

# **DW1000 DEVICE DRIVER APPLICATION PROGRAMMING INTERFACE (API) GUIDE**

**USING API FUNCTIONS TO  
CONFIGURE AND PROGRAM THE  
DW1000 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

Copyright © 2013 DecaWave Ltd

## 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

## Table of Contents

<b>1</b>	<b>INTRODUCTION AND OVERVIEW .....</b>	<b>6</b>
<b>2</b>	<b>GENERAL FRAMEWORK .....</b>	<b>7</b>
<b>3</b>	<b>TYPICAL SYSTEM START-UP .....</b>	<b>9</b>
<b>4</b>	<b>INTERRUPT HANDLING .....</b>	<b>10</b>
<b>5</b>	<b>API FUNCTION DESCRIPTIONS .....</b>	<b>11</b>
5.1	DWT_READDEVID .....	11
5.2	DWT_GETPARTID.....	12
5.3	DWT_GETLOTID .....	12
5.4	DWT_SOFTRESET.....	13
5.5	DWT_INITIALISE .....	13
5.6	DWT_CONFIGURE.....	14
5.7	DWT_GETOTPTXPOWER.....	18
5.8	DWT_CONFIGURETXRF .....	19
5.9	DWT_SETSMARTTXPOWER .....	21
5.10	DWT_READANTENNADELAY .....	21
5.11	DWT_SETRXANTENNADELAY .....	22
5.12	DWT_SETTXANTENNADELAY .....	22
5.13	DWT_WRITETXDATA .....	23
5.14	DWT_WRITETXCTRL.....	24
5.15	DWT_STARTTX.....	24
5.16	DWT_SETDELAYEDTRXTIME .....	26
5.17	DWT_READTXTIMESTAMP .....	27
5.18	DWT_READTXTIMESTAMPHI32 .....	27
5.19	DWT_READRXTIMESTAMP .....	28
5.20	DWT_READRXTIMESTAMPHI32 .....	28
5.21	DWT_READSYSTEMTIMESTAMP.....	29
5.22	DWT_READSYSTEMTIMESTAMPHI32 .....	29
5.23	DWT_FORCETRXOFF.....	29
5.24	DWT_SYNCRXBUPTRS .....	30
5.25	DWT_RXENABLE.....	30
5.26	DWT_SETRXMODE .....	31
5.27	DWT_SETAUTORXREENABLE .....	32
5.28	DWT_SETDBLRXBUFFMODE .....	32
5.29	DWT_SETRXTIMEOUT .....	33
5.30	DWT_SETPREAMBLEDETECTTIMEOUT .....	33
5.31	DWT_CONFIGURESLEEPCNT .....	34
5.32	DWT_CALIBRATESLEEPCNT .....	35
5.33	DWT_CONFIGURESLEEP .....	35
5.34	DWT_ENTERSLEEP .....	38
5.35	DWT_ENTERSLEEPAFTERTX.....	38
5.36	DWT_SPICSWAKEUP.....	39
5.37	DWT_SETCALLBACKS .....	40
5.38	DWT_SETINTERRUPT .....	41
5.39	DWT_ISR .....	42
5.40	DWT_SETPANID .....	44

5.41	DWT_SETADDRESS16.....	45
5.42	DWT_SETEUI .....	45
5.43	DWT_GETEUI.....	46
5.44	DWT_ENABLEFRAMEFILTER .....	46
5.45	DWT_ENABLEAUTOACK .....	47
5.46	DWT_SETRXAFTERTXDELAY.....	47
5.47	DWT_READRXDATA.....	48
5.48	DWT_READACCDATA.....	48
5.49	DWT_READDIAGNOSTICS .....	49
5.50	DWT_CONFIGEVENTCOUNTERS .....	51
5.51	DWT_READEVENTCOUNTERS .....	51
5.52	DWT_READTEMPVBAT .....	53
5.53	DWT_READWAKEUPTEMP .....	53
5.54	DWT_READWAKEUPVBAT.....	54
5.55	DWT_OTPWRITEANDVERIFY.....	54
5.56	DWT_GETRANGEBIAS .....	56
5.57	DWT_SETLEDS .....	57
5.58	DWT_XTALTRIM .....	57
5.59	DWT_CONFIGCWMODE .....	57
5.60	DWT_CONFIGCONTINUOUSFRAMEMODE.....	59
5.61	SPI DRIVER FUNCTIONS .....	61
5.61.1	<i>writetospi</i> .....	61
5.61.2	<i>readfromspi</i> .....	62
5.62	MUTUAL-EXCLUSION API FUNCTIONS .....	63
5.62.1	<i>decamutexon</i> .....	63
5.62.2	<i>decamutexoff</i> .....	64
5.63	SUBSIDIARY FUNCTIONS .....	64
5.63.1	<i>dwt_writetodevice</i> .....	64
5.63.2	<i>dwt_readfromdevice</i> .....	65
5.63.3	<i>dwt_read32bitreg</i> .....	65
5.63.4	<i>dwt_read32bitoffsetreg</i> .....	65
5.63.5	<i>dwt_write32bitreg</i> .....	65
5.63.6	<i>dwt_write32bitoffsetreg</i> .....	65
5.63.7	<i>dwt_read16bitoffsetreg</i> .....	65
5.63.8	<i>dwt_write16bitoffsetreg</i> .....	65
6	APPENDIX 1 – BIBLIOGRAPHY:.....	66
7	ABOUT DECAWAVE .....	67

## List of Tables

TABLE 1: CONFIG PARAMETER TO DWT_INITIALISE() FUNCTION .....	14
TABLE 2: DW1000 SUPPORTED UWB CHANNELS AND RECOMMENDED PREAMBLE CODES .....	17
TABLE 3: RECOMMENDED PREAMBLE LENGTHS.....	17
TABLE 4: RECOMMENDED PAC SIZE .....	18
TABLE 5: PGDLY RECOMMENDED VALUES.....	19
TABLE 6: TX POWER RECOMMENDED VALUES (WHEN SMART POWER IS DISABLED) .....	20
TABLE 7: TX POWER RECOMMENDED VALUES (WHEN SMART POWER IS ENABLED).....	20
TABLE 8: MODE PARAMETER TO DWT_STARTTX() FUNCTION.....	25
TABLE 9: MODE PARAMETER TO DWT_SETRXMODE() FUNCTION.....	31

TABLE 10: BITMASK VALUES FOR DWT_CONFIGURESLEEP() <i>MODE</i> BIT MASK.....	36
TABLE 11: BITMASK VALUES FOR DWT_CONFIGURESLEEP() <i>WAKE</i> BIT MASK.....	36
TABLE 12: BITMASK VALUES FOR DWT_SETINTERRUPT() INTERRUPT MASK ENABLING/DISABLING .....	41
TABLE 13: LIST OF TX EVENTS HANDLED BY THE <i>DWT_ISR()</i> FUNCTION AND SIGNALLED IN TX CALL-BACK.....	42
TABLE 14: LIST OF RX EVENTS HANDLED BY THE <i>DWT_ISR()</i> FUNCTION AND SIGNALLED IN RX CALL-BACK.....	42
TABLE 15: BITMASK VALUES FOR FRAME FILTERING ENABLING/DISABLING.....	47
TABLE 16: OTP MEMORY MAP .....	55

## List of Figures

FIGURE 1: GENERAL SOFTWARE FRAMEWORK OF DW1000 DEVICE DRIVER .....	7
FIGURE 2: TYPICAL FLOW OF INITIALISATION.....	9
FIGURE 3: INTERRUPT HANDLING .....	10
FIGURE 4: INTERRUPT HANDLING .....	44

# 1 INTRODUCTION AND OVERVIEW

The DW1000 IC is a radio transceiver IC implementing the UWB physical layer defined in IEEE 802.15.4-2011 standard [3]. For more details of this device the reader is referred to:

- The DW1000 data sheet [1]
- The DW1000 user manual [2]

This document, “*DW1000 Device Driver - Application Programming Interface (API) Guide*” is a guide to the device driver software developed by DecaWave to drive DecaWave’s DW1000 UWB radio transceiver IC.

The device driver is essentially a set of low-level functions providing a means to exercise the main features of the DW1000 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<sup>1</sup>. The source code employs the C programming language.

The DW1000 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: **"DW1000 Device Driver Version 02.05.00"**

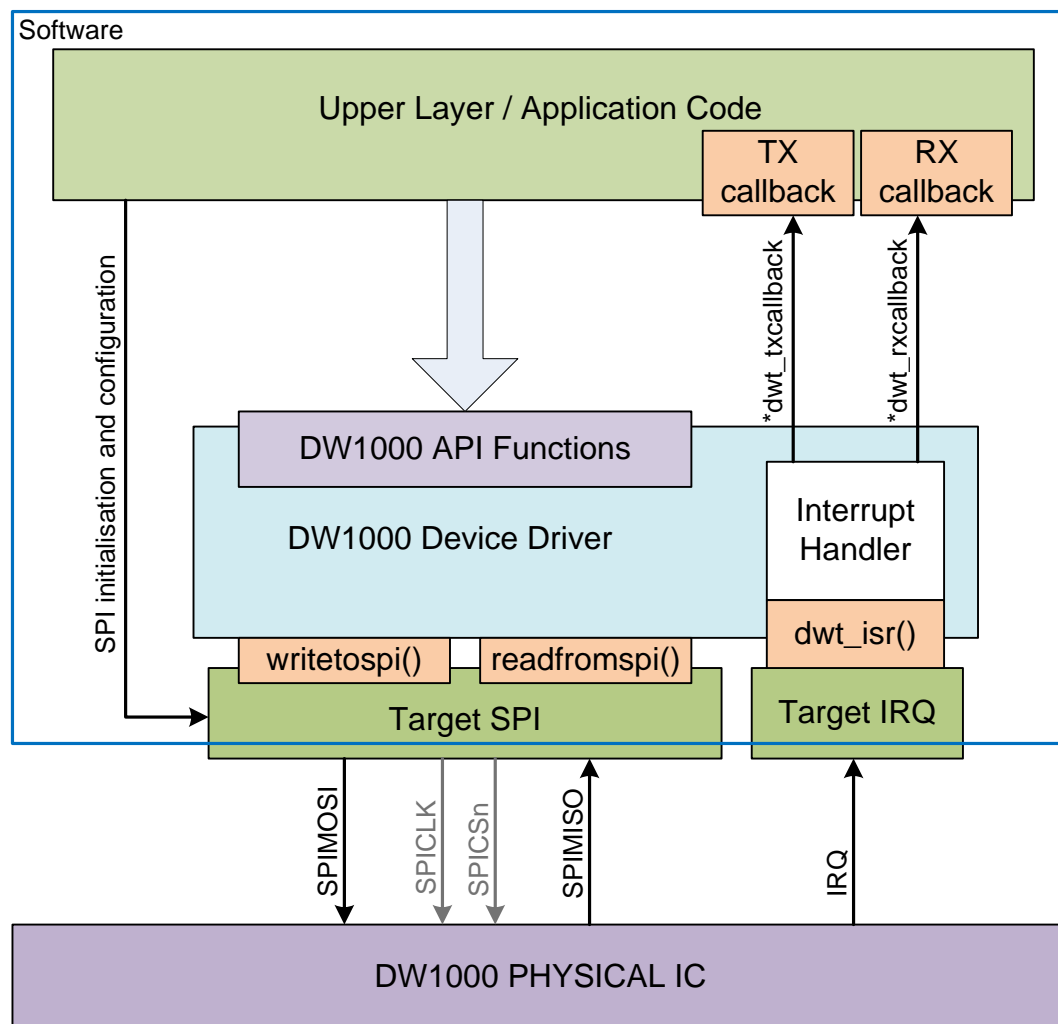
The device driver version information may be found in source code file “[deca\\_version.h](#)”.

---

<sup>1</sup> Since the DW1000 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 DW1000 device driver. The DW1000 device driver controls the DW1000 IC through its SPI interface. The DW1000 device driver abstracts the target SPI device by calling it through generic functions `writetospi()` and `readfromspi()`. In porting the DW1000 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 DW1000 device driver.



**Figure 1: General software framework of DW1000 device driver**

The control of the DW1000 IC through the DW1000 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 DW1000 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 DW1000 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 DW1000's IRQ and will as a result call the DW1000 device driver's interrupt handling function `dwt_irq()` to process the interrupt.

The DW1000 device driver's `dwt_irq()` function, processes the DW1000 interrupts and calls TX and RX call-back functions in the upper layer application code. This is done via function pointers `*dwt_txcallback()` and `*dwt_rxcallback()` 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 DW1000 without employing interrupts. In this case the background loop can periodically call the DW1000 device driver's `dwt_isr()` function, which will poll the DW1000 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 DW1000 device driver's interrupt handler accesses the DW1000 IC through the `writetospi()` and `readfromspi()` functions, and, it is generally expected that the call-back functions will also access the DW1000 IC through the DW1000 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 DW1000 from being recognised while the *background* SPI access is in progress.

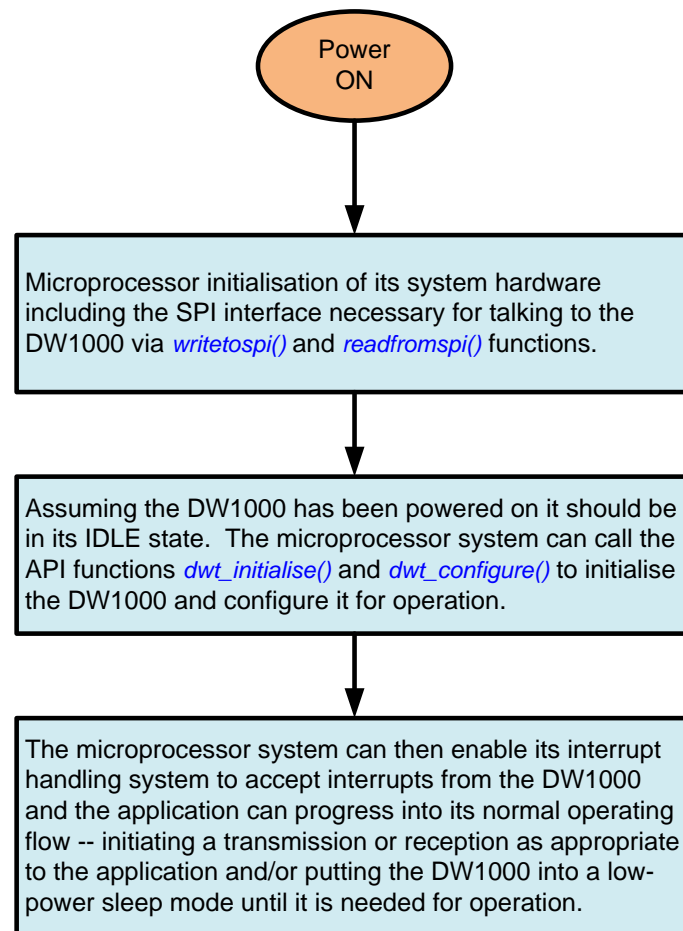
Examples of be `decamutexon()` and `decamutexoff()` within the `writetospi()` and `readfromspi()` functions found in source code file "`deca_irq.c`" and the definitions of the `writetospi()` and `readfromspi()` functions in "`deca_spi.c`" source file.

Other than the provisions for interrupt handling, the DW1000 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.



### 3 TYPICAL SYSTEM START-UP

Figure 2 shows the typical flow of initialisation of the DW1000 in a microprocessor system.

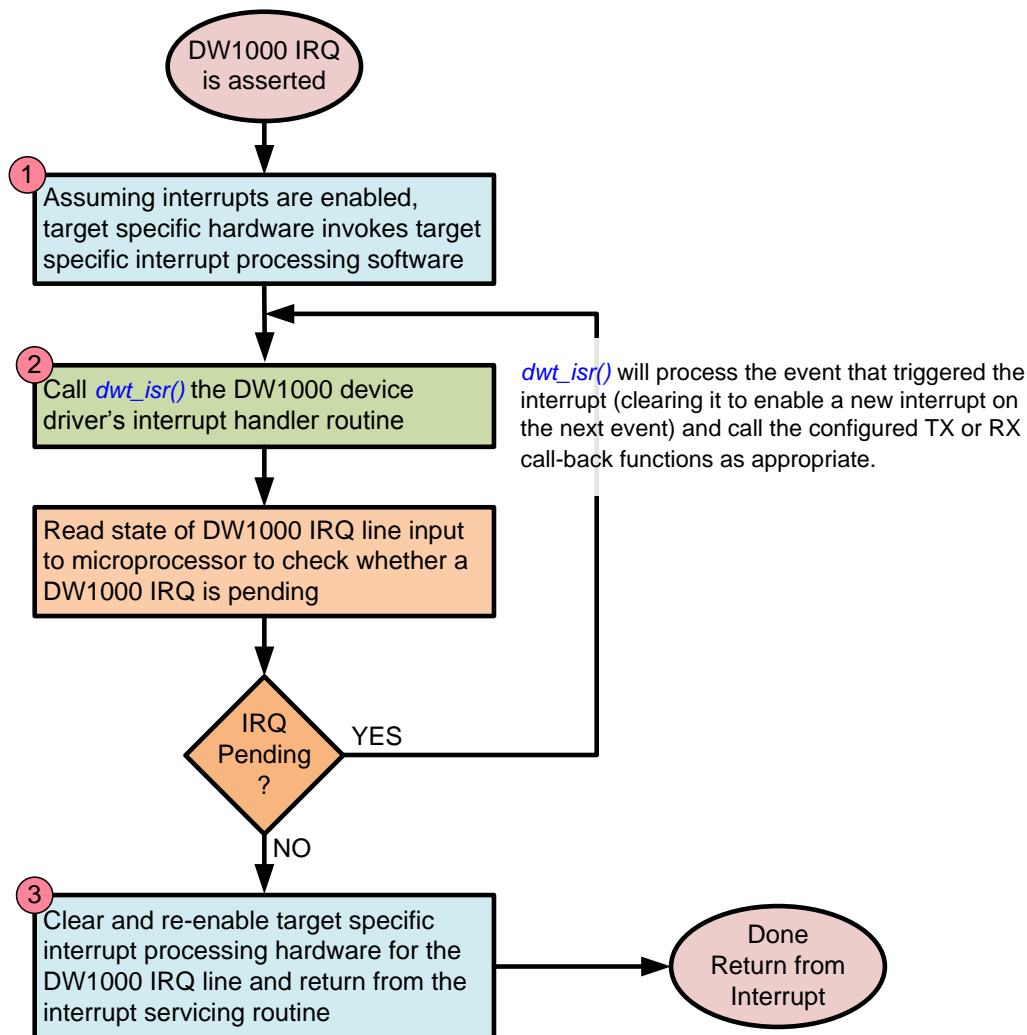


**Figure 2: Typical flow of initialisation**

## 4 INTERRUPT HANDLING

Figure 3 shows how the DW1000 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 DW1000 device driver's interrupt handler service routine, the

`dwt_isr()` API function, which processes the event that triggered the interrupt.



**Figure 3: Interrupt handling**

The flow shown above, with the rechecking of DW1000 to check for 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 4.

## 5 API FUNCTION DESCRIPTIONS

This section describes DW1000 device driver's API function calls. The API functions are provided to aid developers in driving the DW1000 (DecaWave's ScenSor IEEE 802.15.4 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 DW1000 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 *dwt\_readdevId*

```
uint32 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 it to the caller device type and revision information. 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	32-bit device ID value, e.g. for DW1000 the device ID is 0xDECA0130.

**Notes:**

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

**Example code:**

```
uint32 devID = dwt_readdevId();
```

## 5.2 *dwt\_getpartid*

```
uint32 dwt_getpartid(void);
```

This function returns the part identifier as programmed in the factory during device test and qualification.

**Parameters:**

none

**Return Parameters:**

type	description
uint32	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 partID = dwt_getpartid();
```

## 5.3 *dwt\_getlotid*

```
uint32 dwt_getlotid(void);
```

This function returns the lot identifier as programmed in the factory during device test and qualification.

**Parameters:**

none

**Return Parameters:**

type	description
uint32	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 lotID = dwt_getlotid();
```

## 5.4 *dwt\_softreset*

```
void dwt_softreset(void);
```

This function performs a software controlled reset of DW1000. All of the IC configuration will be reset back to default.

### Parameters:

none

### 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 DW1000 will be in the IDLE state, and all of the registers will have default values. Any programmed values in the into the always on (AON) low-power configuration array store will also be cleared.

Note: DW1000 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. **It should never be driven high.**

## 5.5 *dwt\_initialise*

```
int dwt_initialise(uint16 config);
```

This function initialises the DW1000 transceiver and sets up values in an internal static data structure used within the device driver functions, which is private data for use in the device driver implementation. The *dwt\_initialise()* function also kicks off loading of LDE microcode, if *config* parameter has DWT\_LOADUCODE bit set, (from the IC ROM into its runtime location) so that it is available to for future receiver use. The LDE algorithm is responsible for generating an accurate RX timestamp and calculating some signal quality statistics related to the received packet. Other *config* parameter bits are defined below which enable loading of calibration data from one-time-programmable (OTP) memory.

### Parameters:

type	name	description
int	config	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_DECA_SUCCESS = 0 or DWT_DECA_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. DW1000 32-bit device ID value is 0xDECA0130). Then it does some initial once only device configurations (e.g. configures the clocks for normal TX/RX functionality) needed for use. It also reads any calibration data from OTP, which can then be used by the application to configure the device.

If the DWT\_DECA\_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_LOADNONE	0x0	Do not load any data from OTP.
DWT_LOADXTALTRIM	0x1	Load the XTAL trim value stored in OTP. This value will be programmed as part of device calibration.
DWT_LOADANTDLY	0x2	Load antenna delay values into the local <code>dw1000local</code> structure. These antenna delay values will be programmed into OTP as part of device calibration. The read values will be stored in the local structure and can be used by the application to configure the device's antenna delay.
DWT_LOADTXCONFIG	0x4	Load the transmit power calibration values, into the local <code>dw1000local</code> structure. These will be programmed as part of device calibration. The read values should be used as part of device channel configuration to configure the transmit power with a call to <a href="#">dwt_configuretxrf()</a> function.
DWT_LOADUCODE	0x800	Loads LDE microcode (from the IC ROM into its runtime location) so that it is available to for future receiver use. The LDE algorithm is responsible for generating an accurate RX timestamp and calculating some signal quality statistics related to the received packet.

#### Notes:

For more details of the OTP memory programming please refer to section [5.55](#) [dwt\\_otpwritewandverify\(\)](#). 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 DW1000's ability to function normally.

## 5.6 dwt\_configure

```
int dwt_configure(dwt_config_t *config, uint16 useotp);
```

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\\_configuretxrf\(\)](#) for setting certain TX parameters. This is described in section 5.8 below). Also the `useotp` parameter specifies if to use any of the OTP calibration data that was read as part of the device initialisation (see call [dwt\\_initialise\(\)](#) function).

**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.
uint8	useotp	This is a bitmask which specifies which configuration already loaded from OTP as part of initialisation should be used for device configuration. Only DWT_LOADXTALTRIM and DWT_LOADXTALTRIM values (from Table 1) can be used in this function. The OTP values must be read from OTP as part of a call <a href="#">dwt_initialise()</a> function.

```
typedef struct
{
    uint8 chan ;                //!< channel number {1, 2, 3, 4, 5, 7 }
    uint8 prf ;                 //!< Pulse Repetition Frequency
                                //!< {DWT_PRF_16M or DWT_PRF_64M}
    uint8 txPreambleLength;    //!< DWT_PLEN_64..DWT_PLEN_4096
    uint8 rxPAC ;              //!< Acquisition Chunk Size (Relates to RX
                                //!< preamble length)
    uint8 txCode ;              //!< TX preamble code
    uint8 rxCode ;              //!< RX preamble code
    uint8 nsSFD ;              //!< Boolean, use non-std SFD for better
                                //!< performance
    uint8 dataRate ;            //!< Data Rate {DWT_BR_110K, DWT_BR_850K or
                                //!< DWT_BR_6M8}
    uint8 phrMode ;            //!< PHR mode:
                                //!< // 0x0 - standard DWT_PHRMODE_STD
                                //!< // 0x3 - extended frames
                                DWT_PHRMODE_EXT
    uint16 sfdTO ;              //!< SFD timeout value (in symbols)
} dwt_config_t ;
```

**Return Parameters:**

type	description
int	Return values can be either DWT_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes:**

The [dwt\\_configure\(\)](#) function should be to configure the DW1000 channel (TX/RX) parameters before receiver enable or before issuing a start transmission command. It can be called again to change configurations as needed.

The [config](#) parameter points to a [dwt\\_config\\_t](#) structure that has various fields to select and configure different parameters within the DW1000. The fields of the [dwt\\_config\\_t](#) structure are identified individually described below:

Fields	Description of fields within the <i>dwt_config_t</i> structure
<i>chan</i>	The <i>chan</i> parameter sets the UWB channel number, (defining the centre frequency and bandwidth). The supported channels are 1, 2, 3, 4, 5, and 7.
<i>txCode</i> and <i>rxCode</i>	The <i>txCode</i> and <i>rxCode</i> 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 DW1000, the selected preamble code should follow the rules of IEEE 802.15.4-2011 UWB with respect to which codes are allowed in the particular channel and PRF configuration, this is shown in Table 2 below.
<i>prf</i>	The <i>prf</i> parameter allows selection of the nominal PRF (pulse repetition frequency) being used by the receiver which can be either 16 MHz or 64 MHz, via the symbolic definitions DWT_PRF_16M and DWT_PRF_64M.
<i>nsSFD</i>	The <i>nsSFD</i> parameter enables the use of an alternate non-standard SFD (Start Frame Delimiter) sequence, which DecaWave has found to be more robust than that specified in the IEEE 802.15.4 standard, and which therefore gives improved performance.
<i>dataRate</i>	The <i>dataRate</i> parameter specifies the data rate to be one of 110kbps, 850kbps or 6800kbps, via symbolic definitions DWT_BR_110K, DWT_BR_850K and DWT_BR_6M8.
<i>txPreambleLength</i>	The <i>txPreambleLength</i> 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. Table 3 gives recommended preamble sequence lengths to use depending on the data rate.
<i>rxPAC</i>	The <i>rxPAC</i> parameter specifies preamble acquisition chunk size used (DWT_PAC8, DWT_PAC16, DWT_PAC32 or DWT_PAC64). Table 4 below gives the recommended PAC size to use in the receiver depending on the preamble length being used in the transmitter.
<i>phrMode</i>	The <i>phrMode</i> 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.
<i>sfdTO</i>	The <i>sfdTO</i> 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 ( <i>DWT_SFDTOK_DEF</i> ) will be used).



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 return DWT\_DECA\_ERROR for error. If DWT\_API\_ERROR\_CHECK switch is not defined the code always returns DWT\_DECA\_SUCCESS, meaning success.

NOTE: SFD timeout cannot be set to 0; if 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 1024 preamble length, the corresponding receiver should have `sfdTO` parameter set to 1089 (1024+64+1).

**Table 2: DW1000 supported UWB channels and recommended preamble codes**

Channel number	Preamble Codes (16 MHz PRF)	Preamble Codes (64 MHz PRF)
1	1, 2	9, 10, 11, 12
2	3, 4	9, 10, 11, 12
3	5, 6	9, 10, 11, 12
4	7, 8	17, 18, 19, 20
5	3, 4	9, 10, 11, 12
7	7, 8	17, 18, 19, 20

The preamble sequence used at all data rates is the same, 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	64 or 128 or 256
850kbps	256 or 512 or 1024
110kbps	2048 or 4096

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 64 will never receive frames with just 64 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	Recommended PAC size
64	8
128	8
256	16
512	16
1024	32
1536	64
2048	64
4096	64

See also: [dwt\\_configuretxrf\(\)](#) for setting certain TX parameters  
[dwt\\_setrxmode\(\)](#) for setting certain RX (preamble hunt) operating mode.

## 5.7 dwt\_getotptxpower

```
uint32 dwt_getotptxpower(uint8 prf, uint8 chan);
```

This function reads the transmit power value from local structure, the value will only be valid if DWT\_LOADTXCONFIG bit was set in the [config](#) parameter of [dwt\\_initialise\(\)](#) function. The value returned will be the transmit power value saved in the OTP for the given PRF and channel. The value returned is the calibrated power value when the transmit smart power is enabled.

Parameters:

type	name	Description
uint8	prf	The value is either DWT_PRF_16M or DWT_PRF_64M.
uint8	chan	The channel number. The supported channels are 1, 2, 3, 4, 5, and 7.

Return Parameters:

type	Description
uint32	The tx power value. This is the calibrated power value for a given channel and PRF, this is only valid if DWT_LOADTXCONFIG bit was set in the <a href="#">config</a> parameter of <a href="#">dwt_initialise()</a> function.

Notes:

This function is used to read the calibrated txpower register value for a given PRF and channel. It is only valid if DWT\_LOADTXCONFIG bit was set in the [config](#) parameter of [dwt\\_initialise\(\)](#) function. This value should be used when configuring the TX parameters in the [dwt\\_configuretxrf\(\)](#) function. If the smart power is disabled then the low byte needs to be copied into the higher bytes and the resultant value used in the call to the [dwt\\_configuretxrf\(\)](#) function. E.g. If the value returned is 0x15355575,

then to configure the tx power in smart mode the `config.power` is set to 0x15355575 before the call to `dwt_configuretxrf()` function and to configure the tx power with smart mode disabled the `config.power` is set to 0x75757575 before the call to `dwt_configuretxrf()` function.

For more details of the OTP memory programming please refer to section 5.55

`dwt_otpwritandverify()`. 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 DW1000's ability to function normally.

## 5.8 dwt\_configuretxrf

```
void dwt_configuretxrf(dwt_txconfig_t *config);
```

The `dwt_configuretxrf()` function is responsible for setting up the transmit RF configuration parameters. One is the pulse generator delay value which sets the width of transmitted pulses effectively setting the output bandwidth. The other value is the transmit output power setting.

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  PGdly;           //Pulse generator delay value
    uint32 power;           //the TX power - 4 bytes
} dwt_txconfig_t ;
```

Return Parameters:

none

Notes:

This function can be called any time and it will configure the DW1000 spectrum parameters. The `config` parameter points to a `dwt_txconfig_t` structure (shown below) with fields to configure the pulse generator delay (`PGdly`) and TX power (`power`). Recommended values for `PGdly` are given in Table 5 below.

**Table 5: PGdly recommended values**

TX Channel	recommended PGdly value
1	0xC9
2	0xC2
3	0xC5
4	0x95
5	0xC0

TX Channel	recommended PGdly value
7	0x93

**Table 6: TX power recommended values (when smart power is **disabled**)**

TX Channel	recommended TX power value 16 MHz	recommended TX power value 64 MHz
1	0x75757575	0x67676767
2	0x75757575	0x67676767
3	0x6F6F6F6F	0x8B8B8B8B
4	0x5F5F5F5F	0x9A9A9A9A
5	0x48484848	0x85858585
7	0x92929292	0xD1D1D1D1

Table 6 above includes the recommended TX power spectrum values, for use in the case of smart power being disabled using the [\*dwt\\_setsmarttxpower\(\)\*](#) API function, while Table 7 below applies when smart power is enabled.

**Table 7: TX power recommended values (when smart power is **enabled**)**

TX Channel	recommended TX power value 16 MHz	recommended TX power value 64 MHz
1	0x15355575	0x07274767
2	0x15355575	0x07274767
3	0x0F2F4F6F	0x2B4B6B8B
4	0x1F1F3F5F	0x3A5A7A9A
5	0x0E082848	0x25456585
7	0x32527292	0x5171B1D1

NB: The values in Table 6 and Table 7 have been chosen to suit DecaWave's EVB1000 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 place in OTP memory where the calibrated values can be stored and then read as part of device initialisation (see function [\*dwt\\_initalise\(\)\*](#)). Please consult with DecaWave's applications support team for details of transmit power calibration procedures and considerations.

## 5.9 *dwt\_setsmarttxpower*

```
void dwt_setsmarttxpower(int enable);
```

This function enables or disables smart TX power functionality of DW1000.

**Parameters :**

type	name	description
int	enable	1 to enable, 0 to disable the smart TX power feature.

**Return Parameters:**

none

**Notes :**

This function enables or disables smart TX power functionality.

Regional power output regulations typically specify the transmit power limit as -41 dBm in each 1 MHz of channel bandwidth, and generally measure this using a 1 ms dwell time in each 1 MHz segment. When sending short frames at 6.8 Mbps it is possible for a single frame to be sent in a fraction of a millisecond, and then as long as the transmitter does not transmit again within that same millisecond the power of that transmission can be increased and still comply with the regulations. This power increase will increase the transmission range. To make use of this the DW1000 includes functionality we call "Smart Transmit Power Gating" which automatically boosts the TX power for a transmission when the frame is short.

Smart TX power control acts at the 6.8 Mbps data rate. When sending short data frames at this rate (and providing that the frame transmission rate is < 1 frame per millisecond) it is possible to increase the transmit power and still remain within regulatory power limits which are typically specified as average power per millisecond.

NB: When enabling/disabling smart TX power, the TX power values programmed via the [dwt\\_configuretxrf\(\)](#) function also need to be set accordingly. When smart TX power is disabled the appropriate value from Table 6 should be used, and when smart TX power is enabled the appropriate value from Table 7 should be used. The values in Table 6 and 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. Please consult with DecaWave's applications support team for details of transmit power calibration procedures and considerations.

## 5.10 *dwt\_readantennadelay*

```
uint16 dwt_readantennadelay(uint8 prf);
```

This function reads the antenna delay from local structure, the value will only be valid if DWT\_LOADANTDLY bit was set in the *config* parameter of [dwt\\_initialise\(\)](#) function. The value returned will be the total (tx + rx) value saved in the OTP memory for the given PRF. The value returned is in DWT\_TIME\_UNITS (15.65 picoseconds ticks).

**Parameters :**

type	name	description
uint8	prf	The value is either DWT_PRF_16M or DWT_PRF_64M.

**Return Parameters:**

type	description
uint16	Antenna delay value in DWT_TIME_UNITS (15.65 picoseconds ticks).

**Notes:**

This function is used to read the total antenna delay for a given PRF.

**5.11 dwt\_setrxantennadelay**

```
void dwt_setrxantennadelay(uint16 antennaDelay);
```

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	antennaDelay	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.

**5.12 dwt\_settxantennadelay**

```
void dwt_settxantennadelay(uint16 antennaDelay);
```

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:**

type	name	description
------	------	-------------

type	name	description
uint16	antennaDelay	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.

**5.13 dwt\_writetxdata**

```
int dwt_writetxdata(uint16 txFrameLength, uint8 *txFrameBytes, uint16 txBufferOffset) ;
```

This function is used to write the TX message data into the DW1000 TX buffer.

**Parameters:**

type	name	description
uint16	txFrameLength	This is the total frame length, including the two byte CRC.
uint8*	txFrameBytes	Pointer to the user's buffer containing the data to send.
uint16	txBufferOffset	This specifies an offset in the DW1000's TX Buffer at which to start writing data.

**Return Parameters:**

type	description
int	Return values can be either DWT_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes:**

This function writes *txFrameLength* - 2 bytes from memory pointed to by the *txFrameBytes* parameter into the DW1000 IC's transmit data buffer, starting at particular offset (*txBufferOffset*). During transmission the DW1000 will automatically add the 2 CRC bytes to complete the TX frame to its full *txFrameLength*. NB: If the length + the offset is greater than the IC TX buffer length 1024 the function will return DWT\_DECA\_ERROR.

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\\_writetxdata\(\)](#) function does not error check the *txFrameLength* input parameter unless the DWT\_API\_ERROR\_CHECK code switch is defined. If this is defined it will return DWT\_DECA\_ERROR if length is longer than the permitted frame length for the configured [phrMode](#) setting.

**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(frameLength,DataBufferPtr,0); // write the frame data at
                                              // offset 0
dwt_writetxfctrl(frameLength,0);           // set the frame control
                                              // register
dwt_starttx(DWT_START_TX_IMMEDIATE);       // send the frame
```

### 5.14 dwt\_writetxfctrl

```
int dwt_writetxfctrl(uint16 txFrameLength, uint16 txBufferOffset);
```

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

Parameters:

type	name	description
uint16	txFrameLength	This is the total frame length, including the two byte CRC.
uint16	txBufferOffset	This specifies an offset in the DW1000's TX Buffer at which to start writing data.

Return Parameters:

type	description
int	Return values can be either DWT_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

Notes:

This function configures the TX frame control register parameters, namely the length of the frame and the offset in the DW1000 IC's transmit data buffer where the data starts.

The `dwt_writetxfctrl()` function does not error check the `txFrameLength` input parameter unless the `DWT_API_ERROR_CHECK` code switch is defined. If this is defined it will return -1 if length is longer than the permitted frame length for the configured `phrMode` setting.

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

### 5.15 dwt\_starttx

```
int dwt_starttx(uint8 mode);
```

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

Parameters:



type	name	description
uint8	mode	This is a bit mask defining the operation of the function, see notes and Table 8 below.

**Return Parameters:**

type	description
int	Return values can be either DWT_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes:**

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

Transmission begins immediately if the *mode* parameter is zero. When the *mode* parameter is 1 transmission begins when the system time reaches the *starttime* specified in the call to the *dwt\_setdelayedtrxtime()* function described below. The *mode* parameter, when 2 or 3, is used to turn the receiver immediately 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 DW1000 can start receiving data (e.g. ACK/response) which might come within 12 symbol times from the end of transmission. It returns 0 for success, or -1 for error.

In performing a delayed transmission, if the host microprocessor is late in invoking the *dwt\_starttx()* function, (i.e. so that the DW1000'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 8: 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 <i>starttime</i> is reached. See <i>dwt_setdelayedtrxtime()</i> .
DWT_RESPONSE_EXPECTED	0x2	Response is expected, once the frame is sent the transceiver will enter receive mode to wait for response message. See <i>dwt_setrxaftertxdelay()</i> .
DWT_START_TX_DELAYED + DWT_RESPONSE_EXPECTED	0x3	The transmitter will start sending a frame once the programmed delayed TX time is reached, see <i>dwt_setdelayedtrxtime()</i> , and once the frame is sent the transceiver will enter receive mode to wait for response message.

**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(frameLength,DataBufferPtr,0); // write the frame data at
                                              // offset 0
dwt_writetxfctrl(frameLength,0);           // set the frame control
```

```
dwt_starttx(DWT_START_TX_IMMEDIATE);           // register
                                                // send the frame
```

## 5.16 dwt\_setdelayedtrxtime

```
void dwt_setdelayedtrxtime (uint32 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	starttime	<p>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 <math>512/(499.2e6*128)</math> seconds)</p> <p>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.</p>

### 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 reached this time (minus the times it needs to send preamble etc) then the frame sending 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 read and new TX time calculated by adding 100 ms to it. The full 40-bit representation of 100ms would be 0x17CDC0000, however as the code is operating on just the high 32 bits a value of 0x17CDC00 is used. (The TX timestamp value should be read after a TX done interrupt triggers.)

```

uint32 dlyTxTime ;
dlyTxTime = dwt_readtxtimestampphi32() ;           // read last TX time
dlyTxTime = dlyTxTime + 0x17CDC00;                 // add 100ms
dwt_writetxdata(frameLength,dataBufferPtr,0);      // write the frame data at
                                                    // offset 0
dwt_writetxfctrl(frameLength,0);                   // set the frame control
                                                    // register
dwt_setdelayedtrxtime(dlyTxTime);                   // set previously calculated
                                                    // TX time
r = dwt_starttx(DWT_START_TX_DELAYED);              // send the frame at
                                                    // appropriate time

if (r != DWT_SUCCESS)
{
    // start TX was late, TX has been aborted.
    // Application should take appropriate recovery activity
}

```

### 5.17 *dwt\_readtxtimestamp*

```
void dwt_readtxtimestamp(uint8* 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\\_settxantennadelay\(\)](#) API function. The function returns a 40-bit timestamp value in the buffer passed in as the input parameter.

#### Parameters:

type	name	description
uint8*	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).

### 5.18 *dwt\_readtxtimestampphi32*

```
uint32 dwt_readtxtimestampphi32(void);
```

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

#### Parameters:

none

#### Return Parameters:

type	description
------	-------------

type	description
uint32	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).

**5.19 dwt\_readrxtimestamp**

```
void dwt_readrxtimestamp(uint8* 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*	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).

**5.20 dwt\_readrxtimestampphi32**

```
uint32 dwt_readrxtimestampphi32(void);
```

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

**Parameters:**

none

**Return Parameters:**

type	description
uint32	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).

### 5.21 *dwt\_readsystimestamp*

```
void dwt_readsystimestamp(uint8* timestamp);
```

This function returns the system time. The function returns a 40-bit value.

**Parameters :**

type	name	description
uint8*	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 to read the DW1000 system time.

### 5.22 *dwt\_readsystimestampphi32*

```
uint32 dwt_readsystimestampphi32(void);
```

This function returns the high 32-bits of the 40-bit system time.

**Parameters :**

none

**Return Parameters :**

type	description
uint32	High 32-bits of the 40-bit system timestamp.

**Notes :**

This function can be called to read the DW1000 system time.

### 5.23 *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 DW1000 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 DW1000 IC and also clears status register event flags, so that there should be no outstanding/pending events for processing.

### 5.24 dwt\_syncrxbufptrs

```
void dwt_syncrxbufptrs(void);
```

This function synchronizes RX buffer pointers. This is needed to make sure that the host and DW1000 buffer pointers are aligned before starting RX.

#### Parameters:

none

#### Return Parameters:

none

#### Notes:

The function is called as part of [dwt\\_rxenable\(\)](#) and [dwt\\_forcetrxoff\(\)](#), to make sure the buffers are synchronized as the receiver is switched off or switched on. For more information see [dwt\\_setdblrxbuffmode\(\)](#) function below.

### 5.25 dwt\_rxenable

```
int dwt_rxenable(int delayed);
```

This function turns on the receiver to wait for a receive frame. The delayed parameter allows for delayed RX where RX is turned on at the specific time, set via [dwt\\_setdelayedtrxtime\(\)](#), this is a lower power option then turning RX on immediately if we know the message will not come for a while.

#### Parameters:

type	name	description
int	delayed	If non zero this means that the receiver will be turned on when the time reaches the delayed start time as set through the <a href="#">dwt_setdelayedtrxtime()</a> function.

#### Return Parameters:

type	description
int	Return values can be either DWT_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

#### Notes:

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

In performing a delayed reception, if the host microprocessor is late in invoking the [dwt\\_rxenable\(\)](#) function, (i.e. so that the DW1000's system clock has passed the [starttime](#) specified in the call to the [dwt\\_setdelayedtrxtime\(\)](#) function and would have to complete almost a whole clock count period before the start time is reached), then the delayed reception is aborted (transceiver off) and the receiver is immediately enabled. An error flag is returned in this case indicating that the RX on was late. It is up to the application whether this behaviour is acceptable, e.g. the application may issue a [dwt\\_forcetrxoff\(\)](#) call to turn off the receiver if some other remedial action is needed other than the immediate RX enabling.

## 5.26 dwt\_setrxmode

```
void dwt_setrxmode(int mode, uint8 rxON, uint8 rxOFF);
```

This function configures different operational modes for preamble detection in the receiver, i.e. Normal mode or SNIFF. In normal mode the receiver in preamble hunt mode is continuously active looking for the preamble sequence. *SNIFF mode* is a lower power preamble hunt mode, also known as pulsed preamble detection mode (PPDM), where the receiver (RF and digital) is sequenced on and off rather than being on all the time. These on and off times are configurable via parameters to this function. Using *SNIFF mode* causes a reduction in sensitivity depending on the ratio and durations of the on and off periods.

### Parameters:

type	name	description
int	mode	See Table 9 below.
uint8	rxON	The receiver ON time in PACs (as per the <a href="#">rxPAC</a> parameter in the <a href="#">dwt_config_t</a> structure parameter to the <a href="#">dwt_configure()</a> API function call) the minimum number of PACs that should be used for the <a href="#">ppmON</a> parameter is 2.
uint8	rxOFF	The receiver OFF time in us (actually in 1.0256 $\mu$ s, (512/499.2MHz) units).

**Table 9: Mode parameter to dwt\_setrxmode() function**

Mode	Description
DWT_RX_NORMAL	normal RX – When the receiver is in preamble hunt mode it is continuously active looking for the preamble sequence
DWT_RX_SNIFF	SNIFF mode – When the receiver is in preamble hunt mode it is alternating between looking for the preamble sequence and returning to idle mode (reducing the power drain). The duty cycle of these ON and OFF states is defined by the PPM ON/OFF times.

### Return Parameters:

none

### Notes:

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

By default the DW1000 will be in Normal mode. If this is changed to SNIFF mode it will be maintained until a reset or it is re-configured by a call to this [dwt\\_setrxmode\(\)](#) function. The setting is not changed by the [dwt\\_configure\(\)](#) function.

## 5.27 [dwt\\_setautorxreenable](#)

```
void dwt_setautorxreenable (int enable);
```

This function enables automatic re-enable of the receiver. The receiver will exit receive mode only on receiver frame wait timeout or a good frame reception. Error frames are ignored and receiver is re-enabled.

**Parameters :**

type	name	description
int	enable	1 to enable, 0 to disable the auto RX re-enable feature.

**Return Parameters :**

none

**Notes :**

This function is called to configure the receiver with auto re-enable functionality. What this does depends also on whether the DW1000 is in single or double buffered mode, (see [dwt\\_setdblrxbuffmode\(\)](#) function below). By default this is disabled, and the IC will not automatically re-enable the receiver but will stop receiving and return to idle mode whenever any receive events happen. This includes receiving a frame but also failing to receive a frame because of some error condition, for example an error in the PHY header. In such cases if the host wants to re-enable the receiver it must do it explicitly. The operation when RX auto re-enable is enabled is as follows:

(a) Double-buffered mode: After a frame reception event or failure (except a frame wait timeout), the receiver will re-enable to receive another frame.

(b) Single-buffered mode: After a frame reception failure (except a frame wait timeout), the receiver will re-enable to re-attempt reception.

In double-buffered mode when automatic frame acknowledgement is enabled the receiver will be re-enabled after the ACK frame has been transmitted. See [dwt\\_enableautoack\(\)](#) function described in section 5.45 below.

## 5.28 [dwt\\_setdblrxbuffmode](#)

```
void dwt_setdblrxbuffmode (int enable);
```

This function enables double buffered receive mode. It is expected that this is only done when auto RX re-enable feature also turned on. This means when a good frame is received, the receiver will automatically re-enable and start receiving next good frame into the second receive buffer.

**Parameters :**

type	name	description
------	------	-------------



type	name	description
int	enable	1 to enable, 0 to disable the double buffer RX feature.

**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\\_rxenable\(\)\*](#) function is called.

**5.29 dwt\_setrxtimeout**

```
void dwt_setrxtimeout (uint16 time) ;
```

The [\*dwt\\_setrxtimeout\(\)\*](#) function sets the receiver to timeout (and disable) when no frame is received within the specified time (*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 1.012 us (128/499.2 MHz) units. The maximum RX timeout is ~ 65 ms.

**Parameters:**

type	name	description
uint16	time	Timeout time in micro seconds (1.012 us). If this is 0, the timeout will be disabled.

**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.

**5.30 dwt\_setpreambledetecttimeout**

```
void dwt_setpreambledetecttimeout (uint16 time);
```

This 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	time	This is the preamble detection timeout duration, if no preamble is detected within this time from the time the receiver is enabled, the receiver will be turned off. The units are PACs. A value of 0 disables

type	name	description
		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.

**5.31 dwt\_configuresleepcnt**

```
void dwt_configuresleepcnt (uint16 sleepcnt);
```

The [dwt\\_configuresleepcnt\(\)](#) function configures the sleep counter to a new value.

**Parameters:**

type	name	description
uint16	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:**

**NB: the SPI frequency has to be set to < 3MHz before a call to this function.**

The units of the [sleepcnt](#) parameter depend on the oscillating frequency of the IC's internal L-C oscillator, which is between approximately 7,000 and 13,000 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 9500 Hz then programming the [sleepcnt](#) with a value of 24 would yield a sleep time of  $24 \times 4096 \div 9500$ , which is approximately 10.35 seconds.

**Example code:**

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

```
double t;
uint32 sleep_time = 0;
uint16 lp_osc_cal = 0;
uint16 sleepTime16;

// MUST SET SPI <= 3 MHz for this calibration activity.

setspibitrate(SPI_3MHz); // target platform function to set SPI rate to 3
                          // MHz
```

```
// Measure low power oscillator frequency

lp_osc_cal = dwt_calibratesleepcnt();

// calibrate low power oscillator
// the lp_osc_cal value is number of XTAL/2 cycles in one cycle of LP OSC
// to convert into seconds (38.4 MHz/2 = 19.2 MHz (XTAL/2) => 1/19.2 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/19.2e6));
sleep_time = (int) t;
sleepTime16 = sleep_time >> 12;

dwt_configuresleepcnt(sleepTime16);          //configure sleep time

// CAN restore/increase SPI clock up to its maximum after the calibration
// activity.

setspibitrate(SPI_20MHz);                    // target platform function to set
                                              // SPI rate to 20 MHz
```

### 5.32 *dwt\_calibratesleepcnt*

```
uint16 dwt_calibratesleepcnt (void);
```

The *dwt\_calibratesleepcnt()* function calibrates the low-power oscillator. It returns the number of XTAL/2 cycles per one low-power oscillator cycle.

#### Parameters:

none

#### Return Parameters:

type	description
uint16	This is number of XTAL/2 cycles per one low-power oscillator cycle.

#### Notes:

**NB: the SPI frequency has to be set to < 3MHz before a call to this function.**

The DW1000's internal L-C oscillator has an oscillating frequency which is between approximately 7,000 and 13,000 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.

### 5.33 *dwt\_configuresleep*

```
void dwt_configuresleep(uint16 mode, uint8 wake);
```

The *dwt\_configuresleep()* function may be called to configure the activity of DW1000 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	mode	A bit mask which configures which configures the SLEEP parameters, see Table 10.
uint8	wake	A bit mask that configures the wakeup event.

**Return Parameters :**

none

**Notes :**

This function is called to configure the DW1000 sleep and on wake parameters.

**Table 10: Bitmask values for `dwt_configuresleep()` *mode* bit mask**

Event	Bit mask	Description
DWT_LOADUCODE	0x0800	Load micro-code for the LDE engine.
DWT_PRESRV_SLEEP	0x0100	Preserves sleep. When this is set to these sleep controls are not cleared upon wakeup, so that the DW1000 can return to sleep after a failed reception (say).
DWT_LOADOPSET	0x0080	On Wake-up load the receiver operating parameter. When the bit is 0 the receiver operating parameter set reverts to its power-on-reset value (the default operating parameter set) when the DW1000 wakes from SLEEP or DEEP-SLEEP.
DWT_CONFIG	0x0040	Restore saved configurations.
DWT_LOADEUI	0x0008	On Wake-up load the EUI value from OTP memory into register 0x1. The 64-bit EUI value will be stored in register 0x1 when the DW1000 wakes from DEEPSLEEP or SLEEP states.
DWT_GOTORX	0x0002	On Wake-up turn on the receiver. With this bit it is possible to make the IC transition into RX automatically as part of IC wake up.
DWT_TANDV	0x0001	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 DW1000 wakes from DEEPSLEEP or SLEEP states. The sampled temperature value may be accessed using the <a href="#"><code>dwt_readwakeuptemp()</code></a> function and, the sampled battery voltage value may be accessed using the <a href="#"><code>dwt_readwakepvbat()</code></a> function.

**Table 11: Bitmask values for `dwt_configuresleep()` *wake* bit mask**

Event	Bit mask	Description
DWT_WAKE_SLPCNT	0x8	Wake up after sleep count expires. By default this configuration is set enabling the sleep counter as a wake-up signal. Setting this configuration bit to 0 will mean that the

Event	Bit mask	Description
		sleep counter cannot awaken the DW1000 from SLEEP.
DWT_WAKE_CS	0x4	Wakeup on chip select, SPICSn, line.
DWT_WAKE_WK	0x2	Wake up on WAKEUP line.
DWT_SLP_EN	0x1	This is the sleep enable configuration bit. This needs to be set to enable DW1000 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 DW1000 from DEEPSLEEP an external pin needs to be activated for the “power-up duration” approximately 300 to 500  $\mu$ s. 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 DW1000 comes out of DEEPSLEEP) which will vary between individual DW1000 IC and will also vary with changes of battery voltage and different temperatures. To ensure the DW1000 reliably wakes up it is recommended to either apply the wakeup signal until the 500  $\mu$ s has passed, or to use the SLP2INIT 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 DW1000 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 three mechanisms to awaken the DW1000:

- By driving the WAKEUP pin (pin 23) of the DW1000 high for a period > 200  $\mu$ s
- Driving SPICSn low for a period > 200  $\mu$ s. This can be achieved by an SPI read of sufficient length write, however as SPI reading is possibly outputting data on the SPI\_MOSI line this should be held low (outputting zeros) for the duration of the read. This is to avoid accidentally initiating a spurious write to the DW1000 when DW1000 wakes
- If the DW1000 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 0x8 (+ 0x1 – to enable sleep).

#### 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 0x0940 which is a combination of parameters to load IC configurations, load LDE microcode, and preserve the sleep setting. The wake parameter value, 5, enables the sleeping with SPICSn as the wakeup signal.

```
dwt_configuresleep(0x0940, 0x5); //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 200us this
// depends on SPI speed - see also dwt_spicswakeup() function
```

### 5.34 *dwt\_entersleep*

```
void dwt_entersleep(void);
```

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 :**

none

**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 on the [dwt\\_configuresleep\(\)](#) function).

### 5.35 *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 DW1000 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(0x0940, 0x5);           //configure the on-wake parameters
                                           //(upload the IC config settings)

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

dwt_setinterrupt(DWT_INT_TFRS, 0);         //disable TX interrupt

// 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);          // set the frame control register

dwt_starttx(DWT_START_TX_IMMEDIATE);       // send the frame immediately

// when TX completes the DW1000 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 200us this
// depends on SPI speed - see also dwt_spicswakeup() function
```

### 5.36 dwt\_spicswakeup

```
void dwt_spicswakeup (uint8 *buff, uint16 length);
```

The [dwt\\_spicswakeup\(\)](#) function uses an SPI read to wake up the DW1000 from SLEEP or DEEPSLEEP.

Parameters:

type	name	description
uint8*	buff	This is the pointer to a buffer where the data from SPI read will be read into.
uint16	length	This is the length of the input buffer.

Return Parameters:

none

Notes:

When the DW1000 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 > 200  $\mu$ s.

See example code below.

**Example code:**

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

```
dwt_configuresleep(0x0940, 0x5); //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 200us. This depends
// on SPI speed
```

**5.37 dwt\_setcallbacks**

```
void dwt_setcallbacks(void (*txcallback)(const dwt_callback_data_t *), void (*rxcallback)(const
dwt_callback_data_t *));
```

This function is used to configure the TX/RX callback function pointers. These callback functions will be called when TX or RX events happen and the dwt\_isr() is called to handle them (described below).

**Parameters:**

type	name	description
void *	txcallback()	This is the function pointer for the txcallback function. The callback function takes one parameter of dwt_callback_data_t pointer type.
void *	rxcallback()	This is the function pointer for the rxcallback function. The callback function takes one parameter of dwt_callback_data_t pointer type.

```
typedef struct{

    uint8 event;
    uint8 aatset;
    uint16 datalength;

}dwt_callback_data_t;
```

**Return Parameters:**

none

**Notes:**

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

Fields	Description of fields within the <i>dwt_callback_data_t</i> structure
<i>event</i>	The <i>event</i> parameter notifies the application which event triggered the interrupt. The list of events is shown in Table 13 and Table 14 below.



Fields	Description of fields within the <i>dwt_callback_data_t</i> structure
<i>aatset</i>	The <i>aatset</i> parameter notifies the application that the received frame has ACK request bit set. Value of 1 means that the ACK has been requested.
<i>datalength</i>	The <i>datalength</i> parameter specifies the length of the received frame.

For more detailed information on interrupt events, see

*dwt\_isr()* function description below.

### 5.38 *dwt\_setinterrupt*

```
void dwt_setinterrupt( uint32 bitmask, uint8 enable);
```

This function sets the events which will generate an interrupt. Here are the main events that can be enabled:

**Parameters :**

type	name	description
uint32	bitmask	This is the bitmask of the events that will generate the DW1000 interrupt, see Table 12.
uint8	enable	When the enable parameter is set to 1 the function enables the interrupt mask bits (specified in the bitmask parameter) allowing them to cause interrupts, otherwise the selected interrupt mask bits are cleared disallowing them to cause interrupts.  To disable particular interrupt or a set of interrupts enable needs to be set to 0.

**Return Parameters:**

none

**Notes:**

This function is called to enable/set events which are going to generate interrupts.

For the transmitter it is sufficient to enable the SY\_STAT\_TFRS event which will trigger when a frame has been sent, and for the receiver it is sufficient to enable the good frame reception event and also any error events which will disable the receiver.

**Table 12: Bitmask values for *dwt\_setinterrupt()* interrupt mask enabling/disabling**

Event	Bit mask	Description
DWT_INT_TFRS	0x00000080	Transmit Frame Sent: This is set when the transmitter has completed the sending of a frame.
DWT_INT_RPHE	0x00001000	Receiver PHY Header Error: Reception completed, Frame Error
DWT_INT_RFCG	0x00004000	Receiver FCS Good: The CRC check has matched the transmitted CRC, frame should be good

Event	Bit mask	Description
DWT_INT_RFCE	0x00008000	Receiver FCS Error: The CRC check has not matched the transmitted CRC, frame has some error
DWT_INT_RFSL	0x00010000	Receiver Frame Sync Loss: The RX lost signal before frame was received, indicates excessive Reed Solomon decoder errors
DWT_INT_RFTO	0x00020000	Receiver Frame Wait Timeout: The RX_FWTO time period expired without a Frame RX.
DWT_INT_SFDT	0x04000000	SFD Timeout
DWT_INT_RXPTO	0x00200000	Preamble detection timeout
DWT_INT_ARFE	0x20000000	ARFE - frame rejection status

### 5.39 *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 DW1000 – 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 DW1000 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\_isr()* function reads the DW1000 status register and recognises the following events:

**Table 13: List of TX events handled by the *dwt\_isr()* function and signalled in TX call-back**

Event	Activity
Transmit Frame Sent	It clears the event and signals “Frame Sent” (DWT_SIG_TX_DONE) to the application so that the transmit time stamp can be read and appropriate action taken.

**Table 14: List of RX events handled by the *dwt\_isr()* function and signalled in RX call-back**

Event	Activity
Frame Received	Signals “Frame Received” (DWT_SIG_RX_OKAY) 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.  If a frame with ACK request bit has been received then the “ACK request” is signalled ( <i>aatset</i> field of <i>dwt_callback_data_t</i> structure will be set).
Receive wait timeout	Signals “Receive timeout” (DWT_SIG_RX_TIMEOUT).
SFD timeout	Signals “SFD timeout” (DWT_SIG_RX_SFDTIMEOUT).
RX Preamble Timeout	Signals “RXPTO timeout” (DWT_SIG_RX_PTOTIMEOUT).
Receiver PHY Header Error	Signals “PHY Header Error” (DWT_SIG_RX_PHR_ERROR).
RX Frame Sync Loss	Signals “Reed Solomon Error” (DWT_SIG_RX_SYNCLOSS). Sync Loss means

Event	Activity
	excessive Reed Solomon decoder errors.
CRC Error	Signal RX error/ bad CRC" (DWT_SIG_RX_ERROR).

When an event is recognised and processed the status register bit is cleared to clear the event interrupt. The Figure 4 below shows the *dwt\_isr()* function flow diagram.

**Parameters:**

none

**Return Parameters:**

none

**Notes:**

This function should be called from the microprocessors interrupt handler that is used to process the DW1000 interrupt.

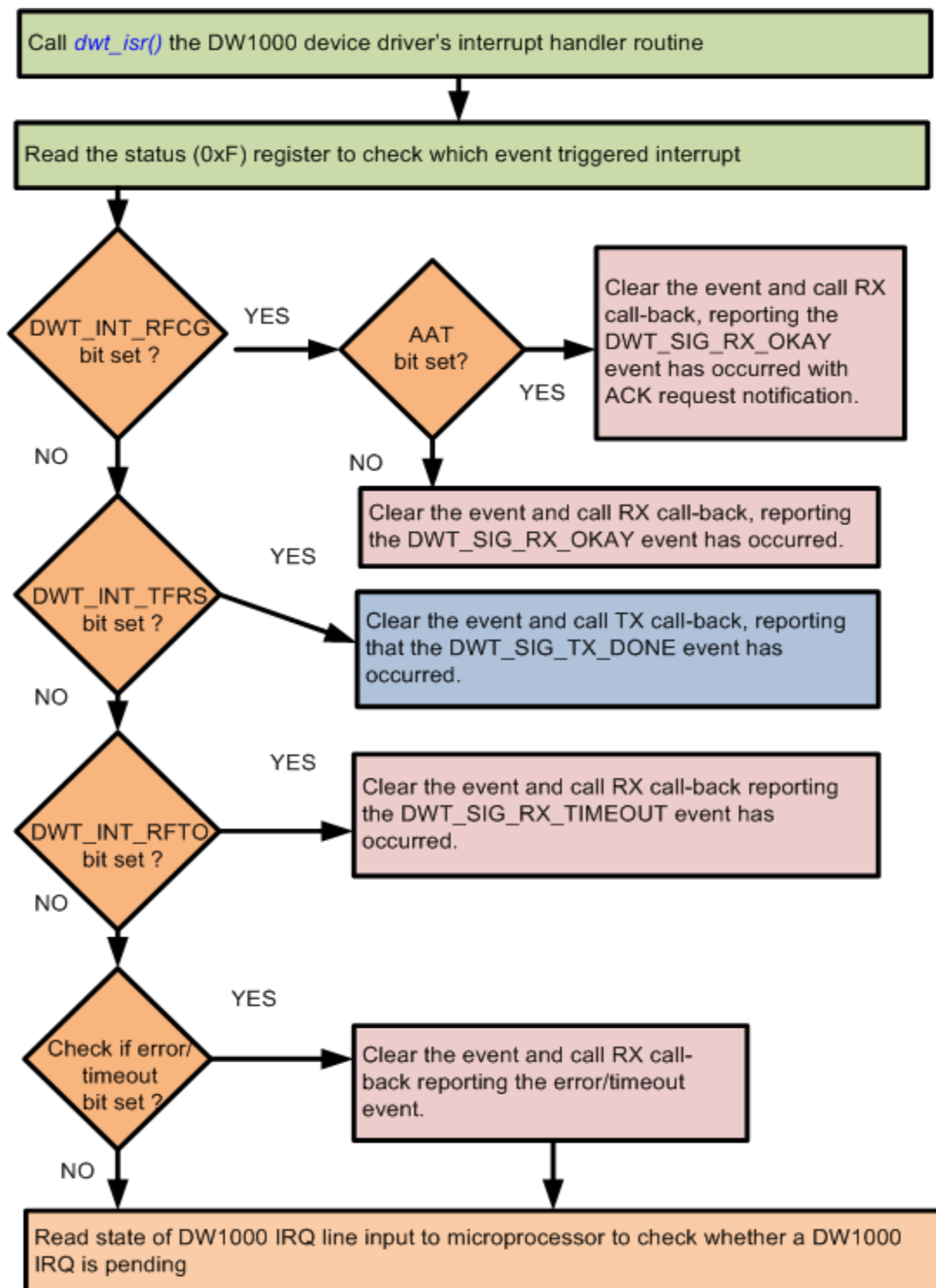


Figure 4: Interrupt handling

### 5.40 `dwt_setpanid`

```
void dwt_setpanid(uint16 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 DW1000 for frame filtering. See the `dwt_enableframefilter()` function.

**Parameters:**

type	name	description
uint16	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.

**5.41 dwt\_setaddress16**

```
void dwt_setaddress16(uint16 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 DW1000 for frame filtering. See the [dwt\\_enableframefilter\(\)](#) function.

**Parameters:**

type	name	description
uint16	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.

**5.42 dwt\_seteui**

```
void dwt_seteui (uint8* eui) ;
```

The [dwt\\_seteui\(\)](#) function sets the 64-bit address.

**Parameters:**

type	name	description
uint8*	eui	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 into the DW1000 internal register 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 DW1000'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.

### 5.43 *dwt\_geteui*

```
void dwt_geteui (uint8* eui) ;
```

The [dwt\\_geteui\(\)](#) function gets the programmed 64-bit EUI value from the DW1000.

Parameters :

type	name	description
uint8*	eui	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 programmed the DW1000 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 DW1000'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.

### 5.44 *dwt\_enableframefilter*

```
void dwt_enableframefilter(uint16 mask) ;
```

This [dwt\\_enableframefilter\(\)](#) function enables frame filtering according to the [mask](#) parameter.

Parameters :

type	name	description
uint16	mask	The bit mask which enables particular frame filter options, see Table 15.

Return Parameters :

none

Notes :

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

**Table 15: Bitmask values for frame filtering enabling/disabling**

Definition	Value	Description
DWT_FF_NOTYPE_EN	0x000	no frame types allowed – frame filtering will be disabled
DWT_FF_COORD_EN	0x002	behave as coordinator (can receive frames with no destination address (PAN ID has to match))
DWT_FF_BEACON_EN	0x004	beacon frames allowed
DWT_FF_DATA_EN	0x008	data frames allowed
DWT_FF_ACK_EN	0x010	ACK frames allowed
DWT_FF_MAC_EN	0x020	MAC command frames allowed
DWT_FF_RSVD_EN	0x040	reserved frame types allowed

### 5.45 *dwt\_enableautoack*

```
void dwt_enableautoack(uint8 responseDelayTime) ;
```

This function enables automatic ACK to be sent after a frame with ACK request is received. The ACK frame is sent after a specified responseDelayTime (in preamble symbols, max is 255).

**Parameters :**

type	name	description
uint8	responseDelayTime	The delay between the ACK request reception and ACK transmission.

**Return Parameters:**

none

**Notes:**

This 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 the frame transmitter to turn around and be ready to receive the response. If the host system is using the [RESPONSE\\_EXP](#) 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  $\mu$ s) without loss of preamble symbols in the receiver awaiting the ACK.

### 5.46 *dwt\_setrxaftertxdelay*

```
void dwt_setrxaftertxdelay(uint32 rxDelayTime) ;
```

This function sets the delay in turning the receiver on after a frame transmission has completed. The delay, [rxDelayTime](#), is in micro seconds. 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 [RESPONSE\\_EXP](#) mode). E.g. transmission of a frame with an ACK request bit set.

**Parameters :**

type	name	description
uint32	rxDelayTime	The turnaround time, $\mu$ s (1.0256 $\mu$ s, (512/499.2MHz)), 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 DW1000 will turn the RX on as soon as possible, which approximately takes 6.2  $\mu$ s. So if setting a value smaller than 7  $\mu$ s it will still take 6.2  $\mu$ s to switch to receive mode.

**5.47 dwt\_readrxdata**

```
void dwt_readrxdata(uint8 *buffer, uint16 len, uint16 bufferOffset);
```

This function reads a number, *len*, bytes of rx buffer data, from a given offset, *bufferOffset*, into the given buffer, *buffer*.

**Parameters :**

type	name	description
uint8*	buffer	The pointer to the buffer into which the data will be read.
uint16	len	The length of data to be read (in bytes).
uint16	bufferOffset	The offset at which to start to read the data.

**Return Parameters:**

none

**Notes:**

This function should be called on good frame reception 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.

**5.48 dwt\_readaccddata**

```
void dwt_readaccddata(uint8 *buffer, uint16 len, uint16 bufferOffset);
```

This function reads a number, *len*, bytes of accumulator buffer data, from a given offset, *bufferOffset*, into the given buffer, *buffer*. Note the data starts from *buffer[1]*. The first byte, *buffer[0]*, is dummy byte.

**Parameters :**

type	name	description
------	------	-------------



type	name	description
uint8*	buffer	The pointer to the buffer into which the data will be read.
uint16	len	The length of data to be read (in bytes).
uint16	bufferOffset	The offset at which to start to read the data.

**Return Parameters:**

none

**Notes:**

This function should be called on good frame or error reception to read the accumulator data.

**5.49 *dwt\_readdiagnostics***

```
void dwt_readdiagnostics(dwt_diag_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
{
    uint16 maxNoise ;           // LDE max value of noise
    uint16 firstPathAmp1 ;      // Amplitude at floor(index FP) + 1
    uint16 stdNoise ;           // Standard deviation of noise
    uint16 firstPathAmp2 ;      // Amplitude at floor(index FP) + 2
    uint16 firstPathAmp3 ;      // Amplitude at floor(index FP) + 3
    uint16 maxGrowthCIR ;       // Channel Impulse Response max growth CIR
    uint16 rxPreamCount;        // count of preamble symbols accumulated
    double firstPath ;          // First path index
}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\(\)](#)).

Fields	Description of fields within the <i>dwt_rxdiag_t</i> structure
<i>maxNoise</i>	The <i>maxNoise</i> parameter.
<i>firstPathAmp1</i>	First path amplitude is a 16-bit value reporting the magnitude of the leading edge signal seen in the accumulator data memory during the LDE algorithm's analysis. The amplitude of the sample reported in this <i>firstPathAmp</i> parameter is the value of the accumulator tap at index given by floor( <i>firstPath</i> ) reported below. This amplitude value can be used in assessing the quality of the received signal and/or the receive timestamp produced by the LDE.
<i>firstPathAmp2</i>	Is a 16-bit value reporting the magnitude of signal at index floor ( <i>firstPath</i> ) +2.
<i>firstPathAmp3</i>	Is a 16-bit value reporting the magnitude of signal at index floor ( <i>firstPath</i> ) + 3.
<i>stdNoise</i>	The <i>stdNoise</i> parameter is a 16-bit value reporting the standard deviation of the noise level seen during the LDE algorithm's analysis of the accumulator data. This value can be used in assessing the quality of the received signal and/or the receive timestamp produced by the LDE.
<i>maxGrowthCIR</i>	Channel impulse response max growth is a 16-bit value reporting a growth factor for the accumulator which is related to the receive signal power. This value can be used in assessing the quality of the received signal and/or the receive timestamp produced by the LDE.
<i>rxPreamCount</i>	This reports the number of symbols of preamble accumulated. This may be used to estimate the length of TX preamble received and also during diagnostics as an aid to interpreting the accumulator data. It is possible for this count to be a little larger than the transmitted preamble length, because of very early detection of preamble and because the accumulation count may include accumulation that continues through the SFD (until the SFD is detected).

Fields	Description of fields within the <i>dwt_rxdiag_t</i> structure
<i>firstPath</i>	<p>First path index is a 16-bit value reporting the position within the accumulator that the LDE algorithm has determined to be the first path. This value is set during the LDE algorithm's analysis of the accumulator data. This value may be of use during diagnostic graphing of the accumulator data, and may also be of use in assessing the quality of the received message and/or the receive timestamp produced by the LDE.</p> <p>The first path (or leading edge) is a sub-nanosecond quantity. Each tap in the accumulator corresponds to a sample time, which is roughly 1 nanosecond (or 30 cm in terms of the radio signal's flight time through air). To report the position of the leading edge more accurately than this 1-nanosecond step size, the index value consist of a whole part and a fraction part. The 16-bits of <i>firstPath</i> are arranged in a fixed point "10.6" style value where the low 6 bits are the fractional part and the high 10 bits are the integer part. Essentially this means if the <i>firstPath</i> is read as a whole number, then it has to be divided by 64 to get the fractional representation.</p>

### 5.50 *dwt\_configeventcounters*

```
void dwt_configeventcounters (int enable) ;
```

This function enables event counters (TX, RX, error counters) in the DW1000.

Parameters :

type	name	description
int	enable	Set to 1 to clear and enable the DW1000's internal digital counters. Set to 0 to disable.

Return Parameters:

none

Notes:

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

### 5.51 *dwt\_readeventcounters*

```
void dwt_readeventcounters (dwt_devicecnts_t *counters) ;
```

This function reads the event counters (TX, RX, error counters) in the DW1000.

Parameters :

type	name	description
dwt_deviceentcnts_t *	counters	Pointer to the device event counters structure.

```
typedef struct
{
    uint16 PHE ;           //number of received header errors
    uint16 RSL ;           //number of received frame sync loss events
    uint16 CRCG ;          //number of good CRC received frames
    uint16 CRCB ;          //number of bad CRC (CRC error) received frames
    uint16 ARFE ;          //number of address filter errors
    uint16 OVER ;          //number of receiver overflows (used in double buffer
mode)
    uint16 SFDT0 ;         //SFD timeouts
    uint16 PTO ;           //Preamble timeouts
    uint16 RTO ;           //RX frame wait timeouts
    uint16 TXF ;           //number of transmitted frames
    uint16 HPW ;           //half period warnings
    uint16 TXW ;           //power up warnings
} 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 <i>dwt_deviceentcnts_t</i> structure
<i>PHE</i>	PHR error counter is a 12-bit counter of PHY header errors.
<i>RSL</i>	RSE error counter is a 12-bit counter of the non-correctable error events that can occur during Reed Solomon decoding.
<i>CRCG</i>	Frame check sequence good counter is a 12-bit counter of the frames received with good CRC/FCS sequence.
<i>CRCB</i>	Frame check sequence error counter is a 12-bit counter of the frames received with bad CRC/FCS sequence.
<i>ARFE</i>	Frame filter rejection counter is a 12-bit counter of the frames rejected by the receive frame filtering function.
<i>OVER</i>	RX overrun error counter is a 12-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.
<i>SFDT</i>	SFD timeout errors counter is a 12-bit counter of SFD timeout error events.

Fields	Description of fields within the <i>dwt_devicecnts_t</i> structure
<i>PTO</i>	Preamble detection timeout event counter is a 12-bit counter of preamble detection timeout events.
<i>RTO</i>	RX frame wait timeout event counter is a 12-bit counter of receive frame wait timeout events.
<i>TXF</i>	TX frame sent counter is a 12-bit counter of transmit frames sent events. This is incremented every time a frame is sent.
<i>HPW</i>	Half period warning counter is a 12-bit counter of “Half Period Warning” events. These relate to late invocation of delayed transmission or reception functionality.
<i>TXW</i>	TX power-up warning counter is a 12-bit counter of “Transmitter Power-Up Warning” events. These relate to a delayed sent time that is too short to allow proper power up of TX blocks before the delayed transmission.

### 5.52 *dwt\_readtempvbat*

```
uint16 dwt_readtempvbat(void);
```

This function reads the temperature and battery voltage.

**Parameters :**

none

**Return Parameters:**

type	description
uint16	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 of DW1000. It enables the DW1000 internal convertors to sample the current IC temperature and battery.

**NB: the SPI frequency must be set to < 3MHz before a call to this function or an incorrect value will be returned.**

### 5.53 *dwt\_readwakeuptemp*

```
uint8 dwt_readwakeuptemp(void);
```

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

**Parameters :**

none

**Return Parameters:**

type	description
uint16	The low 8-bits are voltage value, and the high 8-bits are temperature value.

**Notes:**

This function may be used to read the temperature sensor value that was sampled by DW1000 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.

**5.54 dwt\_readwakeupvbat**

```
uint8 dwt_readwakeupvbat (void);
```

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

**Parameters:**

none

**Return Parameters:**

type	description
uint16	The low 8-bits are voltage value, and the high 8-bits are temperature value.

**Notes:**

This function may be used to read the battery voltage sensor value that was sampled by DW1000 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.

**5.55 dwt\_otpwriteandverify**

```
int dwt_otpwriteandverify(uint32 value, uint16 address);
```

This function is used to program 32-bit value into the DW1000 OTP memory.

**Parameters:**

type	name	description
uint32	value	this is the 32-bit value to be programmed into OTP memory
uint16	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_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes :**

The DW1000 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 DW1000 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 DW1000’s ability to function normally.**

The OTP memory locations are as defined in Table 16. 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 16: OTP memory map**

OTP Address	Size (Used Bytes)	Byte [3]	Byte [2]	Byte [1]	Byte [0]	Programmed By
0x000	4	64 bit EUID (These 64 bits get automatically copied over to register ID 0x1 on each reset.)				Customer
0x001	4					
0x002	0	-	-	-	-	Reserved
0x003	0	-	-	-	-	
0x004	0	-	-	-	-	Reserved
0x005	0	-	-	-	-	
0x006	4	{“0001,0000,0001”, “CHIP ID (20 bits)”}				DecaWave Test
0x007	4	{“0001”, “LOT ID (28 bits)”}				DecaWave Test
0x008	2	-	-	V <sub>meas</sub> @ 3.7 V	V <sub>meas</sub> @ 3.3 V	DecaWave Test
0x009	1	-	-	T <sub>meas</sub> @ Ant Cal	T <sub>meas</sub> @ 23 °C	Customer / DecaWave Test
0x00A	0	-				Reserved
0x00B	4	-				Reserved
0x00C	2	-				Reserved
0x00D	4	-				Reserved
0x00E	4	-				Reserved
0x00F	4	-				Reserved
0x010	4	CH1 TX Power Level PRF 16				Customer
0x011	4	CH1 TX Power Level PRF 64				Customer
0x012	4	CH2 TX Power Level PRF 16				Customer
0x013	4	CH2 TX Power Level PRF 64				Customer
0x014	4	CH3 TX Power Level PRF 16				Customer
0x015	4	CH3 TX Power Level PRF 64				Customer
0x016	4	CH4 TX Power Level PRF 16				Customer
0x017	4	CH4 TX Power Level PRF 64				Customer

OTP Address	Size (Used Bytes)	Byte [3]	Byte [2]	Byte [1]	Byte [0]	Programmed By
0x018	4	CH5 TX Power Level PRF 16				Customer
0x019	4	CH5 TX Power Level PRF 64				Customer
0x01A	4	CH7 TX Power Level PRF 16				Customer
0x01B	4	CH7 TX Power Level PRF 64				Customer
0x01C	4	TX/RX Antenna Delay - PRF 64		TX/RX Antenna Delay - PRF 16		Customer
0x01D	0	-	-	-	-	Customer
0x01E	1	-	-	-	XTAL_Trim[4:0]	Customer
0x01F	0	-	-	-	-	Customer
:	:	:	:	:	:	Reserved
0x400	4	SR Register (see below)				Customer

The SR ("Special 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 0x400. The value of the SR register can be directly read back at address.

**For more information on OTP memory programming please consult the DW1000 User Manual and Data Sheet.**

## 5.56 *dwt\_getrangebias*

```
double dwt_getrangebias(uint8 chan, float range, uint8 prf);
```

This function is used to return the range bias correction that may be applied to a two-way ranging result.

**Parameters:**

type	name	description
uint8	chan	Specifies the operating channel (e.g. 1, 2, 3, 4, 5, 6 or 7)
float	range	The calculated distance before correction.
uint8	prf	This is the PRF e.g. DWT_PRF_16M or DWT_PRF_64M.

**Return Parameters:**

type	description
double	Returns range correction needed in meters.

**Notes:**

A small but potentially significant error in range measurement is observed units are in close proximity. This gives a bias due to unequal growth of low and high powered signals in the accumulator. Depending on the application the bias correction given by this *dwt\_getrangebias()* function may be used to return a more accurate ranging result estimate.

For more information on range bias please consult with DecaWave.



### 5.57 *dwt\_setleds*

```
void dwt_setleds(uint8 value);
```

This is used to set up Tx/Rx GPIOs which are then used to control (for example) the LEDs. This is not completely IC dependent, also needs board with LEDs fitted on the GPIO I/O lines.

**Parameters :**

type	name	description
uint8	value	If value is set to 1 the LEDs will be enabled, if it is 0 the LED control is disabled. If value is 2 the LEDs will flash once after enable.

**Return Parameters:**

none

**Notes :**

For more information on GPIO control and configuration please consult the DW1000 User Manual and Data Sheet.

### 5.58 *dwt\_xtaltrim*

```
void dwt_xtaltrim(uint8 value);
```

This function writes the crystal trim value.

**Parameters :**

type	name	description
uint8	value	Crystal trim value (in range 0x0 to 0x1F, 31 steps (~1.5ppm per step)).

**Return Parameters:**

none

**Notes :**

This function can be called any time to set the crystal trim value register. 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 with DecaWave applications support team for details of crystal trimming procedures and considerations.

### 5.59 *dwt\_configcwmode*

```
void dwt_configcwmode(uint8 chan);
```

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

**Parameters:**

type	name	description
uint8	chan	This sets the UWB channel number, (defining the centre frequency and bandwidth). The supported channels are 1, 2, 3, 4, 5, and 7.

**Return Parameters:**

none

**Notes:**

Example code below of how to use this function in conjunction with [xtaltrim\(\)](#) function :

**Example code:**

```
// The table below specifies the default TX spectrum configuration
// parameters... this has been tuned for DW EVK hardware units

const tx_struct tx_spectrumconfig[NUM_CH] =
{
    // Channel 1
    {
        0xc9,          //PG_DELAY
        {
            0x75757575, //16M prf power
            0x67676767  //64M prf power
        }
    },
    // Channel 2
    {
        // Add other channels here
    },
    // Channel 7
    {
        0x93,          //PG_DELAY
        {
            0x92929292, //16M prf power
            0xd1d1d1d1  //64M prf power
        }
    }
};

void xtalcalibration(void)
{
    int i;
    uint8 chan = 2 ;
    uint8 prf = DWT_PRF_16M ;
    dwt_txconfig_t configTx ;

    // MUST SET SPI <= 3 MHz for this calibration activity.

    setspibitrate(SPI_3MHz); // target platform function to set SPI rate
                             // to 3 MHz

    //
    //     reset device
    //
    dwt_softreset();

    //
```

```

//      configure TX channel parameters
//

configTx.PGdly = tx_spectrumconfig[chan-1].PG_DELAY ;

configTx.power = tx_spectrumconfig[chan-1].tx_pwr[prf - DWT_PRF_16M];

dwt_configuretxrf(&configTx);

dwt_configcwmode(chan);

for(i=0; i<=0x1F; i++)
{
    dwt_xtaltrim(i);
    // measure the frequency
    // Spectrum Analyser set:
    // FREQ to be channel default e.g. 3.9936 GHz for channel 2
    // SPAN to 10MHz
    // PEAK SEARCH
} // end for

// when the crystal trim has completed, the device should be reset
// with a call to dwt_softreset() after which it can be programmed
// using the API functions for desired operation

return;
} // end xtalcalibration()

```

## 5.60 dwt\_configcontinuousframemode

```
void dwt_configcontinuousframemode(uint32 framerepetitionrate);
```

This function configures the DW1000 in continuous frame mode. This enables spectrum power measure.

### Parameters:

type	name	description
uint32	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 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 <4 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.

#### Example code:

```
// The table below specifies the default TX spectrum configuration
// parameters... this has been tuned for DW EVK hardware units

const tx_struct tx_s [NUM_CH] =
{
    { // Channel 1
        0xc9, //PG_DELAY
        {
            0x75757575, //16M prf power
            0x67676767 //64M prf power
        }
    },
    { // Channel 2
        ... Add other channels should be added here
    },
    { // Channel 7
        0x93, //PG_DELAY
        {
            0x92929292, //16M prf power
            0xd1d1d1d1 //64M prf power
        }
    }
};

int powertest(void)
{
    dwt_config_t    config ;
    dwt_txconfig_t  configTx ;

    uint8 msg[127]= "The quick brown fox jumps over the lazy dog."
                   "The quick brown fox jumps over the lazy dog."
                   "The quick brown fox jumps over the l";

    // MUST SET SPI <= 3 MHz for this calibration activity.

    setspibitrate(SPI_3MHz); // target platform function to set SPI rate
                             // to 3 MHz

    //    reset device

    dwt_softreset();

    //    configure channel parameters

    config.chan = 2;
    config.rxCode = 9;
    config.txCode = 9;
    config.prf = DWT_PRF_64M;
    config.dataRate = DWT_BR_110K;
    config.txPreambleLength = DWT_PLEN_2048;
    config.rxPAC = DWT_PAC64;
    config.nsSFD = 1;

    dwt_configure(&config) ;

    configTx.PGdly = tx_s[config.chan-1].PG_DELAY ;

    configTx.power = tx_s[config.chan-1].tx_pwr[config.prf - DWT_PRF_16M];

    dwt_configurertxrf(&configTx);

    // the value here 0x1000 gives a period of 32.82 µs

    dwt_configcontinuousframemode(0x1000);
```

```

dwt_writetxdata(127, (uint8 *) msg, 0) ;
dwt_writetxfctrl(127, 0);

//to start the first frame - set TXSTRT

dwt_starttx(DWT_START_TX_IMMEDIATE);

//measure the channel power
//Spectrum Analyser set:
//FREQ to be channel default e.g. 3.9936 GHz for channel 2
//SPAN to 1GHz
//SWEEP TIME 1s
//RBW and VBW 1MHz

// After the power is measured, the values in configTx can be changed
// to tune the spectrum. To stop the continuous frame mode, a call to
// dwt_softreset() is needed, after which the device can be programmed
// using the API functions for desired operation

return DWT_SUCCESS ;
}

```

## 5.61 SPI driver functions

These functions are platform specific SPI read and write functions, external to the DW1000 driver code, used by the device driver to send and receive data over the SPI interface to and from the DW1000. The DW1000 device driver abstracts the target SPI device by calling it through generic functions [writetospi\(\)](#) and [readfromspi\(\)](#). In porting the DW1000 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 DW1000 device driver functions.

### 5.61.1 writetospi

```
int writetospi (uint16 hLen, const uint8 *hbuff, uint32 bLen, const uint8 *buffer) ;
```

This function is called by the DW1000 device driver code (from the [dwt\\_writetodevice\(\)](#) function) when it wants to write to the DW1000's SPI interface (registers) over the SPI bus.

#### Parameters:

type	name	description
uint16	hLen	This is gives the length of the header buffer ( <a href="#">hbuff</a> )
uint8*	hbuff	This is a pointer to the header buffer byte array. The LSB is the first element.
uint32	bLen	This is gives the length of the data buffer ( <a href="#">buffer</a> ), to write.
uint8*	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_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes:**

The return values can be used to notify the upper application layer that there was a problem with SPI write. In DW1000 API [dwt\\_writetodevice\(\)](#) function the return value from this function is returned. However it should be noted that the DW1000 device driver itself does not take any notice of success/error return value but instead assumes that SPI accesses succeed without error.

**5.61.2 readfromspi**

```
int readfromspi (uint16 hLen, const uint8 *hbuff, uint32 bLen, uint8 *buffer) ;
```

This function is called by the DW1000 device driver code (from the [dwt\\_readfromdevice\(\)](#) function) when it wants to read from the DW1000's SPI interface (registers) over the SPI bus.

**Parameters:**

type	name	description
uint16	hLen	This is gives the length of the header buffer ( <a href="#">hbuff</a> )
uint8*	hbuff	This is a pointer to the header buffer byte array. The LSB is the first element.
uint32	bLen	This is gives the number of bytes to read.
uint8*	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_DECA_SUCCESS = 0 or DWT_DECA_ERROR = -1.

**Notes:**

The return values can be used to notify the upper application layer that there was a problem with SPI write. In DW1000 API [dwt\\_readfromdevice\(\)](#) function the return value from this function is returned. However it should be noted that the DW1000 device driver itself does not take any notice of success/error return value but instead assumes that SPI accesses succeed without error.

**5.62 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 DW1000 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 DW1000 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 DW1000 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.

### 5.62.1 decamutexon

```
decalrqStatus_t decamutexon (void) ;
```

This function is used 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 the DW1000's interrupt. Let's call this "IRQ\_State" Then it should disable the interrupt relating to the DW1000, and then return the original IRQ\_State.

#### Parameters:

none

#### Return Parameters:

Type	Description
decalrqStatus_t	This is the state of the target microcontroller's interrupt logic with respect to the handling the DW1000's interrupt, as it was on entry to the <a href="#">decamutexon()</a> function before it did any interrupt disabling.

```
typedef int decalrqStatus_t ;
```

#### Notes:

The [decamutexon\(\)](#) function returns the DW1000 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.

### 5.62.2 decamutexoff

```
void decamutexoff (decalrqStatus_t state) ;
```



This function is used to restore the DW1000's interrupt state as returned by [decamutexon\(\)](#) function. It is used 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
decalrqStatus_t	state	This is the state of the target microcontroller's interrupt logic with respect to the handling the DW1000's interrupt, as it was on entry to the <a href="#">decamutexon()</a> 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.

## 5.63 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.

### 5.63.1 dwt\_writetodevice

```
int dwt_writetodevice (uint16 regID, uint16 index, uint32 length, const uint8 *buffer) ;
```

This function is used to write to the DW1000's registers and buffers. The *regID* specifies the main address of the register or parameter block being accessed, e.g. a *regID* of 9 selects the transmit buffer. The *index* parameter selects a sub-address within the register file. A *regID* 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. The function returns 0 for success, or, -1 for error.

### 5.63.2 dwt\_readfromdevice

```
int dwt_readfromdevice (uint16 regID, uint16 index, uint32 length, uint8 *buffer) ;
```

This function is used to read from the DW1000's registers and buffers. The parameters the same as for the writeto4adevice function above except that the *buffer* parameter points at a location where the bytes being read are placed by the function call. The function returns 0 for success, or, -1 for error.

### 5.63.3 dwt\_read32bitreg

```
uint32 dwt_read32bitreg(int regFileID) ;
```

This function is used to read 32-bit DW1000 registers.

### 5.63.4 dwt\_read32bitoffsetreg

```
uint32 dwt_read32bitoffsetreg(int regFileID,int regOffset) ;
```

This function is used to read a 32-bit DW1000 register that is part of a sub-addressed block.

### 5.63.5 dwt\_write32bitreg

```
int dwt_write32bitreg(int regFileID, uint32 regval);
```

This function is used to write a 32-bit DW1000 register. The function returns 0 for success, or, -1 for error.

### 5.63.6 dwt\_write32bitoffsetreg

```
int dwt_write32bitoffsetreg(int regFileID,int regOffset,uint32 regval);
```

This function is used to write to a 32-bit DW1000 register that is part of a sub-addressed block. . The function returns 0 for success, or, -1 for error.

### 5.63.7 dwt\_read16bitoffsetreg

```
uint16 dwt_read16bitoffsetreg(int regFileID,int regOffset) ;
```

This function is used to read a 16-bit DW1000 register that is part of a sub-addressed block.

### 5.63.8 dwt\_write16bitoffsetreg

```
int dwt_write16bitoffsetreg(int regFileID, uint16 regval);
```

This function is used to write a 16-bit DW1000 register. The function returns 0 for success, or, -1 for error.

## 6 APPENDIX 1 – BIBLIOGRAPHY:

1	DecaWave DW1000 Datasheet
2	DecaWave DW1000 User Manual
3	<p>IEEE 802.15.4-2011 or “IEEE Std 802.15.4™-2011” (Revision of IEEE Std 802.15.4-2006).</p> <p>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.</p> <p>Available from <a href="http://standards.ieee.org/">http://standards.ieee.org/</a></p>

## 7 ABOUT DECAWAVE

DecaWave is a pioneering fabless semiconductor company whose flagship product, the DW1000, 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 8,  
Ireland.

<mailto:sales@decawave.com>

<http://www.decawave.com/>