SOURCE CODE GUIDE

DECARANGING (ARM) SOURCE CODE

Understanding and using the DW1000 DecaRanging source code

Version 1.6

This document is subject to change without notice.

TABLE OF CONTENTS

This document, "*DecaRanging (ARM) Source Code Guide*" is a guide to the application source code of DecaWave's "DecaRanging" two-way ranging demonstration running on the ARM microcontroller on the EVB1000 development platform.

This document should be read in conjunction with the "*EVK1000 User Guide*" which gives an overview of DW1000 evaluation kit (EVK1000) and describes how to operate the DecaRanging Application.

This document discusses the source code of the DecaRanging application, covering the structure of the software and the operation of the ranging demo application particularly the way the range is calculated.

Appendix A – Operational flow of execution is written in the style of a walkthrough of execution flow of the software. It should give a good understanding of the basic operational steps of transmission and reception, which in turn should help integrating/porting the ranging function to customers platforms.

This document relates to the following versions:

`"DecaRanging MP 2.23"` application version and
`"DW1000 Device Driver Version 02.08.00"` driver version

The device driver version information may be found in source code file "deca_version.h", and the application version is specified in "main.c".

1 OVERVIEW

Figure 1 below shows the layered structure of the DecaRanging application, giving the names of the main files associated with each layer and a brief description of the functionality provided at that layer.

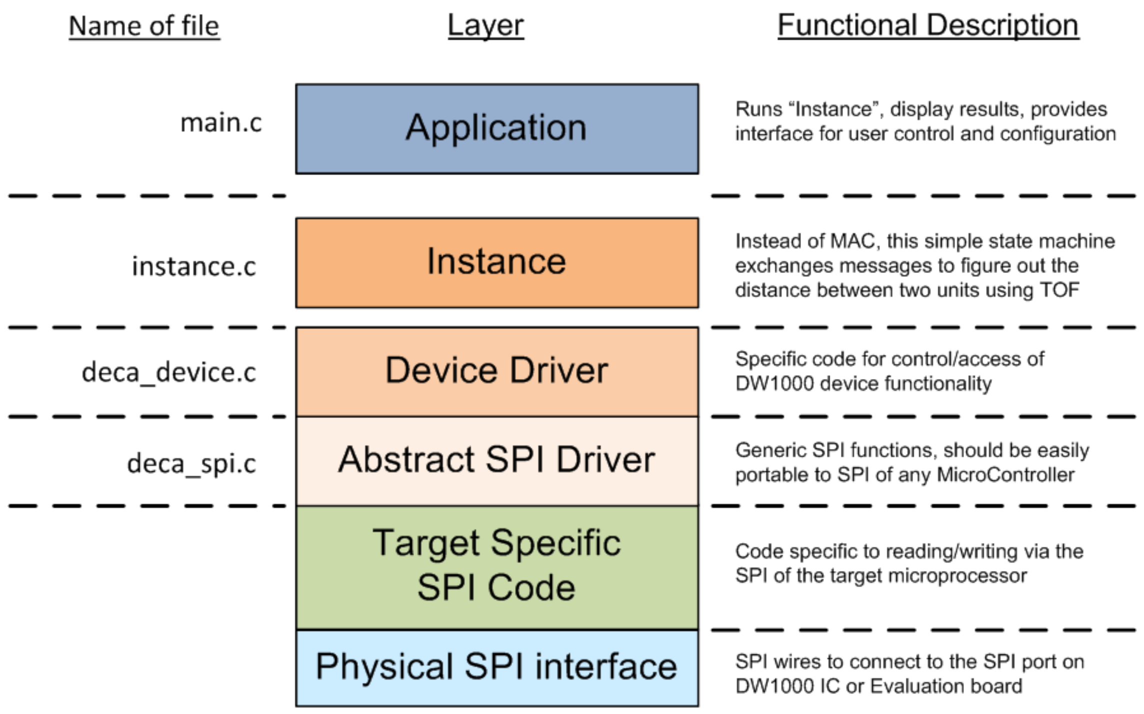

Name of file
Layer
Functional Description
main.c
Application
Runs "Instance", display results, provides interface for user control and configuration
instance.c
Instance
Instead of MAC, this simple state machine exchanges messages to figure out the distance between two units using TOF
deca_device.c
Device Driver
Specific code for control/access of DW1000 device functionality
deca_spi.c
Abstract SPI Driver
Generic SPI functions, should be easily portable to SPI of any MicroController
Target Specific SPI Code
Code specific to reading/writing via the SPI of the target microprocessor
Physical SPI interface
SPI wires to connect to the SPI port on DW1000 IC or Evaluation board

Figure 1: Software layers in DecaRanging

The layers, functions and files involved are described in the following section.

2 DETAILED DESCRIPTION OF DECARANGING CODE STRUCTURE

With reference to Figure 1, the identified layers are described in more detail below.

2.1 *Target Specific Code*

The low-level ARM specific code can be found in \src\platform\ – the two files *port.c* and *port.h* define target peripherals and GPIOs which are enabled and in use i.e. SPI1 for SPI communications with DW1000, SPI2 for SPI communications with LCD, other GPIO lines for application configuration and control.

SPI1:

```
#define SPIx                            SPI1
#define SPIx_GPIO                       GPIOA
#define SPIx_CS                         GPIO_Pin_4
#define SPIx_CS_GPIO                    GPIOA
#define SPIx_SCK                        GPIO_Pin_5
#define SPIx_MISO                       GPIO_Pin_6
#define SPIx_MOSI                       GPIO_Pin_7
```

The SPI1 peripheral is used to communicate to the DW1000 SPI bus.

Interrupt line:

```
#define DECAIRQ                       GPIO_Pin_8
#define DECAIRQ_GPIO                  GPIOA
#define DECAIRQ_EXTI                  EXTI_Line8
#define DECAIRQ_EXTI_PORT             GPIO_PortSourceGPIOA
#define DECAIRQ_EXTI_PIN              GPIO_PinSource8
#define DECAIRQ_EXTI_IRQn             EXTI9_5_IRQn
#define DECAIRQ_EXTI_USEIRQ           ENABLE
```

The DW1000 interrupt line is connected to GPIOA pin 8. Note: For MP the line is active high.

LCD driver:

```
#define SPIy                         SPI2
#define SPIy_GPIO                    GPIOB
#define SPIy_CS                      GPIO_Pin_12
#define SPIy_CS_GPIO                 GPIOB
#define SPIy_SCK                     GPIO_Pin_13
#define SPIy_MISO                    GPIO_Pin_14
#define SPIy_MOSI                    GPIO_Pin_15
```

The SPI2 peripheral is used to communicate with the LCD.

Application configuration switches (S1):

```
#define TA_SW1_3                     GPIO_Pin_0
#define TA_SW1_4                     GPIO_Pin_1
#define TA_SW1_5                     GPIO_Pin_2
#define TA_SW1_6                     GPIO_Pin_3
#define TA_SW1_7                     GPIO_Pin_4
#define TA_SW1_8                     GPIO_Pin_5
#define TA_SW1_GPIO                  GPIOC
```

Configuration switch (**S1**) is used to choose between the *Anchor* and *Tag* modes and various channel configurations. See "*EVK1000 User Guide*" for more details.

The src\compiler\compiler.h contains the standard library files which can be replaced if desired, (e.g. if one wishes to use functions optimised for smaller code size, say).

2.2 *Abstract SPI Driver – SPI Level code*

The file *deca_spi.c* provides abstract SPI driver functions *openspi()*, *closespi()*, *writetospi()* and *readfromspi()*. These are mapped onto the ARM microcontroller SPI interface driver.

2.3 *Device Driver – DW1000 Device Level Code*

The file deca_device_api.h provides the interface to a library of API functions to control and configure the DW1000 registers and implement functions for device level control. The API functions are described in the "*DW1000 Device Driver Application Programming Interface (API) Guide*" document.

2.4 *Instance Code*

The instance code (in instance.c) provides a simple ranging demonstration application. This instance code sits where the MAC would normally reside. For expediency in developing the ranging demonstration to showcase ranging and performance of the DW1000, the ranging demo application was implemented directly on top of the DW1000 driver API.

The ranging demo application is implemented by the state machine in function *testapprun()*, called from function *instance_run()*, which is the main entry point for running the instance code. The instance runs in different modes (*Tag* or *Anchor*) depending on the role configuration set at the application layer. The *Tag* and *Anchor* modes operate as a pair to provide the two-way ranging demo functionality between two units.

Initially the unpaired anchor and tag are in a discovery phase where the unpaired tag sends a *Blink* message that contains its own address, after which it listens for a *Ranging Initiation* response from an anchor. If it does not get one it sleeps for a period (default of 1 second) before blinking again. The unpaired anchor listens for tag blink messages. The anchor will then pair with a first tag it gets the *Blink* message from, and send the *Ranging Initiation* message to exit from the *Discovery Phase* and enter *Ranging Phase*.

The ranging method uses a set of three messages to complete two-round trip measurements from which the range is calculated. As messages are sent and received the *DecaRanging* application retrieves the message send and receive times from the DW1000. These transmit and receive timestamps are used to work out a round trip delay and calculate the range. Figure 2 shows the arrangement and general operation of the two-way ranging as implemented by the DecaRanging application.

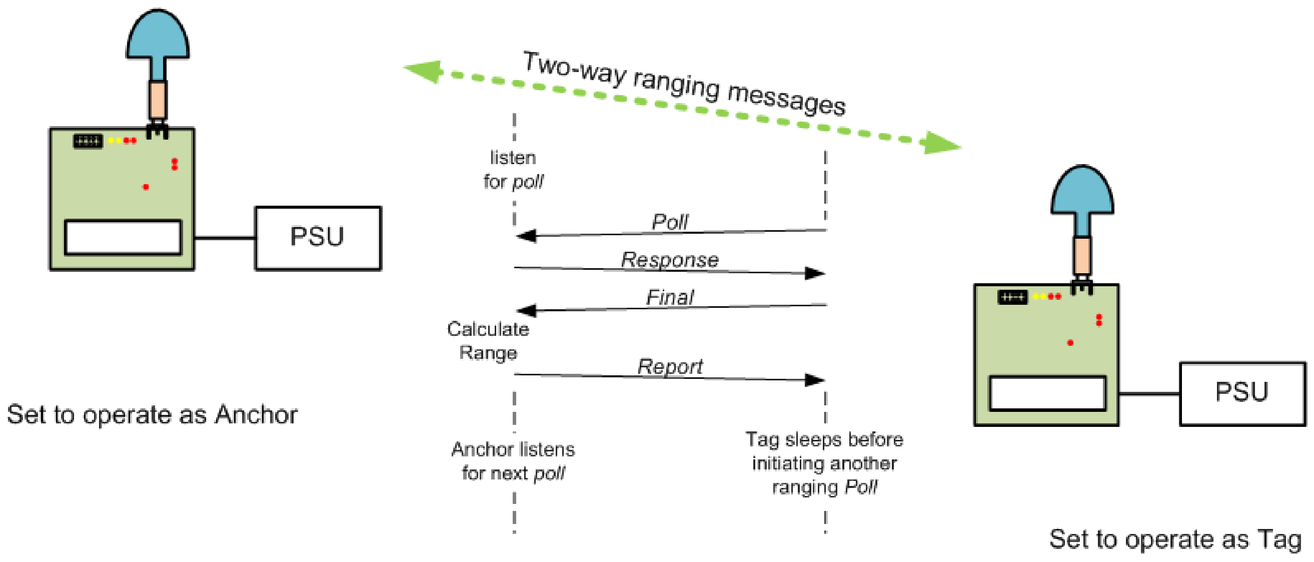


Figure 2: Two way ranging in DecaRanging

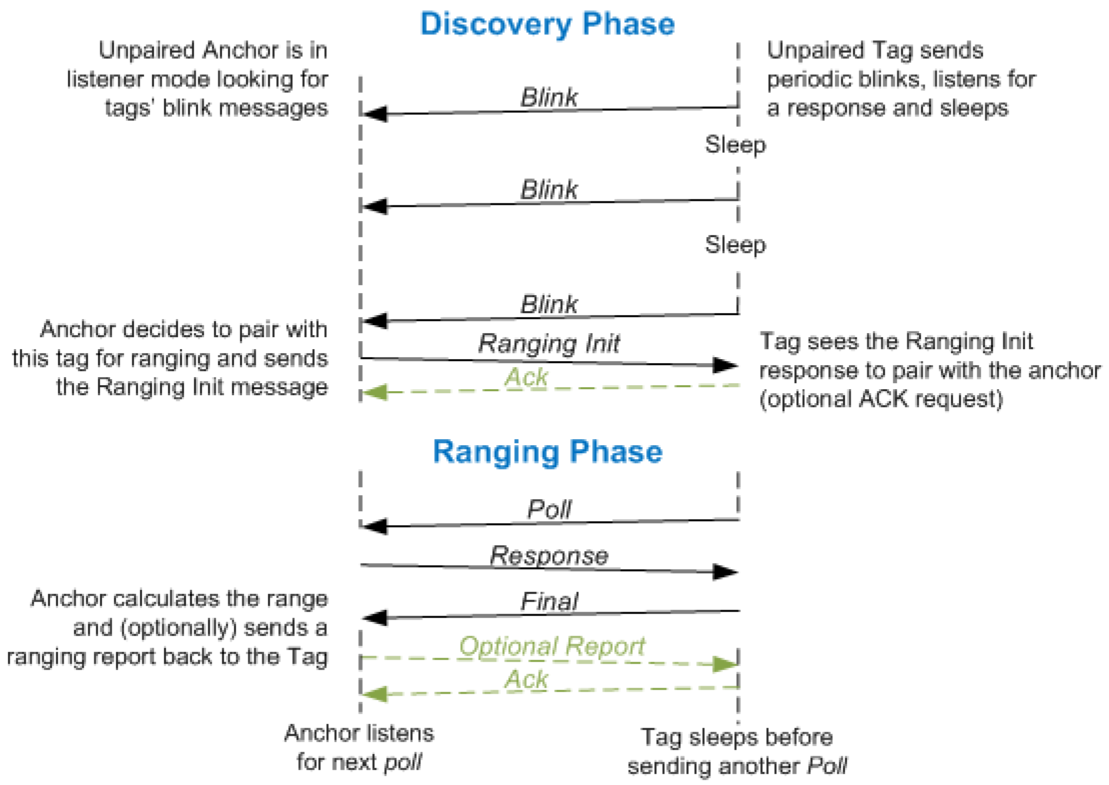

Discovery Phase
Unpaired Anchor is in listener mode looking for tags' blink messages
Unpaired Tag sends periodic blinks, listens for a response and sleeps
Blink
Sleep
Blink
Sleep
Blink
Anchor decides to pair with this tag for ranging and sends the Ranging Init message
Ranging Init
Ack
Tag sees the Ranging Init response to pair with the anchor (optional ACK request)
Ranging Phase
Poll
Response
Anchor calculates the range and (optionally) sends a ranging report back to the Tag
Final
Optional Report
Ack
Anchor listens for next poll
Tag sleeps before sending another Poll

Figure 3: Discovery and Ranging phase message exchanges

Once the anchor enters the *Ranging* phase it turns on its receiver and waits indefinitely for a poll message. The tag sends a *Poll* message, and then waits for a *Response* message from the anchor, after which it sends a *Final* message. At the end of this exchange the anchor calculates the range to the tag, and sends a ranging report message to the tag for it to display. This report may not be needed in a practical implementation depending on whether the initiating end needed to know the resulting range. If the anchor response is not received the tag times out and sends the *Poll* message again (after 400 ms sleep period). Section 3 describes the ranging algorithm in more detail including the format of the messages exchanged and the calculations performed.

Above this instance level is the application that provides the user interface described in section 2.5 below. In addition to the *instance_run()* function, the instance code provides control functions to the application. These functions are listed here to give the reader a quick idea of the functionality: -

instance_init(), instance_config(), instance_run(),instancespiopen(), instance_close(), instancedevicesoftreset(), instancesetantennadelays(), instancesetdisplayfeetinches(), instanceclearcounts(), instancesetrole(), instancesetaccumulatorreadenable(), instancelowlevelreadreg(), instancelowlevelwritereg(), instancereaddeviceid(), instancegetaccumulatordata(), instanceaccumlogenable(), instancesetreporting(), instancegetcurrentrangeinginfo(), instancesetaddresses(), instance_sleeping(), instancesettagsleepdelay(), instancesetreplydelay(), instancesetspibitrate(), instance_readsymbolcounters(), instancereadevents(), instance_rxcallback(), instance_txcallback().

The reader is directed to the code in file instance.c for more details of these functions.

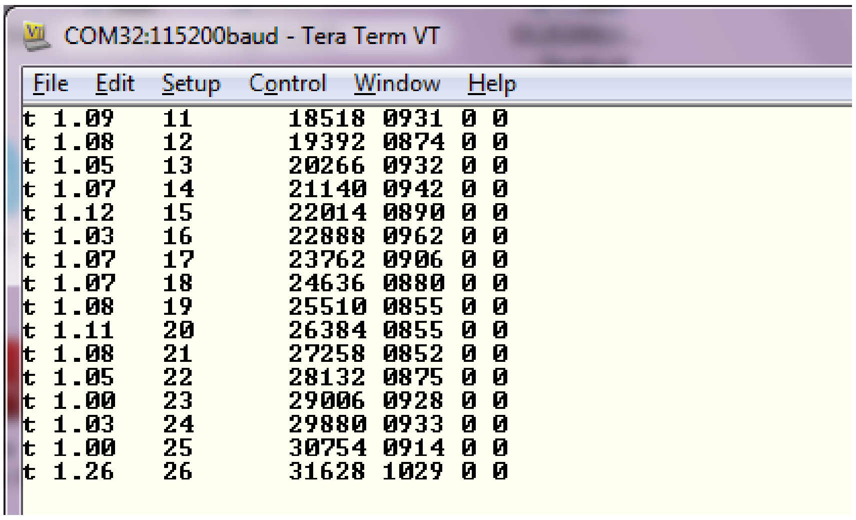
2.5 *Top level Application code*

The top level application (main.c) contains the main entry point for the DecaRanging ranging demo application and also all the user interface code.

The ability of the application layer to display results depends on the capability of the hardware platform. On the EVB1000 evaluation board the LCD is used to display the resultant range from a range measurement.

Figure 4: Example LCD display showing last and average range

The application also outputs some debug information over the virtual COM port. Figure 5 shows example output from a tag. The first column is "*t*" for a tag and "*a*" for and anchor, the second column is the range (received via the ToF *Report* message), the third column is the number of ranges received, the fourth is the micro time in ms, the fifth is the number of preamble symbols accumulated when receiving the *Response* message from the anchor, the sixth and the seventh are the "LATE" tx and rx counters.


COM32:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
t 1.09    11       18518 0931 0 0
t 1.08    12       19392 0874 0 0
t 1.05    13       20266 0932 0 0
t 1.07    14       21140 0942 0 0
t 1.12    15       22014 0890 0 0
t 1.03    16       22888 0962 0 0
t 1.07    17       23762 0906 0 0
t 1.07    18       24636 0880 0 0
t 1.08    19       25510 0855 0 0
t 1.11    20       26384 0855 0 0
t 1.08    21       27258 0852 0 0
t 1.05    22       28132 0875 0 0
t 1.00    23       29006 0928 0 0
t 1.03    24       29880 0933 0 0
t 1.00    25       30754 0914 0 0
t 1.26    26       31628 1029 0 0
```

Figure 5: Example Teraterm window showing the debug info sent via COM port

2.6 *Complete list of source code files*

Table 1 gives a list of the files that make up the source code of the DecaRanging application. The file name is given along with a brief description of the file and its purpose. The reader is referred to the other sections of this document for more details on the code structure and organisation.

Table 1: List of source files in the DecaRanging application

Filename	Brief description
deca_version.h	DecaWave's version number for the DW1000 driver/API code
main.c	Application – source code (main line)
deca_device.c	Device level Functions – source code
deca_device_api.h	Device level Functions – header
deca_mutex.c	Place holder for IRQ disable for mutual exclusion – source code
deca_param_types.h	Header defining the parameter and configuration structures
deca_params_init.c	Initialisation of configuration data for setting up the DW1000
deca_range_tables.c	Contains the ranging correction tables
deca_regs.h	Device level – header (Device Register Definitions)
deca_spi.c	SPI interface driver – source code
deca_spi.h	SPI interface driver – header
deca_types.h	Data type definitions – header
port.c	ARM peripheral and GPIO configuration
port.h	ARM peripheral and GPIO configuration definitions
Instance_calib.c	Calibration functions and data for the application – source code
Instance_common.c	Common application functions – source code
instance.c	Ranging Application Instance – source code
instance.h	Ranging Application Instance – header
compiler.h	Contains the standard library files
stm32f10x_conf.h	STM library configuration/inclusion files
stm32f10x_it.c	Interrupt handlers are defined here.
stm32f10x_it.h	Interrupt handlers are declared here.

3 Ranging Algorithm

This section describes the ranging algorithm used in the DecaRanging ranging demo application. In contrast to some earlier versions of DecaRanging, the ranging algorithm in this code is quite efficient for two-way ranging requiring just three messages to be exchanged for an accurate range to be calculated. This is described below.

3.1 *DecaRanging's Tag/Anchor Two-way ranging algorithm*

For this algorithm one end acts as a Tag, periodically initiating a range measurement, while the other end acts as an Anchor listening and responding to the tag and calculating the range.

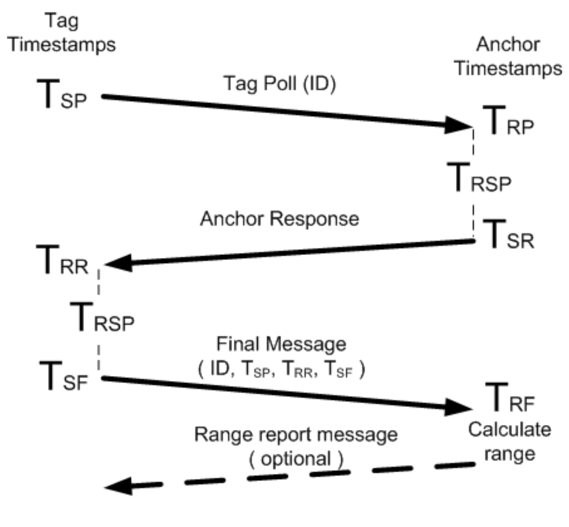
- The tag sends a *Poll* message addressed to the target anchor and notes the send time, T_{SP}. The tag listens for the *Response* message. If no response arrives after some period the tag will time out and send the poll again.

- The anchor listens for a *Poll* message addressed to it. When the anchor receives a poll it notes the receive time T_{RP}, and sends a *Response* message back to the tag, noting its send time T_{SR}.

- When the tag receives the *Response* message it notes the receive time T_{RR} and sets the future send time of the *Final* response message T_{SF}, (a feature of DW1000 IC), it embeds this time in the message before initiating the delayed sending of the *Final* message to the anchor.

- The anchor receiving this *Final* response message now has enough information to work out the range.

- In the DecaRanging ranging demo the anchor sends a ranging report of the calculated range to the Tag to give it some thing to display. Figure 6 shows this exchange and gives the formula used in the calculation of the range.



The tag sees Round Trip Delay time T_{TRT} of (T_{RR} - T_{SP})
The anchor sees Round Trip Delay time T_{ART} of (T_{RF} - T_{SR})

After receiving the Final message, the anchor knows all the timestamps, so it can:
(a) remove its response time: (T_{SR} - T_{RP}) from the Tag's T_{TRT}, and
(b) remove tags response time: (T_{SF} - T_{RR}) from Anchor's T_{ART},
to give antenna to antenna round trip times

The anchor then averages these two resultant round trip times to remove the effects of each end's clock frequency differences, (a technique which works best when the RX-to-TX response T_{RSP} is the same at both ends), and divides by 2 to get a one-way trip time.

Putting all this into one equation gives a Time of Flight:
$((T_{RR} - T_{SP}) - (T_{SR} - T_{RP}) + (T_{RF} - T_{SR}) - (T_{SF} - T_{RR})) / 4$

or TOF = $(2T_{RR} - T_{SP} - 2T_{SR} + T_{RP} + T_{RF} - T_{SF}) / 4$.

Multiplying the TOF by c, the speed of light (and radio waves), gives the distance (or range) between the two devices.

Figure 6: Range calculation in DecaRanging

- After this the anchor turns on its receiver again to await the next poll message, while the tag meanwhile sleeps or counts off the delay period to the next ranging attempt.

3.2 *Practical considerations for ranging in a real product*

In a microprocessor implementation the following is recommended to improve the system's efficiency:

- Delayed send is used to make time T_{RSP} as near as possible to the same value at both ends, i.e. by setting the transmit time to be the receive time + 150 ms. (The 150 ms time is larger than is needed in a microprocessor system but is used in our code to allow PC based DecaRanging enough time to interoperate with the ARM based implementation). Keeping the response time the same at both ends is important since the error in each single round trip delay (RTD) measurement (due to the different local clock rates) is of opposite sign and cancels out when the two RTD measurements are averaged.

  Notes:

  (i) On EVB1000 HW the user can set S1 DIP switch S1-2 to ON (on both *Tag* and *Anchor* units) to use a faster (1 ms) response times. When in this mode 6.81 Mb/s rate is used, so only modes 2, 4, 6 and 8 can be used.

  (ii) Only one transmission is allowed per 1ms when using smart TX power setting (which applies for the 6.81 Mb data rate). Blink TX time is approximately 180 µs with 128 preamble length. Thus in discovery phase the device spends 180 µs in transmit state, then 600 µs in idle, then 220 µs in receive, and then back to idle until it sends a poll or new blink.

  (iii) Anchor is in receive, then sends a response (180 µs), then idle for 300 µs, then receives a final 220 µs after which it goes back to idle and receive.

  (iv) After the tag processes the response it prepares the final message and after transmitting it goes to sleep. The minimum sleep/idle time before next poll transmission is 880µs. So the tag sends a poll (176 µs), then idle for (312 µs) then RX for (192 µs (the time-out is set to 198 µs)), and then idle again for 320 µs. Thus the time from poll TX to final TX start is 176 + 312 + 192 + 320 = 1000 µs.

  (v) These times/state transitions can be seen on the scope when GPIO5 is configured as EXTTXE output and GPIO6 as EXTRXE output.

  (vi) The ranging init message in this mode does not request an ACK as it would violate the 1ms regulation spec, and also the anchor does not send a TOF report as this example is trying to achieve low power tag application.

- In a microprocessor system the T_{RSP} time is best reduced to complete the ranging as quickly as possible. This improves the accuracy of the result as it minimises the difference in T_{RSP} due to the different local clock rates at each end. A time should be chosen that is a little greater than the worst case response latency possible, so that the command to do a delayed TX is not issued after that time has gone past, which would essentially delay TX until the sys clock wraps around to the specified TX time again. A status bit is provided to warn of this "*delayed TX more than half a clock period away"* event indicating the TX start was late. If this late error occurs frequently then it would probably be a good idea to use a longer delay, but if this was a very rare event it may be better to keep the specified period and just recover from the error when it occurs. This is a system design choice.

- In a microprocessor based *Tag* and *Anchor* the ranging can be (and is best) completed quickly. Then assuming we do not want to be doing hundreds of range measurements every second, it is likely that we would want to put the tag into a low power state between range measurements. That is, to conserve battery power, both the DW1000 and the controlling microprocessor would typically be put into a low power mode with just a single timer running to reawaken the system for the next ranging attempt. The DW1000 has a low power oscillator that may be used for this purpose.

- Where two peer mobile devices are ranging between each other (e.g. in a separation alarm say) then for battery conservation it is not practical for either device to have to listen for long periods. Therefore the devices have to operate in a more synchronised fashion, turning on the receiver only when the message from the peer device is expected, (i.e. long enough before the expected message and for a period long enough to detect it, given the maximum possible timing drifts between the two devices clocks). In such a scheme initial pairing will probably be initiated by some manual means like coordinated button pressing on both devices, or perhaps by employing *low-powered listening*, a technique that samples for preamble occasionally (e.g. once per second) looking for a wakeup sequence sent for the whole second.

3.3 *Messages used in DecaRanging's Tag/Anchor Two-way ranging*

Six messages are employed in the tag/anchor two-way ranging, two in the *Discovery* phase (the blink and ranging initiation messages) and four in the *Ranging* phase (the poll message, the response message, the final message, and the optional range report message), as shown in Figure 3. Although these follow IEEE message conventions, these are NOT standard RTLS messages, the reader is referred to ISO/IEC 24730-62 (currently a draft international standard) for details of message formats being standardised for use in RTLS systems based on IEEE 802.15.4 UWB. The formats of the messages used in the demo are given below.

3.3.1 General ranging frame format

The general message format is the IEEE 802.15.4 standard encoding for a data frame. Figure 7 shows this format. The two byte Frame Control octets are constant for the DecaRanging application because it always uses data frames with 8-octet (64-bit) source and destination addresses, and a single 16-bit PAN ID (value 0xDECA). The only exception is the *Blink* message which is described in 3.3.2 below. In a 802.15.4 network, the PAN ID might be negotiated as part of associating with a network or it might be a defined constant based on the application.

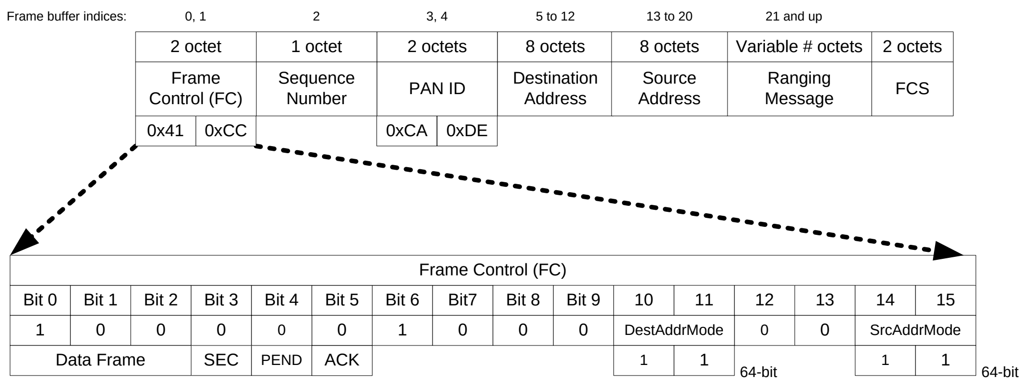


Figure 7: General ranging frame format

The sequence number octet is incremented modulo-256 for every frame sent, in line with IEEE rules.

The source and destination addresses are 64-bit numbers programmed uniquely into each device (during EVB1000 manufacture). This can be used by the application to give each DW1000 based product a unique address.

The 2-octet FCS is a CRC frame check sequence. This is generated automatically by the DW1000 IC (under software control) and appended to the transmitted message.

The content of the ranging message portion of the frame depends on which of the four ranging messages it is. These are shown in Figure 9 and described in sections 3.3.3 to 3.3.6 below. In these only the ranging message portion of the frame is shown and discussed. This data is encapsulated in the general ranging frame format of Figure 7 to form the complete ranging message in each case.

The DecaRanging application's payload setup dialog allows the user to specify a data message transfer at the end of the ranging messages. This is not used other than to display on the receiver screen, and record in any log file created. It demonstrates that data is being sent, and might be used to add a note of something in the log file. The length of user data allowed is defined by the space that is free in the longest of the ranging messages, which is the final message. Max Frame length (127 octets) minus general frame overhead (23 octets) minus final message content (16 octets) leaves 88 octets for user data, as shown in the Figure 9 messages. By default user payload is not programmed, and so the optional part of the messages is empty, i.e. of zero length.

3.3.2 Blink frame format

The special *Blink message* frame format is used for sending of the Tag Blink messages. The blink frame is simply sent without any additional application level payload, i.e. the application data field of the blink frame is zero length. The result is a 12-octet blink frame. The encoding of the minimal blink is as shown in Figure 8.

1 octet FC	1 octet	8 octets	2 octets
0xC5	Seq. Num	64-bit Tag ID	FCS

Figure 8: the 12-octet minimal blink frame

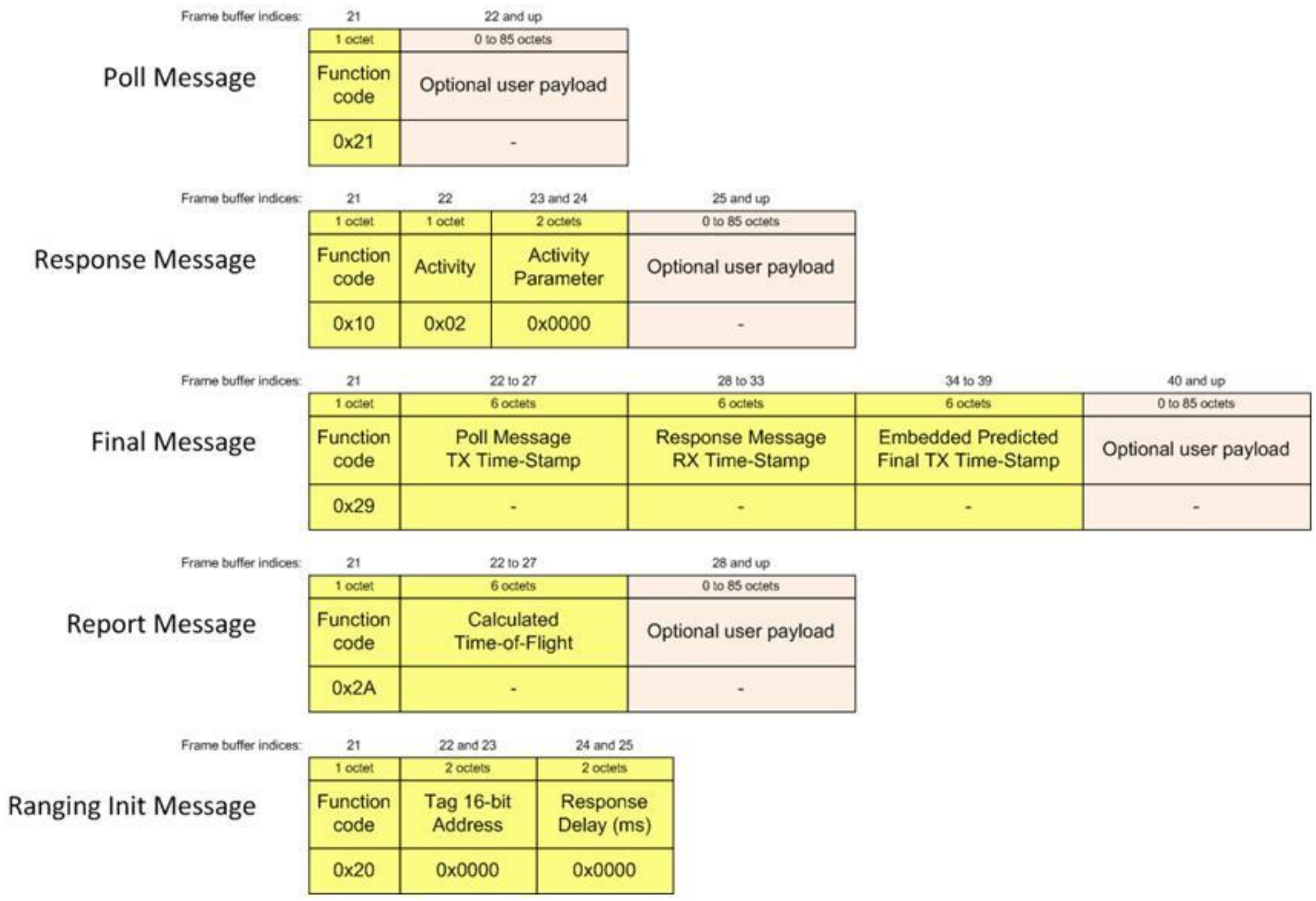


Figure 9: Ranging message encodings

3.3.3 Poll message

The poll message is sent by the tag to initiate a range measurement. For the poll message, the ranging message portion of the frame is a single octet, with value: 0x21.

The *optional user payload* part of the poll message may be empty, or used to carry data not directly related to the ranging. There are also two optional bytes which can be used to transmit tag's voltage and temperature levels. See section 3.3.8 below for more detail of this.

3.3.4 Response message

The response message is sent by the anchor in response to a poll message from the tag. For the response message a single octet would be sufficient, but to allow for some future expansion possibilities a more complex encoding has been included. Table 2 lists and describes the individual fields within the response message.

Table 2: Fields within the ranging response message

Octet #'s	Value	Description
1	0x10	This octet 0x10 identifies this as an anchor response controlling the activity of the tag

2	0x02	This activity octet tells the tag to continue with the ranging exchange
3	0x00	If 0x0 then anchor does not send the report back to Tag, if 0x1 it sends the report.
4	0x00	This two octet parameter is unused for activity 0x02.
5 and up	-	Optional user payload, see section 3.3.8 below.

3.3.5 Final message

The final message is sent by the tag after receiving the anchor's response message. The final message is 16 octets in length. Table 3 lists and describes the individual fields within the final message.

Table 3: Fields within the ranging final message

Octet #'s	Value	Description
1	0x29	This octet identifies the message as the tag "Final" message
2 to 6	-	This five octet field is the TX timestamp for the tag's poll message, i.e. the precise time the frame was transmitted.
7 to 11	-	This five octet field is the RX timestamp for the response poll message, i.e. the time the tag received the response frame from the anchor.
12 to 16	-	This five octet field is the TX timestamp of this final message, i.e. the precise time the frame was (or will be) transmitted, this needs to be calculated by the tag as described in section 3.3.5.1 below.
17 and up	-	Optional user payload, see section 3.3.8 below.

3.3.5.1 Final message embedded TX timestamp

The final message includes a field that is its own transmit timestamp. The tag microprocessor needs to pre-calculate this and embed it in the message buffer before initiating the transmission of the final message. Assuming that it has already calculated DT, the reply time to programme as the delayed send time for the message, the embedded time is then just DT masked to clear the lower 9 bits, plus the TX antenna delay value.

In the DecaRanging source code this calculation is done in file instance.c in state TA_TXFINAL_WAIT_SEND.

3.3.6 Range report message

Upon receiving the *Final message* the anchor node can calculate the time of flight between tags and anchor. Depending on the application this information may be used directly, or may be sent on to a location engine. For DecaRanging, to facilitate displaying the measured tag-to-anchor separation at both ends, the anchor unit sends a range report message to the tag unit.

The range report message is 5 octets in length. Table 4 lists and describes the individual fields within the range report message.

Table 4: Fields within the range report message

Octet #'s	Value	Description
1	0x2A	This octet 0x2A identifies the message as a range report
2 to 6	-	This five octet field is the anchor calculated time-of-flight, representing the estimated distance between the tag and the anchor.
7 and up	-	Optional user payload, see section 3.3.8 below.

3.3.7 Ranging Initiation message

Upon receiving the *Blink message* the unpaired anchor will send the *Ranging Initiation message* to the tag that has sent the blink message.

The ranging initiation message is 5 octets in length. Table 5 lists and describes the individual fields within the ranging initiation message.

Table 5: Fields within the ranging initiation message

Octet #'s	Value	Description
1	0x20	This octet 0x20 identifies the message as a range report
2 to 3	-	This 16-bit field can be used by Tag to change to use the specified 16-bit address. Instead of 64-bit address.
4 to 5	-	This 16-bit number gives the response time to be used in the following ranging exchange.

3.3.8 Optional user payload

The DecaRanging application allows user payload to be added to verify operation sending longer messages. In each of the messages of in Figure 9 the optional user payload is shown as 0 to 88 octets at the end of the message. The standard payload size of an IEEE 802.15.4 UWB frame is 127 octets. The value of 88 then comes from subtracting the 23 octets of general MAC frame overhead (as shown Figure 7) and the 16 octets of the final message, (the longest ranging message as shown in Figure 9) from this 127 octets.

For simplicity this limit of 88 octets of optional user data is applied to all ranging messages within DecaRanging code, but clearly there is more room for data in the shorter messages, which could be utilised in a real application if desired.

3.4 *Frame Time Adjustments*

Successful ranging relies on the system being able to accurately determine the TX and RX times of the messages as they leave one antenna and arrive at the other antenna. This is needed for antenna-to-antenna time-of-flight measurements and the resulting antenna-to-antenna distance estimation.

The significant event making the TX and RX times is defined in IEEE 802.15.4 as the "Ranging Marker (RMARKER): The first ultra-wide band (UWB) pulse of the first bit of the physical layer (PHY) header (PHR) of a ranging frame (RFRAME)". The time stamps should reflect the time instant at which the RMARKER leaves or arrives at the antenna. However, it is the digital hardware that marks the generation or reception of the RMARKER, so adjustments are needed to add the TX antenna delay to the TX timestamp, and, subtract the RX antenna delay from the RX time stamp.

The EVB1000 units as part of the EVK1000 kit are paired and the antenna delays are calibrated, and programmed into the DW1000 OTP (one-time-programmable) memory. However if a value has not been programmed the DecaRanging application will use the default antenna delay value as set in the instance.h file, there are two values, one for each PRF configuration (`DWT_PRF_64M_RFDLY (515.6f); DWT_PRF_16M_RFDLY (515.0f)`). The value specified is divided equally between TX and RX antenna delays. The default value has been experimentally set by adjusting it until the reported distance averaged to be the measured distance. The need to re-tune the Antenna Delay is discussed in section 5.1 below.

The individual adjustments made to correct the timestamps are discussed below.

3.4.1 Frame Transmit-Time Adjustment

In the DW1000 the transmit time stamp is made as the RMARKER is sent by the digital circuitry.
If the TX_ANTD register value is programmed it will be automatically added to the reported TX timestamp, and no software adjustment is necessary.

3.4.2 Frame Receive-Time Adjustment

In the DW1000, the receive time stamp is initially made as an appropriate event representing the receipt of the RMARKER detected digital circuitry, and then a first path seek algorithm is run to find the first path more precisely, and finally the value is adjusted by subtracting the configured RX antenna delay value. This final adjusted RX timestamp is saved in the register and the software does not have to make any further adjustments to the time of arrival read from the IC register.

4 Practical Location / Localisation Algorithms

DecaRanging provides ranging between cooperating nodes, and section 3.2 above covers some practical considerations for this. To locate a mobile node individual distance measurements with reference to a set of fixed known location anchor nodes have to be combined at some location engine function using Trilateration. For 2-D location a minimum of three range measurements have to be combined. For 3-D location a minimum of four range measurements have to be combined.

Another method of location is the Time Difference of Arrival method. This scheme yields the lowest power tag design, since the tag only has to send its ID blink and sleep. The blink rate is chosen to cope with highest speed of tracked items, with a view to number of tags potentially contending for the same air space, (some random element in blink rate is also good to ensure that no two tags are always blinking on top of each other's transmissions. For TDOA location the blink needs to be received by four or more fixed known location readers (or three for 2-D location) and the arrival time noted by each. The difference in arrival times of the same message at any two readers yields a curve on which the tag must lie. The intersection of curves from multiple readers gives the tags location, this is called multilateration. The TDOA scheme requires that the readers are working on a synchronised time base, or that the location engine is tracking each reader's clock drift and correcting for it.

5 Code / System Issues

5.1 *Antenna Delay*

The antenna delay may need changing if a different antenna is being used.

Note: If the antenna delay value is set too large, it results in negative RTD calculation results (internally to the software) and these RTD values are discarded as bad and no RTD / distance measurement will be reported. In using the system, if the communication seems to be working, , but the Time-of-Flight status lines are not updating, then this may be because the antenna delay is set to too large a value. This can be checked by clearing the antenna delays to zero. To tune the antenna delay to the correct value is a process of trial and error, tweaking the antenna delay until the average distance reported matches the real antenna-to-antenna distance measured with a tape measure.

6 Building and Running the Code

6.1 *Building the code*

As an example development environment this code can be built under eclipse IDE. This is covered in a separate document. Please contact DecaWave.

6.2 *Building configuration options*

The example application has a number of different configuration options (see instance.h for more details):

```
#define DEEP_SLEEP     (1) //To enable deep-sleep in the tag set this to 1


#define DR_DISCOVERY   (1) //To use discovery ranging mode (tag will blink until it receives
ranging request from an anchor) after which it will pair with that anchor and start ranging
exchange
```

Note: if DR_DISCOVERY is set to 0, then each anchor needs to be configured with a unique address and tag needs to have a list of the anchors and the poll mask (anchorPollMask) set to enable it to poll up to 4 anchors. Non-discovery mode means the tag will range to the first anchor in its anchorAddressList and then depending on the poll mask value it will poll the next anchor and so on.

7 APPENDIX A – OPERATIONAL FLOW OF EXECUTION

This appendix is intended to be a guide to the flow of execution of the software as it runs, reading this and following it at the same time by looking at the code should give the reader a good understanding of the basic way the software operates as control flows through the layers to achieve transmission and reception. This understanding should be an aid to integrating/porting the ranging function to other platforms.

To use this effectively, the reader is encouraged to browse the source code (e.g. in the eclipse IDE) at the same time as reading this description, and find each referred item in the source code and follow the flow as described here.

7.1 *The main application entry*

The application is initialised and run from the *main()*. Firstly we initialise the HW and various ARM microcontroller peripherals, *peripherals_init()* and *spi_peripheral_init()* functions are used for this. Then the instance roles (Tag or Anchor) and channel configurations (channel, PRF, data rate etc.) are set up by a call to *inittestapplication()* function. Finally the *instance_run()* is called periodically from *while(1)* loop which runs the instance state machine described below. In parallel the DW1000 interrupt line is enabled so any events (e.g. transmitted frames or received frames) are processed in the *dwt_isr()* call.

7.2 *Instance state machine*

The instance state machine delivers the primary DecaRanging function of range measurement. The instance state machine does two-way ranging by forming the messages for transmit (TX), commanding their transmission, by commanding the receive (RX) activities, by recording the TX and RX timestamps, by extracting the remote end's TX and RX timestamps from the received *Final* messages, and, by performing the time-of-flight calculation.

The instance code is invoked using the function *testapprun()*, the paragraphs below trace the flow of execution of this instance state machine from initialisation through the TX and RX operations of a ranging exchange. This is done primarily by looking at the operation of the Tag end. It starts by sending a *Blink* message and waiting to receive a *Ranging Initiation* message before starting ranging exchange. Then it will send a *Poll* message, await a *Response* and then send the *Final* message to complete the ranging exchange. If the anchor is sending the ToF report the tag will enable its receiver after the final message to receive the report and display the range.

The anchor transitions are not discussed in detailed here, but after reading the description of tag execution flow below the reader should be well equipped to similarly follow the anchor flow of execution.

The *instance_run()* function is the main function for the instance; it can be run periodically or as a result of a pending interrupt. It checks if there are any outstanding events that need to be processed and calls the *testapprun()* function to process them. It also reads the message/event counters and checks if any timers have expired. Below paragraphs describe the *testapprun()* sate machine in detail:

7.2.1 Initial state: TA_INIT

Function *testapprun()* contains the state machine that implements the two-way ranging function, the part of the code executed depends on the state and is selected by the "`switch (inst->testAppState)`" statement at the start of the function. The initial state "`case TA_INIT`"[1] performs initialisation and determines the next state to run depending on whether the "`inst->mode`" is selecting Tag or Anchor operation. Let's assume it is a tag and follow the execution of the next state. In the case of a tag we want to send a *Blink* message to allow an anchor to discover the tag and then initiate a ranging exchange, thus the state "`inst->testAppState`" is changed to "`TA_TXBLINK_WAIT_SEND`".

7.2.2 State: TA_TXBLINK_WAIT_SEND

In the state "`case TA_TXBLINK_WAIT_SEND`", we want to send the *Blink* message, so firstly we set up the message frame control data and then fill the rest of the message with the tag address. After sending the *Blink* message (using immediate send option with response expected parameter set), the state machine state will be changed to "`TA_TX_WAIT_CONF`", where the Tag awaits confirmation of the frame transmission.

As the *testapprun()* state machine state is set to "`TA_TX_WAIT_CONF`", and as that state has more than one use, "`inst->previousState = TA_TXBLINK_WAIT_SEND`" is set to as a control variable.

Before starting the transmission we also configure the receiver turn on delay and RX frame wait timeout. Receiver turn on delay is specified by `inst->rnginitW4Rdelay_sy` and RX frame wait timeout is specified by `inst->fwtoTimeB_sy` . The delays and timeouts are calculated as part of initialisation of the application by `instancesetreplydelay()` function.

As the transmission command had `DWT_RESPONSE_EXPECTED` set the receiver will turn on automatically and then time out if no message is received. After timing out the tag will go to sleep (enter DEEP SLEEP mode) and the microprocessor will wake it up after `tagBlinkSleepTime_ms` to restart blinking (this is done in "`TA_SLEEP_DONE`" state).

7.2.3 State: TA_TXPOLL_WAIT_SEND

In the state "`case TA_TXPOLL_WAIT_SEND`", we want to send the *Poll* message, so firstly we set up the destination address and then we call function *setupmacframedata()*, which sets up the all the other parameters/bytes of the *Poll* message.

The *testapprun()* state machine state is set to "`TA_TX_WAIT_CONF`", and as that state has more than one use, "`inst->previousState = TA_TXPOLL_WAIT_SEND`" is set to as a control variable.

Note: In the case if a tag sending the *Poll* message, this message is sent immediately. However in the case of the anchor responses (state "`case TA_TXRESPONSE_WAIT_SEND`" not documented here), and tag's *Final* message (state "`case TA_TXFINAL_WAIT_SEND`" as described in section 7.2.10 below), it is required to send the message at an exact and specific time with respect to the arrival of the message soliciting the response. To do this we use delayed send. This is selected by the "`delayedTx`" second parameter to function *instancesendpacket()*.

[1] The "TA_" prefix is because these are states in the "Test Application".

We also configure and enable the RX frame wait timeout, so that if the response is not coming, the Tag times-out and restarts the ranging.

7.2.4 State: TA_TXE_WAIT

This is the state for the tag which is called before the next ranging exchange starts (i.e. before the sending of next *Poll* message) or before the next *Blink* message is sent. Here we check if tag needs to enter a sleep mode before the next *Poll* or *Blink* messages are sent, and call *dwt_entersleep()* if sleep is required.

7.2.5 State: TA_TX_WAIT_CONF

In the state "`case TA_TX_WAIT_CONF`", we await the confirmation that the message transmission has completed. When the IC completes the transmission a "TX done" status bit is picked up by the device driver interrupt routine which generates an event which is then processed by the TX callback function (*instance_txcallback()*). The instance, after a confirmation of a successful transmission, will read and save the TX time and then proceed to the next state (`TA_RXE_WAIT`) to turn on the receiver and await a response message. The next state is thus set "`inst->testAppState = TA_RXE_WAIT`". See 7.2.6 below for details of what this does.

7.2.6 State: TA_RXE_WAIT

This is the pre-receiver enable state. Here the receiver is enabled and the instance will then proceed to the `TA_RX_WAIT_DATA` where it will wait to process any received messages or will timeout. Since the receiver will be turned on automatically (as we had `DWT_RESPONSE_EXPECTED` set as part of TX command), the state changes to `TA_RX_WAIT_DATA` to wait for the expected response message from the Anchor or timeout. We use automatic delayed turning on of the receiver as we know the exact times the responses are sent, as they are using delayed transmissions. This it is possible (and desirable for power efficiency) to delay turning on the receiver until just before the response is expected. (Delayed RX is not part of the IEEE standard primitive but is an extension to support this DW1000 feature). The next state is: "`inst->testAppState = TA_RXE_WAIT_DATA`".
Note: If a delayed transmission fails the transceiver will be disabled and the receiver will then be enabled normally in this state.

7.2.7 State: TA_RX_WAIT_DATA

The state "`case TA_RX_WAIT_DATA`" is quite long because it handles all the RX messages expected. This is not very robust behaviour. The tag should really only look for the messages expected from the anchor, (and vice versa). We "`switch (message)`", and handle message arrival as signalled by a received event. If a good frame has been received (`SIG_RX_OKAY`) we look at the first byte of MAC payload data (beyond the IEEE MAC frame header bytes) and "`switch(rxmsg->messageData[FCODE])`". FCODE is a DecaWave defined identifier for the different DecaRanging messages; see Figure 9, for details.

For the point of view of the discussions here the tag is awaiting the anchor's response or ranging initiation message so we would expect the FCODE to match "`RTLS_DEMO_MSG_ANCH_RESP`" or "`RTLS_DEMO_MSG_RNG_INIT`" when in Discovery phase. In this code, we note the RX timestamp of the message "anchorRespRxTime" and calculate "delayedReplyTime" which is when we should send the *Final* message to complete the ranging exchange. In this case our next (and subsequent states) are set to:

```
    inst->testAppState = TA_TXFINAL_WAIT_SEND ; // then send the final response
```

If the inst->tag2rxReport is set to rxmsg->messageData[RES_R2]; If this is set it means the anchor will send a report message, so that tag should wait for it before sending the next poll.

The state "case TA_RX_WAIT_DATA" also includes code to handles the "SIG_RX_TIMEOUT" message, for the case where the expected message does not arrive and the DW1000 triggers a frame wait timeout event. The DW1000 has an RX timeout function to allow the host wait for IC to signal either data message interrupt or no-data timeout interrupt[2]. When the timeout happens the Tag will go back to restart the ranging exchange.

```
    inst->testAppState = TA_TXE_WAIT ;       // check if should go to sleep before next
ranging
    inst->nextState = TA_TXPOLL_WAIT_SEND ; // then send the poll
```

7.2.8 State: TA_SLEEP_DONE

In this state the microprocessor will wake up the DW1000 from DEEP SLEEP once the sleep timeout expires. After waking up any of the DW1000 registers that are not preserved will be re-programmed and the state will change to inst->testAppState = inst->nextState;

7.2.9 State: TA_TXE_WAIT

In this state "case TA_TXE_WAIT", the Tag checks if it needs to go to sleep (low power state) before starting a new ranging exchange. If it comes into this state from sleep it will proceed to send the next *Poll* or *Blink* message.

Note: To save power the tag could poll one anchor and then "sleep" for a length of time before polling the same anchor again. In the ToF RTLS system, a tag might poll and range with a number of anchors and then enter a sleep mode before starting the process again.

7.2.10 State: TA_TXFINAL_WAIT_SEND

In the state "case TA_TXFINAL_WAIT_SEND", we want to send the *Final* message.

The *Final* message includes embedded the TX time-stamp of the tag's poll message "inst->tagPollTxTime" along with the RX time-stamp of the anchors response message "inst->anchorRespRxTime" and the embedded predicted (calculated) TX time-stamp for the final message itself which includes adding the antenna delay "inst->txantennaDelay".

So, now the *Final* message is composed and we call the "setupmacframedata()" function to prepare the rest of the message structure. The final message is sent at a specific time with respect to the arrival of the

[2] This idea here (although no code is yet written for this) is to facilitate the host processor entering a low power state until awakened by either the RX data arriving or the no data timeout.

message soliciting the response, this is done using delayed send, selected by the "`delayedTx`" second parameter to function "`instancesendpacket()`".

We finish the processing by setting control variable "`inst->previousState = TA_TXFINAL_WAIT_SEND`" to indicate where we are coming from and we set the "`inst->testAppState = TA_TX_WAIT_CONF`" selecting this as the new state for the next call of the "`testapprun()`" state machine.

7.2.11 State: TA_TX_WAIT_CONF (for *Final* message TX)

In the state "`case TA_TX_WAIT_CONF`", (as detailed in section 7.2.5 above) we await the confirmation that the message transmission has completed.

When we get this, we use the "`inst->previousState == TA_TXFINAL_WAIT_SEND`" to identify that we are a tag who has just sent the final and we either: (a) Go on to send another poll message (perhaps after a sleep period of inactivity), or, (b) Await the reception of a ranging report from the anchor. The choice of (a) or (b) is dependent on Boolean control variable "`inst->tag2rxReport`.

Also the tag will not use the delayed receive, but turn on the receiver immediately and wait for the reception of the ToF report message.

7.2.12 CONCLUSION

The above should be enough of a walkthrough of the state machine that the reader should be able to decipher the anchor activity (and any remaining activity of tag).

In summary the anchor waits indefinitely in the state "`case TA_RX_WAIT_DATA`" until it receives a blink message. Then it will associate with the tag that sent it and send the ranging initiation message. Once it receives the poll it starts the ranging exchange and finishes with a calculation of ToF (range) report, which it reports to the LCD and also, optionally, sends back to the tag.

8 APPENDIX B – BIBLIOGRAPHY

1	DecaWave DW1000 Datasheet
2	DecaWave DW1000 User Manual
3	EVK1000 User Manual
4	IEEE 802.15.4-2011 or “IEEE Std 802.15.4™-2011” (Revision of IEEE Std 802.15.4-2006). IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Computer Society Sponsored by the LAN/MAN Standards Committee. Available from http://standards.ieee.org/

9 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/