we have a good FCS and a valid timestamp.

5.3.12 dwt_readrxtimestampunadj

void dwt_readrxtimestampunadj(uint8_t* timestamp);

This function returns the raw time at which the frame's RMARKER is received before any CIA first path algorithm adjustments. This will be the high 32-bits of the 40-bit system time.

Parameters:

Type	Name	Description

uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 4 bytes long.) The low order byte is the first element.

Return Parameters:

none

Notes:

5.3.13 dwt_readrxtimestamplo32

uint32_t dwt_readrxtimestamplo32(void);

This function returns the low 32-bits of the 40-bit received timestamp (*dwt_readrxtimestamp()*).

Parameters:

none

Return Parameters:

Type	Description
uint32_t	Low 32-bits of the 40-bit received timestamp.

Notes:

This function can be called after the frame received event, DWT_INT_RFCG (see *dwt_isr ()* function). Which means we have a good FCS and a valid timestamp.
This API should not be used when using Double RX buffer mode (see *dwt_setdblrxbuffmode ()*), as it will not return a valid RX time. *dwt_readrxtimestamp_ipatov()* or *dwt_readrxtimestamp_sts()* should be used instead.

5.3.14 dwt_readrxtimestamphi32

uint32_t dwt_readrxtimestamphi32(void);

This function returns the high 32-bits of the 40-bit received timestamp (*dwt_readrxtimestamp()*).

Parameters:

none

Return Parameters:

Type	Description
uint32_t	High 32-bits of the 40-bit received timestamp.

Notes:

This function can be called after the frame received event, DWT_INT_RFCG (see *dwt_isr()* function). Which means we have a good FCS and a valid timestamp.
This API should not be used when using Double RX buffer mode (see *dwt_setdblrxbuffmode()*), as it will not return a valid RX time. *dwt_readrxtimestamp_ipatov()* or *dwt_readrxtimestamp_sts()* should be used instead.

5.3.15 dwt_readsystime

void dwt_readsystime(uint8_t* timestamp);

This function returns the system time, which is a high 32-bit value of internal 40-bit system counter. The time will only be valid when the IC is in IDLE, TX or RX states because the system timer is running, the timer is disabled in IDLE_RC or SLEEP states.

Parameters:

Type	Name	Description
uint8_t*	timestamp	The pointer to the buffer into which the timestamp value is read. (The buffer needs to be at least 4 bytes long.) The low order byte is the first element. The low order bit will always be 0, as the system timer runs in units of approximately 8 ns. (more precisely 512/(499.2e6*128) seconds or 63.8976GHz).

Return Parameters:

none

Notes:

This function can be called to read the system time, when the IC is not in the INIT state.

DW3000 note: once this register is read the system time value is latched and any subsequent read will return the same value. To clear the current value in the register an SPI write transaction is necessary, the following read of the SYS_TIME register will return a new value.

5.3.16 dwt_readsystimestamphi32

uint32_t dwt_readsystimestamphi32(void);

This function returns the high 32-bits of the 40-bit system time. The LSB will always be 0, as the system timer runs in units of approximately 8 ns. (more precisely 512/(499.2e6*128) seconds or 63.8976GHz).

Parameters:

none

Return Parameters:

Type	Description
uint32_t	High 32-bits of the 40-bit system timestamp.

Notes:

This function can be called to read the IC system time.

DW3000 note: once this register is read the system time value is latched and any subsequent read will return the same value. To clear the current value in the register an SPI write transaction is necessary, the following read of the SYS_TIME register will return a new value.

5.3.17 dwt_reset_system_counter

void dwt_reset_system_counter(void);

This function will reset the internal system time counter. The counter will be momentarily reset to 0, and then will continue counting as normal. The system time/counter is only available when device is in IDLE or TX/RX states.).

Parameters:

none

Return Parameters:

none

Notes:

5.3.18 dwt_forcetrxoff

void dwt_forcetrxoff(void);

This function may be called at any time to disable the active transmitter or the active receiver and put the IC back into idle mode (transceiver off).

Parameters:

none

Return Parameters:

none

Notes:

The *dwt_forcetrxoff()* function can be called any time and it will disable the active transmitter or receiver and put the device in IDLE mode. It issues a transceiver off command to the IC and also clears status register event flags, so that there should be no outstanding/pending events for processing.

5.3.19 dwt_rxenable

int dwt_rxenable(int mode);

This function turns on the receiver to wait for a receive frame. The mode parameter is a bit field that allows for selection of number of different RX operations as defined in Table 14. In delayed RX

modes the receiver is not turned on until as specific time, set via *dwt_setdelayedtrxtime()* and *dwt_setreferencerxtime()*. This facility is useful to save power in the case when the timing of a response is known. The mode parameter also controls whether the receiver is enabled in case of error, i.e. the delayed RX being late, see notes below for details.

Parameters:

Type	Name	Description
int	mode	This is a bit mask defining the operation of the function, see notes and

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

This function can be called any time to enable the receiver. The device should be initialised and have its RF configuration set.

In performing a delayed RX, the host microprocessor can be late in invoking the *dwt_rxenable()*, (i.e. the system clock has passed the *starttime* specified in the call to the *dwt_setdelayedtrxtime()* function). The IC has a status flag warning when the specified start time is more than a half period (of the system clock) away. If this is the case, since the clock has a period of over 17 seconds, it is assumed that such a long RX delay is not needed, and the delayed RX is cancelled. The receiver is then either immediately enabled or left off depending on whether DWT_IDLE_ON_DLY_ERR was set in the supplied "mode" parameter, and error flag is returned indicating that the RX on was late. It is up to the application to take whatever remedial action is needed in the case of this late error.

Table 14: Mode parameter to dwt_rxenable() function

Mode	Mask Value	Description
DWT_START_RX_IMMEDIATE	0x00	The receiver is activated immediately.
DWT_START_RX_DELAYED	0x01	The receiver, otherwise the receiver will be turned on when the time reaches the *starttime* set through the *dwt_setdelayedtrxtime()*.
DWT_IDLE_ON_DLY_ERR	0x02	This applies only when a delayed start is determined to be late (see notes above). If this is set the receiver will not be enabled in case of a late error, i.e. the IC will be left in IDLE mode. Otherwise, the receiver will be enabled
DWT_START_RX_DLY_REF	0x04	The receiver will be enabled at specified time (which is the sum of time in DREF_TIME register + any time in DX_TIME register, i.e. programmed *refttime* and *starttime*) See *dwt_setreferencerxtime()* and *dwt_setdelayedtrxtime().*

Mode	Mask Value	Description
DWT_START_RX_DLY_RS	0x08	The receiver will be enabled at specified time (which is the sum of RX timestamp + any time in DX_TIME register, i.e. last received timestamp and *starttime*) See *dwt_readrxtimestamphi32()*and *dwt_setdelayedtrxtime().*
DWT_START_RX_DLY_TS	0x10	The receiver will be enabled at specified time (which is the sum of TX timestamp + any time in DX_TIME register, i.e. last transmitted timestamp and *starttime*) See *dwt_readtxtimestamphi32*() and *dwt_setdelayedtrxtime().*

5.3.20 dwt_setsniffmode

void dwt_setsniffmode(int enable, uint8_t timeOn, uint8_t timeOff);

When the receiver is enabled, it begins looking for preamble sequence symbols, and by default, in this preamble-hunt mode the receiver is continuously active. This *dwt_setsniffmode()* function allows the configuration of a lower power preamble-hunt mode. In *SNIFF mode* the receiver (RF and digital) is not on all the time, but rather is sequenced on and off with a specified duty-cycle. Using *SNIFF mode* causes a reduction in RX sensitivity depending on the ratio and durations of the on and off periods. See "Low-Power SNIFF mode" chapter in the User Manual [2] for more details.

Parameters:

Type	Name	Description
int	enable	1 to activate SNIFF mode or 2 to enable SNIFF with LDC, 0 to deactivate it and go back to the normal higher-powered reception mode.
uint8_t	timeOn	The receiver ON time in PACs (as per the *rxPAC* parameter in the *dwt_config_t* structure parameter to the *dwt_configure()* API function call). The minimum value for correct operation is 2, giving an on time of 2 PACs. The maximum value is 15. Value of 1 is not allowed and will disable SNIFF mode.
uint8_t	timeOff	The receiver OFF time, expressed in multiples of 128/125 µs (~1 µs). Max value is 255.

Return Parameters:

none

Notes:

This function can be called as part of device receiver configuration.

By default (where this API is not invoked) the IC will operate its receiver in normal reception mode. If this API is used to enable SNIFF mode this will be maintained until a reset or it is disabled or re-

configured by another call to this *dwt_setsniffmode()* function. The SNIFF mode setting is not affected by the *dwt_configure()* function.

5.3.21 dwt_setdblrxbuffmode

void dwt_setdblrxbuffmode (dwt_dbl_buff_state_e dbl_buff_state, dwt_dbl_buff_mode_e dbl_buff_mode);

This function enables double buffered receive mode.

Parameters:

Type	Name	Description
dwt_dbl_buff_state_e	dbl_buff_state	DBL_BUF_STATE_EN to enable, DBL_BUF_STATE_DIS to disable the double buffer RX feature.
dwt_dbl_buff_mode_e	dbl_buff_mode	DBL_BUF_MODE_AUTO to enable RX auto re-enable on good frame reception. If mode is DBL_BUF_MODE_MAN, then the host needs to re-enable the receiver following a good frame reception

Return Parameters:

none

Notes:

The *dwt_setdblrxbuffmode()* function is used to configure the receiver in double buffer mode. This should not be done when the receiver is enabled. It should be selected in idle mode before the *dwt_setdblrxbuffmode()* function is called.

Once the data for the received frame is read, the host should call the *dwt_signal_rx_buff_free()* to let the device know the buffer is free again, the host will then be ready to read the next received frame. This is done in the *dwt_isr()* which handles the IRQ.

The reader is referred to “Double Receive Buffer” chapter in the User Manual [2] for more details.

5.3.22 dwt_signal_rx_buff_free

void dwt_signal_rx_buff_free (void);

This function signals to the DW3xxx that the host is finished with current buffer and the buffer is free for the DW3xxx to receive into again. This function is only relevant if device has double RX buffer mode enabled.

Parameters:

none

Return Parameters:

none

Notes:

5.3.23 dwt_setrxtimeout

void dwt_setrxtimeout (uint32_t time);

The *dwt_setrxtimeout()* function sets the receiver to timeout (and disable) when no frame is received within the specified time. This function should be called before the *dwt_rxenable()* function is called to turn on the receiver. The time parameter used here is in 1.0256 us (*UWB microseconds*, i.e. 512/499.2 MHz) units. The maximum RX timeout is ~ 1.0754s.

Parameters:

Type	Name	Description
uint32_t	time	Timeout time in microseconds (1.0256 us). If this is 0, the timeout will be disabled. The max value is 0xFFFFF.

Return Parameters:

none

Notes:

If RX timeout is being employed then this function should be called before *dwt_rxenable()* to configure the frame wait timeout time, and enable the frame wait timeout.

5.3.24 dwt_setrxaftertxdelay

void dwt_setrxaftertxdelay(uint32_t rxDelayTime);

This function sets the delay in turning the receiver on after a frame transmission has completed. The delay, *rxDelayTime*, is in *UWB microseconds* (1 *UWB microsecond* is 512/499.2 microseconds). It is a 20-bit wide field. This should be set before start of frame transmission after which a response is expected, i.e. before invoking the *dwt_starttx()* function (above) to initiate the transmission (in DWT_RESPONSE_EXPECTED mode). E.g. transmission of a frame with an ACK request bit set.

Parameters:

Type	Name	Description
uint32_t	rxDelayTime	The turnaround time, in UWB microseconds, between the TX completion and the RX enable.

Return Parameters:

none

Notes:

This function is used to set the delay time before automatic receiver enable after a frame transmission. The smallest value that can be set is 0. If 0 is set the IC will turn the receiver on as soon as possible, which approximately takes 6.2 µs. If setting a value smaller than 6.2 µs, the device will still take 6.2 µs to switch to receive mode.

5.3.25 dwt_setpreambledetecttimeout

void dwt_setpreambledetecttimeout (uint16_t timeout);

This *dwt_setpreambledetecttimeout()* API function sets the receiver to timeout (and disable) when no preamble is received within the specified time. This function should be called before the *dwt_rxenable()* function is called to turn on the receiver. The time parameter units are PACs (as per the *rxPAC* parameter in the *dwt_config_t* structure parameter to the *dwt_configure()* API function call).

Parameters:

Type	Name	Description
uint16_t	timeout	This is the preamble detection timeout duration. If preamble is not detected within this time, counted from the time the receiver is enabled, the receiver will be turned off. The time here is specified in multiples of PAC size, (as per the *rxPAC* parameter in the *dwt_config_t* structure parameter to the *dwt_configure()* API function call). The IC automatically adds 1 to the configured value. A value of 0 disables the timer and timeout.

Return Parameters:

none

Notes:

If preamble detection timeout is being employed, then this function should be called before *dwt_rxenable()* is called.

5.3.26 dwt_readrxdata

void dwt_readrxdata(uint8_t *buffer, uint16_t length, uint16_t rxBufferOffset);

This function reads a number, *len*, bytes from the IC receive data buffer, beginning at the specified offset, *bufferOffset*, into the given buffer, *buffer*.

Parameters:

Type	Name	Description
uint8_t*	buffer	The pointer to the buffer into which the data will be read.
uint16_t	length	The length of data to be read (in bytes).
uint16_t	rxBufferOffset	The offset at which to start to read the data.

Return Parameters:

none

Notes:

This function should be called on the reception of a good frame to read the received frame data. The offset might be used to skip parts of the frame that the application is not interested in or has read previously.

5.3.27 dwt_read_rx_scratch_data

void dwt_read_rx_scratch_data(uint8_t *buffer, uint16_t length, uint16_t rxBufferOffset);

This is used to read the data from the RX scratch buffer, from an offset location given by offset parameter.

Parameters:

Type	Name	Description
uint8_t*	buffer	The pointer to the buffer into which the data will be read.
uint16_t	length	The length of data to be read (in bytes).
uint16_t	rxBufferOffset	The offset at which to start to read the data.

Return Parameters:

none

Notes:

This function should be called on the reception of a good frame to read the received frame data. The offset might be used to skip parts of the frame that the application is not interested in or has read previously.

5.3.28 dwt_write_rx_scratch_data

void dwt_write_rx_scratch_data(uint8_t *buffer, uint16_t length, uint16_t bufferOffset);

This function is used to write the data from the RX scratch buffer, from an offset location given by the offset parameter.

Parameters:

Type	Name	Description
uint8_t*	buffer	The pointer to the buffer into which the data will be read.
uint16_t	length	The length of data to be read (in bytes).
uint16_t	bufferOffset	The offset at which to write the data.

Return Parameters:

none

5.4 Diagnostic APIs

5.4.1 dwt_readaccdata

void dwt_readaccdata(uint8_t *buffer, uint16_t len, uint16_t accOffset);

This API function reads data from the IC's accumulator memory. This data represents the channel impulse response (CIR) of the RF channel. Reading this data is not required in normal operation but it may be useful for diagnostic purposes. The accumulator contains complex values, each comprised of an 18-bit real integer and an 18-bit imaginary integer, for each tap of the accumulator. Each complex value represents a 1 ns sample interval (or more precisely half a period of the 499.2 MHz fundamental frequency).

When STS mode is enabled there are two separate accumulations and two CIRs: one during the Ipatov sequence and one during the STS, both may be read using this *dwt_readaccdata()* API function. The Ipatov sequence begins at offset 0 and has a span of one symbol time (This is 992 samples for the nominal 16 MHz mean PRF, or, 1016 samples for the nominal 64 MHz mean PRF). The STS begins at offset 1024 and has a span of half a symbol time (512 samples irrespective of PRF setting). If PDOA mode 3 is used, the STS CIR will be split into two. One half of STS symbols and corresponding CIR will be received through one antenna port and saved into CIR memory from 1024 to 1535, and the second half of STS symbols will be received through the other antenna port and saved into CIR memory from 1536 to 2047.

The *dwt_readaccdata()* function reads, *len*, bytes of accumulator buffer data, from a given offset, *sampleOffset*, into the memory pointed to by the supplied *buffer* parameter. Each 18-bit complex sample has 3 bytes of real and 3 bytes of imaginary data (delivered by the IC as signed 24-bit numbers). The accumulator data starts from *buffer[1]*. The first byte written to *buffer[0]* is always a dummy byte, and to allow for this the specified length should always be 1 bigger than the length required.

When reading from CIR memory with an offset less than 127, a normal SPI read can be used, however to read data from CIR memory with offset greater than 127, an indirect SPI read has to be done. To perform an indirect SPI read indirect pointers need to be used: PTR_ADDR_A or PTR_ADDR_B. Firstly the register address needs to be programmed into e.g. indirect pointer A (PTR_ADDR_A) and offset into PTR_OFFSET_A and then the indirect pointer register (INDIRECT_PRT_A) needs to be read as normal to read out the required data. Please see more details on this in DW3XXX User Manual [2].

For example to read the first 2 complex samples of the CIR the function should be done as shown in the example below:

Example code:

```
uint8_t cir_buiffer[xx] ;

dwt_readaccdata(uint8_t *buffer, uint16_t len, uint16_t sampleOffset);
```

Note that the length is in bytes while the offset is in complex samples.

Both accumulators can be read together in this case length should be 1536*6 + 1 bytes.

Parameters:

Type	Name	Description
uint8_t*	buffer	The pointer to the destination buffer into which the read accumulator data will be written.
uint16_t	len	The length of data to be read (in bytes). Since each complex value occupies six octets, the value used here should naturally be a multiple of six, plus 1 for the dummy byte as described above. Maximum length is 9217
uint16_t	bufferOffset	The offset at which to start to read the data. Offset 0 should be used when reading the full accumulator.

Return Parameters:

none

Notes:

dwt_readaccdata() may be called after frame reception to read the accumulator data for diagnostic purposes. The accumulator is not double buffered so this access must be done before the receiver is re-enabled otherwise the accumulator data may be overwritten during the reception of the next frame. The data returned in the buffer has the following format (for *bufferOffset* input of zero):

buffer index	Description of elements within buffer
0	Dummy Octet
1	Low 8 bits of real part of accumulator sample index 0
2	Mid 8 bits of real part of accumulator sample index 0
3	High 8 bits of real part of accumulator sample index 0
4	Low 8 bits of imaginary part of accumulator sample index 0
5	Mid 8 bits of imaginary part of accumulator sample index 0
6	High 8 bits of imaginary part of accumulator sample index 0
7	Low 8 bits of real part of accumulator sample index 1
8	Mid 8 bits of real part of accumulator sample index 1
9	High 8 bits of real part of accumulator sample index 1
10	Low 8 bits of imaginary part of accumulator sample index 1
:	:

In examining the CIR it is normal to compute the magnitude of the complex values.

5.4.2 dwt_configciadiag

void dwt_configciadiag (uint8_t enable_mask);

This function can be used to enable full or partial CIA diagnostic calculations in the IC during reception processing of frame. Note partial diagnostics are enabled by default in the IC.

Parameters:

Type	Name	Description

int	enable_mask	Table 15 lists the allowed values.

Table 15: Values for dwt_configciadiag() *enable_mask* parameter

Event	Bit mask	Description
DW_CIA_DIAG_LOG_MIN	0x0	CIA to log reduced set of diagnostic registers
DW_CIA_DIAG_LOG_ALL	0x1	CIA to log the whole set of diagnostic registers
DW_CIA_DIAG_LOG_MIN	0x2	CIA to copy to swinging set a minimal set of diagnostic registers in Double Buffer mode
DW_CIA_DIAG_LOG_MID	0x4	CIA to copy to swinging set a medium set of diagnostic registers in Double Buffer mode.
DW_CIA_DIAG_LOG_MAX	0x8	CIA to copy to swinging set a maximum set of diagnostic registers in Double Buffer mode.

Return Parameters:

none

Notes:

Turing on diagnostics, means that the reception of a frame consumes some more power and takes more time while the IC performs the calculations to generate the diagnostic values. The diagnostic values may be read, as part of the RX callback for instance, using the *dwt_readdiagnostics()* API.

5.4.3 dwt_readdiagnostics

void dwt_readdiagnostics(dwt_rxdiag_t * diagnostics);

This function reads receiver frame quality diagnostic values.

Parameters:

Type	Name	Description
dwt_rxdiag_t*	diagnostics	Pointer to the diagnostics structure which will contain the read data.

```
typedef struct
{
    uint8_t       ipatovRxTime[5] ;
    uint8_t       ipatovRxStatus ;
    uint16_t      ipatovPOA ;
    uint8_t       stsRxTime[5] ;
    uint16_t      stsRxStatus ;
    uint16_t      stsPOA;
    uint8_t       sts2RxTime[5];
    uint16_t      sts2RxStatus;
    uint16_t      sts2POA;
    uint8_t       tdoa[6];
    int16_t       pdoa;
```

```
    int16_t       xtalOffset ;
    uint32_t      ciaDiag1 ;
    uint32_t      ipatovPeak ;
    uint32_t      ipatovPower ;
    uint32_t      ipatovF1 ;
    uint32_t      ipatovF2 ;
    uint32_t      ipatovF3 ;
    uint16_t      ipatovFpIndex ;
    uint16_t      ipatovAccumCount ;
    uint32_t      stsPeak ;
    uint32_t      stsPower ;
    uint32_t      stsF1 ;
    uint32_t      stsF2 ;
    uint32_t      stsF3 ;
    uint16_t      stsFpIndex ;
    uint16_t      stsAccumCount ;
    uint32_t      sts2Peak;
    uint32_t      sts2Power;
    uint32_t      sts2F1;
    uint32_t      sts2F2;
    uint32_t      sts2F3;
    uint16_t      sts2FpIndex;
    uint16_t      sts2AccumCount;
}dwt_rxdiag_t ;
```

Return Parameters:

none

Notes:

This function is used to read the received frame diagnostic data. They can be read after a frame is received (e.g. after DWT_SIG_RX_OKAY event reported in the RX call-back function called from *dwt_isr()*). CIA diagnostic level must be configured with the *dwt_configciadiag()* otherwise only the minimum diagnostics will be available, please see DW3XXX User Manual [2].

Fields	Description of fields within the *dwt_rxdiag_t* structure
ipatovRxTime	40-bit RX timestamp from Ipatov sequence, arranged as array of octets, with least significant octet first.
ipatovRxStatus	8-bit RX status info for Ipatov sequence
ipatovPOA	POA from Ipatov preamble CIR
stsRxTime	40-bit RX timestamp from the STS, arranged as array of octets, with least significant octet first.
stsRxStatus	16-bit RX status info for STS
stsPOA	POA from STS CIR
sts2RxTime	40-bit RX timestamp from the 2nd STS, arranged as array of octets, with least significant octet first. (this is used in PDOA mode 3, when the STS is split)

Fields	Description of fields within the *dwt_rxdiag_t* structure
sts2RxStatus	16-bit RX status info for the 2nd STS (this is used in PDOA mode 3, when the STS is split)
ciphe2rPOA	POA from the 2nd STS CIR (this is used in PDOA mode 3, when the STS is split)
tdoa	TDOA from two STS RX timestamps (valid when PDOA mode 3 is configured)
pdoa	PDOA from two POAs, signed int [1:-11] in radians
xtalOffset	Estimated crystal offset of remote device. This is PPM x16, i.e. divide integer number by 16 to get the value in PPM.
ciaDiag1	Diagnostics common to both sequences
ipatovPeak	index and amplitude of peak sample in Ipatov sequence CIR
ipatovPower	channel area allows estimation of channel power for the Ipatov sequence
ipatovF1	First path amplitude for the Ipatov sequence value reporting the magnitude for the sample at the index 1 after the reported first path index value.
ipatovF2	First path amplitude for the Ipatov sequence value reporting the magnitude for the sample at the index 2 after the reported first path index value.
ipatovF3	First path amplitude for the Ipatov sequence value reporting the magnitude for the sample at the index 3 after the reported first path index value.
ipatovFpIndex	First path index for Ipatov sequence
ipatovAccumCount	Number accumulated symbols for Ipatov sequence
stsPeak	index and amplitude of peak sample in STS CIR
stsPower	channel area allows estimation of channel power for the STS
stsF1	First path amplitude for the STS value reporting the magnitude for the sample at the index 1 after the reported first path index value.
stsF2	First path amplitude for the STS value reporting the magnitude for the sample at the index 2 after the reported first path index value.
stsF3	First path amplitude for the STS value reporting the magnitude for the sample at the index 3 after the reported first path index value.
stsFpIndex	First path index for STS
stsAccumCount	Number accumulated symbols for STS

Fields	Description of fields within the *dwt_rxdiag_t* structure
sts2Peak	index and amplitude of peak sample in the 2nd STS CIR, (valid when PDOA mode 3 is configured)
sts2Power	channel area allows estimation of channel power for the 2nd STS, (valid when PDOA mode 3 is configured)
sts2F1	First path amplitude for the 2nd STS value reporting the magnitude for the sample at the index 1 after the reported first path index value. (valid when PDOA mode 3 is configured)
sts2F2	First path amplitude for the 2nd STS value reporting the magnitude for the sample at the index 2 after the reported first path index value. (valid when PDOA mode 3 is configured)
sts2F3	First path amplitude for the 2nd STS value reporting the magnitude for the sample at the index 3 after the reported first path index value. (valid when PDOA mode 3 is configured)
sts2FpIndex	First path index for the 2nd STS, (valid when PDOA mode 3 is configured)
sts2AccumCount	Number accumulated symbols for the 2nd STS, (valid when PDOA mode 3 is configured)

5.4.4 dwt_readpdoa

int16_t dwt_readpdoa (void);

This function is used to read the PDOA result, it will return either the phase difference between Ipatov and STS POAs, or the two STS POAs, depending on the PDOA mode of operation.

Parameters:

none

Return Parameters:

Type	Description
int16_t	The PDOA result in radians (signed number [1:-11]). To convert to degrees: pdoa_deg = ((pdoa_rad/1<<11))*180/π

Notes:

5.4.5 dwt_readtdoa

void dwt_readtdoa(uint8_t * tdoa);

This function is used to read the TDOA (Time Difference Of Arrival). The TDOA value that is read from the register is 41-bits in length. However, 6 bytes (or 48 bits) are read from the register. The remaining 7 bits at the 'top' of the 6 bytes that are not part of the TDOA value should be set to zero and should not interfere with rest of the 41-bit value. However, there is no harm in masking the returned value.

Parameters:

Type	Name	Description
uint8_t*	tdoa	Time difference on arrival - buffer of 6 bytes that will be filled with TDOA value by calling this function.

Return Parameters:

none

Notes:

5.4.6 dwt_get_dgcdecision

uint8_t dwt_get_dgcdecision(void);

This function is used to read the DGC_DECISION index when RX_TUNING is enabled, this value is used to adjust the RX level and FP level estimation.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	The index value to be used in RX level and FP level formulas.

Notes:

5.4.7 dwt_configeventcounters

void dwt_configeventcounters (int enable);

This function enables event counters (TX, RX, error counters) in the IC.

Parameters:

Type	Name	Description
int	enable	Set to 1 to clear and enable the internal digital counters. Set to 0 to disable.

Return Parameters:

none

Notes:

This function is used to enable counters that count the number of frames transmitted, and received, and various types of error events.

5.4.8 dwt_readeventcounters

void dwt_readeventcounters (dwt_deviceentcnts_t *counters);

This function reads the event counters (TX, RX, error counters).

Parameters:

Type	Name	Description
dwt_deviceentcnts_t *	counters	Pointer to the device event counters structure.

```
Typedef struct
{
       uint16_t PHE ;        //number of received header errors
       uint16_t RSL ;        //number of received frame sync loss events
       uint16_t CRCG ;       //number of good CRC received frames
       uint16_t CRCB ;       //number of bad CRC (CRC error) received frames
       uint8_t  ARFE ;       //number of address filter rejections
       uint8_t  OVER ;       //number of RX overflows (used in double buffer mode)
       uint16_t SFDTO ;      //number of SFD timeouts
       uint16_t PTO ;        //number of preamble timeouts
       uint8_t  RTO ;        //number of RX frame wait timeouts
       uint16_t TXF ;        //number of transmitted frames
       uint8_t  HPW ;        //number of half period warnings
       uint8_t  CRCE;        //number of SPI CRC write errors
       uint16_t PREJ;        //number of preamble rejections
      uint16_t SFDD;         //number of SFD detection events (only valid in QM33120)
       uint8_t  STSE;        //number of STS error + warning events


} dwt_deviceentcnts_t ;
```

Return Parameters:

none

Notes:

This function is used to read the internal counters. These count the number of frames transmitted, received, and also number of errors received/detected.

Fields	Description of fields within the *dwt_deviceentcnts_t* structure
PHE	PHR error counter is a 12-bit counter of PHY header errors.

Fields	Description of fields within the *dwt_deviceentcnts_t* structure
RSL	RSE error counter is a 12-bit counter of the non-correctable error events that can occur during Reed Solomon decoding.
CRCG	Frame check sequence good counter is a 12-bit counter of the frames received with good CRC/FCS sequence.
CRCB	Frame check sequence error counter is a 12-bit counter of the frames received with bad CRC/FCS sequence.
ARFE	Frame filter rejection counter is an 8-bit counter of the frames rejected by the receive frame filtering function.
OVER	RX overrun error counter is an 8-bit counter of receive overrun events. This is essentially a count of the reporting of overrun events, i.e. when using double buffer mode, and the receiver has already received two frames, and the host has not processed the first one. The receiver will flag an overrun when it starts receiving a third frame.
SFDT	SFD timeout errors counter is a 12-bit counter of SFD timeout error events.
PTO	Preamble detection timeout event counter is a 12-bit counter of preamble detection timeout events.
RTO	RX frame wait timeout event counter is an 8-bit counter of receive frame wait timeout events.
TXF	TX frame sent counter is a 12-bit counter of transmit frames sent events. This is incremented every time a frame is sent.
HPW	Half period warning counter is an 8-bit counter of “Half Period Warning” events. These relate to late invocation of delayed transmission or reception functionality.
CRCE	SPI CRC write error is an 8-bit counter of “SPI write CRC error” events.
PREJ	Preamble rejection events, this is a 12-bit counter.
SFDD	SFD detection events, this is a 12-bit counter.
STSE	STS error + warning events, this is an 8-bit counter.

5.4.9 dwt_readclockoffset

int16_t dwt_readclockoffset (void);

This function can be used to read the estimated clock offset between the local clock and the remote. The value is calculated as part of the reception of the frame. It relates to last received frame and

should be read after frame reception. This function is an alternative to *dwt_readcarrierintegrator()* which can also be used to calculate clock offset between two devices.

Parameters:

none

Return Parameters:

Type	Description
int16_t	Signed 16-bit clock offset value. To convert to ppm the value should be divided by 2^26 and multiplied by 10e6.

Notes:

Positive value means that the local receiver's clock is running faster than that of the remote transmitter.

If the CIA is not running, this function will return 0 and cannot be used to read the clock offset.

5.4.10 dwt_readcarrierintegrator

int32_t dwt_readcarrierintegrator(void);

The *dwt_readcarrierintegrator()* API function reads the receiver carrier integrator value and returns it as a 32-bit signed value. The receive carrier integrator value is valid at the end of reception of a frame, (and before the receiver is re-enabled). It reflects the frequency offset of the remote transmitter with respect to the local receive clock. A positive carrier integrator value means that the local receive clock is running slower than that of the remote transmitter device.

Parameters:

none

Return Parameters:

Type	Description
int32_t	Receiver carrier integrator value

Notes:

This *dwt_readcarrierintegrator()* API may be called after receiving a frame to determine the clock offset of the remote transmitter the sent the frame. The receive frame should be valid (i.e. with good CRC) otherwise the clock offset information may be incorrect. The following constants are defined to allow the returned carrier integrator be converted to a frequency offset in Hertz and from that to a clock offset in PPM (which depends on the channel centre frequency): FREQ_OFFSET_MULTIPLIER, and HERTZ_TO_PPM_MULTIPLIER_CHAN_5.

The HERTZ_TO_PPM_xxx multipliers are negative quantities, so when the resultant clock offsets are positive it means that the local receiver's clock is running slower than that of the remote transmitter.

Example code:

```
int32_t ci ;
```

```
float clockOffsetHertz ;
float clockOffsetPPM ;

ci = dwt_readcarrierintegrator() ; // Read carrier integrator value

// at 6.81Mb/s data rate convert carrier integrator to clock offset in Hz.
clockOffsetHertz = ci * FREQ_OFFSET_MULTIPLIER;

// On channel 5 convert this to clock offset in PPM.
clockOffsetPPM = clockOffsetHertz * and HERTZ_TO_PPM_MULTIPLIER_CHAN_5 ;
```

5.4.11 dwt_readstsquality

int dwt_readstsquality (int16_t *rxSTSQualityIndex);

This function may be used in any STS mode. It reads the STS quality index and also returns an indication of whether the STS reception quality is good or bad. After a frame is received the *dwt_readstsquality()* API can be used to assess the quality of the STS and hence decide whether to trust the RX timestamp.

The STS is considered good when the (STS) quality index is greater than a specified threshold value, which is a percentage of the configured STS length. These thresholds have been set as hard coded values in the device driver code.

Parameters:

Type	Name	Description
int16_t*	rxSTSQualityIndex	The reported quality index will be stored in this parameter on function exit.

Return Parameters:

Type	Description
int	This value is the STS quality index minus the threshold value which depends on the configured STS PRF and the configured STS length. If this return value is > 0 this indicates that the STS quality is good, however if this return value is negative this indicates that the received frame has bad quality STS and the resulting timestamps are less trustworthy

Notes:

5.4.12 dwt_readstsstatus

int dwt_readstsstatus(uint16_t* stsStatus, int sts_num);

This function may be used in any STS mode. It will read the STS status register in order to show if there are any errors present in the STS signals. It can be used in conjunction with the *dwt_readstsquality()* API after a packet/frame is received.

A 16-bit buffer is passed into the function and is populated with a '1' or '0' depending on whether the 9 different STS statuses are set high or not. The remaining upper 7 bits of the 16 bits are ignored. Only bits 8 through to 0 are set.

Parameters:

Type	Name	Description
uint16_t*	stsStatus	This 16-bit buffer is populated with the various STS statuses that are described in the CY1_TOA_HI register.
int	sts_num	This parameter is used to select which STS packet/frame to analyse the status of. In regular operation, there will only be one STS packet/frame to analyse. However, when PDOA Mode 3 is used, there are two separate STS packets/signals to analyse. '0' will select the first STS while '1' will select the second STS.

Return Parameters:

Type	Description
int	DWT_SUCCESS is returned for good/valid STS status. Otherwise, DWT_ERROR is returned for a bad STS status.

Notes:

The available STS statuses that are available to read as part of the *stsStatus* are described in the table below:

Table 16: stsStatus values

Buffer Bits	Default Value	Description
stsStatus[8]	0x0	Peak growth rate warning
stsStatus[7]	0x0	ADC count warning
stsStatus[6]	0x0	SFD count warning
stsStatus[5]	0x0	Late first path estimation
stsStatus[4]	0x0	Late coarse estimation
stsStatus[3]	0x0	Coarse estimation empty
stsStatus[2]	0x0	High noise threshold
stsStatus[1]	0x0	Non-triangle
stsStatus[0]	0x0	Logistic regression failed

5.4.13 dwt_readctrdbg

uint32_t dwt_readctrdbg(void);

This function is used to read CTR_DBG_ID register. This should be done after packet reception, please see User Manual for more details on this register.

Parameters:

none

Return Parameters:

Type	Description
uint32_t	This value of CTR_DBG_ID register

Notes:

5.4.14 dwt_readdgcdbg

uint32_t dwt_readdgcdbg(void);

This function is used to read DGC_DBG_ID register. This should be done after packet reception, please see User Manual for more details on this register.

Parameters:

none

Return Parameters:

Type	Description
uint32_t	This value of DGC_DBG_ID register

Notes:

5.4.15 dwt_readCIAversion

uint32_t dwt_readCIAversion(void);

This function is used to read the internal CIA version. Is generally used in logging/diagnostic applications.

Parameters:

none

Return Parameters:

Type	Description

uint32_t	This devices CIA version.

Notes:

5.4.16 dwt_getcirregaddress

uint32_t dwt_getcirregaddress (void);

This function is used to return ACC_MEM_ID register address. Is generally used in logging/diagnostic applications when logging CIR data following packet reception, see also *dwt_readaccdata()*.

Parameters:

none

Return Parameters:

Type	Description
uint32_t	The 32-bit address of ACC_MEM_ID register.

Notes:

5.4.17 dwt_get_reg_names

register_name_add_t* dwt_get_reg_names(void);

This function returns a list of register name/value pairs, to enable debug output / logging in external applications e.g. DecaRanging. The implementation is device specific, i.e. DW3000 device values are different from QM331XX devices. This API is not enabled unless _DGB_LOG is defined.

Parameters:

none

Return Parameters:

Type	Description
register_name_add_t*	Pointer to the array of register name/value pairs.

Notes:

5.4.18 dwt_nlos_alldiag

uint8_t dwt_nlos_alldiag(dwt_nlos_alldiag_t *all_diag);

This function will read the device's diagnostics registers regarding IPATOV, STS1, STS2 CIRs. This data can then be used to help in determining if packet has been received in LOS (line-of-sight) or NLOS (non-line-of-sight) condition. To help determine/estimate NLOS condition either Ipatov, STS1 or STS2 CIR diagnostics can be used, (or all three).

Parameters:

Type	Name	Description
dwt_nlos_alldiag_t *	all_diag	Pointer to the all diagnostics structure which will contain the read data.

```
typedef struct
{
    uint32_t        accumCount ;
    uint32_t        F1 ;
    uint32_t        F1 ;
    uint32_t        F1 ;
    uint32_t        cir_power ;
    uint8_t        D ;
    dwt_diag_type_e diag_type ;
    uint8_t         result ;
} dwt_nlos_alldiag_t ;
```

Return Parameters:

Type	Description
uint8_t	DWT_SUCCESS is returned for successful reads of registers. Otherwise, DWT_ERROR is returned.

Notes:

This function is used to read the received frame diagnostic data. They can be read after a frame is received. CIA diagnostic level must be configured with the *dwt_configciadiag(DW_CIA_DIAG_LOG_ALL)* otherwise the diagnostic registers will read back as zero, please see DW3XXX User Manual [2] section 8.2.13.

Fields	Description of fields within the ***dwt_nlos_alldiag_t*** structure
accumCount	the number of preamble symbols accumulated when reading Ipatov diagnostics, or accumulated STS length.
F1	the First Path Amplitude (point 1) magnitude value.
F2	the First Path Amplitude (point 2) magnitude value.
F3	the First Path Amplitude (point 3) magnitude value.
cir_power	the Channel Impulse Response Power value.
D	the DGC_DECISION, treated as an unsigned integer in range 0 to 7.
diag_type	enum to select which diagnostic type to be read: 0x0 for IP_DIAG, 0x1 for STS_DIAG and 0x2 for STS1_DIAG
result	return value to (-1) or (0) on failure or success respectively.

5.4.19 dwt_nlos_ipdiag

void dwt_nlos_ipdiag(dwt_nlos_ipdiag_t *index);

This function will read the IPATOV Diagnostic Registers to get the First Path and Peak Path Index value. This function is used when signal power is low to determine the signal type (LOS or NLOS). Hence only Ipatov diagnostic registers are used to determine the signal type.

Parameters:

Type	Name	Description
dwt_nlos_ipdiag_t *	index	Pointer to the Ipatov diagnostics structure which will contain the read data.

```
typedef struct
{
    uint32_t        index_fp_u32 ;
    uint32_t        index_pp_u32 ;
} dwt_nlos_ipdiag_t;
```

Return Parameters:

none

Notes:

This function is used to read the received frame Ipatov diagnostic data. They can be read after a frame is received. CIA diagnostic level must be configured with the *dwt_configciadiag(DW_CIA_DIAG_LOG_ALL)* otherwise the diagnostic registers will read back as zero, please see DW3XXX User Manual [2] section 8.2.13, “IP_DIAG_0” for peak path index and “IP_DIAG_8” for first path index.

Fields	Description of fields within the *dwt_nlos_alldiag_t* structure
index_fp_u32	the First Path Index.
index_pp_u32	the Peak Path Index.

5.5 Sleep/Wakeup APIs

5.5.1 dwt_calibratesleepcnt

uint16_t dwt_calibratesleepcnt (void);

The *dwt_calibratesleepcnt()* function calibrates the low-power oscillator. It returns the number of XTAL cycles per one low-power oscillator cycle.

Parameters:

none

Return Parameters:

Type	Description
uint16_t	This is number of XTAL cycles per one low-power oscillator cycle.

Notes:

The IC's internal L-C oscillator has an oscillating frequency which is between approximately 15,000 and 34,000 Hz depending on process variations within the IC and on temperature and voltage. To do more precise setting of sleep times its calibration is necessary. See also example code given under the *dwt_configuresleepcnt()* function. This function need to be run before *dwt_configuresleepcnt()* in order to ascertain the counter units required to calculated the sleep time.

5.5.2 dwt_configuresleepcnt

void dwt_configuresleepcnt (uint16_t sleepcnt);

The *dwt_configuresleepcnt()* function configures the sleep counter to a new value. This function needs to be run before *dwt_entersleep()* if sleep mode is used.

Parameters:

Type	Name	Description
uint16_t	sleepcnt	This is the sleep count value to set. The high 16-bits of 28-bit counter. See note below for details of units and code example for configuration detail.

Return Parameters:

none

Notes:

The units of the *sleepcnt* parameter depend on the oscillating frequency of the IC's internal L-C oscillator, which is between approximately 15,000 and 34,000 Hz depending on process variations within the IC and on temperature and voltage. This frequency can be measured using the *dwt_calibratesleepcnt()* function so that sleep times can be more accurately set.

The *sleepcnt* is actually setting the upper 16 bits of a 28-bit counter, i.e. the low order bit is equal to 4096 counts. So, for example, if the L-C oscillator frequency is 15000 Hz then programming the *sleepcnt* with a value of 24 would yield a sleep time of 24 × 4096 ÷ 15000, which is approximately 6.55 seconds.

Example code:

This example shows how to calibrate the low-power oscillator and set the sleep time to 10 seconds.

```
Double t;
uint32_t sleep_time = 0;
uint16_t lp_osc_cal = 0;
uint16_t sleepTime16;

// Measure low power oscillator frequency

lp_osc_cal = dwt_calibratesleepcnt();
```

```
// calibrate low power oscillator
// the lp_osc_cal value is number of XTAL cycles in one cycle of LP OSC
// to convert into seconds (38.4 MHz => 1/38.4 MHz ns)
// so to get a sleep time of 10s we need a value of:
// 10 / period and then >> 12 as the register holds upper 16-bits of 28-bit
// counter

t = ((double) 10.0 / ((double) lp_osc_cal/38.4e6));
sleep_time = (int) t;
sleepTime16 = sleep_time >> 12;

dwt_configuresleepcnt(sleepTime16);      //configure sleep time
```

5.5.3 dwt_configuresleep

void dwt_configuresleep(uint16_t mode, uint8_t wake);

The *dwt_configuresleep()* function may be called to configure the activity of DEEPSLEEP or SLEEP modes. Note TX and RX configurations are maintained in DEEPSLEEP and SLEEP modes so that upon "waking up" there is no need to reconfigure the devices before initiating a TX or RX, although as the TX data buffer is not maintained the data for transmission will need to be written before initiating transmission.

Parameters:

Type	Name	Description
uint16_t	mode	A bit mask which configures which configures the SLEEP parameters, see Table 17.
uint8_t	wake	A bit mask that configures the wakeup event. As defined in Table 18

Return Parameters:

none

Notes:

This function is called to configure the sleep and on wake parameters.

Table 17: Bitmask values for dwt_configuresleep() *mode* bit mask

Event	Bit mask	Description
DWT_PGFCAL	0x0800	On wake-up, run the PGF calibration. **NOTE: on QM33120 – this has the opposite function. Setting this bit will NOT run PGF calibration on wakeup. Thus the API clears this bit if it is set, as most host applications need the PGF calibration to run and there is no harm in running this even for TX only applications, i.e. it will not speed up wake -up time.**
DWT_GOTORX	0x0200	On wake-up, go to the RX state via IDLE.
DWT_GOTOIDLE	0x0100	On wake-up, go to the IDLE state with PLL calibration.
DWT_SEL_GEAR	0x0040 | 0x0080	On wake-up, select which gear table to load.
DWT_LOADGEAR	0x0020	On wake-up, load the gear table specified by DWT_SEL_GEAR.
DWT_LOADLDO	0x0010	On wake-up, load the LDO tune codes from OTP.
DWT_LOADDGC	0x0008	On wake-up, populate the DGC table from settings in OTP.
DWT_LOADBIAS	0x0004	On wake-up, load the bias settings from OTP.
DWT_RUNSAR	0x0002	On Wake-up run the (temperature and voltage) ADC. Setting this bit will cause the automatic initiation of temperature and input battery voltage measurements when the IC wakes from DEEPSLEEP or SLEEP states. The sampled temperature value may be accessed using the *dwt_readwakeuptemp()* function and, the sampled battery voltage value may be accessed using the *dwt_readwakeupvbat()* function
DWT_CONFIG	0x0001	Restore saved configurations.

Table 18: Bitmask values for dwt_configuresleep() *wake* bit mask

Event	Bit mask	Description
DWT_SLP_CNT_RPT	0x40	sleep counter loop after expiration.
DWT_PRESERVE_SLP	0x20	The sleep enable (bit 0) will be restored after wakeup.
DWT_WAKE_WK	0x10	Wakeup on chip select, SPICSn, line.
DWT_WAKE_CS	0x8	Wakeup on chip select, SPICSn, line.
DWT_BR_DET	0x4	Enable brownout detector during SLEEP/DEEPSLEEP
DWT_SLEEP	0x2	Device will use DEEPSLEEP mode unless this is set, then it will use SLEEP mode.
DWT_SLP_EN	0x1	This is the sleep enable configuration bit. This needs to be set to enable the SLEEP/DEEPSLEEP functionality.

The DEEPSLEEP state is the lowest power state except for the OFF state. In DEEPSLEEP all internal clocks and LDO are off and the IC consumes approximately 100 nA. To wake the IC from DEEPSLEEP an external pin needs to be activated for the "power-up duration" approximately 300 to 500 µs. This can be either be the SPICSn line pulled low or the WAKEUP line driven high. The duration quoted here is dependent on the frequency of the low power oscillator (enabled as the IC comes out of DEEPSLEEP) which will vary between individual IC and will also vary with changes of battery voltage and different temperatures. To ensure the IC reliably wakes up it is recommended to either apply the wakeup signal until the 500 µs has passed, or to use the SPIRDY event status bit (in Register file: 0x0F – System Event Status Register) to drive the IRQ interrupt output line high to confirm the wake-up. Once the IC has detected a "wake up" it progresses into the WAKEUP state. While in DEEPSLEEP power should not be applied to GPIO, SPICLK or SPIMISO pins as this will cause an increase in leakage current.

There are four mechanisms to awaken the IC:

a) By driving the WAKEUP pin (pin 23) high for a period > 500 µs (as per the Data Sheet [1])
b) Driving SPICSn low for a period > 500 µs. This can also be achieved by an SPI read (of register 0, offset 0) of sufficient length
c) If the IC is sleeping using its own internal sleep counter it will be awoken when the timer expires. This is configured by setting the *wake* parameter to 0x10 (+ 0x1 – to enable sleep).
d) By resetting the device, setting RSTn pin to low.

Example code:

This example shows how to configure the device to enter DEEPSLEEP mode after some event e.g. frame transmission. The mode parameter into the *dwt_configuresleep()* function has value 0x01 which configures QM33120 to load IC configurations. The wake parameter value, 0x29, which enables the sleeping with SPICSn as the wakeup signal, and also sets the preserve sleep bit setting.

```
dwt_configuresleep(0x01, 0x29);   //configure sleep and wake parameters

// then ... later... after some event we can instruct the IC to go into
// DEEPSLEEP mode

dwt_entersleep();                 //go to sleep

/// then ... later ... when we want to wake up the device

dwt_spicswakeup(buffer, len);

// buffer is declared locally and needs to be of length (len) which must be
// sufficiently long keep the SPI CSn pin low for at least 500us this
// depends on SPI speed - see also dwt_spicswakeup() function
```

5.5.4 dwt_entersleep

void dwt_entersleep(int idle_rc);

This function is called to put the device into DEEPSLEEP or SLEEP mode.

NOTE: *dwt_configuresleep()* needs to be called before calling this function to configure the sleep and on wake parameters.

(Before entering DEEPSLEEP, the device should be programmed for TX or RX, then upon "waking up" the TX/RX settings will be preserved and the device can immediately perform the desired action TX/RX see *dwt_configuresleep()*).

Parameters:

Type	Name	Description
int	idle_rc	If this is set to DWT_DW_IDLE_RC, the auto INIT2IDLE bit will be cleared prior to going to sleep. Thus, after wake-up, device will stay in IDLE_RC state.

Return Parameters:

none

Notes:

This function is called to enable (put the device into) DEEPSLEEP mode. The *dwt_configuresleep()* should be called first to configure the sleep/wake parameters. (See code example in the *dwt_configuresleep()* function).

5.5.5 dwt_entersleepaftertx

void dwt_entersleepaftertx (int enable);

The *dwt_entersleepaftertx()* function configures the "enter sleep after transmission completes" bit. If this is set, the device will automatically go to DEEPSLEEP/SLEEP mode after a TX event.

Parameters:

Type	Name	Description
int	enable	If set the "enter DEEPSLEEP/SLEEP after TX" bit will be set, else it will be cleared.

Return Parameters:

none

Notes:

When this mode of operation is enabled the IC will automatically transition into SLEEP or DEEPSLEEP mode (depending on the sleep mode configuration set in *dwt_configuresleep()*) after transmission of a frame has completed so long as there are no unmasked interrupts pending. See *dwt_setinterrupt()* for details of controlling the masking of interrupts.

To be effective *dwt_entersleepaftertx()* function should be called before *dw_starttx()* function and then upon TX event completion the device will enter sleep mode.

Example code:

This example shows how to configure the device to enter DEEP_SLEEP mode after frame transmission.

```
dwt_configuresleep(0x01, 0x25);          //configure the on-wake parameters
                                         //(upload the IC config settings)

dwt_entersleepaftertx(1);                //configure the auto go to sleep
                                         //after TX
```

```
// disable TX interrupts
dwt_setinterrupt(
       DWT_INT_TXFRS_BIT_MASK | \
       DWT_INT_TXPHS_BIT_MASK | \
       DWT_INT_TXPRS_BIT_MASK | \
       DWT_INT_TXFRB_BIT_MASK,
       0, DWT_DISABLE_INT
);

// won't be able to enter sleep if any other unmasked events are pending

dwt_writetxdata(frameLength,DataBufferPtr,0);  // write the frame data at
                                               //offset 0

dwt_writetxfctrl(frameLength,0,0)       // set the frame control register

dwt_starttx(DWT_START_TX_IMMEDIATE);    // send the frame immediately

// when TX completes the IC will go to sleep....then…..later...when we
// want to wake up the device

dwt_spicswakeup(buffer, len);

// buffer is declared locally and needs to be of length (len) which must be
// sufficiently long keep the SPI CSn pin low for at least 500us this
// depends on SPI speed – see also dwt_spicswakeup() function
```

5.5.6 dwt_entersleepafter

void dwt_entersleepafter (int event_mask);

The *dwt_entersleepafter()* function makes the device automatically enter deep sleep or sleep mode after a frame transmission and/or reception.

Parameters:

Type	Name	Description
int	event_mask	Bitmask to go to sleep after: DWT_TX_COMPLETE; to configure the device to enter sleep or deep sleep after TX; DWT_RX_COMPLETE; to configure the device to enter sleep or deep sleep after RX.

Return Parameters:

none

Notes:

The IC will only transition to sleep or deep sleep mode if no interrupt events are active. See *dwt_setinterrupt()* for details of controlling the masking of interrupts.

To be effective the *dwt_entersleepafter()* function should be called before calling the *dw_starttx()* or *dwt_rxenable()* function and then upon TX or RX event completion the device will enter sleep mode.

Example code:

This example shows how to configure the device to enter DEEP_SLEEP mode after frame reception.

```
dwt_configuresleep(0x01, 0x25);          // configure the on-wake parameters
                                         // (upload the IC config settings)

dwt_entersleepafter(DWT_RX_COMPLETE);    // configure the auto go to sleep
                                         // after RX

dwt_setinterrupt(DWT_INT_RX, 0, DWT_DISABLE_INT);     //disable TX interrupt

// won't be able to enter sleep if any other unmasked events are pending

dwt_setrxtimeout(0);

dwt_rxenable(); // Receive a frame and go to sleep after reception
```

5.5.7 dwt_spicswakeup

int dwt_spicswakeup (uint8_t *buff, uint16_t length);

The *dwt_spicswakeup()* function uses an SPI read to wake up the IC from SLEEP or DEEPSLEEP.

Parameters:

Type	Name	Description
uint8_t*	buff	This is the pointer to a buffer where the data from SPI read will be read into.
uint16_t	length	This is the length of the input buffer.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

When the IC is in DEEPSLEEP or SLEEP mode, this function can be used to wake it up, assuming SPICSn has been configured as a wakeup signal in the *dwt_configuresleep()*) call. This is done using an SPI read. The duration of the SPI read, keeping SPICSn low, has to be long enough to provide the low for a period > 500 µs.

See example code below.

Example code:

This example shows how to configure the device to enter DEEPSLEEP mode after some event e.g. frame transmission.

```
dwt_configuresleep(0x01, 0x25);  //configure sleep and wake parameters
```

```
// then ... later....after some event we can instruct the IC to go into
// DEEPSLEEP mode

dwt_entersleep();                //go to sleep

// then ... later ... when we want to wake up the device

dwt_spicswakeup(buffer, len);

// buffer is declared locally and needs to be of length (len) which must be
// sufficient to keep the SPI CSn pin low for at least 500us This depends
// on SPI speed
```

5.5.8 dwt_readwakeuptemp

uint8_t dwt_readwakeuptemp(void);

This function reads the IC temperature sensor value that was sampled during IC wake-up. This should be only used with QM331XX as it does not work on DW3000 – see DW3000 errata.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	The 8-bits are temperature value sampled at wake-up event.

Notes:

This function may be used to read the temperature sensor value that was sampled by the IC on wake up, assuming the DWT_TANDV bit in the mode parameter was set in a call to *dwt_configuresleep()* before entering sleep mode. If the wakeup sampling of the temperature sensor was not enabled then the value returned by *dwt_readwakeuptemp()* will not be valid.

5.5.9 dwt_readwakeupvbat

uint8_t dwt_readwakeupvbat(void);

This function reads the battery voltage sensor value that was sampled during IC wake-up.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	The 8-bits are voltage value sampled at wake-up event.

Notes:

This function may be used to read the battery voltage sensor value that was sampled by the IC on wake up, assuming the DWT_TANDV bit in the mode parameter was set in the call to

dwt_configuresleep() before entering sleep mode. If the wakeup sampling of the battery voltage sensor was not enabled then the value returned by *dwt_readwakeupvbat()* will not be valid.

5.5.10 dwt_wakeup_ic

void dwt_wakeup_ic(void);

This function will wake up the device by toggling the correct IO pin. DW3xxx SPI_CS or WAKEUP pins can be used for this.

Parameters:

none

Return Parameters:

none

Notes:

This function is platform dependent. This is due to the fact that each platform may configure IO pins differently. Please view the source code of this function to see how it can be ported to other platforms.

5.5.11 dwt_ds_en_sleep

void dwt_ds_en_sleep(dwt_host_sleep_en_e host_sleep_en);

With this function, each host can prevent the device going into SLEEP/DEEPSLEEP state. By default, it is possible for either host to place the device into SLEEP/DEEPSLEEP. This may not be desirable, thus a host once it is granted access can set a SLEEP_DISABLE bit in the register to prevent the other host from putting the device to sleep once it gives up its access. This does not exist in DW3000.

Parameters:

Type	Name	Description
dwt_host_sleep_en_e	host_sleep_en	Sets or clears the bit to prevent or allow the device to go to sleep respectively. Valid values are as follows: HOST_EN_SLEEP (0x00): clears the bit allowing the device to go to sleep. HOST_DIS_SLEEP (0x60): sets the bit to prevent the device from going to sleep.

Return Parameters:

None

Notes:

5.6 ISR and callback APIs

5.6.1 dwt_setcallbacks

void dwt_setcallbacks(dwt_cb_t cbTxDone, dwt_cb_t cbRxOk, dwt_cb_t cbRxTo, dwt_cb_t cbRxErr, dwt_cb_t cbSPIErr, dwt_cb_t cbSPIRdy, dwt_cb_t dsSPIavail);

This function is used to configure the TX/RX callback function pointers, and SPI CRC error callback function pointer. These callback functions will be called when TX, RX or SPI error events happen and the *dwt_isr()* is called to handle them (See *dwt_isr()* description below for more details about the events and associated callback functions).

Parameters:

Type	Name	Description
dwt_cb_ t	cbTxDone	Function pointer for the cbTxDone function. See type description below.
dwt_cb _t	cbRxOk	Function pointer for the cbRxOk function. See type description below.
dwt_cb _t	cbRxTo	Function pointer for the cbRxTo function. See type description below.
dwt_cb _t	cbRxErr	Function pointer for the cbRxErr function. See type description below.
dwt_cb_t	cbSPIErr	Function pointer for the cbSPIErr function. See type description below.
dwt_cb_t	cbSPIRdy	Function pointer for the cbSPIRdy function. See type description below.
dwt_cb_t	dsSPIavail	Function pointer for the dsSPIavail function. See type description below.

```
	// Call-back type for all events
	typedef void (*dwt_cb_t)(const dwt_cb_data_t *);

// TX/RX call-back data
typedef struct
{
	uint32_t status;	//initial value of register as ISR is entered
	uint16_t status_hi;	//SYS_STATUS_HI_ID register as read at entry to dwt_isr
	uint16_t datalength;	//length of frame
	uint8_t  rx_flags;	//RX frame flags - see dwt_cb_data_rx_flags_e
	uint8_t  dss_stat;	//Dual SPI status register
	struct dwchip_s *dw	//pointer to local device structure
}dwt_cb_data_t;

typedef enum
{
	DWT_CB_DATA_RX_FLAG_RNG = 0x01,	// Ranging bit
	DWT_CB_DATA_RX_FLAG_ND = 0x02,	// No data mode
	DWT_CB_DATA_RX_FLAG_CIA = 0x04,	// CIA done
	DWT_CB_DATA_RX_FLAG_CER = 0x08,	// CIA error
	DWT_CB_DATA_RX_FLAG_CPER = 0x10,	// STS error
} dwt_cb_data_rx_flags_e;
```

Return Parameters:

none

Notes:

This function is used to set up the TX and RX events call-back functions.

Fields	Description of fields within the *dwt_cb_data_t* structure
status	The *status* parameter holds the initial value of the SYS_STATUS_ID register as read on entry into the ISR.
status_hi	The *status_hi* parameter holds the initial value of the SYS_STATUS_HI_ID register as read on entry into the ISR.
datalength	The *datalength* parameter specifies the length of the received frame. Only valid for RX events and only if not SP3 packet.
rx_flags	The *rx_flags* parameter is a bit field value valid only for received frames. It is interpreted as follows: - Bit 0: 1 if the ranging bit was set for this frame, 0 otherwise. - Bit 1: 1 if no data STS mode (no RX data but timestamps are valid) - Bit 2: CIA done (the RX timestamps and diagnostics are valid) - Bit 3: CIA error (the RX timestamps are not valid) - Bit 4: STS error (the STS status is a non-zero value) - 5-7: Reserved. See *dwt_cb_data_rx_flags_e* above.
dss_stat	This callback is not used for DW3000.
**dw*	The **dw* parameter is a pointer to the local device structure.

For more detailed information on interrupt events and especially for details on which status events trigger each one of the different callback functions, see *dwt_isr()* function description below.

5.6.2 dwt_setinterrupt

void dwt_setinterrupt(uint32_t bitmask_lo, uint32_t bitmask_hi, dwt_INT_options_e INT_options);

This function sets the events which will generate an interrupt. The bit mask parameters may be used to enable or disable single events or multiple events at the same time. Table 19 shows the main events that are typically configured as interrupts:

Parameters:

Type	Name	Description
uint32_t	bitmask_lo	This specifies the events being acted on by this API. See Table 19 for the relevant events.
uint32_t	bitmask_hi	This specifies the additional events being acted on by this API. For more information, please see details of the SYS_ENABLE_HI register in the DW3xxx User Manual [2].
dwt_INT_options_e	INT_options	The operation parameter selects the operation being applied to the selected event bits. This can be: DWT_DISABLE_INT (0) = clear only selected bits (other bits settings unchanged). DWT_ENABLE_INT (1) = set only selected bits (other bits settings unchanged). DWT_ENABLE_INT_ONLY (2) = set only selected bits, force other bits to clear. DWT_ENABLE_INT_DUAL_SPI (3) = set only selected bits (other bits settings unchanged) for dual SPI mode. DWT_ENABLE_INT_ONLY_DUAL_SPI (4) = set only selected bits, force other bits to clear for dual SPI mode.

Return Parameters:

none

Notes:

This function is called to enable/disable events for which to generate interrupts.

For the transmitter, it is generally sufficient to enable the SY_STAT_TFRS event which will trigger when a frame has been sent. For the receiver, it is generally sufficient to enable the good frame reception event (DWT_INT_RFCG) and also any error events which will disable the receiver.

Table 19: bitmask_lo values for control of common event interrupts

Event	Bit mask	Description
DWT_INT_IRQS_BIT_MASK	0x00000001	Interrupt set.
DWT_INT_CP_LOCK_BIT_MASK	0x00000002	PLL locked.
DWT_INT_SPICRCE_BIT_MASK	0x00000004	SPI write CRC error event. This is set when IC detects a mismatch between the 8-bit CRC set by the host at the end of the host SPI write transaction and the CRC calculated by the IC. When SPI CRC mode is enabled this may be used to detect SPI errors. The event bit is always set when SPI CRC mode is disabled, therefore the mask should not be set to avoid this causing interrupts.
DWT_INT_AAT_BIT_MASK	0x00000008	Automatic ACK transmission pending.

Event	Bit mask	Description
DWT_INT_TXFRB_BIT_MASK	0x00000010	Frame transmission begins.
DWT_INT_TXPRS_BIT_MASK	0x00000020	Frame preamble sent.
DWT_INT_TXPHS_BIT_MASK	0x00000040	Frame PHR sent.
DWT_INT_TFRS_BIT_MASK	0x00000080	Transmit Frame Sent: This is set when the transmitter has completed the sending of a frame.
DWT_INT_RXPRD_BIT_MASK	0x00000100	Preamble detected.
DWT_INT_RXSFDD_BIT_MASK	0x00000200	SFD detected.
DWT_INT_CIADONE_BIT_MASK	0x00000400	CIA done.
DWT_INT_RXPHD_BIT_MASK	0x00000800	PHY header detected.
DWT_INT_RPHE_BIT_MASK	0x00001000	Receiver PHY Header Error: Reception completed, Frame Error.
DWT_INT_RXFR_BIT_MASK	0x00002000	Receiver Frame Good: A frame (of any type/mode) has been received and is good.
DWT_INT_RFCG_BIT_MASK	0x00004000	Receiver FCS Good: The CRC check has matched the transmitted CRC, frame should be good.
DWT_INT_RFCE_BIT_MASK	0x00008000	Receiver FCS Error: The CRC check has not matched the transmitted CRC, frame has some error.
DWT_INT_RFSL_BIT_MASK	0x00010000	Receiver Frame Sync Loss: The RX lost signal before frame was received, indicates excessive Reed Solomon decoder errors.
DWT_INT_RFTO_BIT_MASK	0x00020000	Receiver Frame Wait Timeout: The RX_FWTO time period expired without a Frame RX.
DWT_INT_CIAERR_BIT_MASK	0x00040000	CIA error.
DWT_INT_VWARN_BIT_MASK	0x00080000	Brownout event detected.
DWT_INT_RXOVRR_BIT_MASK	0x00100000	RX overrun event when double RX buffer in use.
DWT_INT_RXPTO_BIT_MASK	0x00200000	Preamble detection timeout
DWT_INT_SPIRDY_BIT_MASK	0x00800000	SPI ready flag.
DWT_INT_RCINIT_BIT_MASK	0x01000000	Device has entered IDLE_RC.
DWT_INT_PLL_HILO_BIT_MASK	0x02000000	PLL calibration flag.
DWT_INT_RXSTO_BIT_MASK	0x04000000	SFD timeout.
DWT_INT_HPDWARN_BIT_MASK	0x08000000	Half period warning flag when delayed TX/RX is used.
DWT_INT_ARFE_BIT_MASK	0x20000000	ARFE – frame rejection status.
DWT_INT_CPERR_BIT_MASK	0x10000000	STS quality warning/error.

5.6.3 dwt_setinterrupt_db

void dwt_setinterrupt_db(uint8_t bitmask, dwt_INT_options_e INT_options);

This function sets the events which enables the specified double RX buffer to trigger an interrupt.

This function sets the events which will generate an interrupt. The bit mask parameter may be used to enable or disable single events or multiple events at the same time. Table 20 shows the main events that are typically configured as interrupts:

Parameters:

Type	Name	Description
Uint8_t	bitmask	This specifies the events being acted on by this API. See Table 20 for the relevant events.
dwt_INT_options_e	INT_options	The operation parameter selects the operation being applied to the selected event bits. This can be: DWT_DISABLE_INT (0) = clear only selected bits (other bits settings unchanged). DWT_ENABLE_INT (1) = set only selected bits (other bits settings unchanged). DWT_ENABLE_INT_ONLY (2) = set only selected bits, force other bits to clear. DWT_ENABLE_INT_DUAL_SPI (3) = set only selected bits (other bits settings unchanged) for dual SPI mode. DWT_ENABLE_INT_ONLY_DUAL_SPI (4) = set only selected bits, force other bits to clear for dual SPI mode.

Return Parameters:

none

Notes:

This function is only available in QM331XX devices.

This function is called to enable/disable events for which to generate interrupts.

Table 20: bitmask values for control of RX buffer event interrupts

Event	Bit mask	Description
DWT_DB_INT_RXFCG0_EN	0x01	Frame CC good in RX buffer 0
DWT_DB_INT_RXFR0_EN	0x02	Frame ready in RX buffer 0
DWT_DB_INT_RXCIADONE0_EN	0x04	CIA done for frame in RX buffer 0
DWT_DB_INT_CPERR0_EN	0x08	STS quality warning/error in RX buffer 0
DWT_DB_INT_RXFCG1_EN	0x10	Frame CC good in RX buffer 1

Event	Bit mask	Description
DWT_DB_INT_RXFR1_EN	0x20	Frame ready in RX buffer 1
DWT_DB_INT_RXCIADONE1_EN	0x40	CIA done for frame in RX buffer 1
DWT_DB_INT_CPERR1_EN	0x80	STS quality warning/error in RX buffer 1

5.6.4 dwt_checkirq

uint8_t dwt_checkirq(void);

This API function checks the interrupt line status.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	1 if the IC interrupt line is active (IRQS bit in STATUS register is set), 0 otherwise.

Notes:

This function is typically intended to be used in a PC based system using (Cheetah or ARM) USB to SPI converter, where there can be no interrupts. In this case we can operate in a polled mode of operation by checking this function periodically and calling *dwt_isr()* if it returns 1.

5.6.5 dwt_isr

void dwt_isr(void);

This function processes device events, (e.g. frame reception, transmission). It is intended that this function be called as a result of an interrupt from the IC – the mechanism by which this is achieved is target specific. Where interrupts are not supported this function can be called from a simple runtime loop to poll the status register and take the appropriate action, but this approach is not as efficient and may result in reduced performance depending on system characteristics.

The *dwt_isr()* function makes use of call-back functions in the application to indicate that received data is available to the upper layers (application) or to indicate when frame transmission has completed. The *dwt_setcallbacks()* API function is used to configure the call back functions.

The *dwt_isr()* function reads the status register and recognises the following events:

Table 21: List of events handled by the *dwt_isr()* function and signalled in call-backs

Event	Corresponding status register event flags	Comments
Reception of a good frame (cbRxOk callback)	RXFCG	This means that a frame with a good CRC has been received and that the RX data and the frame receive time stamp can be read. Frame length and frame control information are reported through “datalength” and “fctrl” fields of the *dwt_cb_data_t* structure. The value of the Ranging bit (from the PHY header), is reported through RNG bit in the rx_flags field of the *dwt_cb_data_t* structure. When automatic acknowledgement is enabled (via the *dwt_enableautoack()* API function), if a frame is received with the ACK request bit set then the AAT bit will be set in the “status” field of the *dwt_cb_data_t* structure, indicating that an ACK is being sent (or has been sent).
Reception of good STS Mode 3 (no data) packet (cbRxOk callback)	RXFR	Since the STS mode 3 (no data) packet contains no PHY payload, the way in which a ‘good’ packet is checked for is different. The required event for this callback is a simplified version of the callback described above (for a good frame). There is no CRC and no RX data to be checked in this case. Nor is the any frame length or control to cater for. All that is required is the RXFR bit set and STS mode 3 is configured.
Reception timeout (cbRxTo callback)	RXRFTO/RXPTO	These events indicate that a timeout occurred while waiting for an incoming frame. If needed, the “status” field of the *dwt_cb_data_t* structure can be examined to distinguish between these events.
Reception error (cbRxErr callback)	RXRXPHE/RXSFDTO/ RXRFSL/RXRFCE/ LDEERR/AFFREJ/ LCSSERR	This means that an error event occurred while receiving a frame. If needed, the “status” field of *dwt_cb_data_t* structure can be examined to determine which event caused the interrupt.
Transmission of a frame completed (cbTxDone callback)	TXFRS	This means that the transmission of a frame is complete and that the transmit time stamp can be read.
SPI write CRC error detected (cbSPIErr callback)	SPICRCERR	This means that the CRC byte written by the host as the last byte of SPI write transaction did not match the CRC generated by the IC on the header and data bytes. This will only be generated when IC is using SPI CRC mode.
Device powered on or wake-up (cbSPIRdy callback)	SPIRDY/ RCINIT	This callback is used to check if the device has powered up or has woken up from a sleep state. It checks for the SPI to be ready and that the device has gone from wakeup to the RXINIT state.

When an event is recognised and processed the status register bit is cleared to clear the event interrupt. Figure 7 shows the *dwt_isr()* function flow diagram.

Parameters:

none

Return Parameters:

none

Notes:

The *dwt_isr()* function should be called from the microprocessor's interrupt handler that is used to process the IC interrupt.

It is recommended to read the User Manual [2], especially chapters 3, 4, and 5 to become familiar with IC events and their operation.

In addition, if the microprocessor is not fast enough and two events are set in the status register, the order in which they are processed is as shown in Figure 7. This may not be the order in which they were triggered.

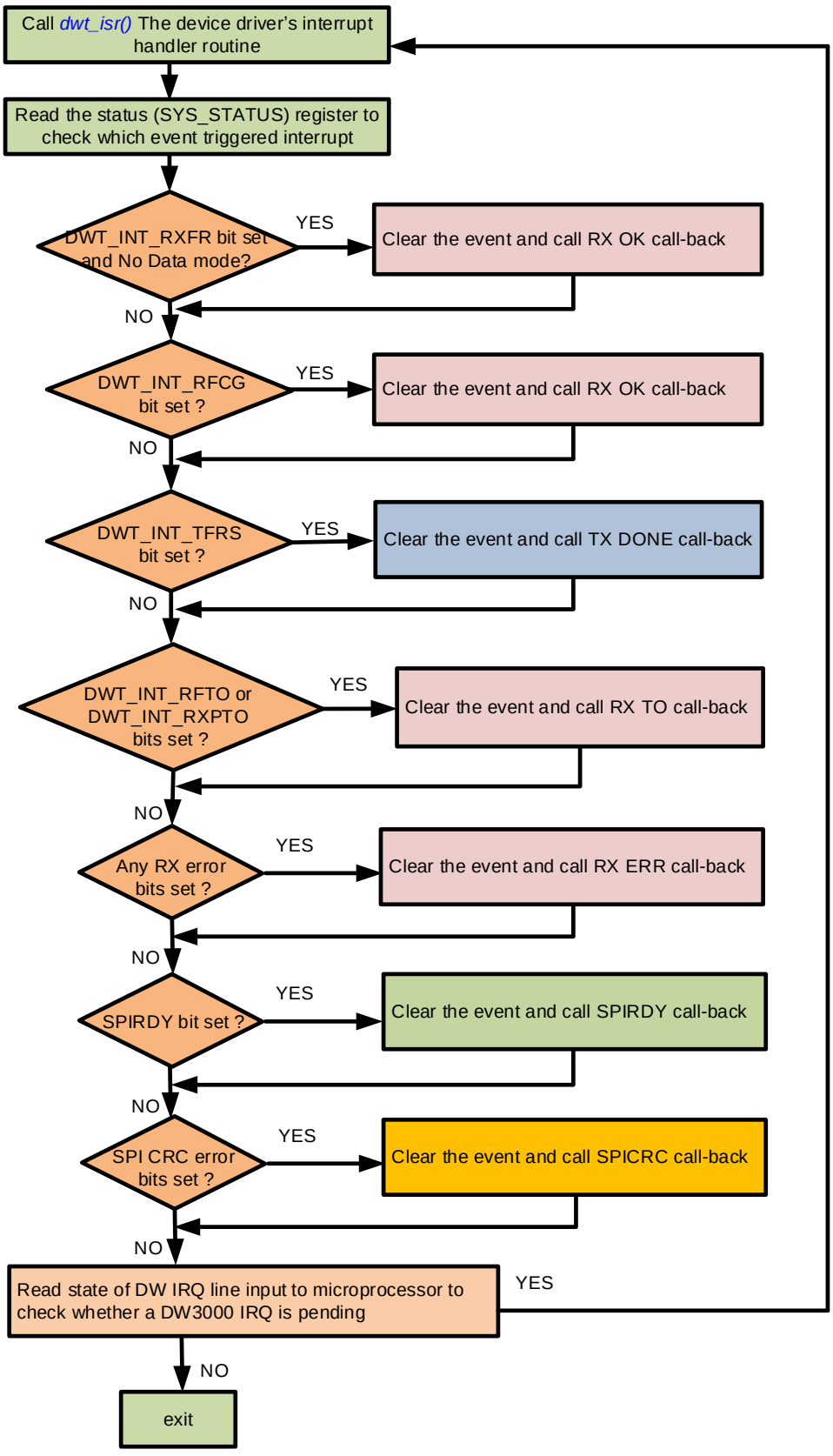


Figure 7: Interrupt handling

5.6.6 dwt_writesysstatuslo

void dwt_writesysstatuslo(uint32_t mask);

This function writes a value to the system status register (lower). Host can do this to clear events in the status register and any associated interrupts.

Parameters:

Type	Name	Description
uint32_t	mask	Value to send to the system status register (lower 32-bits),

Return Parameters:

none

5.6.7 dwt_writesysstatushi

void dwt_writesysstatuslo(uint32_t mask);

This function writes a value to the system status register (higher). Host can do this to clear events in the status register and any associated interrupts.

Parameters:

Type	Name	Description
uint32_t	mask	Value to send to the system status register (higher bits),

Return Parameters:

None

Notes:

Be aware that the size of this register varies per device

DW3000 devices only require a 16-bit mask value typecast to 32-bit register

QM33120 devices require 32-bit mask value.

5.6.8 dwt_readsysstatuslo

uint32_t dwt_readsysstatuslo(void);

This function reads the current value of the system status register (lower 32 bits).

Parameters:

None

Return Parameters:

Type	Description
uint32_t	A uint32_t value containing the value of the system status register (lower 32 bits)

5.6.9 dwt_readsysstatushi

uint32_t dwt_readsysstatushi(void);

This function reads the current value of the system status register (higher bits).

Parameters:

None

Return Parameters:

Type	Description
uint32_t	A uint32_t value containing the value of the system status register (higher bits)

Notes:

Be aware that the size of this register varies per device

DW3000 devices only require a 16-bit mask value typecast to 32-bit register

QM33120 devices require 32-bit mask value.

5.6.10 dwt_writerdbstatus

void dwt_writerdbstatus (uint8_t mask);

This function writes a value to the receiver double buffer status register. Host can do this to clear events in the status register and any associated interrupts.

Parameters:

Type	Name	Description
uint8_t	mask	Value to write to the register.

Return Parameters:

None

5.6.11 dwt_readrdbstatus

uint8_t dwt_readrdbstatus(void);

This function reads the current value of the receiver double buffer register.

Parameters:

None

Return Parameters:

Type	Description
uint8_t	A uint8_t value containing the value of the receiver double buffer register

5.7 MAC configuration APIs

5.7.1 dwt_setpanid

void dwt_setpanid(uint16_t panID);

This function sets the PAN ID value. These are typically assigned by the PAN coordinator when a node joins a network. This value is only used by the IC for frame filtering. See the *dwt_configureframefilter()* function.

Parameters:

Type	Name	Description
uint16_t	panID	This is the PAN ID.

Return Parameters:

none

Notes:

This function can be called to set device's PANID for frame filtering use, it does not need to be set if frame filtering is not being used. Insertion of PAN ID in the TX frames is the responsibility of the upper layers calling the *dwt_writetxdata()* function.

5.7.2 dwt_setaddress16

void dwt_setaddress16(uint16_t shortAddress);

This function sets the 16-bit short address values. These are typically assigned by the PAN coordinator when a node joins a network. This value is only used by the IC for frame filtering. See the *dwt_configureframefilter()* function.

Parameters:

Type	Name	Description
uint16_t	shortAddress	This is the 16-bit address to set.

Return Parameters:

none

Notes:

This function is called to set device's short (16-bit) address, it does not need to be set if frame filtering is not being used. Insertion of short (16-bit) address, in the TX frames is the responsibility of the upper layers calling the *dwt_writetxdata()* function.

5.7.3 dwt_seteui

void dwt_seteui (uint8_t* eui64);

The *dwt_seteui()* function sets the 64-bit address.

Parameters:

Type	Name	Description
uint8_t*	eui64	This is a pointer to the 64-bit address to set, arranged as 8 unsigned bytes. The low order byte comes first.

Return Parameters:

none

Notes:

This function may be called to set a long (64-bit) address used for address filtering. If address filtering is not being used, then this register does not need to be set.

It is possible for a 64-bit address to be programmed into the IC's one-time programmable memory (OTP memory) during customers' manufacturing processes and automatically loaded into this register on power-on reset or wake-up from sleep. *dwt_seteui()* may be used subsequently to change the value automatically loaded.

5.7.4 dwt_geteui

void dwt_geteui (uint8_t* eui64);

The *dwt_geteui()* function gets the programmed 64-bit EUI value.

Parameters:

Type	Name	Description
uint8_t*	eui64	This is a pointer to the 64-bit address to read, arranged as 8 unsigned bytes. The low order byte comes first.

Return Parameters:

none

Notes:

This function may be called to get the EUI value. The value will be 0xFFFFFFFF00000000 if it has not been programmed into OTP memory or has not been set by a call to *dwt_seteui()* function.

It is possible for a 64-bit address to be programmed into the IC's one-time programmable memory (OTP memory) during customers' manufacturing processes and automatically loaded into this register on power-on reset or wake-up from sleep. *dwt_seteui()* may be used subsequently to change the value automatically loaded.

5.7.5 dwt_configureframefilter

void dwt_configureframefilter(uint16_t enabletype, uint16_t filtermode) ;

This *dwt_configureframefilter()* function enables frame filtering according to the *mask* parameter.

Parameters:

Type	Name	Description
uint16_t	enabletype	This enables 802.15.4 frame filter (DWT_FF_ENABLE_802_15_4) or disables the frame filter (DWT_FF_DISABLE)
uint16_t	filtermode	This enables a particular frame filter options, see Table 22.

Return Parameters:

none

Notes:

This function is used to enable frame filtering, the device address and pan ID should be configured beforehand.

Table 22: Bitmask values for frame filtering enabling/disabling

Definition	Value	Description
DWT_FF_BEACON_EN	0x001	Beacon frames allowed
DWT_FF_DATA_EN	0x002	Data frames allowed
DWT_FF_ACK_EN	0x004	ACK frames allowed
DWT_FF_MAC_EN	0x008	MAC command frames allowed
DWT_FF_RSVD_EN	0x010	Reserved frame types allowed
DWT_FF_MULTI_EN	0x020	Multipurpose frame types allowed
DWT_FF_FRAG_EN	0x040	Fragmented frame types allowed
DWT_FF_EXTEND_EN	0x080	Extended frame types allowed
DWT_FF_COORD_EN	0x100	behave as coordinator (can receive frames with no destination address (PAN ID has to match))
DWT_FF_IMPBRCAST_EN	0200	MAC implicit Broadcast allowed.

5.7.6 dwt_configure_le_address

void dwt_configure_le_address(uint16_t addr, int leIndex);

This function is used to write a 16 bit address to a desired Low-Energy device (LE) address. For frame pending to function when the correct bits are set in the frame filtering configuration via the *dwt_configureframefilter()*. See *dwt_configureframefilter()* for more details.

Parameters:

Type	Name	Description
uint16_t	addr	The uint16_t address value to be written to the selected LE register
int	leIndex	The Low-Energy device (LE) address to write to. There are four options for this index: 0, 1, 2 & 3. The index the LE_PEND_01 register with offset 0 (LE_ADDR0), LE_PEND_01 register with offset 16 (LE_ADDR1), LE_PEND_23 register with offset 0 (LE_ADDR2) and LE_PEND_23 with offset 16 (LE_ADDR3) respectively.

Return Parameters:

none

5.7.7 dwt_enableautoack

void dwt_enableautoack(uint8_t responseDelayTime, int enable);

This function enables automatic ACK to be automatically sent when a frame with ACK request is received. The ACK frame is sent after a specified responseDelayTime (in preamble symbols, max is 255). It can also be enabled or disabled depending on how the *enable* parameter is set.

Parameters:

Type	Name	Description
uint8_t	responseDelayTime	The delay between the ACK request reception and ACK transmission.
int	enable	This argument enables the Auto-ACK feature with '1' and disables it with '0'.

Return Parameters:

none

Notes:

This *dwt_enableautoack()* function is used to enable the automatic ACK response. It is recommended that the *responseDelayTime* is set as low as possible consistent with the ability of unit requesting the ACK to turn around and be ready to receive the response. If the host system is using the DWT_RESPONSE_EXPECTED mode (with *rxDelayTime* in *dwt_setrxaftertxdelay()* function set to 0) in the *dwt_starttx()* function then the *responseDelayTime* can be set to 3 symbols (3 µs) without loss of preamble symbols in the receiver awaiting the ACK.

5.7.8 dwt_getframelength

uint16_t dwt_getframelength(void);

This function will read the length of the last received frame. This function presumes that a good frame or packet has been received.

Parameters:

None

Return Parameters:

Type	Description
uint16_t	A uint16_t value with the number of octets in the received frame.

5.8 Temperate and voltage reading APIs

5.8.1 dwt_readtempvbat

uint16_t dwt_readtempvbat(void);

This function reads the temperature and battery voltage. Note: the DW3XXX needs to be in IDLE_PLL mode or the call will return 0.

Parameters:

none

Return Parameters:

Type	Description
uint16_t	The low 8-bits are voltage value, and the high 8-bits are temperature value.

Notes:

This function can be called to read the battery voltage and temperature. It enables the IC's internal convertors to sample the current IC temperature and battery. Must be called when DW3xxx is in IDLE.

5.8.2 dwt_convertrawtemperature

float dwt_convertrawtemperature(uint8_t raw_temp);

This function takes a raw temperature value and applies the conversion factor to return a temperature in degrees.

Parameters:

Type	Name	Description
uint8_t	raw_temp	Raw 8-bit temperature value, as returned by *dwt_readtempvbat()*

Return Parameters:

Type	Description
float	The temperature value in degrees.

Notes:

This function is called to convert the raw IC temperature to degrees, the conversion is given by:

Temperature (°C) = ((SAR_LTEMP – OTP_READ(Vtemp @ 23°C)) x 1.05) + 22

5.8.3 dwt_convertrawvoltage

float dwt_convertrawvoltage (uint8_t raw_voltatge);

This function takes a raw voltage value and applies the conversion factor to return a voltage in volts.

Parameters:

Type	Name	Description
uint8_t	raw_voltage	Raw 8-bit voltage value, as returned by *dwt_readtempvbat()*

Return Parameters:

Type	Description
float	The voltage value in volts.

Notes:

This function is called to convert the raw IC voltage to volts, the conversion is given by:

Voltage (V) = ((SAR_LVBAT – OTP_READ(Vmeas @ 3.0 V)) *0.4 * 16/ 255) + 3.0

5.9 OTP and AON access APIs

5.9.1 dwt_otpread

void dwt_otpread(uint32_t address, uint32_t *array, uint8_t length);

This function is used to read a number (given by length) of 32-bit values from the IC's OTP memory, starting at given address. The given array will contain the read values.

Parameters:

Type	Name	Description
uint32_t	address	This is starting address in the OTP memory from which to read
uint16_t*	array	This is the 32-bit array that will hold the read values. It should be of at least *length* 32-bit words long.
uint8_t	length	The number of values to read

Return Parameters:

none

Notes:

None

5.9.2 dwt_otpwriteandverify

int dwt_otpwriteandverify(uint32_t value, uint16_t address);

This function is used to program 32-bit value into OTP memory.

Parameters:

Type	Name	Description
uint32_t	value	this is the 32-bit value to be programmed into OTP memory
uint16_t	address	this is the 16-bit OTP memory address into which the 32-bit value is programmed

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

The IC has a small amount of one-time-programmable (OTP) memory intended for device specific configuration or calibration data. Some areas of the OTP memory are used to save device calibration values determined during IC testing, while other OTP memory locations are intended to be set by the customer during module manufacture and test.

Programming OTP memory is a one-time only activity, any values programmed in error cannot be corrected. Also, please take care when programming OTP memory to only write to the designated areas – programming elsewhere may permanently damage the IC's ability to function normally.

The OTP memory locations are as defined in Table 23. The OTP memory locations are each 32-bits wide; OTP addresses are word addresses, so each increment of address specifies a different 32-bit word.

Table 23: OTP memory map

Address	Size (Used Bytes)	Byte [3]	Byte [2]	Byte [1]	Byte [0]	Programmed By
0x00	4	64 bit EUID				Customer
0x01	4					
0x02	4	Alternative 64bit EUID (Selected via reg/SR register)				Customer
0x03	4					
0x04	4	LDOTUNE_CAL				Prod Test
0x05	4					
0x06	4	{"0001,0000,0001", "CHIP ID 5 nibbles (20 bits)"}				Prod Test
0x07	4	{"0001" , "LOT ID – 7 nibbles (28bits)"}				Prod Test
0x08	4	-	Vbat @ 3.0 V [23:16]	Vbat @ 3.62 V [15:8]	Vbat @ 1.62 V [7:0]	Prod Test
0x09	2				Temp @ 22 °C +/- 2 °C [7:0]	Prod Test
0x0A	0	BIASTUNE_CAL				Prod Test
0x0B	4	Antenna Delay – RFLoop				Prod Test
0x0C	4	AoA Iso CH9 RF2->RF1	AoA Iso CH9 RF1->RF2	AoA Iso CH5 RF2 -> RF1	AoA Iso CH5 RF1->RF2	Prod Test
0x0D	0	W.S. Lot ID [3]	W.S. Lot ID [2]	W.S. Lot ID [1]	W.S. Lot ID [0]	Prod Test
0x0E	0			W.S. Lot ID [5]	W.S. Lot ID [4]	Prod Test
0x0F	0		W.S. Wafer Number	W.S. Y Loc	W.S. X Loc	Prod Test
0x10	4					Customer
0x11	4					Customer
0x12	4					Customer
0x13	4					Customer

0x14	4					Customer
0x15	4					Customer
0x16	4					Customer
0x17	4					Customer
0x18	4					Customer
0x19	4					Customer
0x1A	4					Customer
0x1B	4					Customer
0x1C	4					Customer
0x1D	4					Customer
0x1E	1				XTAL_Trim[6:0]	Customer
0x1F	1				OTP Revision	Customer
0x20	4	RX_TUNE_CAL: DGC_CFG0				Prod Test
0x21	4	RX_TUNE_CAL: DGC_CFG1				Prod Test
0x22	4	RX_TUNE_CAL: DGC_CFG2				Prod Test
0x23	4	RX_TUNE_CAL: DGC_CFG3				Prod Test
0x24	4	RX_TUNE_CAL: DGC_CFG4				Prod Test
0x25	4	RX_TUNE_CAL: DGC_CFG5				Prod Test
0x26	4	RX_TUNE_CAL: DGC_CFG6				Prod Test
0x27	4	RX_TUNE_CAL: DGC_LUT_0 – CH5				Prod Test
0x28	4	RX_TUNE_CAL: DGC_LUT_1 – CH5				Prod Test
0x29	4	RX_TUNE_CAL: DGC_LUT_2 – CH5				Prod Test
0x2A	4	RX_TUNE_CAL: DGC_LUT_3 – CH5				Prod Test
0x2B	4	RX_TUNE_CAL: DGC_LUT_4 – CH5				Prod Test
0x2C	4	RX_TUNE_CAL: DGC_LUT_5 – CH5				Prod Test
0x2D	4	RX_TUNE_CAL: DGC_LUT_6 – CH5				Prod Test
0x2E	4	RX_TUNE_CAL: DGC_LUT_0 – CH9				Prod Test
0x2F	4	RX_TUNE_CAL: DGC_LUT_1 – CH9				Prod Test
0x30	4	RX_TUNE_CAL: DGC_LUT_2 – CH9				Prod Test
0x31	4	RX_TUNE_CAL: DGC_LUT_3 – CH9				Prod Test
0x32	4	RX_TUNE_CAL: DGC_LUT_4 – CH9				Prod Test
0x33	4	RX_TUNE_CAL: DGC_LUT_5 – CH9				Prod Test
0x34	4	RX_TUNE_CAL: DGC_LUT_6 – CH9				Prod Test
0x35	4	PLL_LOCK_CODE				Prod Test
0x36 – 0x5F		UNALLOCATED				Customer
0x60	1	QSR Register (Special function register)				Reserved
0x61					Q_RR Register [7:0]	Reserved
0x62 – 0x77	4	UNALLOCATED				Customer
0x78	4	AES_KEY[127:96] (big endian order)				Customer
0x79	4	AES_KEY[95:64] (big endian order)				Customer
0x7A	4	AES_KEY[63:32] (big endian order)				Customer
0x7B	4	AES_KEY[31:0] (big endian order)				Customer

0x7C	4	AES_KEY[255:224] (big endian order)	Customer
0x7D	4	AES_KEY[223:192] (big endian order)	Customer
0x7E	4	AES_KEY[191:160] (big endian order)	Customer
0x7F	4	AES_KEY[159:128] (big endian order)	Customer

The QSR ("Special function register") is a 32-bit segment of OTP that is directly readable via the register interface upon power up. To program the SR register follow the normal OTP programming method but set the OTP address to 0x60. As this is part of OTP boot sequence the new value will be present in the QSR register following the next boot up sequence.
For more information on OTP memory programming please consult the User Manual [2] **and Data Sheet [1].**

5.9.3 dwt_otpwrite

int dwt_otpwrite(uint32_t value, uint16_t address);

This function is used to program 32-bit value into OTP memory. It will not validate/check the written value (*dwt_otpwriteandverify()* checks if the value has been saved correctly).

Parameters:

Type	Name	Description
uint32_t	value	this is the 32-bit value to be programmed into OTP memory
uint16_t	address	this is the 16-bit OTP memory address into which the 32-bit value is programmed

Return Parameters:

Type	Description
int	Return value is always DWT_SUCCESS = 0

Notes:

5.9.4 dwt_aon_read

uint8_t dwt_aon_read (uint16_t aon_address);

The *dwt_aon_read()* function reads from the AON memory. It returns an 8-bit read from the given AON memory address.

Parameters:

Type	Name	Description
uint16_t	aon_address	This is the address of the memory location to read.

Return Parameters:

Type	Description
uint8_t	8-bit value of the AON memory address given.

Notes:

This function allows the user to read addresses from AON memory. Please see the implementation of *dwt_aon_read()* for an example of how this function is utilised.

5.9.5 dwt_aon_write

void dwt_aon_read (uint16_t aon_address , uint8_t aon_write_data);

The *dwt_aon_write()* function writes to the AON memory given a 16-bit address and 8-bit value. It has no return values.

Parameters:

Type	Name	Description
uint16_t	aon_address	This is the address of the memory location to write to.
uint8_t	aon_write_data	This is the 8-bit value that is written to the specified AON address.

Return Parameters:

none

Notes:

This function allows the user to write a value to AON memory. Please see the implementation of *dwt_aon_write()* for an example of how this function is utilised.

5.9.6 dwt_clearaonconfig

void dwt_clearaonconfig(void);

This function allows the user to clear the AON configuration. This will clear any previously programmed configurations such as AON on-wake / wake-up configurations. Default values will be restored.

Parameters:

none

Return Parameters:

none

Notes:

When this function is called, anything set in the AON_DIG_CFG register will be cleared. The same applies for the ANA_CFG register. The default configuration will be loaded into the AON_CTRL register also.

5.10 TX test APIs

5.10.1 dwt_setfinegraintxseq

void dwt_setfinegraintxseq(int enable);

This is used to activate/deactivate fine grain TX sequencing. In some applications/use cases the fine grain TX sequencing needs to be disabled, e.g. continuous wave mode or when driving an external PA. Please refer to [2] for more details about those modes.

Parameters:

Type	Name	Description
int	enable	Set to 1 to enable fine grain TX sequencing, 0 to disable it.

Return Parameters:

none

5.10.2 dwt_setxtaltrim

void dwt_setxtaltrim(uint8_t value);

This function writes the crystal trim value parameter into the IC crystal trimming register.

Parameters:

Type	Name	Description
uint8_t	value	Crystal trim value (in range 0x0 to 0x3F, 63 steps (~1.5ppm per step).

Return Parameters:

none

Notes:

This function can be called any time to set the crystal trim register value. This is used to fine tune and adjust the XTAL frequency. Better long-range performance may be achieved when crystals are more closely matched. Crystal trimming may allow this without using expensive TCXO devices. Please consult the User Manual [2], Data Sheet [1] and application notes available on www.decawave.com.

5.10.3 dwt_configcwmode

void dwt_configcwmode(void);

This function configures the device to transmit a Continuous Wave (CW) at a specified channel frequency. This may be of use as part of crystal trimming procedure. Please consult with Decawave's applications support team for details of crystal trimming procedures and considerations.

Parameters:

none

Return Parameters:

none

Notes:

Example code of how to use this function in conjunction with *dwt_setxtaltrim()* function is given by the Example 04a: continuous wave mode sample example in the API package [5]

5.10.4 dwt_configcontinuousframemode

void dwt_configcontinuousframemode(uint32_t framerepetitionrate);

This function configures continuous frame mode. This facilitates measurement of the power in the transmitted spectrum.

Parameters:

Type	Name	Description
uint32_t	framerepetitionrate	This is a 32-bit value that is used to set the interval between transmissions. The minimum value is 4. The units are approximately 8 ns. (or more precisely 512/(499.2e6*128) seconds)).

Return Parameters:

none

Notes:

This function is used to configure continuous frame (transmit power spectrum test) mode, used in TX power spectrum measurements. This test mode is provided to help support regulatory approvals spectral testing. Please consult with Decawave's applications support team for details of regulatory approvals considerations. The *dwt_configcontinuousframemode()* function enables a repeating transmission of the data from the transmit buffer. To use this test mode, the operating channel, preamble code, data length, offset, etc. should all be set-up as if for a normal transmission.

The *framerepetitionrate* parameter value is programmed in units of one quarter of the 499.2 MHz fundamental frequency, (~ 8 ns). To send one frame per millisecond, a value of 124800 or 0x0001E780 should be set. A value <2 will not work properly, and a time value less than the frame length will cause the frames to be sent back-to-back without any pause.

We expect there to be two use cases for the *dwt_configcontinuousframemode()* function:

(a) Testing to figure out the TX power/pulse width to meet the regulations.

(b) In the approvals house to enable the spectral test.

To end the test and return to normal operation the device can be rest with *dwt_softreset()* function.

Please see Example 04b: continuous frame mode, of the API package [5] for an example of the use of this API function.

5.10.5 dwt_readpgdelay

uint8_t dwt_readpgdelay(void);

This is used to read the pulse generator delay value of the TX signal.

Parameters:

none

Return Parameters:

Type	Description
uint8_t	Pulse generator delay read from TX_CTRL_HI register.

Notes:

5.10.6 dwt_repeated_cw

void dwt_repeated_cw(int cw_enable, int cw_mode_config);

This function will enable a repeated continuous waveform on the selected device given a pulse generator channel and pulse generator coefficient.

Parameters:

Type	Name	Description
int	cw_enable	CW mode enable
int	cw_mode_config	CW configuration mode

Return Parameters:

none

Notes:

5.10.7 dwt_repeated_frames

void dwt_repeated_frames(uint32_t framerepetitionrate);

This function enables repeated frames to be generated given a frame repetition rate.

Parameters:

Type	Name	Description
uint32_t	framerepetitionrate	Value specifying the rate at which frames with be repeated. If the value is less than the frame duration, the frames are sent back-to-back.

Return Parameters:

none

Notes:

5.10.8 dwt_stop_repeated_frames

uint16_t dwt_stop_repeated_frames(void);

This function disables repeated frames from being generated.

Parameters:

none

Return Parameters:

none

5.10.9 dwt_disablecontinuousframemode

void dwt_disablecontinuousframemode(void)

This function stops the continuous TX frame mode.

Parameters:

none

Return Parameters:

none

5.10.10 dwt_calcbandwidthadj

uint8_t dwt_calcbandwidthadj(uint16_t target_count);

This function runs a bandwidth compensation algorithm that adjusts the bandwidth of the output spectrum to correct for the effects of different temperatures. This ensures that the bandwidth is constant at any temperature. The target count parameter is a reference value taken at a known temperature for a known good bandwidth using the *dwt_calcpgcount()* API call, which relates directly to the bandwidth of the spectrum.

Parameters:

Type	Name	Description

uint16_t	target_count	This is a 16-bit value that is used by the IC to calculate a bandwidth adjust value

Return Parameters:

Type	Description
uint8_t	This is an 8-bit value that represents a pulse generator delay (PG_DELAY) value

Notes:

See the app note in [4] for more details. The return value is automatically set into DW3xxx PG delay register, but it is also returned here so host knows what it was set to.

5.10.11 dwt_calcpgcount

uint16_t dwt_calcpgcount(uint8_t pgdly);

This function returns a pulse generator count value that is used as a reference for bandwidth compensation over temperature. The pulse generator delay value that is passed in should be the current bandwidth setting.

Parameters:

Type	Name	Description
uint8_t	pgdly	This is an 8-bit value representing the current pulse generator delay for the current bandwidth setting

Return Parameters:

Type	Description
uint16_t	This is a 16-bit value that represents the pulse generator count value for the current pulse generator delay. It is directly related to the bandwidth.

Notes:

See the app note in [4] for more details. The return value should be stored as a reference to be used with *dwt_configuretxrf()*.

5.11 AES APIs

5.11.1 dwt_configure_aes

void dwt_configure_aes(const dwt_aes_config_t *pCfg)

This function initializes the AES_CFG register that is responsible for the tag size, key size, etc.

Parameters:

Type	Name	Description
dwt_aes_config_t	pCfg	This struct contains all the needed fields for initialization of this reg.

```
typedef struct {
    dwt_aes_otp_sel_key_block_e aes_otp_sel_key_block; //!< Select OTP
                                                       //key, first 128 or 2nd 128
                                                       //bits
    dwt_aes_key_otp_type_e  aes_key_otp_type;
    dwt_aes_core_type_e     aes_core_type;  //!< Core type
    dwt_mic_size_e          mic;            //!< Message integrity code
                                            //size
    dwt_aes_key_src_e       key_src;        //!< Location of the key:
                                            //either as programmed in
                                            //registers(128 bit) or in the
                                            //RAM
    dwt_aes_key_load_e      key_load;       //!< Loads key from RAM
    uint8_t                 key_addr;       //!< Address offset of AES key
                                            //in AES key RAM
    dwt_aes_key_size_e      key_size;       //!< AES key length
                                            //configuration corresponding
                                            //to AES_KEY_128/192/256bit
    dwt_aes_mode_e          mode;           //!< Operation type
                                            //encrypt/decrypt
} dwt_aes_config_t ;
```

Return Parameters:

none

Notes:

Further information on the structures that make up the dwt_aes_config_t structure can be found within the source code.

5.11.2 dwt_set_keyreg_128

void dwt_set_keyreg_128(const dwt_aes_key_t *key);

This function sets the AES key – 128 bits. It updates the AES_KEY0 – AES_KEY3.

Parameters:

Type	Name	Description
dwt_aes_key_t	key	This struct contains all the keys values that is needed for the initialization.

Return Parameters:

none

5.11.3 dwt_do_aes

int8_t dwt_do_aes(dwt_aes_job_t *job, dwt_aes_core_type_e core_type)

This function responsible for data encryption/decryption, filling the relevant buffer (in case of decryption) or updating the TX buffer with encrypted data (in case of encryption). The function also checks for errors and updates the nonce.

Parameters:

Type	Name	Description
dwt_aes_job_t	job	This is a struct pointer that contains the info and buffers for encryption/decryption, header, payload, nonce, …
dwt_aes_core_type_e	core_type	This option refers to the AES core type used by the IC. The options are either GCM core type (0) or CCM core type (1).

Return Parameters:

Type	Description
int8_t	Negative value on error or AES_STS reg value. AES_STS reg value will be checked in the calling function

5.11.4 dwt_mic_size_from_bytes

dwt_mic_size_e dwt_mic_size_from_bytes(uint8_t mic_size_in_bytes)

This function gets mic size in bytes and returns the MIC size to fit into AES_CFG-AES_TAG_SIZE reg.

Parameters:

Type	Name	Description
uint8_t	mic_size_in_bytes	Mic size in bytes.

Return Parameters:

dwt_mic_size_e – Enum that contains the right value for the AES reg.

5.12 UWB Timer APIs

5.12.1 dwt_timers_reset

void dwt_timers_reset(void);

This function will reset the timers block. It will reset both timers. It can be used to stop a timer running in repeat mode. Only available in QM33120 device.

Parameters:

none

Return Parameters:

none

Notes:

5.12.2 dwt_timers_read_and_clear_events

void dwt_timers_read_and_clear_events(void);

This function will read the timers' event counts. When reading from this register the values will be reset/cleared, thus the host needs to read both timers' event counts the events relating to TIMER0 are in bits [7:0] and events relating to TIMER1 in bits [15:8]. Only available in QM33120 device.

Parameters:

none

Return Parameters:

none

Notes:

5.12.3 dwt_configure_timer

void dwt_configure_timer(dwt_timer_cfg_t *tim_cfg);

This function configures selected timer (TIMER0 or TIMER1) as per configuration structure passed in. Only available in QM33120 device.

Parameters:

Type	Name	Description
dwt_timer_cfg_t*	tim_cfg	The timer configuration structure.

```
typedef enum
{
        DWT_TIMER0 = 0,
        DWT_TIMER1
} dwt_timers_e;
```

```
typedef enum
{
        DWT_TIM_SINGLE = 0,
        DWT_TIM_REPEAT
 } dwt_timer_mode_e;

 typedef enum
 {
        DWT_XTAL = 0,        // 38.4 MHz
        DWT_XTAL_DIV2 = 1,   // 19.2 MHz
        DWT_XTAL_DIV4 = 2,   // 9.6 MHz
        DWT_XTAL_DIV8 = 3,   // 4.8 MHz
        DWT_XTAL_DIV16 = 4, // 2.4 MHz
        DWT_XTAL_DIV32 = 5, // 1.2 MHz
        DWT_XTAL_DIV64 = 6, // 0.6 MHz
        DWT_XTAL_DIV128 = 7 // 0.3 MHz
 } dwt_timer_period_e;

 typedef struct
 {
        dwt_timers_e timer;           // Select the timer to use.
        dwt_timer_period_e timer_div; // Select the timer frequency (divider).
        dwt_timer_mode_e timer_mode;  // Select the timer mode.
        uint8_t timer_gpio_stop;      // Set to '1' to halt GPIO on interrupt.
        uint8_t timer_coexout;        // Configure GPIO for WiFi co-ex.
 } dwt_timer_cfg_t;
```

Return Parameters:

none

Notes:

The *tim_cfg* parameter points to a *dwt_timer_cfg _t* structure that has various fields to select and configure different parameters within the IC. The fields of the *dwt_timer_cfg _t* structure are identified are individually described below:

Fields	Description of fields within the *dwt_timer_cfg_t* structure
timer	The *timer* parameter selects the timer to configure. QM33120 has two timers namely TIMER0 and TIMER1. The timer is selected with setting this to either *DWT_TIMER0* or *DWT_TIMER1*.
timer_div	The *timer_div* is a timer frequency divider, I is used to configure desired timer frequency. The timer supports a number of frequencies as shown in *dwt_timer_period_e* above.
timer_mode	The *timer_mode* is used to select either a single expiry timer or configure the timer for continuous/repeat operation.
timer_gpio_stop	If this is set to 1 then the timer will stop when GPIO interrupt is raised. See more details about GPIO interrupt configuration in User Manual [2]
timer_coexout	This is used to configure the timer for WiFi coexistence mode. On timer expiry WiFi coex GPIO (4 or 5) is toggled. See also *dwt_configure_wificoex_gpio()*

5.12.4 dwt_configure_wificoex_gpio

void dwt_configure_wificoex_gpio(uint8_t timer_coexout, uint8_t coex_swap);

This function configures the GPIOs (4 and 5) for COEX_OUT. Only available in QM33120 device.

Parameters:

Type	Name	Description
uint8_t	timer_coexout	This configures whether the timer will control the GPIO COEX_OUT function. A timer needs to be configured for this operation see *dwt_configure_timer().* Set to 1 to enable this function, 0 to disable.
uint8_t	coex_swap	The configures if the COEX_OUT is on GPIO4 or GPIO5, when set to 1 the GPIO4 will be COEX_OUT.

Return Parameters:

none

Notes:

5.12.5 dwt_set_timer_expiration

void dwt_set_timer_expiration(dwt_timers_e timer_name, uint32_t exp);

This function sets timer expiration period, it is a 22-bit number. Only available in QM33120 device.

Parameters:

Type	Name	Description
dwt_timers_e	timer_name	This selects the timer to configure. QM33120 has two timers namely TIMER0 and TIMER1. The timer is selected with setting this to either *DWT_TIMER0* or *DWT_TIMER1*.
uint32_t	exp	The configures the expiry count - e.g. if units are XTAL/64 (1.66 us) then setting 1024 ~= 1.7 ms period.

Return Parameters:

none

Notes:

5.12.6 dwt_timer_enable

void dwt_timer_enable(dwt_timers_e timer_name);

This function enables the timer. In order to enable, the timer enable bit [0] for TIMER0 or [1] for TIMER1 needs to transition from 0->1. Only available in QM33120 device.

Parameters:

Type	Name	Description
dwt_timers_e	timer_name	This selects the timer to enable. QM33120 has two timers namely TIMER0 and TIMER1. The timer is selected with setting this to either *DWT_TIMER0* or *DWT_TIMER1*.

Return Parameters:

none

Notes:

5.13 SPI driver functions

These functions are platform specific SPI read and write functions, external to the driver code, used by the device driver to send and receive data over the SPI interface to and from the IC. The device driver abstracts the target SPI device by calling it through generic functions *writetospi()* and *readfromspi().* In porting the device driver, to different target hardware, the body of these SPI functions should be written, re-written, or provided in the target specific code to drive the target microcontroller device's physical SPI hardware. The initialisation of the target host controller's physical SPI interface mode and its data rate is considered to be part of the target system and is done in the host code outside of the device driver functions.

5.13.1 writetospi

int writetospi (uint16_t hLen, const uint8_t *hbuff, uint32_t bLen, const uint8_t *buffer);

This function is called by the device driver code (from the dwt_xfer3xxx() function) when it wants to write to the IC's SPI interface (registers) over the SPI bus.

Parameters:

Type	Name	Description
uint16_t	hLen	This is gives the length of the header buffer (*hbuff*)
uint8_t*	hbuff	This is a pointer to the header buffer byte array. The LSB is the first element.
uint32_t	bLen	This is gives the length of the data buffer (*buffer*), to write.

Type	Name	Description
uint8_t*	buffer	This is a pointer to the data buffer byte array. The LSB is the first element. This holds the data to write.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

The return values can be used to notify the upper application layer that there was a problem with SPI write. The *writetospi()* function has a return value, however it should be noted that the device driver itself does not take any notice of success/error return value but instead assumes that SPI accesses succeed without error.

5.13.2 writetospiwithcrc

int writetospiwithcrc(uint16_t hLen, const uint8_t *hbuff, uint32_t bLen, const uint8_t *buffer, uint8_t crc);

When the IC is configured to use SPI with 8-bit CRC mode, this function is called by the device driver code (from the dwt_xfer3xxx() function) to write to the SPI interface (registers) over the SPI bus. In this mode the IC is expecting an 8-bit CRC to be sent as the last byte of the write SPI transaction. If the CRC it receives from the host does not match a CRC it generates internally, then the IC will set SPI CRC error bit which will generate an interrupt if this status event has been masked enabled by the *dwt_enablespicrccheck()* function.

Parameters:

Type	Name	Description
uint16_t	hLen	This is gives the length of the header buffer (*hbuff*)
uint8_t*	hbuff	This is a pointer to the header buffer byte array. The LSB is the first element.
uint32_t	bLen	This is gives the length of the data buffer (*buffer*), to write.
uint8_t*	buffer	This is a pointer to the data buffer byte array. The LSB is the first element. This holds the data to write.
uint8_t	crc	This is the 8-bit CRC generated from the header and data bytes, which needs to be at the end of the SPI transaction

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

The return values can be used to notify the upper application layer that there was a problem with SPI write. The *writetospiwithcrc()* function has a return value, however it should be noted that the device driver itself does not take any notice of success/error return value but instead assumes that all SPI accesses succeed without error.

5.13.3 readfromspi

int readfromspi (uint16_t hLen, const uint8_t *hbuff, uint32_t bLen, uint8_t *buffer);

This function is called by the device driver code (from the dwt_xfer3xxx() function) when it wants to read from the IC's SPI interface (registers) over the SPI bus.

Parameters:

Type	Name	Description
uint16_t	hLen	This is gives the length of the header buffer (*hbuff*)
uint8_t*	hbuff	This is a pointer to the header buffer byte array. The LSB is the first element.
uint32_t	bLen	This is gives the number of bytes to read.
uint8_t*	buffer	This is a pointer to the data buffer byte array. The LSB is the first element. This holds the data being read.

Return Parameters:

Type	Description
int	Return values can be either DWT_SUCCESS = 0 or DWT_ERROR = -1.

Notes:

The return values can be used to notify the upper application layer that there was a problem with SPI read. The *readfromspi()* function has a return parameter, however it should be noted that the device driver itself does not take any notice of success/error return value but instead assumes that each SPI access succeeds without error.

5.14 Mutual-exclusion API functions

The purpose of these functions is to provide for microprocessor interrupt enable/disable, which is used for ensuring mutual exclusion from critical sections in the device driver code where interrupts

and background processing may interact. The only use made of this is to ensure SPI accesses are non-interruptible.

The mutual exclusion API functions are *decamutexon()* and *decamutexoff().* These are external to the driver code but used by the device driver when it wants to ensure mutual exclusion from critical sections. This usage is kept to a minimum and the disable period is also kept to a minimum (but is dependent on the SPI data rate). A blanket interrupt disable may be the easiest way to provide this mutual exclusion functionality in the target system, but at a minimum those interrupts coming from the device should be disabled/re-enabled by this activity.

In implementing the *decamutexon()* and *decamutexoff()* functions in a particular microprocessor system, the implementer may choose to use #defines to map these calls transparently to the target system. Alternatively the appropriate code may be embedded in the functions provided in the deca_mutex.c source file.

5.14.1 decamutexon

decaIrqStatus_t decamutexon (void);

This function is used to turn on mutual exclusion (e.g. by disabling interrupts). **This is called at the start of the critical section of SPI access.** The *decamutexon()* function should operate to read the current system interrupt status in the target microcontroller system's interrupt handling logic with respect to the handling of the IC's interrupt. Let's call this "IRQ_State" Then it should disable the interrupt relating to the IC, and then return the original IRQ_State.

Parameters:

none

Return Parameters:

Type	Description
decaIrqStatus_t	This is the state of the target microcontroller's interrupt logic with respect to the handling the IC's interrupt, as it was on entry to the *decamutexon()* function before it did any interrupt disabling.

```
Typedef int decaIrqStatus_t ;
```

Notes:

The *decamutexon()* function returns the IC's interrupt status, which can be noted and appropriate action taken. The returned status is intended to be used in the call to *decamutexoff()* function to be used to restore the interrupt enable status to its original pre-*decamutexon()* state.

5.14.2 decamutexoff

void decamutexoff (decaIrqStatus_t state);

This function is used to restore the IC's interrupt state as returned by *decamutexon()* function. It is used to turn off mutual exclusion (e.g. by enabling interrupts if appropriate). **This is called at the end**

of the critical section of SPI access. The *decamutexoff()* function should operate to restore the system interrupt status in the target microcontroller system's interrupt handling logic to the state indicated by the input "IRQ_State" parameter, *state*.

Parameters:

Type	Name	Description
decaIrqStatus_t	state	This is the state of the target microcontroller's interrupt logic with respect to the handling of the IC's interrupt, as it was on entry to the *decamutexon ()* function before it did any interrupt disabling.

Return Parameters:

none

Notes:

The state parameter passed into *decamutexoff()* function should be used to appropriately set/restore the system interrupt status in the target microcontroller system's interrupt handling logic.

5.15 Sleep function

The purpose of this function is to provide a platform dependent implementation of sleep feature, i.e. waiting for a certain amount of time before proceeding with the application's next step.

This is an external function used by the driver code to wait for the end of a process, e.g. the stabilization of a clock or the completion of a write command. This function is provided in the deca_sleep.c source file.

5.15.1 deca_sleep

void deca_sleep (unsigned int time_ms);

This function is used to wait for a given amount of time before proceeding to the next step of the calling function.

Parameters:

Type	Name	Description
unsigned int	time_ms	The amount of time to wait, expressed in milliseconds.

Return Parameters:

None

Notes:

The implementation provided here is designed for a simple single-threaded system and is blocking, i.e. it will prevent the system from doing anything else during the waiting time.

5.15.2 deca_usleep

void deca_usleep (unsigned int time_us);

This function is used to wait for a given amount of time before proceeding to the next step of the calling function.

Parameters:

Type	Name	Description
unsigned int	time_us	The amount of time to wait, expressed in microseconds.

Return Parameters:

None

Notes:

The implementation provided here is designed for a simple single-threaded system and is blocking, i.e. it will prevent the system from doing anything else during the waiting time.

5.16 Subsidiary functions

These functions are used to provide low-level access to individually numbered registers and buffers (or register files). These may be needed to access IC functionality not included in the main API functions above.

5.16.1 dwt_writetodevice

dwt_writetodevice(uint32_t regFileID, uint16_t index, uint16_t length, const uint8_t *buffer);

This function is used to write to the IC's registers and buffers. The *regID* specifies the main address of the register or parameter block being accessed, e.g. a *regID* of DX_TIME selects the delay TX or RX start time register. The *index* parameter selects a sub-address within the register file. An *index* value of 0 is used for most of the accesses employed in the device driver. The *length* parameter specifies the number of bytes to write, and the *buffer* parameter points at the bytes to actually write. If DWT_API_ERROR_CHECK code switch is defined, this function will check input parameters and assert if an error is detected.

Parameters:

Type	Name	Description
uint32_t	regFileID	ID of register file or buffer being accessed.
uint16_t	index	Byte index into register file or buffer being accessed.
uint16_t	length	Number of bytes being read/written

uint8_t*	buffer	Pointer to buffer containing the 'length' bytes to be written.

Return Parameters:

None

5.16.2 dwt_readfromdevice

void dwt_readfromdevice(uint32_t regFileID, uint16_t index, uint16_t length, uint8_t *buffer);

This function is used to read from the IC's registers and buffers. The parameters are the same as for the dwt_writetodevice() function above except that the *buffer* parameter points at a location where the bytes being read are placed by the function call. If DWT_API_ERROR_CHECK code switch is defined, this function will check input parameters and assert if an error is detected. It is up to the developer to ensure that the assert macro is correctly enabled in order to trap any error conditions that arise.

Parameters:

Type	Name	Description
uint32_t	regFileID	ID of register file or buffer being accessed.
uint16_t	index	Byte index into register file or buffer being accessed.
uint16_t	length	Number of bytes being read/written
uint8_t*	buffer	Pointer to buffer in which to return the read data.

Return Parameters:

None

5.16.3 dwt_xfer3xxx

void dwt_xfer3xxx(uint32_t regFileID, uint16_t index, uint16_t length, uint8_t *buffer, spi_modes_e spi_modes);

This function is used to read from or write to the IC's registers and buffers. The parameters specify the register address to access, the byte index at which to access said register address, the number of bytes being read/written, the buffer to read to / write from and the particular SPI mode used respectively. The SPI modes are specified in Table 24. If DWT_API_ERROR_CHECK code switch is defined, this function will check input parameters and assert if an error is detected. It is up to the developer to ensure that the assert macro is correctly enabled in order to trap any error conditions that arise.

Parameters:

Type	Name	Description
uint32_t	regFileID	ID of register file or buffer being accessed.
uint16_t	index	Byte index into register file or buffer being accessed.
uint16_t	length	Number of bytes being read/written
uint8_t*	Buffer	Pointer to buffer in which to return the read data.
spi_modes_e	spi_modes	Mode of SPI transaction (read or write). See Table 24 for more details.

Return Parameters:

None

Notes:

The implementation provided here is designed for a simple single-threaded system and is blocking, i.e. it will prevent the system from doing anything else during the waiting time.

Both *dwt_writetodevice()* and *dwt_readfromdevice()* will use this function with their specified SPI modes to perform their respective SPI writes and reads.

Table 24: spi_modes_e enum values (SPI read/write modes)

SPI Modes	Value	Description
DW3000_SPI_RD_BIT	0x0000	Standard SPI read mode.
DW3000_SPI_RD_FAST_CMD	0x0001	SPI read with fast command mode.
DW3000_SPI_WR_FAST_CMD	0x0002	SPI write with fast command mode.
DW3000_SPI_WR_BIT	0x8000	Standard SPI Write mode.
DW3000_SPI_AND_OR_8	0x8001	8-bit SPI read modify write mode
DW3000_SPI_AND_OR_16	0x8002	16-bit SPI read modify write mode
DW3000_SPI_AND_OR_32	0x8003	32-bit SPI read modify write mode

5.16.4 dwt_read32bitreg

uint32_t dwt_read32bitreg(int regFileID);

This function is used to read 32-bit IC registers.

5.16.5 dwt_read32bitoffsetreg

uint32_t dwt_read32bitoffsetreg(int regFileID, int regOffset);

This function is used to read a 32-bit IC register that is part of a sub-addressed block.

5.16.6 dwt_write32bitreg

void dwt_write32bitreg(int regFileID, uint32_t regval);

This function is used to write a 32-bit IC register that is part of a sub-addressed block.

5.16.7 dwt_write32bitoffsetreg

void dwt_write32bitoffsetreg(int regFileID, int regOffset, uint32_t regval);

This function is used to write to a 32-bit IC register that is part of a sub-addressed block.

5.16.8 dwt_read16bitoffsetreg

uint16_t dwt_read16bitoffsetreg(int regFileID, int regOffset);

This function is used to read a 16-bit IC register that is part of a sub-addressed block.

5.16.9 dwt_write16bitoffsetreg

void dwt_write16bitoffsetreg(int regFileID, int regOffset, uint16_t regval);

This function is used to write a 16-bit IC register that is part of a sub-addressed block.

5.16.10 dwt_read8bitoffsetreg

uint8_t dwt_read8bitoffsetreg(int regFileID, int regOffset);

This function is used to read an 8-bit IC register that is part of a sub-addressed block.

5.16.11 dwt_write8bitoffsetreg

void dwt_write8bitoffsetreg(int regFileID, int regOffset, uint8_t regval);

This function is used to write an 8-bit IC register that is part of a sub-addressed block.

5.16.12 dwt_modify32bitoffsetreg

void dwt_write32bitoffsetreg(int regFileID, int regOffset, uint32_t andmask, uint32_t ormask);

This function is used to clear or set individual bits in a 32-bit register. The andmask will be AND-ed with the register value, and the ormask OR-ed. Single or multiple bits can be set in a single SPI transaction.

5.16.13 dwt_modify16bitoffsetreg

void dwt_write16bitoffsetreg(int regFileID, int regOffset, uint16_t andmask, uint16_t ormask);

This function is used to clear or set individual bits in a 16-bit register. The andmask will be AND-ed with the register value, and the ormask OR-ed. Single or multiple bits can be set in a single SPI transaction.

5.16.14 dwt_modify8bitoffsetreg

void dwt_modify8bitoffsetreg(int regFileID, int regOffset, uint8_t andmask, uint8_t ormask);

This function is used to clear or set individual bits in an 8-bit register. The andmask will be AND-ed with the register value, and the ormask OR-ed. Single or multiple bits can be set in a single SPI transaction.

5.16.15 dwt_writefastCMD

void dwt_writefastCMD(int cmd);

This function is used to write a single byte special 5-bit command word to the device. The supported commands are listed below:

Table 25: List of supported commands

Command ID	Value	Description
CMD_TXRXOFF	0x0	Puts the device into IDLE state and clears any events.
CMD_TX	0x1	Immediate start TX
CMD_RX	0x2	Immediate RX on
CMD_DTX	0x3	Delayed TX w.r.t. DX_TIME
CMD_DRX	0x4	Delayed RX w.r.t. DX_TIME
CMD_DTX_TS	0x5	Delayed TX w.r.t. TX timestamp + DX_TIME
CMD_DRX_TS	0x6	Delayed RX w.r.t. TX timestamp + DX_TIME
CMD_DTX_RS	0x7	Delayed TX w.r.t. RX timestamp + DX_TIME
CMD_DRX_RS	0x8	Delayed RX w.r.t. RX timestamp + DX_TIME
CMD_DTX_REF	0x9	Delayed TX w.r.t. REF_TIME + DX_TIME
CMD_DRX_REF	0xA	Delayed RX w.r.t. REF_TIME + DX_TIME
CMD_CCA_TX	0xB	TX frame if no preamble detected
CMD_TX_W4R	0xC	Immediate start TX, then enable receiver
CMD_DTX_W4R	0xD	Delayed TX w.r.t. DX_TIME, then enable receiver
CMD_DTX_TS_W4R	0xE	Delayed TX w.r.t. TX timestamp + DX_TIME, then enable receiver
CMD_DTX_RS_W4R	0xF	Delayed TX w.r.t. RX timestamp + DX_TIME, then enable receiver
CMD_DTX_REF_W4R	0x10	Delayed TX w.r.t. REF_TIME + DX_TIME, then enable receiver
CMD_CCA_TX_W4R	0x11	TX frame if no preamble detected, then enable receiver
CMD_CLR_IRQS	0x12	Clear all interrupt events
CMD_DB_TOGGLE	0x13	Toggle double buffer pointer

Command ID	Value	Description
CMD_SEMA_REQ	0x14	Write to the Semaphore and try to reserve access (if it hasn't already been reserved by the other master)
CMD_SEMA_REL	0x15	Release the semaphore if it is currently reserved by this master
CMD_SEMA_FORCE	0x16	Only SPI 2 can issue this command. Force access regardless of current semaphore value
CMD_SEMA_RESET	0x18	Global digital reset including of the semaphore
CMD_SEMA_RESET_NO_SEM	0x19	Global digital reset without reset of the semaphore
CMD_ENTER_SLEEP	0x1A	Enters sleep/deep sleep according to ANA_CFG - DEEPSLEEP_EN

5.16.16 dwt_readfastCMD

void dwt_readfastCMD(uint32_t cmd, uint8_t *data);

This function is used to read a single byte special 5-bit command word from the device and return one byte (4-bit) from that address.

5.16.17 dwt_read_reg

uint32_t dwt_read_reg(uint32_t address);

This function allows read from the DW3xxx device 32-bit register.

Parameters:

Type	Name	Description
uint32_t	address	ID of the DW3xxx register

Return Parameters:

Type	Description
uint32_t	Value of the 32-bit register

5.16.18 dwt_write_reg

void dwt_read_reg(uint32_t address, uint32_t data);

This function allows read from the DW3xxx device 32-bit register.

Parameters:

Type	Name	Description
uint32_t	address	ID of the DW3xxx register
uint32_t	data	Value to write to register

Return Parameters:

None

6 APPENDIX 1 – SIMPLE EXAMPLES

The API package [5] provides, along with the IC driver itself, a set of simple example applications designed to show how to achieve a number of basic features of the IC like sending a frame, receiving a frame, putting the IC to sleep, etc.

All these examples have been designed to be as simple as possible. The main idea is to make the code self-explanatory and include the least possible amount of code not directly involved in the achievement of the example-related feature. One of the consequences of this design is that the examples output very little (or even no) debug information and are designed so that the application flow can be followed using a debugger to examine run-time operations.

On the hardware side, the examples have been designed to run on an DW3XXX Arduino-type shield. The base layers included in this package (see detail below) provide specific implementations for this HW.

6.1 Package structure

The folder structure of the package is the following:

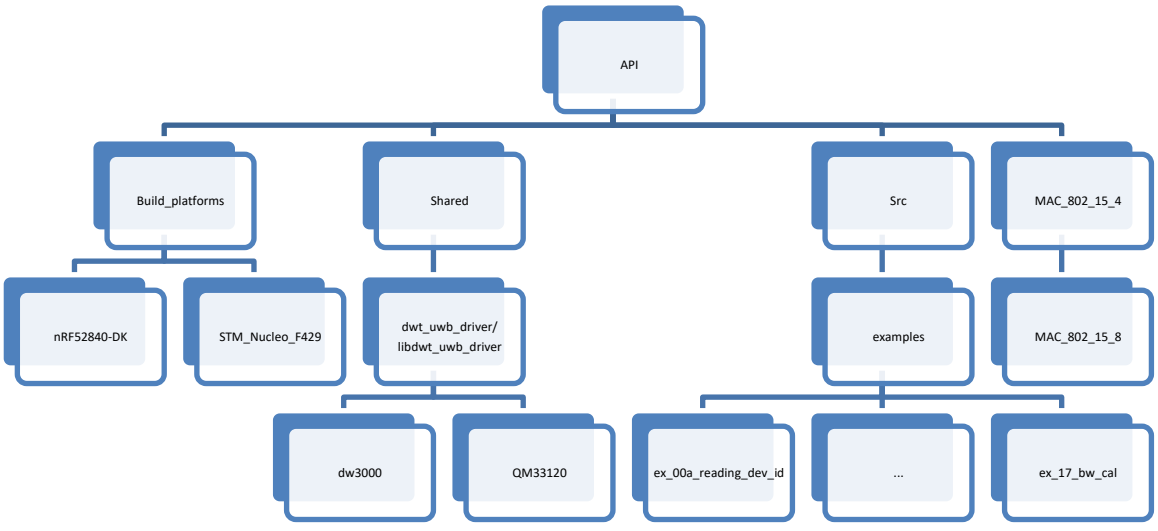


Figure 8: API package structure tree

Table 26: API package structure				**Brief description**
API				Root directory of DW3XXX API and test examples.
├───	Build_Platforms			Directory containing hardware platforms that are supported by the software package.
|	├───	nRF52840-DK		Directory containing SEGGER IDE build files for Nordic platform.
|	├───	STM_Nucleo_F429		Directory containing STM Workbench build files for STM platform.
├───	Shared			Directory that contains shared drivers for Decawave/Qorvo products.
├───	├───	dwt_uwb_driver		Drivers library for the Decawave/Qorvo UWB family of products.
├───	Src			Main source directory of Decawave/Qorvo API code + simple examples.
	├───	examples		Examples directory containing various simple examples.
	|	├───	ex_00a_reading_dev_id	Read Device ID example code.
	|	├───	ex_01a_simple_tx	Simple TX example code.
	|	├───	ex_01b_tx_sleep	TX Sleep example code.
	|	├───	…	Additional example code.
	├───	MAC_802_15_4		MAC layer IEEE 802.15.4 code.
	├───	MAC_802_15_8		MAC layer IEEE 802.15.8 code.

All example applications are named after the feature or set of features they implement.

6.2 Building and running the examples

All examples provide a specific ex_<example number>_<example name>.c source file with a single project build configuration. To build and run the code, just unzip the source code and import the project as “Existing Projects into Workspace” into your ST Workbench IDE. If ST Workbench IDE is already installed on your machine, you should be able to simply double-click the “.cproject” file and the project will load into the IDE.

Once the project is loaded into the IDE, select the example to build by editing the “..\API\Src\example_selection.h” header file. Each simple example has a corresponding “#define” (pre-processor macro definition) in this header file. For example, to build the

"ex_00a_reading_dev_id" simple example, the "//" before the "#define TEST_READING_DEV_ID" would need to be removed. By default, each of the pre-processor macro definitions that correspond to a simple example are commented out (with a "//" in front of each line). If there is any confusion as to which pre-processor macro definition needs to be enabled for a particular simple example, simply look at the source code for the simple example in question. It will have an "#if defined(<example macro definition>)" near the top of the source file. This is the macro definition to enable in the "example_selection.h" file to build that simple example.

ST Workbench IDE (SW4STM32) and CubeMX project generator can be downloaded from ST website. [6]

6.3 Examples list

As all examples have been designed to be self-explanatory and quite straightforward to read. The following is a list of all the examples provided with a brief description of the function of each.

6.3.1 Example 00a: reading device ID

This example is the most basic example which just reads the QM33120 device ID register. This can be used to test that the SPI communications between host MCU and QM33120 are working correctly.

6.3.2 Example 01a: simple TX

This example application repeatedly sends a hard-coded standard blink frame. Hard-coded delay between frames is 1 second.

6.3.3 Example 01b: TX with sleep

This is a variation of example 1a, where the IC is commanded to sleep and then awoken after the delay between each frame.

There are two flavours of this example "tx_sleep" and "sleep_idleRC". In the latter the device remains in IDLE_RC state after wakeup, and only transitions to IDLE prior to transmission of the message, staying in IDLE_RC during the programming of TX data and frame control means QM33120 is in a lower power state, and consumes less power than if it was in IDLE state (as in the former example).

6.3.4 Example 01c: TX with auto sleep

This is a variation of example 1b where the IC automatically goes to sleep after the transmission of a frame. The IC is still commanded to wake up after the desired sleep period has elapsed before sending the next frame.

6.3.5 Example 01d: TX with timed sleep

This is a variation of example 1c where the IC automatically wakes up using an internal sleep timer. Before the IC is put to sleep for the first time, the internal low-power oscillator driving the sleep counter is calibrated so that the desired sleep time can be properly set through the sleep timer counter.

6.3.6 Example 01e: TX with CCA

Here we implement a simple Clear Channel Assessment (CCA) mechanism before frame transmission. The CCA can be used to avoid collisions with other frames on the air.

Note this example is not doing CCA the way a continuous carrier radio would do it by looking for energy/carrier in the band. It is only looking for preamble so will not detect PHR or data phases of the frame. In a UWB data network it is advised to also do a random back-off before re-transmission in the event of not receiving acknowledgement to a data frame transmission.

6.3.7 Example 01g: simple TX with STS

This example is very similar to Example 1a, 6.3.1 above, except that it is using STS configuration.

6.3.8 Example 01h: simple TX for PDOA

This example is very similar to Example 1a, 6.3.1 above, except that it is using TX configuration for PDOA.

6.3.9 Example 01i: simple TX with AES

This example is very similar to Example 1a, 6.3.1 above, except that it is using AES encryption of the data payload (of 802.15.8 sample frame). The encrypted data is fixed bytes array, but the header is changing according to the nonce (changing according to counter) and frame sequence number. This payload is then decrypted by the 6.3.17 companion example.

6.3.10 Example 02a: simple RX

This example application waits indefinitely for an incoming frame. When a frame is received, it is read into a local buffer where it can be examined and then the application re-enables the receiver to start waiting for another frame. It is intended that the simple TX examples (like that in *6.3.1 above*) should be used as a source of frames when running these simple RX examples.

There are two flavours of this example “simple_rx” and “simple_rx_nlos”. In the latter, after the frame is received and validated based on the diagnostics logged, diagnostic register values are read and calculations for First Path Power based on the section 4.7.1 and estimating the receive signal power based on 4.7.2 of the User Manual [2]. The probability of signal being Line of Sight or Non-Line of Sight is calculated based on the Application Notes "APS006 PART 3” [8] revision (1.1).

6.3.11 Example 02c: simple RX with diagnostics

This is a variation of example 2a where RX frame diagnostic information (first path index, channel impulse response power) and accumulator (channel impulse response) values are read for each received frame. This information is read into a local structure where it can be examined.

6.3.12 Example 02d: RX SNIFF mode

This is a variation of example 2a where the RX SNIFF mode of QM33120 is used. When the receiver is enabled, it begins preamble-hunt mode with the receiver on. In SNIFF mode, the receiver is not on

all the time, but is sequenced on and off, with a defined duty-cycle. In this example, these durations are defined to give roughly a 50% duty-cycle, which allows a corresponding reduction in the preamble-hunt power consumption while still being able to receive frames. It is suggested that the simple TX example, from 6.3.1 above, is used as a source of frames to test this.

Note: SNIFF mode reduces RX sensitivity depending on the on and off period configurations. Please see the QM33120 User Manual [2] for more details

6.3.13 Example 02e: Double Buffer RX

This example keeps listening for any incoming frames, storing in a local buffer any frame received before going back to listening. This example activates interrupt handling and the double buffering feature of the DW IC (either auto or manual re-enable of receiver can be used). Frame processing is performed in the RX good frame call-back.

6.3.14 Example 02f: RX with XTAL trimming

This is an example of a receiver that measures the clock offset of a remote transmitter and then uses the XTAL trimming function to modify the local clock to achieve a target clock offset. Note: To keep a system stable it is recommended to only adjust trimming at one end of a link.

6.3.15 Example 02g: simple RX with STS

This example is very similar to Example 2a, 6.3.10 above, except that it is using STS configuration.

6.3.16 Example 02h: simple RX with PDOA

This example is very similar to Example 2a, 6.3.10 above, except that it is using PDOA configuration.

6.3.17 Example 02i: simple RX AES

This example application waits indefinitely for an incoming frame (is expects a 802.15.8 sample frame from companion 6.3.9 example). When a frame is received, it starts to examine the frame residing in the RX buffer. It checks sizes validity and then extract the header from this buffer. According to this header and header size, it builds the nonce and get the payload size, before attempting to decrypt the payload.

6.3.18 Example 03a: TX then wait for a response

This example application is a combination of examples 1a and 2a. This example sends a frame then waits for a response (with receive timeout enabled). If a response is received, it is stored in a local buffer for examination and then flow proceeds to the transmission of the next frame. If a response is not received, the timeout will trigger, and the application will proceed to the next transmission.

6.3.19 Example 03b: RX then send a response

This example application is the complement of example 3a. It waits indefinitely for a frame. When a frame is received, it is stored in a local buffer. If the received frame is the one transmitted by the

example 3a application, then a response is sent. In any case, when the received frame is processed this simple example application re-enables the receiver to start waiting again for another frame.

6.3.20 Example 03d: TX then wait for a response using interrupts

This is a variation of example 3a where interrupts and call-backs are used to process received frames, reception errors and timeouts and transmission confirmation instead of polling with an infinite loop.

6.3.21 Example 04a: continuous wave mode

This example application activates continuous wave mode for 2 minutes with a predefined configuration. On a correctly configured spectrum analyser (use configuration values on the picture below), the output should look like this:

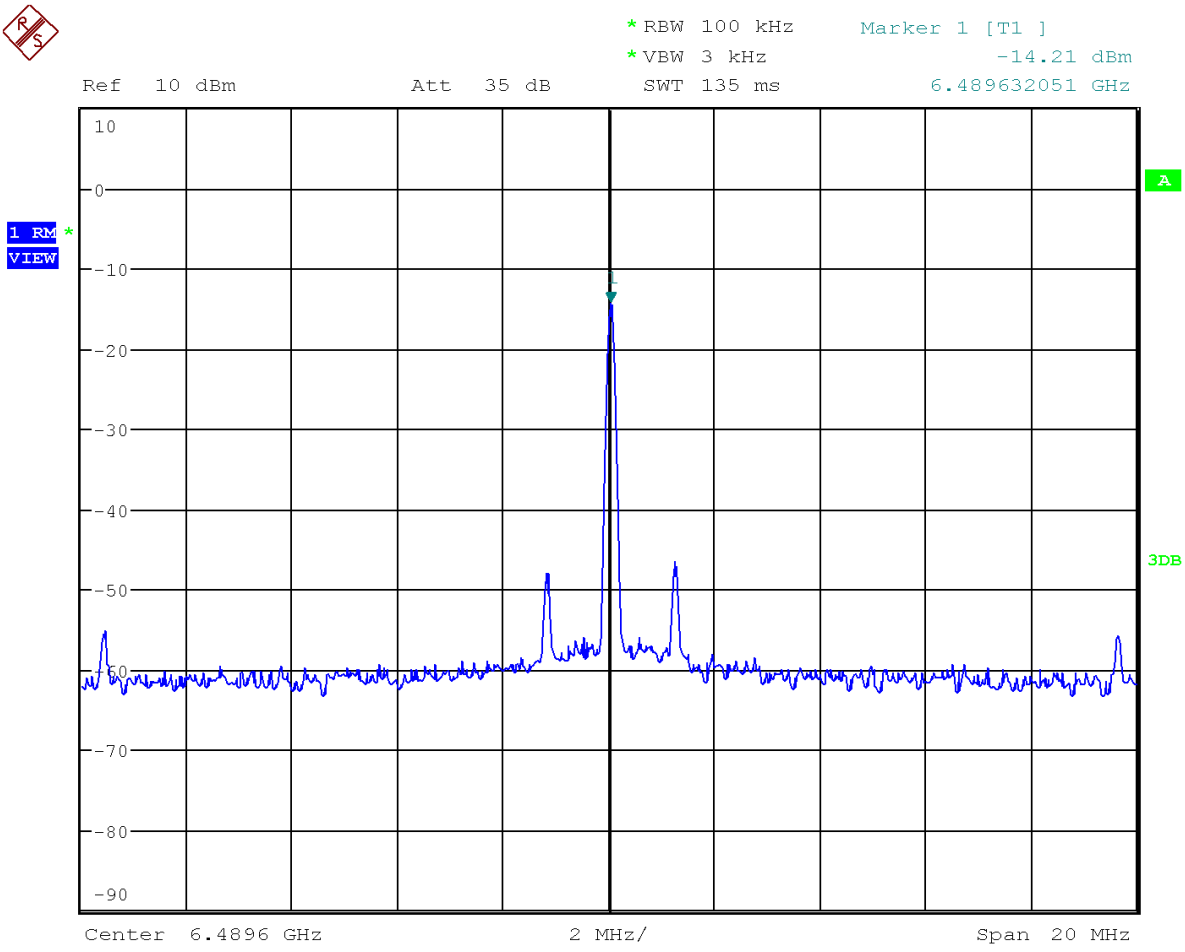


Figure 9: Continuous wave output

6.3.22 Example 04b: continuous frame mode

This example application activates continuous frame mode for 2 minutes with a predefined configuration. On a correctly configured spectrum analyser (use configuration values on the picture below), the output should look like this:

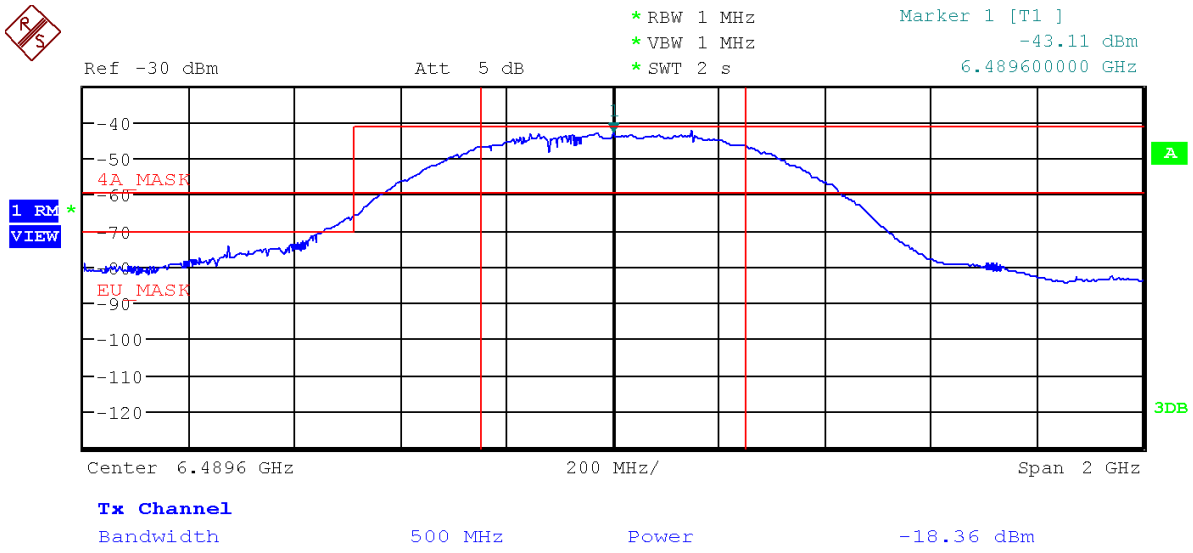


Figure 10: Continuous frame output

6.3.23 Example 05a: double-sided two-way ranging (DS TWR) initiator

This is a simple code example that acts as the initiator in a DS TWR distance measurement exchange. This application sends a "poll" frame (recording the TX time-stamp of the poll), and then waits for a "response" message expected from the "DS TWR responder" example code (companion to this application – see section *6.3.24 below*). When the response is received its RX time-stamp is recorded and we send a "final" message to complete the exchange. The final message contains all the time-stamps recorded by this application, including the calculated/predicted TX time-stamp for the final message itself. The companion "DS TWR responder" example application works out the time-of-flight over-the-air and, thus, the estimated distance between the two devices.

Included in this directory in the examples source code is another version of the code described above that uses STS Mode 1 frames instead of STS Mode 0 frames. This means that the frames that are sent and received use an STS to compute distance measurements. For more details on STS, please read the IEEE 802.15.4z documentation.

6.3.24 Example 05b: double-sided two-way ranging (DS TWR) responder

This is a simple code example that acts as the responder in a DS TWR distance measurement exchange. This application waits for a "poll" message (recording the RX time-stamp of the poll) expected from the "DS TWR initiator" example code (companion to this application), and then sends a "response" message recording its TX time-stamp, after which it waits for a "final" message from the initiator to complete the exchange. The final message contains the remote initiator's time-stamps of poll TX, response RX and final TX. With this data and the local time-stamps, (of poll RX,

response TX and final RX), this example application works out a value for the time-of-flight over-the-air and, thus, the estimated distance between the two devices, which it writes to the LCD.

Included in this directory in the examples source code is another version of the code described above that uses STS Mode 1 frames instead of STS Mode 0 frames. This means that the frames that are sent and received use an STS to compute distance measurements. For more details on STS, please read the IEEE 802.15.4z documentation.

6.3.25 Example 05c: double-sided two-way ranging with STS (DS TWR STS) initiator

This is an extension based on example 5a (see section *6.3.23 above*), except that the STS mode is also configured. Thus when good frame reception occurs, STS quality is checked and validated before the STS timestamps are used to work out the range. The companion to this example is DS TWR STS responder is described in section *6.3.26 below*.

6.3.26 Example 05d: double-sided two-way ranging with STS (DS TWR STS) responder

This is a companion example to DW TWR STS initiator (see section *6.3.25 above*) and is based on DW TWR responder example (see section *6.3.24 above*) with the addition of STS.

6.3.27 Example 06a: single-sided two-way ranging (SS TWR) initiator

This is a simple code example that acts as the initiator in a SS TWR distance measurement exchange. This application sends a “poll” frame (recording the TX time-stamp of the poll), after which it waits for a “response” message from the “SS TWR responder” example code (companion to this application) to complete the exchange. The response message contains the remote responder’s time-stamps of poll RX, and response TX. With this data and the local time-stamps, (of poll TX and response RX), this example application works out a value for the time-of-flight over-the-air and, thus, the estimated distance between the two devices, which it writes to the LCD.

Heretofore, we would have recommended use of double-sided TWR (as per examples 5a and 5b) instead of this single-sided two-way ranging because the SS-TWR time-of-flight estimation typically suffers poor accuracy due to the clock offset between the two nodes participating in the TWR exchange. However since driver version 4.0.6 we are now making use of the carrier integrator diagnostic from the IC (accessible via the new *dwt_readcarrierintegrator()* API function) to measure the clock offset and improve the accuracy SS-TWR range estimate calculation.

Included in this directory in the examples source code is another version of the code described above that uses STS Mode 1 frames instead of STS Mode 0 frames. This means that the frames that are sent and received use an STS to compute distance measurements. For more details on STS, please read the IEEE 802.15.4z documentation.

Also included in this directory in the examples source code is another version of the code described above that uses STS Mode 3 packets instead of STS Mode 0 frames. The “poll” and “response” messages contain no payload and are only used for computing timestamps using STS. An STS Mode 0 frame is sent from the receiver to the initiator which contains the required timestamp data to

compute distance values in this particular transaction of signals. For more details on STS, please read the IEEE 802.15.4z documentation.

6.3.28 Example 06b: single-sided two-way ranging (SS TWR) responder

This is a simple code example that acts as the responder in a SS TWR distance measurement exchange. This application waits for a "poll" message (recording the RX time-stamp of the poll) expected from the "SS TWR initiator" example code (companion to this application), and then sends a "response" message to complete the exchange. The response message contains all the timestamps recorded by this application, including the calculated/predicted TX time-stamp for the response message itself. The companion "SS TWR initiator" example application works out the time-of-flight over-the-air and, thus, the estimated distance between the two devices.

Included in this directory in the examples source code is another version of the code described above that uses STS Mode 1 frames instead of STS Mode 0 frames. This means that the frames that are sent and received use an STS to compute distance measurements. For more details on STS, please read the IEEE 802.15.4z documentation.

Also included in this directory in the examples source code is another version of the code described above that uses STS Mode 3 packets instead of STS Mode 0 frames. The "poll" and "response" messages contain no payload and are only used for computing timestamps using STS. An STS Mode 0 frame is sent from the receiver to the initiator which contains the required timestamp data to compute distance values in this particular transaction of signals. For more details on STS, please read the IEEE 802.15.4z documentation.

6.3.29 Example 06e: single-sided two-way ranging (SS TWR) initiator with AES

This is a simple code example that acts as the initiator in a SS TWR distance measurement exchange. This application sends a "poll" frame (recording the TX time-stamp of the poll), after which it waits for a "response" message from the "SS TWR responder" example code (companion to this application) to complete the exchange. The response message contains the remote responder's timestamps of poll RX, and response TX. With this data and the local time-stamps, (of poll TX and response RX), this example application works out a value for the time-of-flight over-the-air and, thus, the estimated distance between the two devices, which it writes to the LCD.

This example uses a simple MAC frame – 802.15.4. It has a source address; destination address and key index for encryption.

The Initiator sends encrypted tx_poll_msg message to the responder.

The responder will replay with its own encrypted rx_resp_msg, but this time with a different key index and it will switch between the source address and destination address.

Both Initiator and responder will check if the data is for them, meaning data is encrypted and with the right source and destination address. The responder puts its time signature results in the first 8 bytes of the rx_resp_msg.

Heretofore, we would have recommended use of double-sided TWR (as per examples 5a and 5b) instead of this single-sided two-way ranging because the SS-TWR time-of-flight estimation typically suffers poor accuracy due to the clock offset between the two nodes participating in the TWR exchange. However since driver version 4.0.6 we are now making use of the carrier integrator diagnostic from the IC (accessible via the new *dwt_readcarrierintegrator()* API function) to measure the clock offset and improve the accuracy SS-TWR range estimate calculation.

6.3.30 Example 06f: single-sided two-way ranging responder (SS TWR) with AES

This is a simple code example that acts as the responder in a SS TWR distance measurement exchange. This application waits for a "poll" message (recording the RX time-stamp of the poll) expected from the "SS TWR initiator" example code (companion to this application), and then sends a "response" message to complete the exchange. The response message contains all the time-stamps recorded by this application, including the calculated/predicted TX time-stamp for the response message itself. The companion "SS TWR initiator" example application works out the time-of-flight over-the-air and, thus, the estimated distance between the two devices.

This example uses a simple MAC frame – 802.15.4. It has a source address, destination address and key index for encryption.

The Responder replies to the Initiator with encrypted rx_resp_msg message.

The responder replay will be with a different key index and it will switch between the source address and destination address.

Both Initiator and responder will check if the data is for them, meaning data is encrypted and with the right source and destination address. The responder puts its time signature results in the first 8 bytes of the rx_resp_msg.

6.3.31 Example 07a: Auto ACK TX

This example, with its companion example 8b below, demonstrates the operation of the IC's auto-ACK function. The code here is based on example 3a, except that in this case the transmitted frame has the AR (acknowledgement request) bit set in the frame control field of the MAC header, (following the MAC frame definitions of IEEE 802.15.4 [3]), and the turn-around to await response is immediate, reflecting the ACK response timing of the IC.

6.3.32 Example 07b: Auto ACK RX

This complement to example 8a. Here the Auto ACK feature of IC is activated so that frames sent by companion example 8a are automatically acknowledged.

6.3.33 Example 11a: Use of SPI CRC

This example shows the use of SPI CRC feature.

6.3.34 Example 13a: Use of DW3XXX GPIO lines

This example demonstrates how to enable the GPIO lines as inputs and output

6.3.35 Example 14: OTP Write

This example illustrates how a user can write and verify data to addresses in the OTP memory.

6.3.36 Example 15: LE (Low-Energy) pend

This example illustrates how a user can utilise the “LE Pend” (Low-Energy Pending) features of the QM331XX device. A TX device will transmit a frame to an RX device. The RX device will acknowledge this frame (with an ACK frame) if the following conditions are met:

- The TX frame is an IEEE 802.14.5 MAC command frame.
  The frame is received from a device with an address that is pre-programmed into the QM331XX LE_PEND01 or LE_PEND23 registers.

6.3.37 Example 16 PLL Cal

This example will test that the PLL will recalibrate and relock when a significant change in temperature is detected.

6.3.38 Example 17 Bandwidth Calibration

This example will record the initial PG count (emulating what should be done in factory). The example will recalibrate the bandwidth given this reference PG count value in a loop over time. The example should be run in a temperature chamber over a range of operating temperatures. The device will output a continuous frame for bandwidth monitoring on a spectrum analyser.

6.3.39 Example 18: Timer Example

This example demonstrates how to enable one of DW IC internal timers. In this example TIMER0 is configured in the repeating mode with period set to approx. 1s. Every second host count of timer events is printed. Every 20 seconds both host count and DW count of timer event is printed.

6.3.40 Example 19: TX Power Adjustment Example

This example demonstrates how the power adjustment API can be used to perform some adjustment of TX power depending on TX frame duration.

6.3.41 Example 20: Simple AES

This example demonstrates how the use the AES engine to encrypt/decrypt using the AES-CCM* standard that is defined in the IEEE 802.15.4 standard [3]. It uses test vectors defined in that standard also.

7 APPENDIX 2 – BIBLIOGRAPHY:

[1]	The Decawave DW3000 and QM33120 Data Sheet, which is available on available on www.decawave.com.
[2]	User Manual DW3000 and QM33120 User Manual which is available on www.decawave.com.
[3]	IEEE 802.15.4-2011 or "IEEE Std 802.15.4™-2015" (Revision of IEEE Std 802.15.4-2017). IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Computer Society Sponsored by the LAN/MAN Standards Committee. Available from http://standards.ieee.org/
[4]	Application note APS023 Part 2: TX Bandwidth and Power Compensation. This is available on available on www.decawave.com.
[5]	DW3XXX Application Programming Interface with application examples package downloadable from http://www.decawave.com/support/software
[6]	Installation of tools and drivers, www.st.com
[7]	ARIB STD-T91 https://www.arib.or.jp/english/std_tr/telecommunications/desc/std-t91.html
[8]	Application Notes APS003 https://www.decawave.com/application-notes/

Table 27: Bibliography

8 DOCUMENT HISTORY

Table 28: Document History

Revision	Date	Description
1.0	30th March 2021	Initial release.
1.1	13th April 2021	Updated release with “unified driver” changes.
2.0	12th July 2021	Added the NLOS API, TX power adjustment and GPIO driver changes, updated various missing references.
2.1	17th August 2021	Updated document to reflect latest changes in Simple Examples project and dwt_uwb_driver API code. This version is planned to be released alongside the DW3XXX Release 9.

9 MAJOR CHANGES

9.1 Release 2.0

Page	Change Description
-	The document has been updated to cover a whole family of DW3XXX devices, and their APIs.

10 ABOUT DECAWAVE

Decawave is a pioneering fabless semiconductor company whose flagship product, the QM33120, is a complete, single chip CMOS Ultra-Wideband IC based on the IEEE 802.15.4 standard UWB PHY. This device is the first in a family of parts.

The resulting silicon has a wide range of standards-based applications for both Real Time Location Systems (RTLS) and Ultra Low Power Wireless Transceivers in areas as diverse as manufacturing, healthcare, lighting, security, transport, and inventory and supply-chain management.

For further information on this or any other Decawave product contact a sales representative as follows: -

Decawave Ltd,
Adelaide Chambers,
Peter Street,
Dublin D08 T6YA,
Ireland.

mailto:sales@decawave.com
http://www.decawave.com/