SOURCE CODE GUIDE

DECARANGING (PC) SOURCE CODE

Understanding and using the DW1000 DecaRanging source code

Version 4.5

This document is subject to change without notice

DISCLAIMER

Mailing address: -
Decawave Ltd.,
Adelaide Chambers,
Peter Street,
Dublin 8

TABLE OF CONTENTS

1 INTRODUCTION

This document, "*DecaRanging (PC) Source Code Guide*" is a guide to the Decawave's two-way ranging application source code of the "DecaRanging" ranging PC demo.

Companion documents: "*EVK1000 User Guide*" gives an overview of the DW1000 Evaluation kit and describes how to install and 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.

Section 8 - 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: `"DecaRanging MP Version 3.03"` application version and `"DW1000 Device Driver Version 02.16.00"` driver version.

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

2 BUILDING THE CODE

The code is built under Microsoft Visual C++ 2010 Express Edition. Once this is installed, simply open the project file "DecaRanging.vcxproj" and build via a right click on the application within the "Solution Explorer" pane.

3 RUNNING THE CODE

To run DecaRanging.exe on a computer that does not have Visual C++ 2010 installed, the runtime libraries, "Microsoft Visual C++ 2010 Redistributable Package (x86)" available from Microsoft need to be installed. These are available at:

http://www.microsoft.com/download/en/details.aspx?id=5555

4 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.

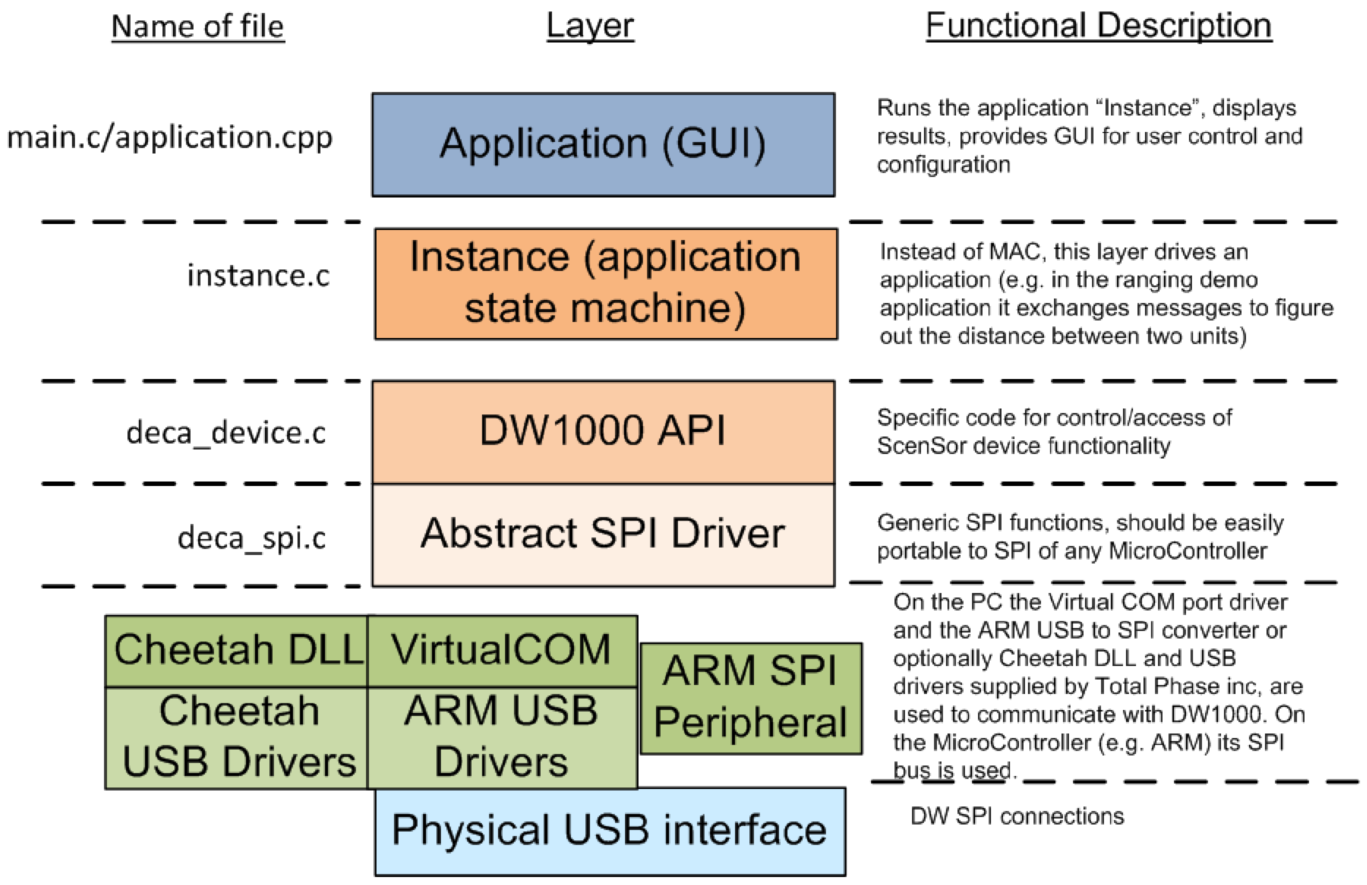

Name of file
Layer
Functional Description
main.c/application.cpp
Application (GUI)
Runs the application "Instance", displays results, provides GUI for user control and configuration
instance.c
Instance (application state machine)
Instead of MAC, this layer drives an application (e.g. in the ranging demo application it exchanges messages to figure out the distance between two units)
deca_device.c
DW1000 API
Specific code for control/access of ScenSor device functionality
deca_spi.c
Abstract SPI Driver
Generic SPI functions, should be easily portable to SPI of any MicroController
Cheetah DLL
VirtualCOM
Cheetah USB Drivers
ARM USB Drivers
ARM SPI Peripheral
On the PC the Virtual COM port driver and the ARM USB to SPI converter or optionally Cheetah DLL and USB drivers supplied by Total Phase inc, are used to communicate with DW1000. On the MicroController (e.g. ARM) its SPI bus is used.
Physical USB interface
DW SPI connections

Figure 1: Software layers in DecaRanging

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

5 DETAILED DESCRIPTION OF DECARANGING CODE STRUCTURE

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

5.1 Virtual COM port/Cheetah and USB interface

The DW1000 is controlled via its SPI bus. There are two ways for the *DecaRanging* PC application to connect and control the DW1000 IC on the DW1000 evaluation boards:

a) Via the USB interface – The EVB1000's on-board ARM microcontroller's a preinstalled application can operate as a simple USB to SPI controller, passing the *DecaRanging* PC applications SPI accesses to/from the DW1000 IC. Please refer to the EVK1000 User Manual document for details of how to configure and enable this mode of operation.

b) Via the SPI interface header – The EVK1000 User Manual document for details of how to configure EVB1000 for DW1000 direct SPI access. The *DecaRanging* PC application can optionally employ a *Cheetah* USB-to-SPI convertor control the DW1000 IC directly. This is a legacy operation supported on earlier versions of DecaRanging. The *Cheetah*[1] USB-to-SPI convertor is a commercial product of Total Phase, Inc. This mode of operation may be useful to control the DW1000 on a customer developed board (through a suitable header or wiring) to validate the board performance compared to the EVB1000.

The Decawave's USB-to-SPI converter is an application that runs on the EVB1000 HW. When using the DecaRanging application in this mode, the SPI commands are formatted and sent from the PC DecaRanging application through the Virtual COM port driver (deca_vcspi.c) over the USB interface to the ARM on the EVB1000. There the ARM USB-to-SPI application reads the commands and talks to the DW1000 though the SPI interface.

The Cheetah USB-to-SPI converter from Total Phase Inc. provides the SPI interface functions on the PC. The Cheetah comes with USB drivers that need to be installed before using the product, a DLL needed at run time and the cheetah.c file built into the application. Please refer to Total Phase documentation for more details, at http://www.totalphase.com/.

5.2 Abstract SPI Driver – SPI Level code.

The file deca_spi.c provides functions *openspi()*, *closespi()*, *writetospi()* and *readfromspi().* These call appropriate Cheetah SPI or Virtual COM port driver functions.

5.3 Device Driver – DW1000 Device Level Code

The file deca_device_api.h provides a library of API functions to control and configure the DW1000 registers and implement certain functions for device level control as listed below.

[1] Cheetah SPI Host Adapter - These are available from http://www.totalphase.com/products/cheetah_spi/ for USD $350.

The API functions are described in the "*DW1000 Device Driver Application Programming Interface (API) Guide*" document.

5.4 Instance Code

The instance code (in instance.c) provides a simple ranging demo 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 (*Listener*, *Tag* or *Anchor*) depending on the role configuration set at the application layer. The *Listener* mode just receives messages and reports their reception via the GUI application layer. The *Tag* and *Anchor* modes operate as a pair to provide the two-way ranging demo functionality between two units.

Initially the tag is in a discovery phase where it 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 waits for a period (default of 1 second) before blinking again. The listener will listen for any blinks. When anchor mode is chosen the user will select the tag it wishes to pair with and then the anchor will wait for the blink message from that tag. The anchor will then pair with the tag when it gets the *Blink* message from it, and send the *Ranging Initiation* message to exit from the *Discovery Phase* and enter *Ranging Phase*.

Figure 2 shows the arrangement and general operation of the two-way ranging as implemented by the DecaRanging application. Section 6 describes the ranging algorithm in more detail including the format of the messages exchanged and the calculations performed.

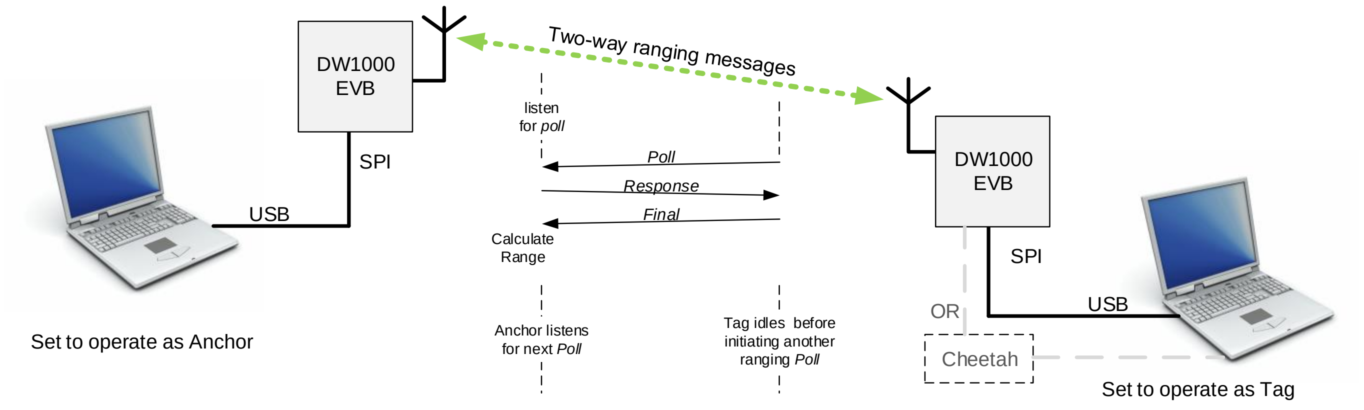


Figure 2: Two way ranging in DecaRanging

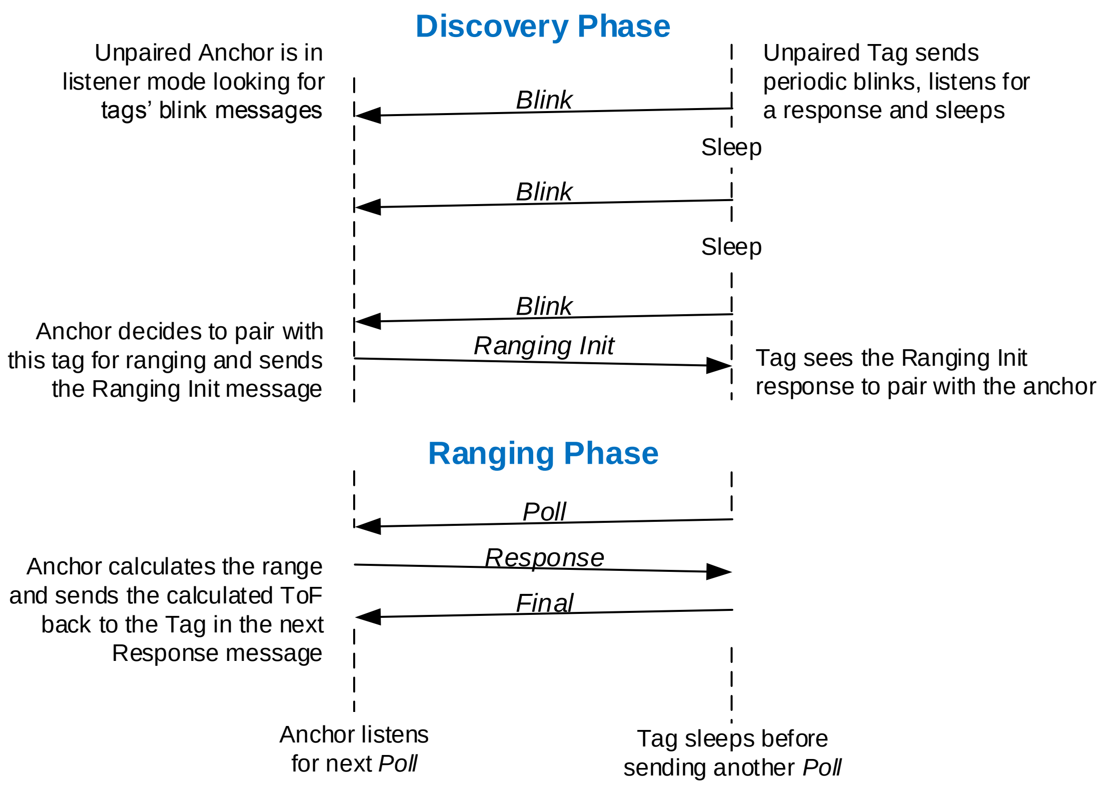

Discovery Phase
Unpaired Anchor is in listener mode looking for tags' blink messages
Unpaired Tag sends periodic blinks, listens for a response and sleeps
Blink
Sleep
Blink
Sleep
Blink
Anchor decides to pair with this tag for ranging and sends the Ranging Init message
Ranging Init
Tag sees the Ranging Init response to pair with the anchor
Ranging Phase
Poll
Anchor calculates the range and sends the calculated ToF back to the Tag in the next Response message
Response
Final
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 to the tag for it to display, included in anchor's next *Response* message. 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. Section 6 describes the ranging algorithm in more detail including the format of the messages exchanged and the calculations performed.

As true interrupt handling is not possible via the cheetah or the virtual COM port drivers so the *dwt_isr()* function is instead called from *instance_run()* essentially implementing the device interface by polling for events "RX Frame Received" or "TX Frame sent". In porting to a microcontroller, at the discretion of this system integrator, the *dwt_isr()* function may be called by system specific handler triggered by the interrupt line from DW1000.

Sitting above this instance level is the windows GUI application that provides the user interface described in section 2.5 below. The reader is directed to the code in file instance.c and instance_common.c for more details on this layer. The file instance_log.c contains logging functions used to log received data and events.

5.5 Windows GUI Application

The Windows GUI Application (decaranging.cpp) contains the main entry point for the DecaRanging ranging demo application and also all the windows graphical user interface code.

There is a main window where status lines display the Time of Flight (TOF) and distance estimated. The main window includes a control panel to pause and resume operation, select tag, anchor and listener roles and to configure other parameters. Sub-menu dialogues allow for display of registers, accumulator (channel response graph), timing setup, log file enable, etc.

Separate companion documents: "*EVK1000 User Guide*" and "*DecaRanging Ranging Demo Application (PC Version) User Guide*" give an overview of DW1000 Evaluation kit (EVK1000), and, describe how to install and operate the DecaRanging Application.

5.6 DW1000 time units and manipulation

File deca_util.c contains functions which are used mainly for converting the DW1000 time units to seconds and vice versa.

The DW1000 employs 40-bit values for timestamps, the lowest order bit aligning with the IEEE 802.15.4a standard timestamp recommendation of being $^{1}/_{128}$ of a 499.2 MHz clock period. To operate on these in the windows DecaRanging application uses the 64-bit native int64 supported by VC++ and modern desktop PC CPUs.

In a practical ranging application in the more modestly sized embedded microprocessor it may not be necessary to operate on 64-bit or even 40-bit numbers since with a moderate response time the complete round trip can fit into a 32-bit number, and where it doesn't, shifting left to discard a few of the low-order bits will generally not reduce the precision of the result.

5.7 Folder structure

Table 1 gives the folder structure of the DecaRanging application source code given along with a brief description of each folder's content. The reader is referred to the other sections of this document for more details on the code structure and organisation.

Table 1: List of folders in the DecaRanging application

Folder		Brief description
DecaRanging		
	application	DecaRanging's high level application and instance layers
	compiler	Standard libraries inclusions
	decadriver	DW1000 device driver
	platform	High level driver (including interrupt management) for various external and DW peripherals

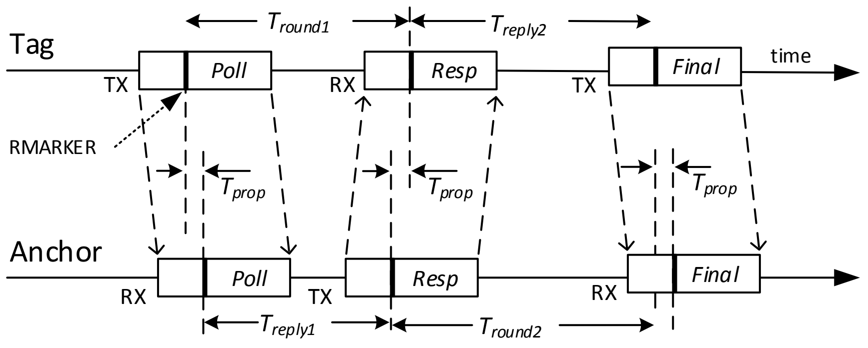
6 Ranging Algorithm

This section describes the ranging algorithm used in the DecaRanging ranging demo application. In contrast to some earlier versions of DecaRanging demo, 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.

6.1 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), it embeds this time in the message before initiating the delayed sending of the *Final* message to the anchor. It will also take the ToF from the previous ranging exchange and display the distance.
- The anchor receiving this *Final* response message (at T_{RF}) now has enough information to work out the range. $T_{round1} = T_{RR} - T_{SP}$; $T_{reply1} = T_{SR} - T_{RP}$; $T_{round2} = T_{RF} - T_{SR}$; $T_{reply2} = T_{SF} - T_{RR}$.
- It is to be noted that, for small ranges, a received signal level bias correction has to be applied to calculated raw range. More details about this bias correction can be found in APS011 "Sources of error in TWR schemes".
- In the DecaRanging ranging demo the anchor will send the calculated ToF to the Tag to give it something to display. This ToF will be sent in the next Response message. Figure 4 shows this exchange and gives the formula used in the calculation of the range.



The *Final* message communicates the tag's T_{round} and T_{reply} times to the anchor, which calculates the range to the tag as follows:

$$T_{prop} = \frac{T_{round1} \times T_{round2} - T_{reply1} \times T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}}$$

Figure 4: Range calculation in DecaRanging

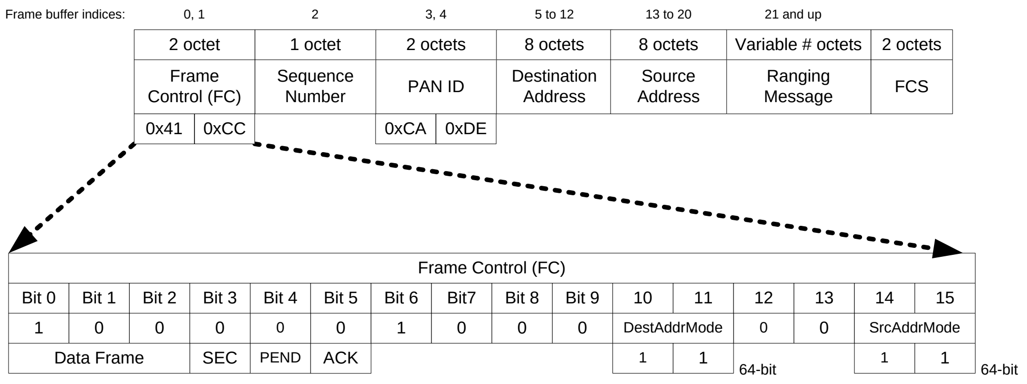
- After this the anchor turns on its receiver again to await the next poll message, while the tag meanwhile counts off the delay period to the next ranging attempt.

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

Five messages are employed in the tag/anchor two-way ranging, two in the *Discovery* phase (the blink and ranging initiation messages) and three in the *Ranging* phase (the poll message, the response message, the final 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.

6.2.1 General ranging frame format

The general message format is the IEEE 802.15.4 standard encoding for a data frame. Figure 5 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 6.2.2 below. In a real 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.



Frame buffer indices:	0, 1	2	3, 4	5 to 12	13 to 20	21 and up	
	2 octet	1 octet	2 octets	8 octets	8 octets	Variable # octets	2 octets
	Frame Control (FC)	Sequence Number	PAN ID	Destination Address	Source Address	Ranging Message	FCS
	0x41 0xCC		0xCA 0xDE				

Frame Control (FC)															
Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit7	Bit 8	Bit 9	10	11	12	13	14	15
1	0	0	0	0	0	1	0	0	0	DestAddrMode		0	0	SrcAddrMode	
Data Frame			SEC	PEND	ACK					1	1	64-bit		1	1	64-bit

Figure 5: 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 three ranging messages it is. These are shown in Figure 7 and described in sections 6.2.3 to 6.2.6. 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 5 to form the complete ranging message in each case.

6.2.2 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 6.

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

Figure 6: the 12 octet minimal blink frame

Poll Message

Frame buffer indices:	21
	1 octet
	Function code
	0x21

Response Message

Frame buffer indices:	21	22	23 and 24	25 to 29
	1 octet	1 octet	2 octets	5 octets
	Function code	Activity	Activity Parameter	Previous ToF
	0x10	0x02	0x0000	-

Final Message

Frame buffer indices:	21	22 to 26	27 to 31	32 to 36
	1 octet	5 octets	5 octets	5 octets
	Function code	Poll Message TX Time-Stamp	Response Message RX Time-Stamp	Embedded Predicted Final TX Time-Stamp
	0x29	-	-	-

Ranging Init Message

Frame buffer indices:	21	22 and 23	24 to 25	26 to 27
	1 octet	2 octets	2 octets	2 octets
	Function code	Tag 16-bit Address	Response Delay Anc	Response Delay Tag
	0x20	-	-	-

Figure 7: Ranging message encodings

6.2.3 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.

6.2.4 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 to 4	0x0000	This two octet parameter is unused for activity 0x02.

Octet #’s	Value	Description
5 to 9	-	This five octet field is the ToF from the previous ranging exchange. 40-bit DW1000 time units.

6.2.5 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 six 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 six 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 six 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 6.2.5.1 below.

6.2.5.1 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.

6.2.6 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 7 octets in length. Table 4 lists and describes the individual fields within the ranging initiation message.

Table 4: 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.

Octet #'s	Value	Description
4 to 5	-	This 16-bit bit field gives the anchor response time to be used in the following ranging exchange: - bit 0 to 14: value - bit 15: 0 for microseconds, 1 for milliseconds
6 to 7	-	This 16-bit bit field gives the tag response time to be used in the following ranging exchange: - bit 0 to 14: value - bit 15: 0 for microseconds, 1 for milliseconds

6.3 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.

In DecaRanging the user sets the antenna delay on the main page control panel. The value specified is divided equally between TX and RX antenna delays. In a real system where multiple vendors may be operating it would be more important to attribute individual TX and Rx delays appropriately. 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.

6.4 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 TX timestamp stored in the register, and no software adjustment is necessary.

6.5 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.

7 CODE / SYSTEM ISSUES

7.1 Antenna Delay

The antenna delay may need changing if operating modes are changed, i.e. changing channel frequency or PRF might mean that delay needs to be changed up/down. The DecaRanging software takes care of the adjustment when PRF setting is changed, maintaining separate antenna delays for 16 MHz or 64 MHz PRF values.

Note: If the antenna delay value is 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, (i.e. TX and RX message counts show interaction without errors), 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.

8 OPERATIONAL FLOW OF EXECUTION

This section 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 at the same time as reading this description, and find each referred item in the source code and follow the flow as described here.

8.1 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.

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:

8.1.1 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`"[2] 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`".

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

8.1.2 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 IDLE mode and wait `blink_period_ms` to restart blinking (this is done in "`TA_SLEEP_DONE`" `state).`

8.1.3 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 8.1.10 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()*.

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.

8.1.4 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.

8.1.5 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 8.1.6 below for details of what this does.

8.1.6 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 using delayed transmissions. It is possible (and desirable for power efficiency) to delay turning on the receiver until just before the response is expected. The next state is: "`inst->testAppState = TA_RXE_WAIT_DATA`".
Note: If a delayed transmission fails (i.e. due to starting it too late) then the recovery disables the transceiver and the receiver will then be enabled normally in this state.

8.1.7 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 7, 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
```

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[3]. When the timeout happens the Tag will go back to restart the ranging exchange.

```
    inst->testAppState = TA_TXE_WAIT ;
    inst->nextState = TA_TXPOLL_WAIT_SEND ; // send next poll
```

8.1.8 State: TA_SLEEP_DONE

In this state the microprocessor will wait for the expiration of the `blink_period_ms` or `poll_period_ms` timeout, depending on the next message to send. Then the state will change to `inst->testAppState = inst->nextState;`

8.1.9 State: TA_TXE_WAIT

In this state "`case TA_TXE_WAIT`", the Tag will update the displayed range from anchor's report and proceed to TA_SLEEP_DONE state if next frame to transmit is a blink or a poll. If next frame to transmit is a final message, tag will proceed to state TA_TXFINAL_WAIT_SEND.

8.1.10 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 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.

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

In the state "`case TA_TX_WAIT_CONF`", (as detailed in section 8.1.5 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 go on to send another poll message (perhaps after a period of inactivity).

[3] 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.

8.1.12 CONCLUSION

That 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 poll 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 GUI.

9 PC USB TO SPI PROTOCOL HANDLING EXPLAINED

9.1 Introduction

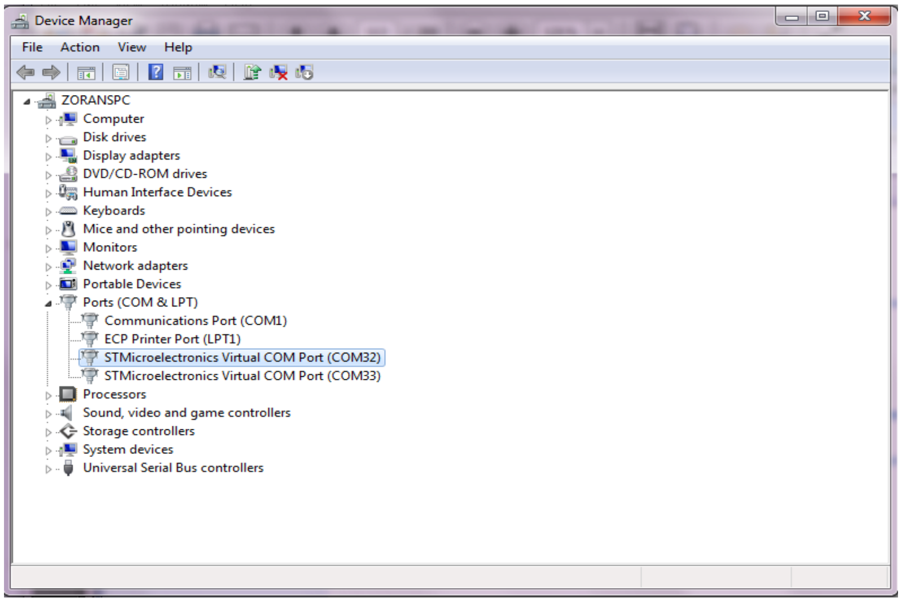
The DecaRanging PC application can communicate to the DW1000 on the EVB1000 board over the Virtual COM port (over USB) interface via “USB-to-SPI” application running on the EVB1000 STM32 processor.

The EVK1000 uses STMicroelectronics STM32 ARM cortex M3 microcontroller and employs the STM32 USB driver from STMicroelectronics. There is a “USB to SPI” application which operates on the EVB1000 microcontroller to read and write data from/to the DW1000 and is controlled from the DecaRanging PC application. When EVB1000 is connected to the PC in this “USB to SPI” mode it appears as a COM port see Figure 8.

Application (PC) software can be written which writes and reads to the COM port needing no knowledge of the translation happening in the provided driver and hardware combination. The protocol used over the USB virtual COM port, between the DecaRanging PC application and the EVK1000 software, and used for DW1000 SPI access, is described below:

9.2 Protocol to Write and Read DW1000 SPI data

Connect the EVB1000 to the PC and the Device Manager will show STMicroelectronics Virtual COM port.



ZORANSPC
- Computer
- Disk drives
- Display adapters
- DVD/CD-ROM drives
- Human Interface Devices
- Keyboards
- Mice and other pointing devices
- Monitors
- Network adapters
- Portable Devices
- Ports (COM & LPT)
  - Communications Port (COM1)
  - ECP Printer Port (LPT1)
  - STMicroelectronics Virtual COM Port (COM32)
  - STMicroelectronics Virtual COM Port (COM33)
- Processors
- Sound, video and game controllers
- Storage controllers
- System devices
- Universal Serial Bus controllers

Figure 8: Device Manager Window showing ST Virtual COM Port

On starting up the Decaranging PC application will get a handle to the COM port and try to connect to the “USB-to-SPI” application on the EVB. This is done by sending `"deca?"` string and checking the reply. Each COM port with *STMicroelectronics Virtual COM Port* name will be checked in turn. The reply should be `"EVB1000 USB2SPI X.0"`, where X can be 1 or 2. The deca_vcspi.c function *findandopenEVB1000COMport()* is used for this.

Then to do an SPI write or read transaction, the PC application sends a command comprising of at least 8 bytes, as defined below:
<0x02> - this is the first byte and specifies the start of command
<0x0X> - here the LSB specifies the SPI speed and SPI read/write operation, bit 0 = 1 for write, 0 for read
<0xTL> - low byte of 16-bit total length of the command
<0xTH> - high byte of 16-bit total length of the command
<0xBL> - low byte of 16-bit *body length* of the command
<0xBH> - high byte of 16-bit *body length* of the command
<N>   - number of data bytes as passed into SPI read/write function (header + body)
<0x03> - this is the last byte and ends the command

Note: *body length* is the length of the SPI data buffer, header length is the length of SPI header and total length comprises 7 bytes + body length + header length, hence the minimum 9 byte length required for a write operation and 8 bytes for reading (the reading does not have any body length fields).

After sending the above command to the EVB, the application waits for a response. In the case of an SPI write the response is <0x02><0x00><0x03>, for reading the response is <0x02><0x00><read data bytes><0x03>.

9.3 Example

To read device ID, i.e. read four bytes from register 0x0, we send following to the EVB1000 (8 bytes):
<0x02><0x00><0x08><0x00><0x00><0x00><0x00><0x03>
and the EVB sends back the following (7 bytes):
<0x02><0x00><0x30><0x01><0xCA><0xDE><0x03>
The register read of register 0x0 returned 0xDECA0130.

10 Bibliography

Ref	Author	Date	Version	Title
[1]	Decawave		Current	DW1000 Data Sheet
[2]	Decawave		Current	DW1000 User Manual
[3]	Decawave		Current	EVK1000 User Manual
[4]	Decawave		Current	DecaRanging Ranging Demo Application (PC Version) User Guide
[4]	IEEE		2011	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/

11 Document History

Table 5: Document History

Revision	Date	Description
3.7	20th December 2013	Initial release for production device.
4.2	11th November, 2014	Scheduled update
4.3	30th September, 2015	Scheduled update
4.4	17th February, 2016	Fix for reference not found, changed DecaWave to Decawave
4.5	22nd June 2018	Document update with new logo

12 Major Changes

12.1 Release 4.3

Page	Change Description
All	Update of version number to 4.3
All	Various typographical changes

12.2 Release 4.4

Page	Change Description
All	Update of version number to 4.4

All	Change DecaWave to Decawave, and copyright date to 2016
12, 14	Fix “Error! Reference source not found.”
Chapter 9	Added USB to SPI protocol explanation

13 FURTHER INFORMATION

For further information on this or any other Decawave product, please refer to our website www.decawave.com or contact a sales representative at sales@decawave.com