Decawave Is Now Qorvo

DW3000-TWR-demo

Version 1.2

TABLE OF CONTENTS

1 INTRODUCTION 4

1.1 BASIC OPERATION 4

1 DESCRIPTION OF JUNIPER ARM PLATFORM 5

1.1 JUNIPER ARCHITECTURE 5

1.1.1 Top-level applications layer 5

1.1.2 Core tasks 5

1.1.3 Drivers 6

1.1.4 FreeRTOS 6

1.1.5 Target HAL 6

1.2 JUNIPER SOURCE CODE - FOLDER STRUCTURE 7

2 OPERATION OF THE MAIN CODE 8

2.1 STARTUP, INITIAL HAL CONFIGURATION AND STARTING OF THE KERNEL 8

2.2 CORE TASKS 9

2.2.1 Default task 9

2.2.2 USB_VBUS driver 9

2.3 CONTROL TASK: MODES OF OPERATION 10

2.3.1 Command mode of Control task 11

2.3.2 Controlling of the embedded applications over a PC GUI app 14

2.4 FLUSH TASK 20

2.5 RTOS EXTENSIONS USED IN THE APPLICATION 21

3 IN DEEP ABOUT TOP-LEVEL APPLICATIONS 22

3.1 NODE TOP-LEVEL APPLICATION 22

3.1.1 Concept of Discovered and Known tags lists 24

3.1.2 Discovery and ranging to tags 24

3.1.3 The superframe, the wakeup timers and the tag's slot correction 25

3.2 TAG TOP-LEVEL APPLICATION 27

3.2.1 The Discovery phase 28

3.2.2 The Ranging phase 28

3.3 USB2SPI TOP-LEVEL APPLICATION 30

3.4 TCWM TOP-LEVEL APPLICATIONS 30

3.5 TCFM TOP-LEVEL APPLICATIONS 31

3.6 LISTENER TOP-LEVEL APPLICATION 31

3.7 LOW POWER MODE IN JUNIPER APPLICATIONS 31

4 BUILDING AND RUNNING THE CODE 32

4.1 BUILDING THE CODE 32

4.2 DOWNLOADING AND RUNNING THE CODE 32

5 APPENDIX A 33

5.1 TWO WAY RANGING ALGORITHM 33

5.2 UWB CONFIGURATION AND TWR TIMING PROFILE USED IN THE SYSTEM 34

5.3 FRAME TIME ADJUSTMENTS 34

5.4 UWB MESSAGES, USED IN TWR PROCESS 35

5.4.1 Tag blink message 35

5.4.2 Ranging Config message 35

5.4.3 Ranging messages 37

5.4.4 The Poll message 37

5.5 Slot Time correction method in between Node and Tag 39
5.6 The application architecture in the flowchart 40
6 BIBLIOGRAPHY 44
7 REVISION HISTORY 45
8 FURTHER INFORMATION 46

LIST OF TABLES

Table 1: Anytime commands 11
Table 2: Commands to change mode of operation 12
Table 3: Commands to change run-time parameters 12
Table 4: The PC to the Node top-level application commands 15
Table 5: Node top-level application JSON outputs 17
Table 6: Top-level applications and corresponded commands 22
Table 7: UWB mode of operation of system 34
Table 8: Fields within the Ranging Config message 36
Table 9: List of Function Codes in the TWR exchange 37
Table 10: Fields within the ranging Poll message 37
Table 11: Fields within the ranging Response message 38
Table 12: Fields within the ranging Final message 38

LIST OF FIGURES

Figure 1: Discovery and ranging phases 4
Figure 2: Architecture of the TWR application 5
Figure 3: Juniper source code folder structure 7
Figure 4: Initial startup workflow 8
Figure 5: Control task (in command mode) sends EVENT to the Default task 10
Figure 6: Control Task (in data mode) sends signal to the Usb2Spi application 10
Figure 7: Output data using shared Report buffer 20
Figure 8: *Node* top-level application 23
Figure 9: Tasks used in the node application 24
Figure 10: Superframe structure and ranging exchange time profile 25
Figure 11: *Tag* top-level application 27
Figure 12: *Tag's* Discovery phase: Rx Thread 28
Figure 13: Tag's Ranging phase: Poll Thread 29
Figure 14: Tag's Ranging phase: Rx Thread 29
Figure 15: Usb2Spi top-level application 30
Figure 16: Distance calculation in TWR 33
Figure 17: TWR timing profile 34
Figure 18: Encoding of Tag's 12-bytes blink message 35
Figure 19: Frame format of Ranging Config message 36
Figure 20: Frame format used for Ranging 37
Figure 21: Node-Tag Slot Time correction method 39
Figure 22 Common platform blocks 41
Figure 23 Operational flow on the "tag" top-level application 42
Figure 24 Operational flow on the "node" top-level application 43

1 INTRODUCTION

This document describes the operation of DW3000 TWR demo (Juniper) applications. The demo consists of two EVK3000 units with DW3000 IC, one running a *Node* application and the other a *Tag* application. The basic operation of the demo is outlined below.

The application source code employs a real-time operating system (FreeRTOS) [www.freertos.org], however it is possible to use another RTOS or potentially remove the operating system and implement the node application using a Round Robin scheduling with interrupts technique (main super loop).

1.1 Basic operation

The basic operation of the system is as follows: The node performs double-sided two-way ranging with a tag, and then calculates the range (and optionally PDoA with tag's x, y location coordinates) and reports the results to an external application (e.g. PC GUI application). The PC GUI application then plots the position of the tags based on the reported values.

Tags start operating in *Discovery* mode, periodically sending Blink messages, 5.4.1.The node listens for Blink messages from tags and when a Blink message is received, the node responds to the tag with a Ranging Config message, 5.4.2. The Ranging Config message provides information to the tag describing how to perform ranging with the node. This information includes the PAN ID, the short address of the node, a short address assignment for the tag, timing parameters for the ranging phase to be used at the start of the next ranging exchange. Upon receiving the Ranging Config message, the tag operation changes to the *Ranging* mode where it periodically initiates ranging exchanges with the node. Figure 1 describes the Discovery and Ranging phases.

Each ranging exchange starts with the tag sending a Poll message, 5.4.4. When the node receives the Poll message, it replies with a Response message, 5.4.4.1, and the tag completes the ranging exchange by sending a Final message, 5.4.4.2. The node then calculates the range to the tag which it reports via USB/UART for displaying by the PC GUI application. NOTE: when using configurations with STS modes, the STS timestamp should be validated before replying to the Poll or Response messages.

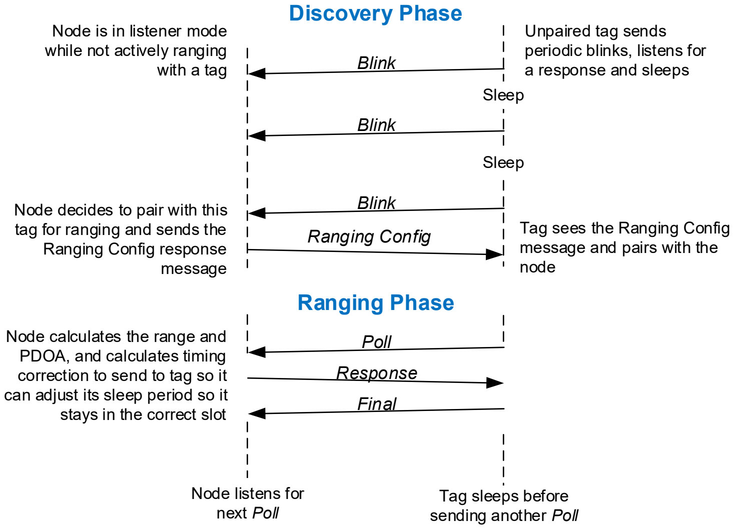

Discovery Phase
Node is in listener mode while not actively ranging with a tag
Unpaired tag sends periodic blinks, listens for a response and sleeps
Blink
Sleep
Blink
Sleep
Node decides to pair with this tag for ranging and sends the Ranging Config response message
Blink
Ranging Config
Tag sees the Ranging Config message and pairs with the node
Ranging Phase
Node calculates the range and PDOA, and calculates timing correction to send to tag so it can adjust its sleep period so it stays in the correct slot
Poll
Response
Final
Node listens for next Poll
Tag sleeps before sending another Poll

Figure 1: Discovery and ranging phases

1 Description of Juniper ARM platform

Juniper platform is based on STM32 ARM Cortex M4F MCU. The sections below discuss the architecture, structure and workflow of the software, residing in the microcontroller. This should enable the developer to understand the philosophy and be able to add functionality or port the project to another platform, if necessary (i.e. to other Cortex M4 or to another architecture).

1.1 Juniper architecture

An overview of the architecture is given by Figure 2 below. The figure shows, that the platform can be structured as layers (or levels) of code, namely: “Top-level applications”, “Core tasks”, “FreeRTOS”, “Drivers” and “HAL” which interacts with the corresponding physical interfaces. The detailed scheme of interaction between software blocks is given in **Appendix A**, section 5.

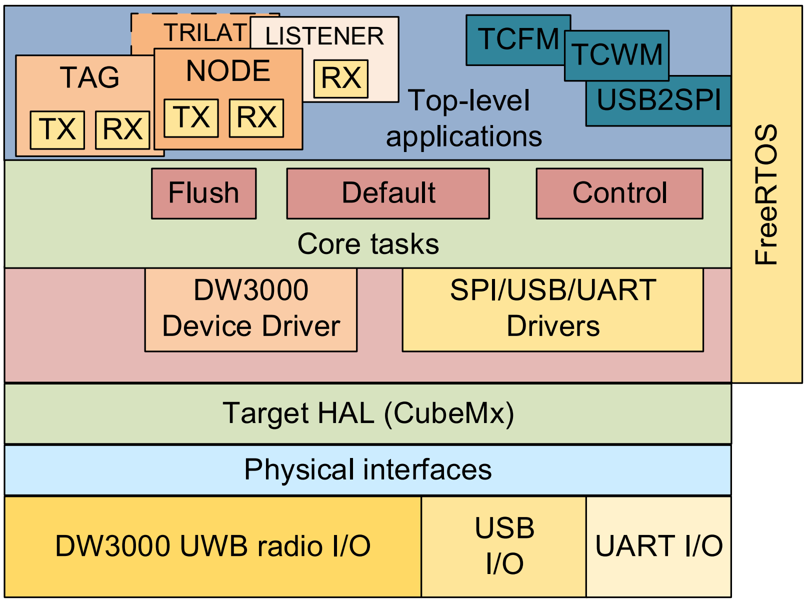


Figure 2: Architecture of the TWR application

1.1.1 Top-level applications layer

This is the top level of the software defining the operation of the Juniper unit. It has seven separate top-level applications / modes of operation. There are two main modes, namely “NODE” and “TAG”, whose operation and functionality was introduced in section 1.1, while the other five operational modes are used for testing purposes.

These are: TCFM – Test Continuous Frame transmission Mode; TCWM – Test Continuous Wave transmission Mode; TRILAT – the Mode when Juniper Runs the Trilateration function on the top of the NODE application; LISTENER puts the device into receive mode and reports any received packets, and USB2SPI test mode, allowing external test software direct access to the DW3000 over SPI.

The top-level applications cannot run concurrently since they use the same resources: e.g. NODE application configures the DW3000 to do two-way ranging, while TCWM application initializes DW3000 to run a Continuous Wave mode test.

1.1.2 Core tasks

The core tasks are always running once the RTOS kernel has been started. These core tasks are described in section 2.2. The core tasks are:

- The *Default* core task is responsible for starting one of the top-level applications. It receives events from the *Control* task to switch to a particular mode of operation and starts corresponding top-level application. This is described in detail in section: 2.2.1
- The *Control* core task is responsible for reception and execution of *commands* from external IO interfaces USB and/or UART. The *Control* task can also pass *data* from those I/O interfaces to the top-application layer. This is described in detail in section: 2.3.
- The *Flush* core task is responsible for transmitting of any output data to the external I/O interfaces USB and/or UART. This is described in detail in section 2.4.

1.1.3 Drivers

The drivers are responsible for translating of higher-layer requests and the specific sequences to control particular peripherals.
To control the DW3000 UWB radio transceivers the DW3000 API and the device driver is incorporated as a library. Other physical interfaces have corresponding drivers, namely UART, USB, etc.

1.1.4 FreeRTOS

Juniper platform runs under the FreeRTOS operating system control. This is a CMSIS compatible RTOS, thus the Juniper software is portable to other CMSIS-RTOS if needed. The CMSIS-RTOS is a common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables therefore software templates, middleware, libraries, and other components that can work across supported the RTOS systems.

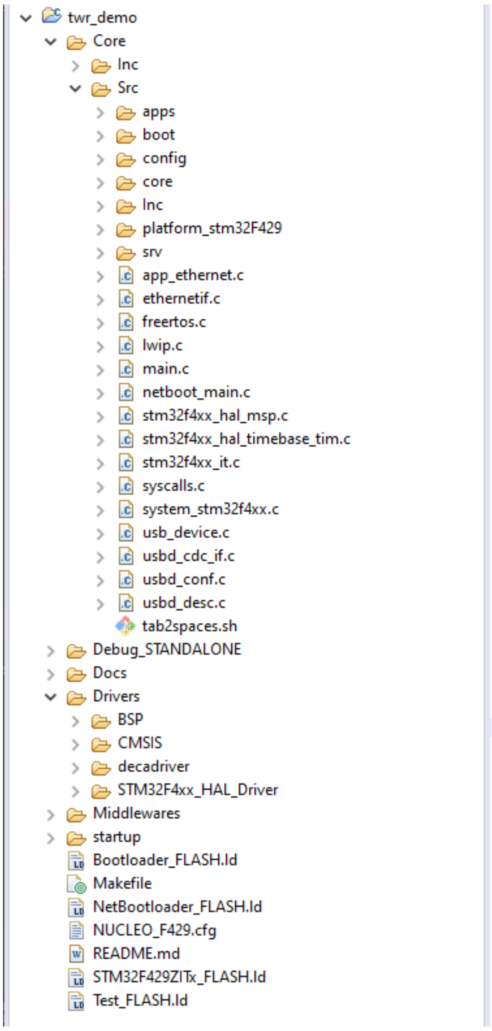
Juniper software has two layers of operation: RTOS tasks, which run concurrently, and bare-metal functions, which run under the operation of the RTOS based application. Potentially it is possible to remove RTOS tasks, and implement their functionality using Round Robin technique (super loop), however this may lead to a complex times management and complexity in the code (spaghetti-code). Some of the bare-metal functions, such as SPI driver are running below the RTOS priority, which means that they cannot be interrupted by the RTOS or use RTOS mechanisms for communications. This is done to increase the performance of running applications.

1.1.5 Target HAL

Juniper software employs a combination of HAL (Hardware Abstraction Layer) and LL (Low-Level) drivers, generated by the CubeMX PC application from ST microelectronics, to interact to the physical peripherals of the ST microcontroller [www.st.com].
For performance reasons, LL drivers are used for SPI & DMA drivers to the DW3000 chips. HAL drivers are used for all other peripherals: UART, USB and GPIO.

1.2 Juniper source code - folder structure

The initial folder structure was created using the CubeMX software from ST microelectronics, see **Figure 3** below. CubeMX is a tool that simplifies the initial creation of a project by providing the complete code for the initialization of the peripherals.



```
twr_demo
  Core
    Inc
    Src
      apps
      boot
      config
      core
      Inc
      platform_stm32F429
      srv
      app_ethernet.c
      ethernetif.c
      freertos.c
      lwip.c
      main.c
      netboot_main.c
      stm32f4xx_hal_msp.c
      stm32f4xx_hal_timebase_tim.c
      stm32f4xx_it.c
      syscalls.c
      system_stm32f4xx.c
      usb_device.c
      usbd_cdc_if.c
      usbd_conf.c
      usbd_desc.c
      tab2spaces.sh
  Debug_STANDALONE
  Docs
  Drivers
    BSP
    CMSIS
    decadriver
    STM32F4xx_HAL_Driver
  Middlewares
  startup
  Bootloader_FLASH.ld
  Makefile
  NetBootloader_FLASH.ld
  NUCLEO_F429.cfg
  README.md
  STM32F429ZITx_FLASH.ld
  Test_FLASH.ld
```

Folder name	Description
Docs	Doxygen documentation script
Debug_ STANDALONE	The target build directory. The name can be changed in the Makefile.
Drivers	This folder contains the HAL and LL drivers for STM32F429, generated by CubeMX, and Decadriver library
Core/Inc	This folder contains header files, generated by CubeMX.
Middlewares	This folder contains the USB device library and FreeRTOS source code, generated by CubeMX.
Core/Src	TWR Demo's main project folder, where: • **"apps"** – collection of top-level applications; • **"boot"** – ant bootloader; • **"config"** – configuration files; • **"core"** – collection of «Core» tasks; • **"Inc"** – common header files; • **"platform_stm32F429"** – folder with platform-dependent files; • **"srv"** – collection of support service utilities.

Figure 3: Juniper source code folder structure

2 OPERATION OF THE MAIN CODE

Please read this chapter with project's code opened in your preferred editor, e.g. *Eclipse* or similar. Initially open the main.c file from *Src* folder, where the *main()* - entry point to the application is located.

2.1 *Startup, initial HAL configuration and starting of the kernel*

At entry point of the *main()* the RTOS is not configured and is not running. The code in *main()* file provides the initial hardware configuration using ST HAL libraries, initialises the core tasks and starts the RTOS kernel.

At the startup, the *main()* loads the saved configuration from the *FConfig*, which is the Non Volatile Memory (NVM) of the MCU (in the target ARM it's organized as a part of ***Flash*** memory), into the RAM segment, called *bssConfig*. On the run-time the application uses the configuration parameters from *bssConfig* only.

The *bssConfig* parameters may be updated by the *Control* task and saved to the *Fconfig* section of NVM.

NOTE: the application has two configuration sections in the NVM memory, called *defaultConfig* and *FConfig*, and one section in the RAM memory, called *bssConfig*. The *defaultConfig* NVM segment stores the default data configuration and this cannot be changed, but can be used to **restore** the initial configuration. The *FConfig* NVM segment stores the current configuration, which can be updated by the *Control* task. The *bssConfig* RAM segment holds the actual run-time working parameters copied from *FConfig* during the startup.

All globally accessible variables are defined in the global "*app*" structure.

```
app_t app;        /**< All global variables are in the "app" structure */
load_bssConfig();/**< load the RAM Configuration parameters from NVM block
*/
app.pConfig = get_pbssConfig();/* app.pConfig is pointed to the RAM
(bssConfig)*/
```

After loading the configuration parameters, the *main()* code initialises the core tasks and enables the Real-Time kernel, see Figure 4. After starting the RTOS kernel, the core tasks will begin to run "in parallel", each executing its dedicated role.

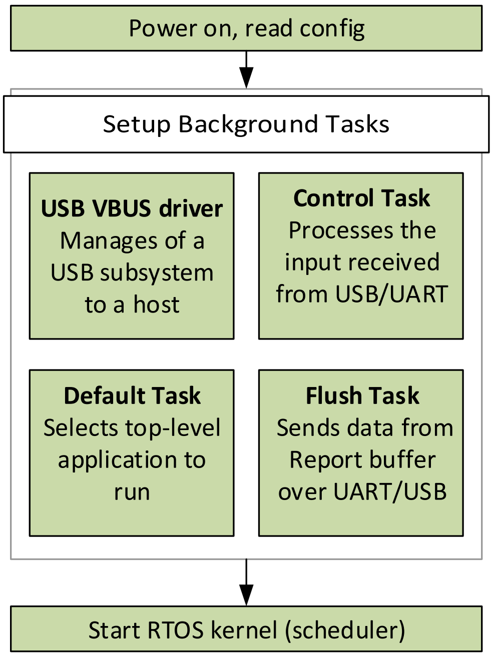

Power on, read config
Setup Background Tasks
USB VBUS driver
Manages of a USB subsystem to a host
Control Task
Processes the input received from USB/UART
Default Task
Selects top-level application to run
Flush Task
Sends data from Report buffer over UART/USB
Start RTOS kernel (scheduler)

Figure 4: Initial startup workflow

2.2 Core tasks

There are several logically different core functions required to run the application. These functions are implemented within core tasks that will constantly run "in parallel".All core tasks have lower priority (less important) than top-level application tasks, thus core tasks can be interrupted by the RTOS kernel when more important thread needs to process the data (i.e. *RxTask, CalcTask*, see 3.1).

The core tasks consist in the following:

- The *Default* task, which is coded in the *DefaultTask()*, is responsible for starting individual top-level application tasks which operate in a mutually exclusive way with the DW3000's because they cannot share this unique resources for their operation, i.e. only one of these top-level tasks is enabled to run at any one time. This is described in more detail in section 2.2.1.
- The USB_VBUS pin driver coded as a dummy loop of the *DefaultTask()*. It periodically polls the status of MCU's VBUS pin and on connection to the +5V source it starts/stops the USB HAL, see 2.2.2.
- The *Control* task, which is coded in the *CtrlTask()*, is responsible for reading of the input from USB and UART, translating it to the appropriate command and executing of that command, see 0.
- The *Flush* task*,* coded in the *FlushTask()* is attempting to output all data to USB and UART from the common circular report buffer *Report.buf*, which used as a common data storage for all output information coming from any running processes, see 2.4.

There is one more special core task, called the *Idle()* task, which is created automatically when the RTOS scheduler is started. It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority tasks in the ready state. The power-saving mode of the MCU can be implemented as a part of this task.

2.2.1 Default task

The *Default* task waits for a global event xStartTaskEvent, which instructs it to enable a particular top-level application task, depending on the requested operation mode. This event can be received from *Control* task or as a part of parsing of initial configuration, see Figure 5 below.
On reception of non-empty xStartTaskEvent, the *Default* task stops all running top-level tasks and their corresponded processes, and then starts the requested top-level application from its initial condition.
Alternatively, if the xStartTaskEvent is empty, the *Default* task periodically executes the USB_VBUS driver, by running it every USB_DRV_UPDATE_MS.\

2.2.2 USB_VBUS driver

The USB_VBUS pin driver, coded in the *usb_vbus_driver()*, runs every USB_DRV_UPDATE_MS as a dummy-loop of the *DefaultTask()* task to check whether the VBUS pin of the MCU is attached to a power source or not. On connection of USB_VBUS pin to the power source, (host PC or wall adapter), the driver configures the USB CDC interface of the MCU and, if the USB handshake is successful, it allows to the *FlushTask()* to output to the USB. Vice versa, on disconnection of VBUS pin of the MCU, the driver will cease output to the USB and release the CDC interface.
Note, the *FlushTask()* can output to both USB and UART simultaneously. If USB is detached, the *FlushTask()* still able to output to the UART, if the parameter "*pConfig->s.uart*" is configured in the *FConfig*, this can be toggled with command *"UART"* from Table 3 below.

2.3 Control task: modes of operation

The *Control* task awaits an input on a USB and/or UART interfaces. The task has two modes of operation, Command mode and Data mode.
The *Control* task in Command operational mode, see Figure 5, parses and executes a command, and can set an event to xStartTaskEvent, which will be received by *Default* task. More details of Command mode of operation of *Control* task is given in the paragraph 2.3.1.

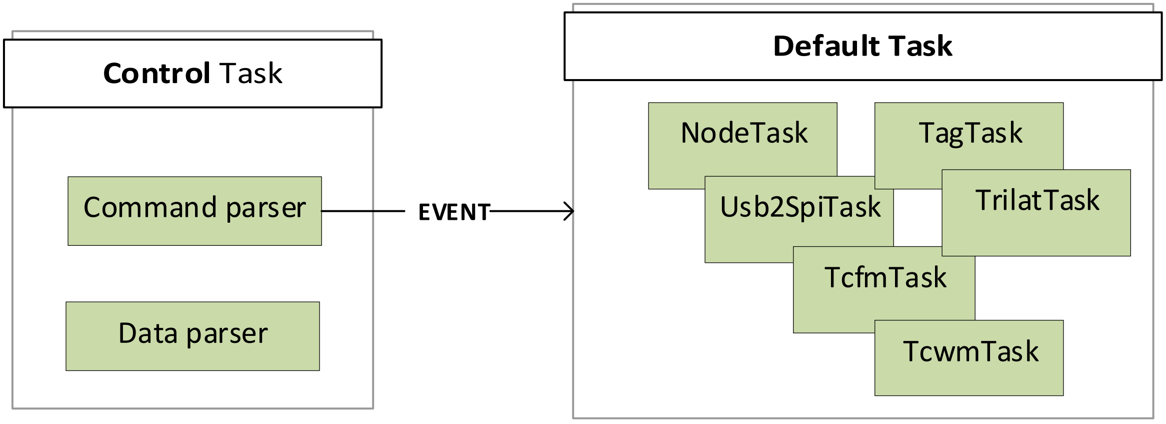


Figure 5: Control task (in command mode) sends EVENT to the Default task

Importantly, on the reception of a *USPI* command from a PC, see Table 2, the *Default* task will be instructed to start the special *Usb2Spi* top-level application, which requires a raw data input from the I/O interface (USB or UART connection to the "DecaRanging" PC application, see [2]).
The start of the *Usb2Spi* top-level application switches the *Control* task to run in transparent mode, called "Data parser" mode, where the *Control* task will not parse commands and will not attempt to execute them.
Instead, the *Control* task will send the SIGNAL and pass the data input directly to the *Usb2Spi top-level* application, which will process the incoming traffic from USB and UART inputs. This is illustrated in Figure 6 below. For more information about *Usb2Spi* top-level application see 3.2.

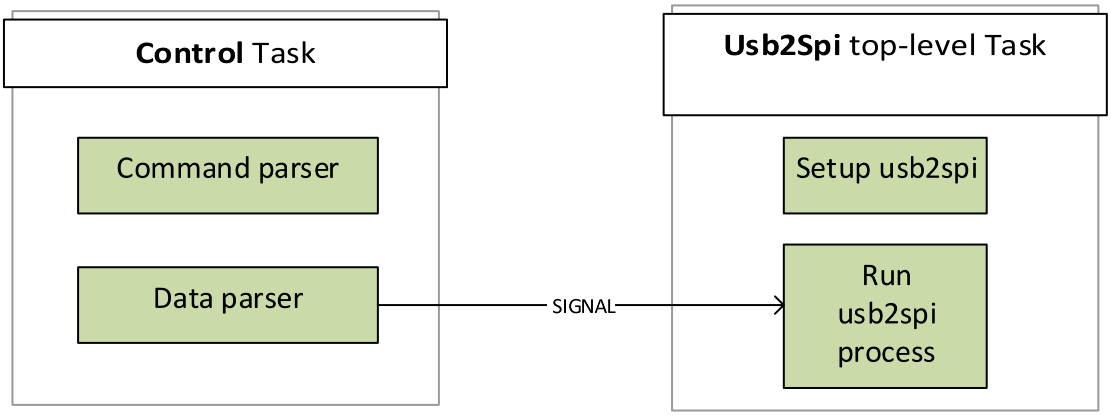


Figure 6: Control Task (in data mode) sends signal to the Usb2Spi application

2.3.1 Command mode of Control task

The command mode of *Control* task includes parsing and execution of several different input command sets. In this section a more advanced description of this mode will be given.

On reception of a valid command by the *Control* task, the command is processed and the corresponding reply is sent for output. The *Flush* task is in charge of sending the output to the USB/UART interfaces.
The twr demo can parse three different types of commands:
- anytime commands
- state changing commands
- run-time parameters access commands.

Some of these commands constitute a "generic commands" set, and some other commands constitute a set of commands controlling the *Node* top-level application.
The generic commands set described in subsections 2.3.1.1, 2.3.1.2, 2.3.1.3, and the Node's application set in subsections 2.3.2 below.

Generic commands set has the general format of **<Command>[<SPACE><Val>]<CR>**

Where **<Command>** is the command string from the Table 1, Table 2 and Table 3 below, **[<SPACE><Val>]** is optional and **<CR>** is representing the carriage return (can be any of <CR>, <LF>, <CRLF>, <LFCR>).

2.3.1.1 Anytime commands

The anytime commands listed in Table 1 can be executed anytime except during operation of the *USPI* mode , i.e. when binary data parser is running (see Figure 6. The only exception is a *STOP* command, which can be executed in all modes.

Table 1: Anytime commands

Command	Definition of functionality
STAT	Reports the status. Gives a dump of software version info, configuration values and the current operation mode (NODE, TCFM, TCWM or STOP).
HELP or "HELP <CMD>"	Outputs a list of all known commands available in the current mode of operation or shows the help of the particular command <CMD>. Equivalent shortcut is "?".
STOP	Stops running any of top-level applications and places the node to the *STOP* mode, where only core tasks are running.
SAVE	Stores the *bssConfig* configuration to the *FConfig*.
Node-application specific commands set *	See section 2.3.2 below

(*) - subject to change

2.3.1.2 Commands to change mode of operation

The commands to change mode of operation can be executed only after the *STOP* command (otherwise the command parser will output the string "error incompatible mode"). They are used to send an appropriate event request to the *Default* task to start a particular top-level application.

On reception of a command to change the mode the *Control* task sets an event for to the xStartTaskEvent which is then received by the *Default* task, as described on Figure 5 above. For a complete list of commands to change the mode of operation see Table 2 below.

Table 2: Commands to change mode of operation

Command	Definition of functionality
NODE	Run the Node top-level application.
TAG	Run the Tag top-level application.
TRILAT	Run the Trilateration top-level application example.
LISTENER <PARM>	Run the Listener top-level application <PARM> is the optional set of parameters to be paseed to the Listener
TCWM	Run the Test Continuous Wave transmission Mode top-level application.
TCFM <PARM>	Run the Test Continuous Frame transmission Mode top-level application. <PARM> is the optional set of parameters to be paseed to the TCFM.
USPI	Run the USB-to-2SPI conversion top-level application. The USB-to-SPI conversion mode gives an external host/PC direct access to the SPI bus of DW3000. Can be used for testing of DW3000 IC and its RF performance.

2.3.1.3 Commands to change run-time parameters

The commands to change run-time parameters, listed in Table 3, can be executed only when a top-level application is not running, i.e. the *STOP* command is needed to place the Juniper into its STOP mode before accessing these parameters. All parameters are a part of the *bssConfig*, and all changes will be applied at the start of a top-level application. The *SAVE* command can be used to store changed parameters to the *FConfig* that they also will be used after the reboot of the device.

Table 3: Commands to change run-time parameters

Command	App	Definition of functionality
ADDR <val>	Node	Set the node's short address to decimal <val>. The default is 1 (which is decimal for node's address 0x0001).
PANID <val>	Node	Set the node's PAN ID to decimal <val>. The default is 57034 (which is decimal for 0xDECA).
NUMSLOT <val>	Node	Set the number of slots been used in the superframe to decimal <val>. The default is 20. This specifies the maximum number of tags in the superframe. This value should be equal or bigger than MAX_KNOWN_TAG_LIST_SIZE, see 3.1.1.
SLOTPER <val>	Node	Set the slot's window period to decimal <val>, milliseconds (ms). The 5 ms slot window is used in the system.

Command	App	Definition of functionality
SFPER <val>	Node	Set the superframe period to decimal <val>, milliseconds (ms). Superframe period shall be at least of time duration to fit all slots, i.e. SFPER ≥ NUMSLOT*SLOTPER. The suerpframe period defines the maximum ranging rate of tags in the system. The default is 100 ms means that all tags in the system can range to the node 10 times a second each.
REPLYDEL <val>	Node	Set the Reply delay to <val>, microseconds (µs). This value is a parameter in the *Ranging Config* message that is sent to each discovered tag to begin ranging. This value is both tag and node hardware dependent. The tags have their `MIN_RESPONSE_CAPABILITY_US`, and if REPLYDEL is less than tag supports, tag will not start ranging to the node.
P2FDEL <val>	Node	Set the Poll-to-Final delay to <val>, microseconds (µs). Default is 1500 µs. This value is a parameter in the *Ranging Config* message that is sent to each discovered tag to begin ranging. This value is mostly tag hardware dependent. The tags have their `MIN_POLL_TX_FINAL_TX_CAPABILITY_US`, and if P2FDEL is less than tag supports, tag will not start ranging to the node.
RCDEL	Node Tag	The <val> is the delay in microseconds (µs) between tag's completion of sending a Blink and start of tag's reception of *Ranging Config* response from the node. Both nodes and tag have this configuration parameter and it should be the same for all the devices in the system. Default is 1000 µs.
UART <val>	all	"0" (default) - Disables the UART. The node will not output data to UART. "1" - Enables the UART. This enables the node to use both USB and UART, which means the top-level application will use both output. In this case the location report rate in Node top-level application will be limited, since UART speed is limited to 115200 b/s. It is recommended to switch off JSON TWR output to maintain maximum location output rate (set PCREP to 2 or 3).
F_NUM <val>	Tag	Tag top-level application Ranging phase configuration: Range Fails number after which tag top-level application will return back to Discovery phase.
ANTTXA <val>	all	Sets the TX antenna delay value to specified val (INT16) decimal value, in device time units of 1/499.2e-6/128 (approx. 15.65 ps). Applied to the TX antenna delay configuration of the DW3000 chip.
ANTRXA <val>	all	Sets the RX antenna delay value to specified val (INT16) decimal value, in device time units of 1/499.2e-6/128 (approx. 15.65 ps). Applied to the RX antenna delay configuration of the DW3000 chip.
PDOFF <val>	Node	Phase difference mean value, externally collected and send back to the node on phase calibration process. The known tag, placed in the known coordinates is used to find the PDoA offset.
RNGOFF <val>	Node	Distance mean value, externally collected and send back to the node after on range calibration process. The known tag, placed in the known coordinates is used to find the range offset.

Command	App	Definition of functionality
PCREP <val>	Node Tag	Select granularity of report used for output. By default "1" (JSON). Used to replace JSON format in report with simple short one to achieve higher locations throughput over UART. If "0" then reports switched off. <val> can be "0", "1", "2".
RESTORE	all	Restore node's configuration to default. This command copies the *defaultConfig* section of NVM to the *bssConfig*. SAVE command shall be used thereafter if user wants to save the *FConfig* section.
UWBCFG <parm>	all	This can be used to modify the default UWB configuration of DW3000. The set of <PARM> equivalent to the order of the output. Should be used only when one understands of the consequence of the opetation.: see code for full details.
XTALTRIM <val>	all	Sets XTAL trim value to adjust the crystal frequency. The value should be in the region 0x00 and 0x7F, with the default parameter set to 0x2E.
POWER <parm>	all	This command sets the default transmit power of the chip. <parm> represents the <0xPWR>, <0xPGDLY> <0xPGCNT>. Should be read in conjunction with the User Manual of the DW chip.
STSKEYIV <key> <iv> <type>	all	Configures the STS KEY and IV to be used when sending/receiving packets when running top-level applications. <key> this is the 128-bit STS KEY value (0xaaaabbbbccccdddd); <iv> this is the 128-bit STS IV value (0x1111222233334444); <type> can be 0 – the STS will update (increment the IV counter for each TX/RX) or 1 the STS will be static – same IV used for each TX/RX. The default parameters sets to a well-defined STS key/data pair and static mode.
LSTAT	Listener	Shows the statistics numbers of received UWB packets and their quality. Should be read in conjunction with the User Manual of the DW chip.

2.3.2 Controlling of the embedded applications over a PC GUI app

When any of the top-level applications is running, it can accept its own specific set of commands to utilize the control of its specific functionality. For example, the Node application can be controlled by PDoA GUI PC app and has its own set of commands and output responses (i.e. interface), appropriate for the Node application.
Below is the specification of the interface in between Node and PDoA GUI PC application.

The rest of the applications do not have their own, specific input commands from the GUI's and can only be started/stopped to produce the output.

2.3.2.1 The PC to the Node-application specific commands

When Node top-level application is running, it can be controlled externally by USB/UART and accepting specific commands, which belongs specifically to the Node top-level application.
The PC to the Node communication commands, listed in Table 4, can be executed at any time and will have immediate effect to the Node's top-level application, if it is running. The node is replying onto that commands, using JSON formatted output, wrapped to TLV format, see 2.3.2.2.
For autonomous mode, e.g. when Node application is not connected to the PC GUI application, but working standalone, (e.g. when installed on a mobile robot), the known tags list (*KList*) can be saved to

the *FConfig NVM* segment, using the *SAVE* command for this.

Table 4: The PC to the Node top-level application commands

Command	Definition of functionality
DECA$	Node will reply with Info JSON object with version string, see Table 5
GETDLIST	Node will reply with KList JSON array of discovered (harvested) tags list, see Table 5
GETKLIST	Node will reply with DList JSON array of known tags list, see Table 5.
ADDTAG <addr64> <addr16> <mFast> <mSlow> <mMode>	Formatted input string with spaces as separator. Instructs the Node to add a tag with *addr64* to known tags list (KList), using *addr16, mFast, mSlow and mMode* as parameters for that tag. <addr64> is address of the tag, hexadecimal, must be 16 characters; <addr16> is request to assign this short address of the tag, hexadecimal; This address may be automatically changed by the node, as KList is protected from adding of identical addresses in it. <mFast> is hexadecimal value which will be used by the tag if it considered it is moving. This is in number of superframes. For example, 1 means that the Tag, when its moving, will range to the Node every superframe, and “0A” (decimal 10) means tag will range to the node every 10-th superframe. <mSlow> is hexadecimal value which will be used by the tag if it considered it is not moving, i.e. stationary. This value usually specified to a big number, 64 hexadecimal means tag will range to the node every 100 superframes, see *Example* below. <mMode> is a hexadecimal bitfield parameter to pass to the tag. Bit 0 indicates tag shall use IMU to detect if it stationary or moving. Bits 1-15 are not used. *Example:* “ADDTAG 001122334455667788 1000 2 64 1” – this instructs the node to add the tag to the KList with long address “0x001122334455667788”, try to assign to this tag a new short address “0x1000”, configure tag to use IMU, the tag should range to the node every 2 superframes if it is moving (giving superframe is 100 ms, this means tag will range 5 times a second), and when tag is stationary, range every 100 superframes, i.e. every 10 seconds. On success the command will return a “TagAdded” JSON object with actual parameters, assigned to the tag, see Table 5 below. It is mandatory to wait for “TagAdded” object or request a full known tags list from the Node to confirm the tag has been added. The external application must use the short address from “TagAdded” or “KList” responses, as KList is protected from adding of identical addresses to it and node will assign unique short address for the tag in case it was erroneous instructed to add a tag with short address, which belongs to other tag.

Command	Definition of functionality
DELTAG <addr64>	Formatted input string. This will delete the tag of with <addr64> from KList. <addr64> is address of the tag, hexadecimal, must be 16 characters; The <addr64> may contain a short address of the tag, followed 12 zeroes. In this case the tag also will be correctly deleted by its short address. *Example:* Tag long 64-bit address is 0x001122334455667788. And assigned short address is 0x1234 Following commands will identically correctly delete the tag from the KList: “DELTAG 0000000000001234” or “DELTAG 001122334455667788”.
D2K	Automatically adds all discovered tags to known tag list. This is a useful command for Terminal only, as it connects to all tags around.

2.3.2.2 The Node top-level application output to PC

The Node top-level application outputs to the PC using JSON formatted output. The JSON object is encapsulated in TLV format (Type-Length-Value) to easier the implementation of parser on the PC side. The reader may find a description of JSON format in the RFC 4627, [https://tools.ietf.org/html/rfc4627].
<TYPE><LENGTH><VALUE>, where <TYPE> is "**JS**", <LENGTH> is 4-byte hexadecimal length of <VALUE> field, which is a JSON object, see Table 5 below.

Table 5: Node top-level application JSON outputs

Action	JSON object	Type	Format of JSON object with TLV wrapper
Reply to the "DECA$"	Info	Info object	JSxxxx{"Info": <info_object>} The device reports the information about it. *Example:* JS0088{"Info":{ "Device":" Node", "Version":"1.0.0", "Build":"Sep 18 2017 14:06:47", "Driver":"DW3000 Device Driver Version 04.00.07"}}

Node application reports a new tag has been discovered	NewTag	String	JSxxxx{"NewTag": <string>}
			Tag's 64 bits hex address is in the <string>.
			Example:
			JS001D{"NewTag":"10205F4910002E5C"}

Reply to the "ADDTAG"	TagAdded	Tag object	JSxxxx{"TagAdded": <tag_obj>} The <tag_obj> is JSON object of following fields: {"slot":<int>,"a64":<string>,"a16":<string>,"F":<int>,"S":<int>,"M":<int>}" **"slot"** - is the assigned slot (controlled by the node); **"a64"** - is the long address of the tag, hex; **"a16"** - is the short address, assigned to the tag, hex; **"F"** - is the how often in numbers of SuperFrames the tag will range if it is considered it is moving, dec; **"S"** - is the how often in numbers of SuperFrames the tag will range to the Node if it is considered it is not moving, dec; **"M"** - tag operational mode bitmask, bit 0 to indicate Tag shall use IMU and decide whether it is moving or stationary and use "F" and "S" fileds described above. If "M" defined to 0, then the tag will use "F" field for its continuously ranging to the node.

Reply to the "GETDLIST"	DList	Array of strings	JSxxxx{"DList": [<string>,<string>,<string>,…]}
			Tag's 64 bits hex addresses are in strings.
			Example:
			JS001F{"DList":["10205F4910002E5C"]}

Action	JSON object	Type	Format of JSON object with TLV wrapper

Reply to the “GETKLIST”	KList	Array of Tag objects	JSxxxx{"KList": [<tag_obj>,<tag_obj>,…]}
			Example:
			JS005D{"KList":[
			{
			"slot":1,
			"a64":"10205F4910002E5C",
			"a16":"012A",
			"F":1,
			"S":64,
			"M":1
			}
			]}

Reply to the “DELTAG”	TagDeleted	String	JSxxxx{"TagDeleted": <string>}
			<string> is the long address of the tag, hex;
			Example reply on DELTAG *command:*
			JS0022{"TagDeleted": "10205f4910002e5c"}

When the Node application is running, it can report to the PC the “TWR” object anytime	TWR	Node’s Twr Object	JSxxxx{"TWR": <twr_obj>} Where <twr_obj> is: {"a16":<string>,"R":<int>,"T":<int>,"D":<int>,"P":<int>,“A”:<int>,"O":<int>,"V":<int>,"X":<int>,"Y":<int>,"Z":<int>} **"a16"** - is tag’s short address, hex; **"R"** - is the range number, dec; **"T"** - is the time of reception of Final wrt node’s SF start in us, dec; **"D"** - is the distance to the tag in centimeters, (float as int), dec **"P"** - is the raw PDoA to the tag in degrees, (int), dec **"Xcm"** - is the X coordinate of the tag in centimeters, (float as int), dec **"Ycm"** - is the Y coordinate of the tag in centimeters, (float as int), dec **"O"** - is a clock offset of the tag in hundreths of ppm (float as int), dec **"V"** - is a service data wrt to the tag, bitfields, dec: bit 0 indicates 1 if tag is stationary and 0 if tag is moving; bit 14 indicates a RNGOFF=0 is used for distance calculation; bit 15 indicates a PDOFF=0 is used for calculation; **"X"** - is the accelerometer X axis data, in milli-G, dec **"Y"** - is the accelerometer Y axis data, in milli-G, dec **"Z"** - is the accelerometer Z axis data, in milli-G, dec *Example:* JS006A{"TWR": {"a16":"4096","R":53,"T":5126,"D":112,"P":-161,"Xcm":112,"Ycm":0,"O":336,"V":0,"X":0,"Y":0,"Z":0}}
Diagnostic object	TWR_DIAG	Node’s Diag object	JSxxxx{“TWR_DIAG”: <twr_diag_obj>} Proprietary reads of registers of the DW3000 / used for developing of some specific algorithms.

2.3.2.3 The Tag top-level application output

The Tag top-level application outputs to the PC using JSON formatted output.

Similarly to the Node, the Tag outputs the JSON TWR object, with a distance from the Tag to the Node, see Table 5.

2.3.2.4 The Trilat top-level application output

Trilat top-level application has been developed to demonstrate RTLS functionality. The node, upon receiving at least 3 range measurements from fixed tags performs trilateration to find its position relative to the fixed tags. The trilateration function has not been optimized for a higher capacity and is suitable to locate only one mobile Node at a time. Trilat sits on the top of the Node application, which should be configured to range to at least 3 known Tags with fixed locations (tags for this demo act as a fixed infrastructure elements and should be configured to supply their fixed position co-ordinates X,Y,Z).

2.4 Flush task

Any functions, including ISR functions, or RTOS tasks can produce and request to send data to the non-blocking output (USB and/or UART). In the data sending function *port_tx_msg()*, the data is copied to the intermediate Report Buffer, which is then flushed by the *Flush* task. This is illustrated in the Figure 7 below.
The *Flush* task is coded as FlushTask() in the task_flash.c source code file.
The Report buffer is a circular buffer, which is statically allocated in the usb_uart_tx.c as txHandle.Report. The *port_tx_msg()* function is copying data to the *txHandle.Report.buf* and then sets the *app.flushTask.Signal* to the *Flush* task to start immediate transmitting of data via USB/UART.
The *Flush* task is emptying the Report buffer onto the USB and the UART. The Report buffer is a statically allocated area of USB_REPORT_BUFSIZE. The size of Report buffer is sufficient that any task/function can send a chunk of data for background output without delaying its throughput, even during an ISR, see Figure 7 below.

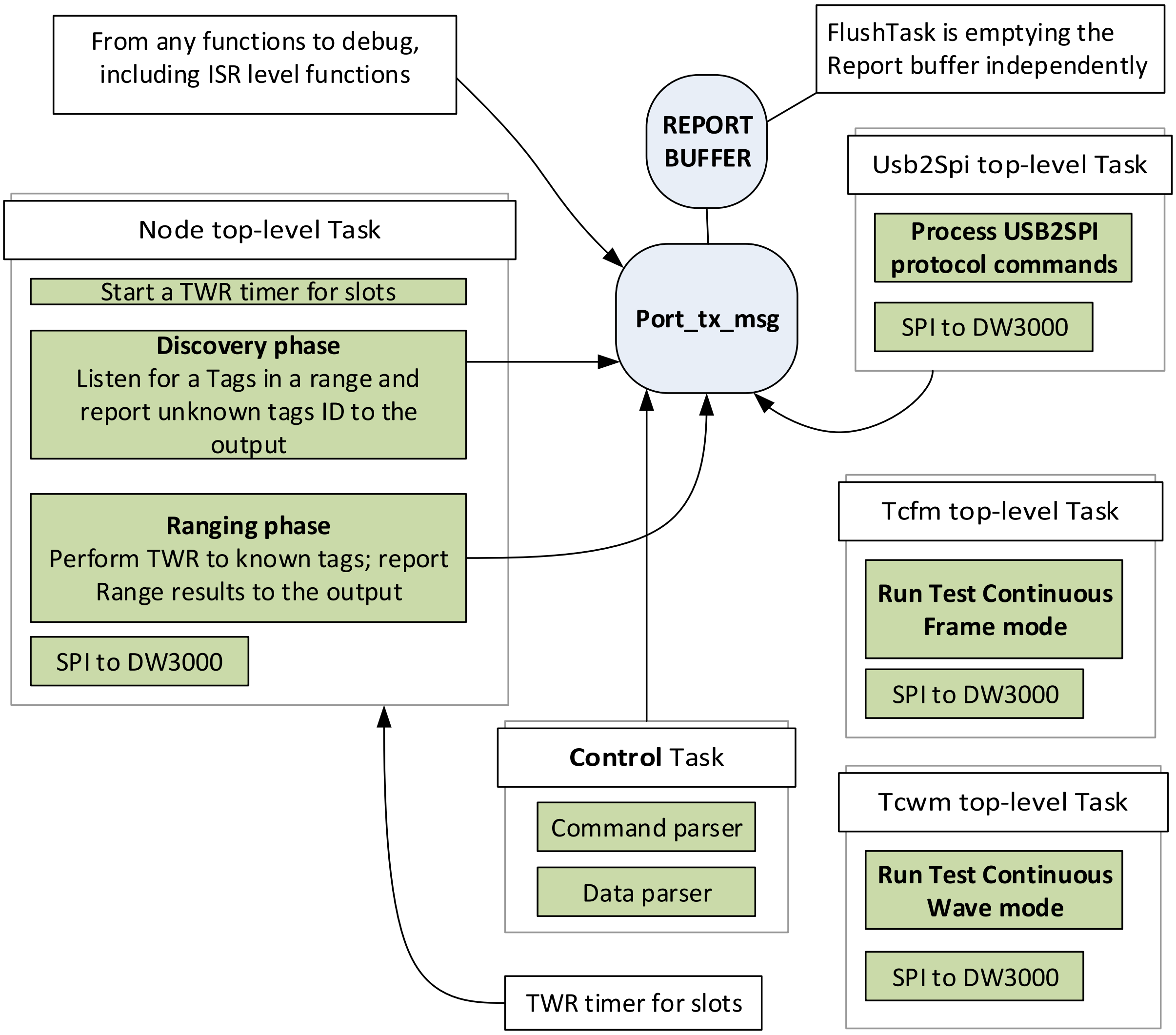

From any functions to debug, including ISR level functions
FlushTask is emptying the Report buffer independently
REPORT BUFFER
Usb2Spi top-level Task
Process USB2SPI protocol commands
SPI to DW3000
Node top-level Task
Start a TWR timer for slots
Discovery phase
Listen for a Tags in a range and report unknown tags ID to the output
Ranging phase
Perform TWR to known tags; report Range results to the output
SPI to DW3000
Port_tx_msg
Tcfm top-level Task
Run Test Continuous Frame mode
SPI to DW3000
Control Task
Command parser
Data parser
Tcwm top-level Task
Run Test Continuous Wave mode
SPI to DW3000
TWR timer for slots

Figure 7: Output data using shared Report buffer

2.5 RTOS extensions used in the application

For performance reason in the Node application only RTOS mailbox is used to pass data between *rxTask* and *calckTask*. The queues and mailboxes are not used to pass the data from ISR to tasks and a circular buffer alternative is widely used instead.

For locking and signalling, the following mechanisms are used: mutexes, events and signals. Mutexes are used to protect task execution from being killed in the *Default* task while they still in the running state.

The *EventGroup* mechanism, is used to send relatively slow events between tasks. E.g. this method is used to instruct the *Default* task to start a particular top-level application.

As a fast and simple alternative to *EventGroup*, the fast task notification mechanism can also be used.

In the CMSIS-RTOS this defined as signals. The signal is delivering a simple message to the specific task. This mechanism is faster that *EventGroup* and in the application is used to organize interconnection from ISR level functions to a RTOS-based tasks. For more information please refer to the FreeRTOS documentation [www.freertos.org].

For the purposes of unification, all tasks (top-level applications and sub-levels), which are capable to receive signals are defined as task_signal_t structures in the global *app* structure. Example of the code is below.

```
/* Application tasks handles & correspondeing signals structure */
typedef struct
{
    osThreadId Handle;    /* Task’s handler */
    osMutexId  MutexId;   /* Task’s mutex */
    int32_t    Signal;    /* Task’s signal */
}task_signal_t;
```

In the code, in the *app* structure, task handlers and signals are defined as follows:

```
//defaultTask is always running and is not accepting signal

task_signal_t  ctrlTask;      /* usb/uart RX: Control task */
task_signal_t  flushTask;     /* usb/uart TX: Flush task */

/* app task for TWR */
task_signal_t  rxTask;        /* Tag/Node */
task_signal_t  calcTask;      /* Node only */

/* tasks for special top-level applications */
task_signal_t  usb2spiTask;   /* USB2SPI top-level application */
task_signal_t  tcfmTask;      /* TCFM top-level application */
task_signal_t  tcwmTask;      /* TCWM top-level application */
```

3 In deep about top-level applications

There are a number of top-level applications, listed in the Table 6, which can run in the dedicated mode on the Juniper platform. Every top-level application consists from a task (or a number of tasks) and a set of non RTOS based functions to implement an application's functionality.

Table 6: Top-level applications and corresponded commands

Top-level application	Corresponded command	Description
Node, section 3.1	NODE	Two-way ranging slotted Node top-level application (PDoA / Non-PDoA)
Tag, section 3.2	TAG	Two-way ranging slotted Tag top-level application
Trilat,	TRILAT	Example of trilateration engine, running on above the Node top-level application
Usb2Spi, section 3.3	USPI	USB (or UART) to SPI converter. Used for testing.
TCWM, section 3.4	TCWM	Test Continuous Wave transmission. Used for testing
TCFM, section 3.4	TCFM	Test Continuous Frame transmission. Used for testing.
Listener, section 3.6	LISTENER	Starts Listener application.

3.1 Node top-level application

If configured in the *app.pConfig->s.default_event* parameter, the juniper platform will start and execute the two-way-ranging *Node* top-level application.

The Node supports ranging to multiple tags. To prevent interference between these tags, a Time-Division Multiple Access method (TDMA) is employed to separate the tags' ranging exchanges into individual "slots" within a repeating "superframe" structure, specified by the node, see 3.1.3.
Initially each tag sends only blink messages to advertise itself and be discovered by the node. For the tag, this is called the *Discovery* phase. After sending the blink, the tag awaits a *Ranging Config* response from the node, and upon its successful reception, the tag configures itself, as instructed by the *Ranging Config* response, to range to the node in a designated slot.

The node only ranges to tags that appear in its KList, which is a list of **known** tag IDs (and their configuration parameters) that are authorised to communicate to the node.

When the node receives a blink from a tag which is not in the KList, it reports this via the "newTag" report, sent over the USB/UART (e.g. to the to the connected PC application). The new tag may be added to the KList individually using the "ADDTAG" command or "D2K" command, which will add all discovered tags and assign to them short 16-bit addresses automatically.

When a blink is received from a tag that is in the KList, the node immediately responds with a *Ranging Config* response, assigning a slot to the tag for its future TWR exchanges. The SAVE command which saves the node configuration also saves the current *KList*. The "DELTAG" command, can be used to remove an individual tag from the *KList.* See Table 5 for more details.

As noted above, after sending a blink, receiving of a *Ranging Config* and been configured, the tag is going to the *Ranging* phase and starts periodic ranging exchanges to the node in its dedicated time slot. Every ranging exchange the tag starts with sending of a *Poll* message (addressed to the node address, specified in the *Ranging Config* message), awaits of a *Response* message from the node, and upon its successful reception, replies with a *Final* message to complete the ranging sequence.

On successful reception of the *Final message*, the node reads data from DW3000, calculates distance and phase difference (on a Non-PDoA variant of DW3000 the PDoA is zero), and reports result to the output. Figure 8 below shows a high-level view to the *Node* application flow and more detailed description is giving in the section 3.1.2.

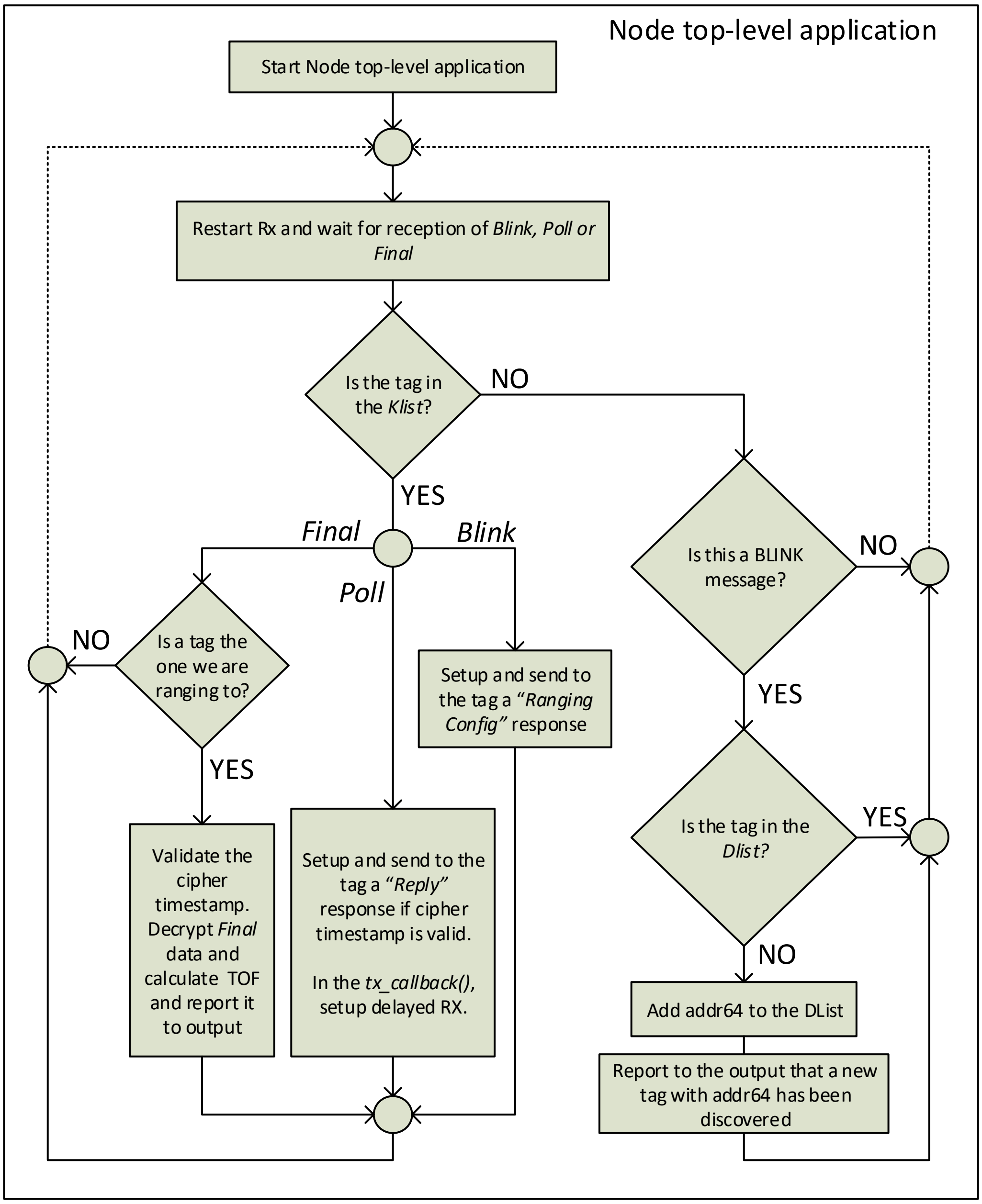

Node top-level application
Start Node top-level application
Restart Rx and wait for reception of Blink, Poll or Final
Is the tag in the Klist?
NO
YES
Final
Blink
Poll
Is this a BLINK message?
NO
YES
Is a tag the one we are ranging to?
NO
YES
Setup and send to the tag a "Ranging Config" response
Validate the cipher timestamp. Decrypt Final data and calculate TOF and report it to output
Setup and send to the tag a "Reply" response if cipher timestamp is valid.
In the tx_callback(), setup delayed RX.
Is the tag in the Dlist?
YES
NO
Add addr64 to the DList
Report to the output that a new tag with addr64 has been discovered

Figure 8: *Node* top-level application

3.1.1 Concept of Discovered and Known tags lists

Before the node starts ranging to a tag, the tag needs to be added to a list specifying tags to which the node is allowed to range with. This list of tags is called “known tags list” or *KList*.
Every record in the *KList* has all necessary information about each tag including: its 64-bit address, assigned 16-bit (short) address, assigned slot number, etc. This information is supplied to the tag in the *Ranging Config* reply by the node following the reception of the tag’s blink message. The *KList* can be saved and it will then be available for use during autonomous working mode of the *Node* (i.e. after start-up).
The user-commands *“*ADDTAG*”* and *“*DELTAG*”*, described in Table 5, can be used to add/remove individual tag information to/from the *KList*. The *“*GETKLIST*”* command can be used to retrieve the *KList* information, e.g. by the PC GUI application so that it can update the list of known tags.

When the node receives a blink from a tag that is not present in the *KList*, the tag’s 64-bit address is stored in a temporary *“discovered”* tags list, called *DList*. The “GETDLIST” command is used to periodically retrieve the discovered *tags* information, e.g. by the PC GUI application, so that it can update its list of tags in the system.
Note: The “GETDLIST” command also clears the discovered tags list in the node.
The maximum sizes for KList and DList can be found in the *tag_list.h* header file:

```
#define MAX_KNOWN_TAG_LIST_SIZE         (20)
#define MAX_DISCOVERED_TAG_LIST_SIZE    (20)
```

3.1.2 Discovery and ranging to tags

There is a set of RTOS tasks (threads) to implement the node’s functionality above. On reception of the *Ev_Node_Task* event, the *Default* task executes the *node_helper()* function, which configures all the HW to operate for the *Node* top-level application, i.e. wakes up and configures the DW3000 IC to run with configured UWB parameters and starts following sub-tasks: *RxTask* and *CalckTask*.

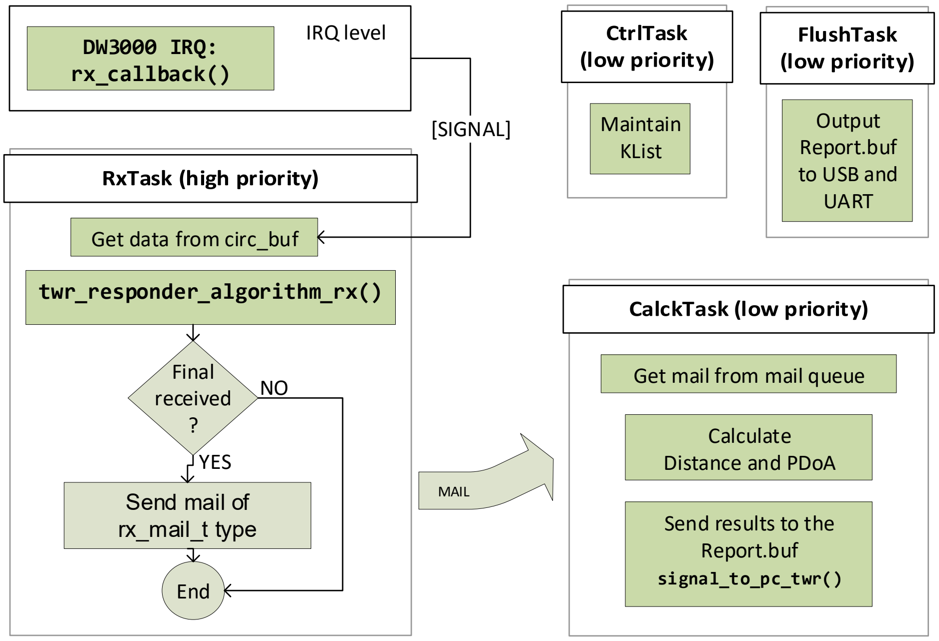


Figure 9: Tasks used in the node application

Please note, the core tasks, i.e. *Control*, for input handling, and *Flush,* for output handling, are always running, see sections: 1.1.2, 2.3.1, 2.4.
On reception of a UWB blink message in the *RxTask* it checks whether the sender is in the *KList*. If it is, the *RxTask* sends to the tag the appropriate *Ranging Config* response, which describes to the tag its personal run-time parameters, for it to use during the *Ranging* phase.
If the sender is not in the KList and is not yet in the *DList*, the *RxTask* reports that a new tag has been

discovered in the range (see the "NewTag" object in Table 5) and stores the tag's 64-bit address in the discovered tags list *DList*, that it will not be reported as "NewTag" to the output anymore, but only if the control application will request for a "GETDLIST", see Table 5.
Once a tag has been sent a *Ranging Config* response, it is expected that it will start ranging to the node, i.e. it will periodically send a *Poll* message to initiate the ranging exchange in its configured time slot.
On reception of *Poll* message from a known tag, the node begins range to that particular tag, also node sends back to the tag a correction value in microseconds, of how far is the tag from its assigned slot, that the tag can correct its internal processes and will range next time closer to the assigned slot, see 3.1.3 and 5.5.
On reception of *Final* message from the tag, the Node's *rxTask* sends the mail using the mailbox mechanism to the lower priority *calcTask*, which will calculate and report the estimated distance, and X-Y coordinates of the tag with respect to the Node.

3.1.3 The superframe, the wakeup timers and the tag's slot correction

To ensure non-overlapping ranging exchanges for multiple tags, the node uses Time-Division Multiple Access method (TDMA), to assign to every tag its own dedicated slot, of T_Slot duration, within node's superframe period, see Figure 10.

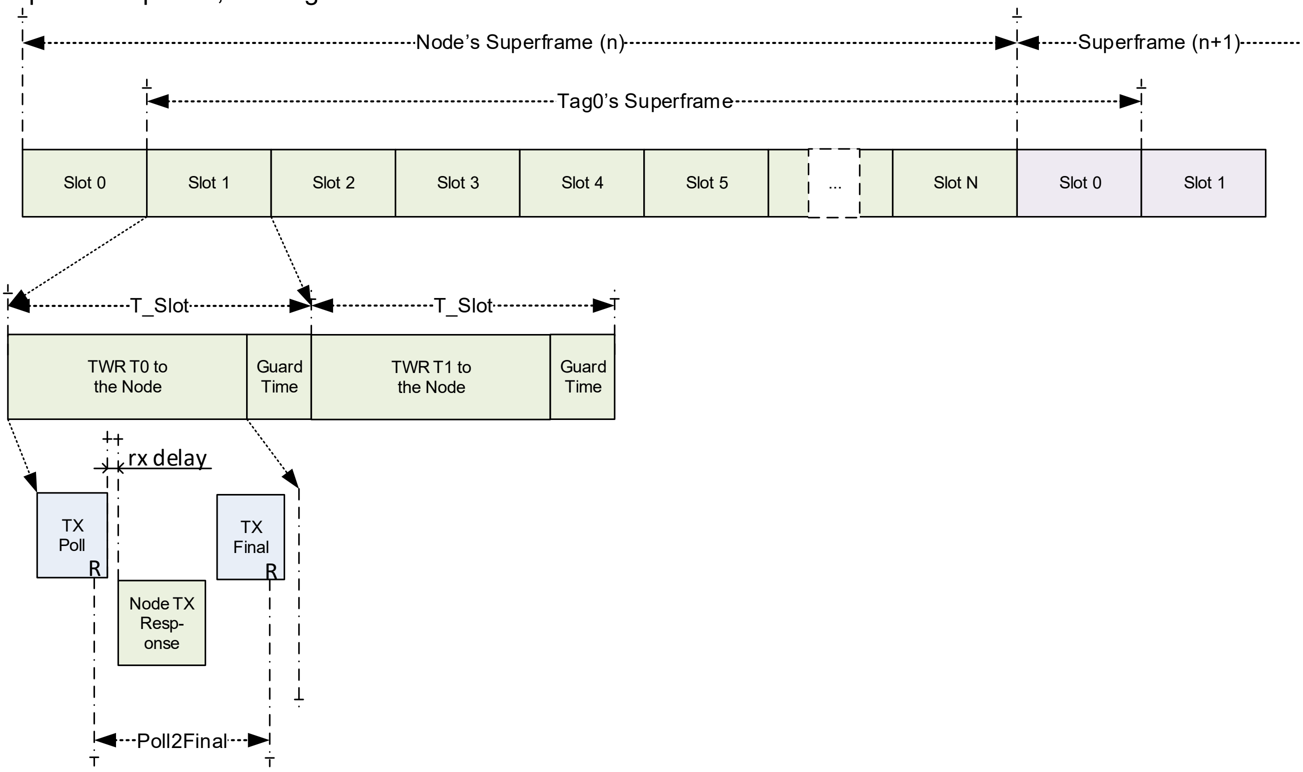


Figure 10: Superframe structure and ranging exchange time profile

On the picture above, the R represents the RMARKER, which is the event nominated by the IEEE 802.15.4 UWB PHY standard for message time-stamping. The time the first symbol of the PHR launches from the antenna (defined as the RMARKER) is the event nominated as the transmit time-stamp, see 5.1, 5.2, 5.3, **[1]**.

The Poll2Final configuration value defined the rough time between transmissions of RMARKERS for Poll and Final messages from the tag. Rx_delay is the time between tag's end transmission of the Poll and its start of reception of a Response from a Node, see P2FDEL and RCDEL parameters in Table 3 above.

Note, the slot number zero is reserved to be used for future enhancement of the system, for example Node can beacon in this slot and Tag can listen for the beacon and be instructed to transmit only on allowed time.

The node specifies the superframe period in the *Ranging Config* message which the tag saves in its `framePeriod_ms` variable.The node counts its local superframe period using a RTC wakeup timer, configured to expire every `pSfConfig->sfPeriod_ms.` On expiry, the timer saves the clock value to the `gRtcSFrameZeroCnt`, which indicates the start of Node's internal superframe and is used in the slot correction process, as described below.

The RTC in the node and the RTC in the tag have a small drift with respect to each other, so to maintain the tag ranging in its assigned slot, every time the node receives a *Poll* from tag, it checks its receive time against the expected receive time and includes in the ranging *Response* message a correction factor, the `slotCorr_us`, which indicates the difference between the start of tag's dedicated slot with respect to node's current start of superframe – `gRtcSFrameZeroCnt` and the actual arrival time of the *Poll*. Using this `slotCorr_us` information, the tag adjusts its wakeup timer for the next period to send its *Poll* in the assigned slot. For more details about slot correction method see section 5.5.

3.2 Tag top-level application

Once tag top-level application is started, it will setup and execute the two-way-ranging *Tag* application which consists of Discovery and Ranging phases.
The system may include more than one tag ranging to the same PDoA node. To prevent interference between tags, a Time-Division Multiple Access method (TDMA) is employed to separate the tags ranging exchanges into individual “slots” within a repeating "superframe" structure specified by the PDoA node, see 3.1.3.
Initially the tag sends blink messages to advertise itself and be discovered by the PDoA node. This is called the Discovery phase. After sending the blink, the tag awaits the response from the PDoA node and upon successful reception it configures itself as instructed by the PDoA node. The *tag* application supports ranging to a single PDoA node only.

For each ranging exchange, the tag sends a *Poll* message (addressed to the PDoA node address, specified in the *Ranging Config* message), and awaits a *Response* message from the PDoA node, and upon its successful reception, sends a *Final* message to complete the ranging exchange. After completing the ranging exchange (or if it fails to complete because of error or timeout), the tag will put the DW3000 into DEEPSLEEP until a configured WakeUp timer period elapses, after which it will wake the DW3000 and the flow will be repeated again, to complete another ranging exchange.

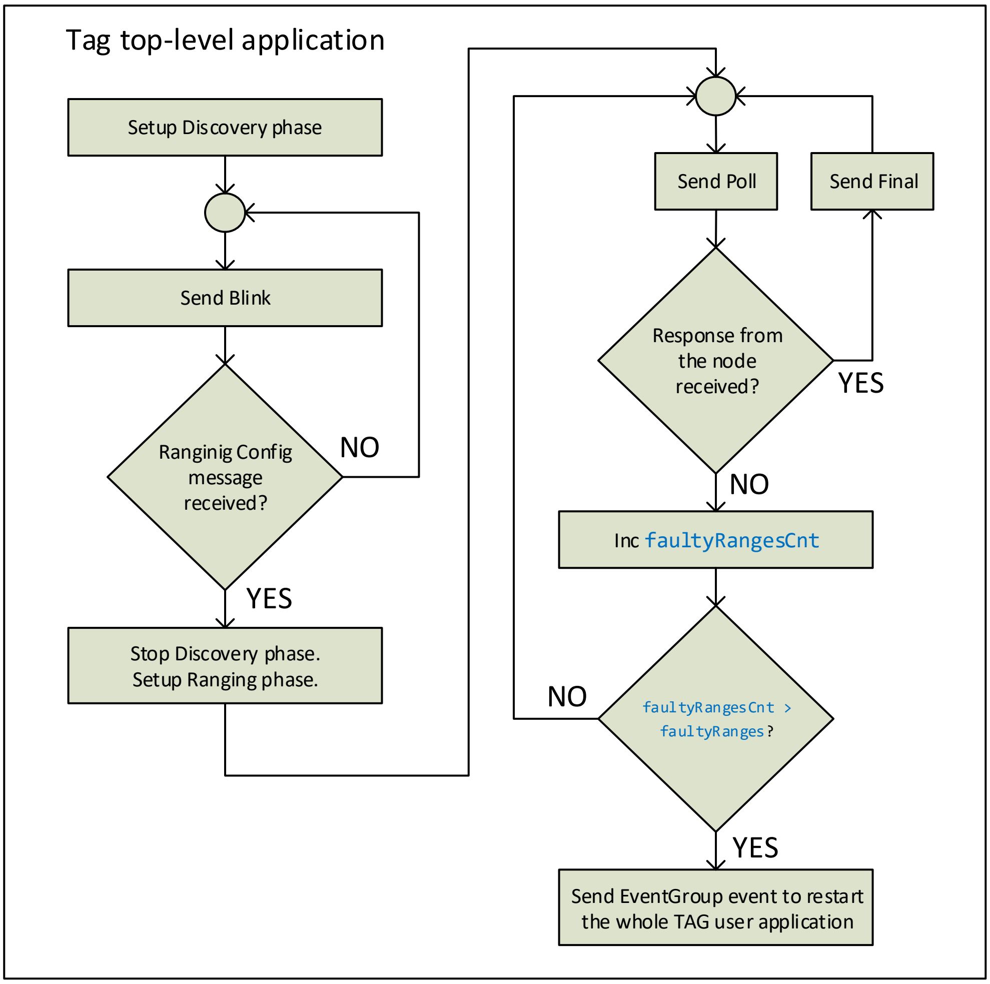

Tag top-level application
Setup Discovery phase
Send Blink
Ranginig Config message received?
NO
YES
Stop Discovery phase. Setup Ranging phase.
Send Poll
Send Final
Response from the node received?
YES
NO
Inc faultyRangesCnt
faultyRangesCnt > faultyRanges?
NO
YES
Send EventGroup event to restart the whole TAG user application

Figure 11: *Tag* top-level application

There is a set of threads to implement the *Tag’s* functionality above. On reception of the *Ev_Tag_Task* event, the *Default* core task executes the *tag_helper()* function, which configures all the HW to operate for the *Tag* top-level application, i.e. initially wakes up of the DW3000, then configures it for running with configured UWB parameters and starts sub-tasks: *BlinkTask*, *PollTask, RxTask*.

3.2.1 The Discovery phase

Once the *BlinkTask* has started, the tag starts sending periodic blink messages and awaits a response in the *RxTask*, see below. This is called the Discovery phase, and continues indefinitely, until a *Ranging Config* message is received from the node.

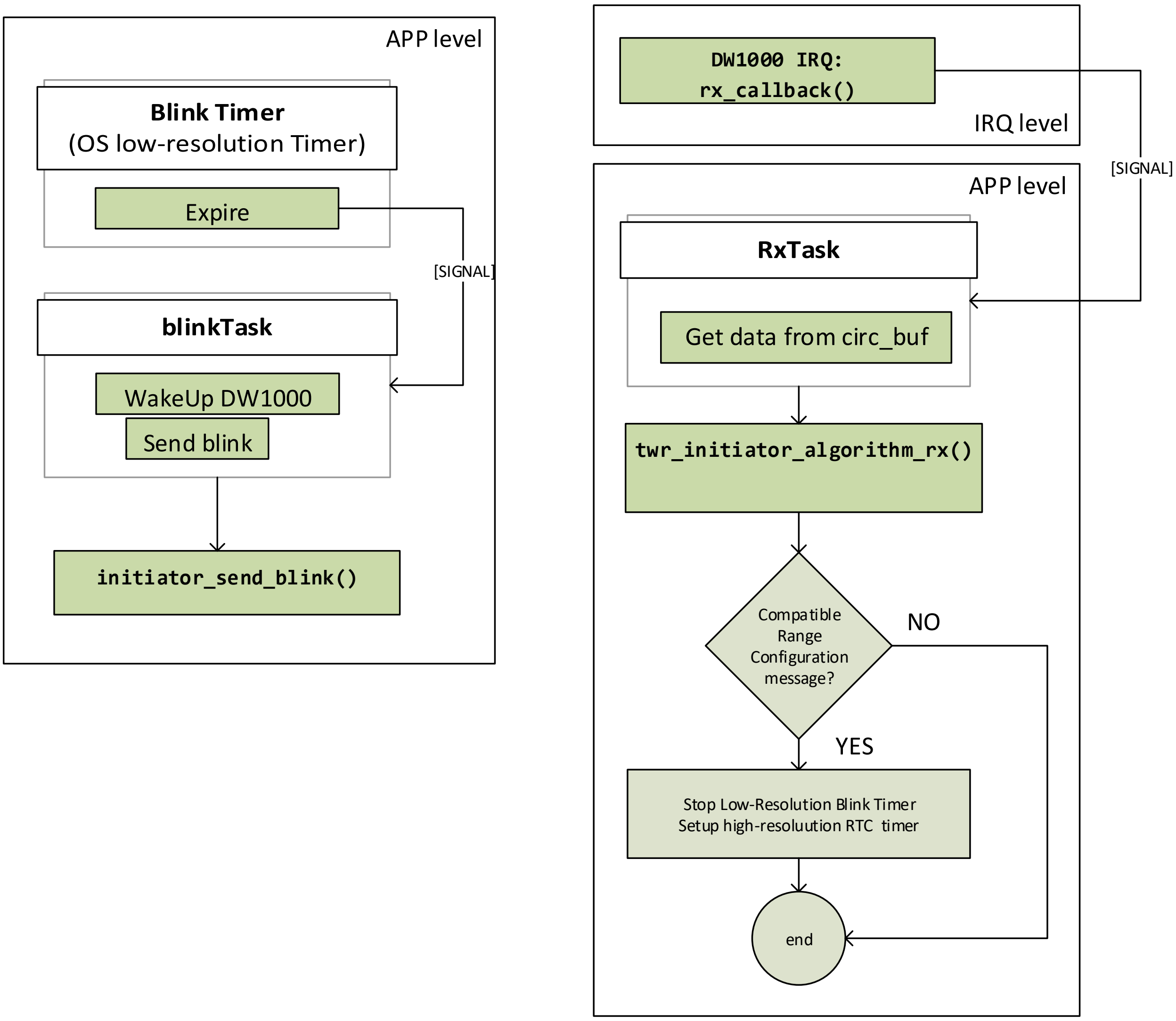


Figure 12: *Tag's* Discovery phase: Rx Thread

The *Blink* message is an IEEE specified 12-byte message with long addressing mode, see 5.4.1. The *Ranging Config* message uses long-short addressing and contains configuration options: the version of *Ranging Config*, Tag's and Node's short addresses, PDoA system PanID, timings for range exchange, etc.
On reception of the *Range Config* response, the *RxTask* stops the low resolution Blink Timer and sets a higher precision RTC wakeup timer to expire and Signal to the PollTask to attempt a first ranging. The Discovery phase has finished at this point and tag application goes to the Ranging phase.

3.2.2 The Ranging phase

The RTC wakeup timer, first time configured at the end of Discovery phase, expires when the tag is due to send the *Poll* message. The Wakeup timer is reloaded with the periodic value, corresponded to the super frame duration value from *Ranging Config* message. It will also be corrected by the PDoA node in every response message. The *PollTask* receives the signal from Wakeup timer and sends the

Poll to the PDoA node, initiating a *Ranging* sequence, see Figure 13. On reception of UWB packet, the *dwt_isr()* rx_callback() sends the SIGNAL to the *rxTask,* which is then responsible for the setup of the *Final* reply message to the node, see Figure 14.

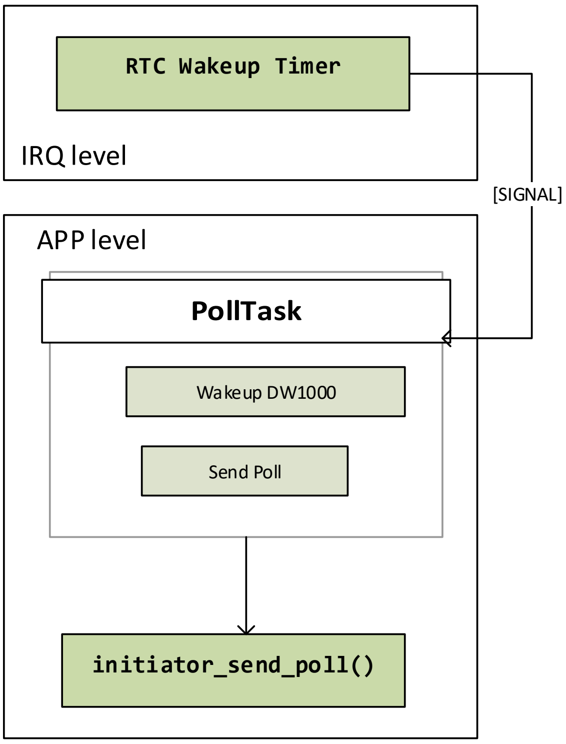


Figure 13: Tag's Ranging phase: Poll Thread

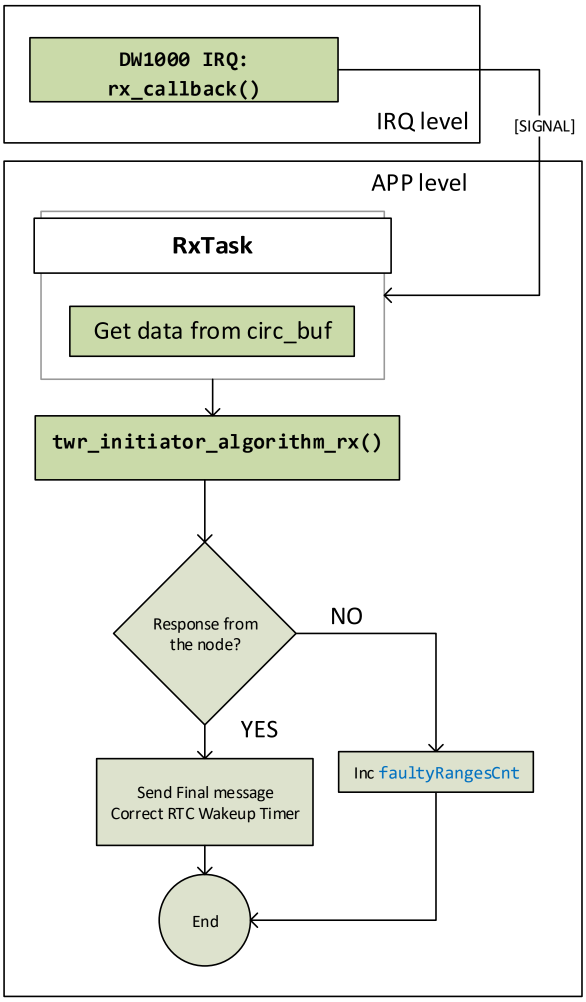


Figure 14: Tag's Ranging phase: Rx Thread

Otherwise Wake up timer sends SIGNALs to the *PollTask* as configured in the fast location rate parameter. Outside of the ranging exchange, the DW3000 is placed into its DEEPSLEEP Mode, and the MCU can be in a low-power mode if other tasks are not running, this is under control of the *Idle* task.

3.3 Usb2Spi top-level application

When the *Control* task receives the command "*USPI*", see Table 2, it sets the *Ev_Usb2Spi_Task* to the xStartTaskEvent. The *Default* task consumes the event, safely ends all running tasks and starts *Usb2Spi* top-level application.
Once *Usb2Spi* top-level application has started, the *Control* task is switched to the "Data parser" mode, see 2.3, where it passes the whole incoming USB/UART stream into the *Usb2Spi* task, which has the implementation of the Usb2Spi protocol to control the DW3000 from an external application. This illustrated on the Figure 15 below.

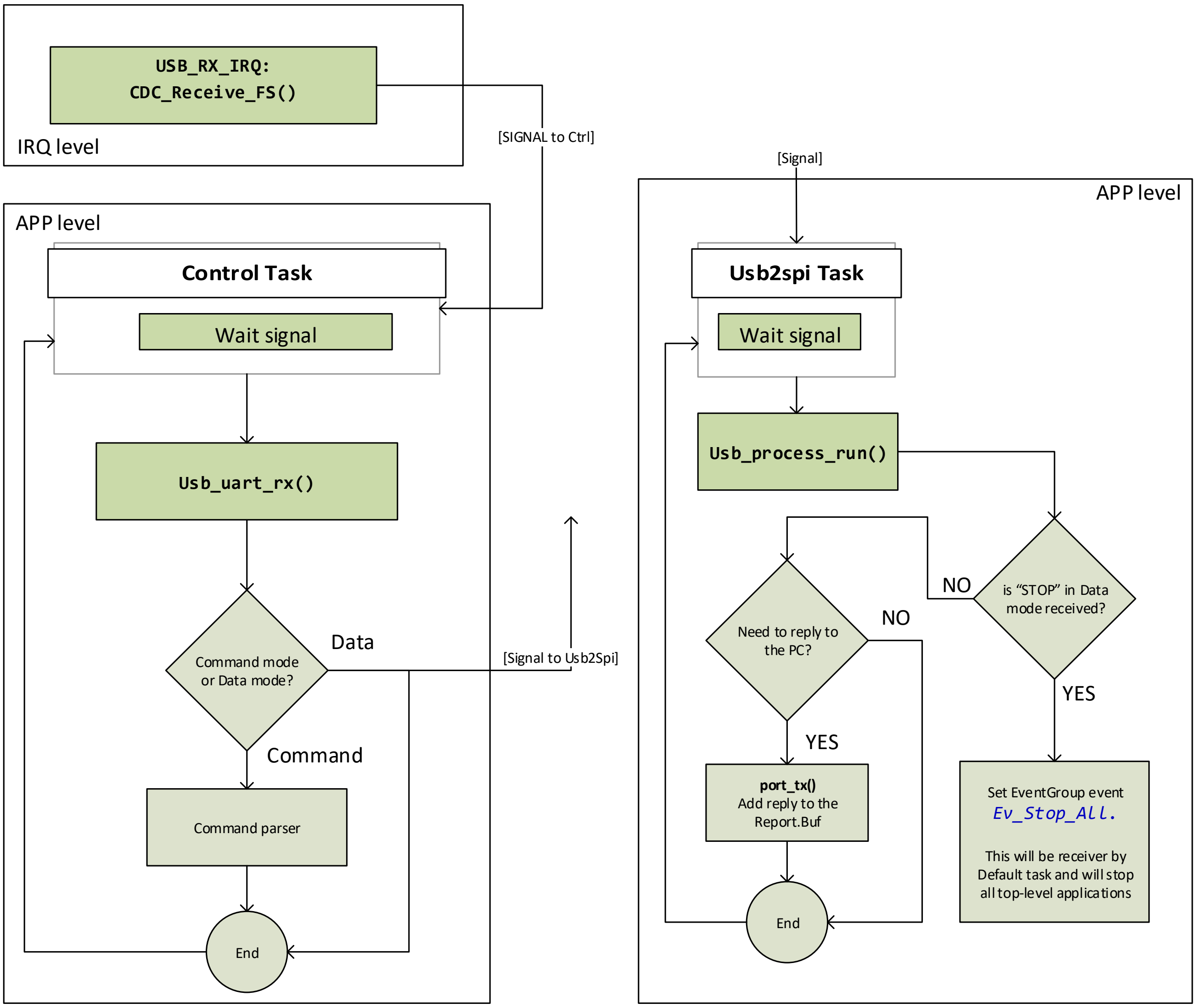


Figure 15: Usb2Spi top-level application

3.4 TCWM top-level applications

This task configure the DW3000, to run in the Test Continuous Wave Mode and awaiting to be stopped from *Control* task. It is an interface to the corresponding bare-metal production test functions, located in the *tcwm.c* file. This command can be used to check the appropriate reaction of the unit to the XTALTRIM command.

3.5 TCFM top-level applications

This task configure the DW3000, to run in the Test Continuous Frame Mode.This is an interface to the corresponding bare-metal production test functions, located in files *tcfm.c* respectively.
TCFM command can accept list of parameters <PARM>.

The format of the command is as follow:

Tcfm <NUM_OF_TX> <PERIOD_MS> <LENGTH_OF_THE_PACKET>

Where
<NUM_OF_TX> - number of packets to be transmitted;
<PERIOD_MS> - time in milliseconds from start of transmission of the packet to start of transmission of the sequential packet.
<LENGTH_OF_THE_PACKET> - length of packet in bytes <5> to <127> to be transmitted.

Please note, if the total duration of the packet would be longer, than the period, then packets would be transmitted back-to back.

The command with <PERIOD_MS> set to 1 (i.e. 1ms) is widely used for the certification purpose when require to measure the Transmit power level of the tested unit.

3.6 Listener top-level application

The Listener application will keep putting the DW3000 into receive mode and will report any received data, to run the Listener application, "listener" command is used from IDLE mode. Listener can accept the specific <PARAM>, which will specify its operational mode.

"listener 0" will start listener in "prefer data" mode. In this mode listener will attempt to receive and output all the received data. Depending on the UWB air load, the incoming data traffic can be faster that the output capability (USB), in that case some data will be dropped and not displayed over the USB backhaul.

"listener 1" (default) starts listener in "prefer speed" mode. In this mode the output of the data is limited to the max amount that can be achieved and still receive any UWB frames once per 1ms. In this mode only the first 5 bytes of the UWB message will be sent over the USB backhaul.

3.7 Low power mode in Juniper applications

The power-save feature is the Top-level application specific.

The Node application does not use a low power mode for DW3000 when it is not actively ranging. This is because the system is currently designed to keep the receiver on all the time to allow for the discovery of new tags. It is possible to modify the Node application to include power save features and put DW3000 to lower power mode for the periods of time when there no tag ranging is expected. This is out of the scope of current document.

The Tag application manually places DW3000 to the deep sleep and can place MCU to the low-power mode. However, the USB block assumes MCU is active all the time, thus this feature is not fully integrated to the Juniper platform for the MCU.

4 BUILDING AND RUNNING THE CODE

4.1 Building the code

The node application consists of ST CubeMX BSP libraries, the Decawave application and the driver sources. All of these are provided in the source code zip file.

The project is a Makefile project, which can be build using the ARM GCC compiler. Decawave currently uses the GNU Tools ARM for Embedded toolchain version 5.4 Q3. The GNU Tools ARM for Embedded compiler can be found at: https://launchpad.net/gcc-arm-embedded .

Unzip the source of the project and either build it manually by executing the “make” or import the project into the Eclipse IDE as a Makefile project and build it from there.

The developer may also import projects to the System Workbench for STM32, which is basically a customized version of Eclipse IDE with pre-installed GCC compiler (version 7.x.x).

4.2 Downloading and running the code

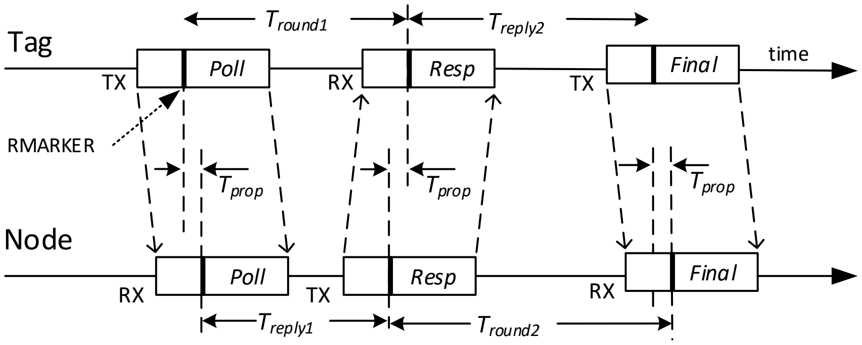
The Juniper hardware has a standard Nucleo (ST-LINK) interface, such it can be used to download the code to the Juniper HW platform. Decawave is currently using OpenOCD for this and the GNU Debugger (GDB) from the recommended toolchain for the debugging.

5 Appendix A

5.1 Two Way Ranging algorithm

The tag (*Initiator*) periodically initiates a range measurement, while the node (*Responder*) listens and responds to the tag and calculates the range. 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 message send and receive times are retrieved from the DW3000. These transmit and receive timestamps are used to work out a round trip delay and calculate the range, see [1].

In the ranging scheme shown in Figure 16 below, the tag sends a *Poll* message which is received by the node. The node replies with a response packets *Resp*, after which the tag sends the *Final* message.



The *Final* message communicates the tag's T_{round} and T_{reply} times to the node, 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 16: Distance calculation in TWR

Since the node also calculates the phase difference on the reception of the *Poll* and *Final* messages, this means that the tag can be located relative to the node after just a single ranging exchange.

The range and measured phase difference used by the node to work out tag's X-Y position.

5.2 *UWB configuration and TWR timing profile used in the system*

The node and tags hardware and software are designed to operate on one UWB configuration. This includes antenna designs, data rate and message timings, used in the TWR exchange. This specified in the Table 7 below.

Table 7: UWB mode of operation of system

UWB Config	Channel	Data Rate	Preamble Length	STS Length	STS Mode	PRF	Preamble Code	SFD	PHR mode	PAC
Value	5	6.81 Mbit/s	64	256	Mode1 SDC	64 MHz	9	DW-8	Standard	8

In the TWR timing profile, see Figure 17, there are two timing parameters, `pollTxToFinalTx_us` and `delayRx_us`. The `pollTxToFinalTx_us` parameter, specifies the rough time between RMARKER of *Poll* and RMARKER of the *Final* Tx messages for the tag, see 5.1. The `delayRx_us` parameter specifies the rough time, the Tag shall activate its receiver after transmission of the *Poll* in order to receive the *Response* from the node, see Figure 17.

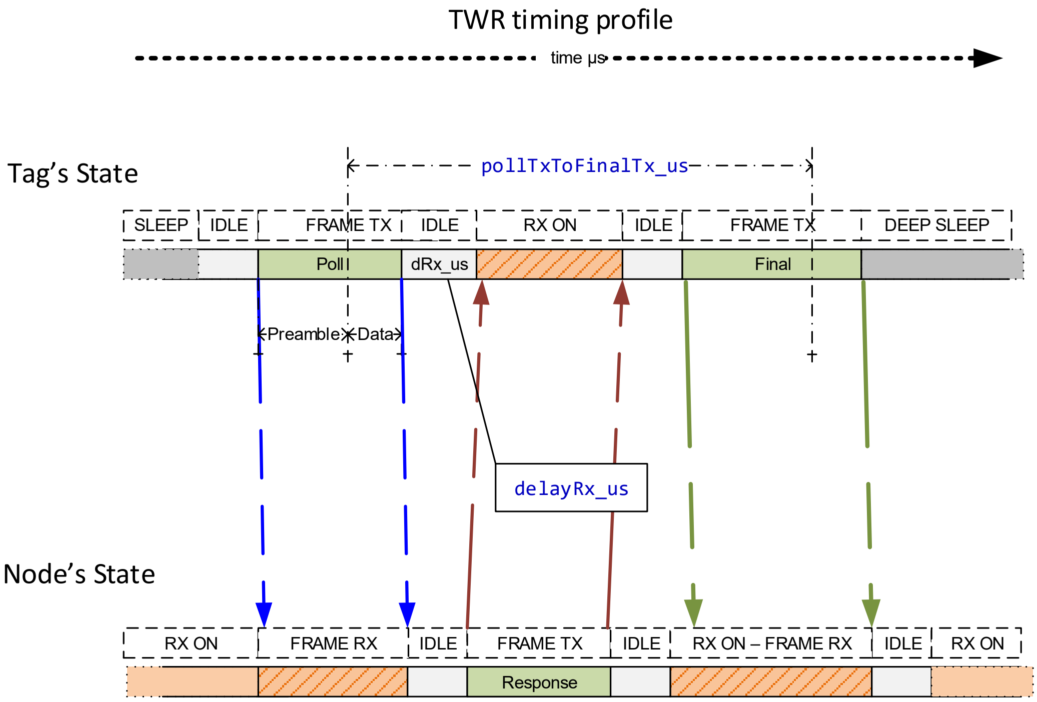


Figure 17: TWR timing profile

5.3 *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 specified in IEEE 802.15.4 [1] as the "Ranging Marker (RMARKER): The RMARKER is defined to be the time when the beginning of the first symbol of the PHR of the RFRAME is at the local antenna. 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. This is done automatically by DW3000, as long as the TX and RX antenna delays are configured.

The node uses configurable antenna delay values (which are initially defined in the default_config.h file). The values have been experimentally set by adjusting them until the average reported distance matches the measured distance. This can be re-programmed in the *FConfig* by using a corresponding control command interface, see 2.3.1.3.

5.4 UWB messages, used in TWR process

There are following message addressing modes employed in the system:

- the *Blink* message uses long (64-bits) source address mode
- the *Ranging Config* message uses short (16-bit) source address and long (64-bits) destination address mode
- the *Poll*, the *Response*, and the *Final* messages use both short (16-bit) source and short (16-bit) destination address mode.

The general message formats, used in the Discovery phase and the Ranging phase follow the IEEE 802.15.4 standard encoding for a data frame, for more description see below.
Note: The messages follow IEEE message encoding conventions, but these are not standardised RTLS messages. The reader is referred to the ISO/IEC 24730-62 international standard for details of standardised message formats for use in RTLS systems based on IEEE 802.15.4 UWB. This may be changed in future software revisions.

5.4.1 Tag blink message

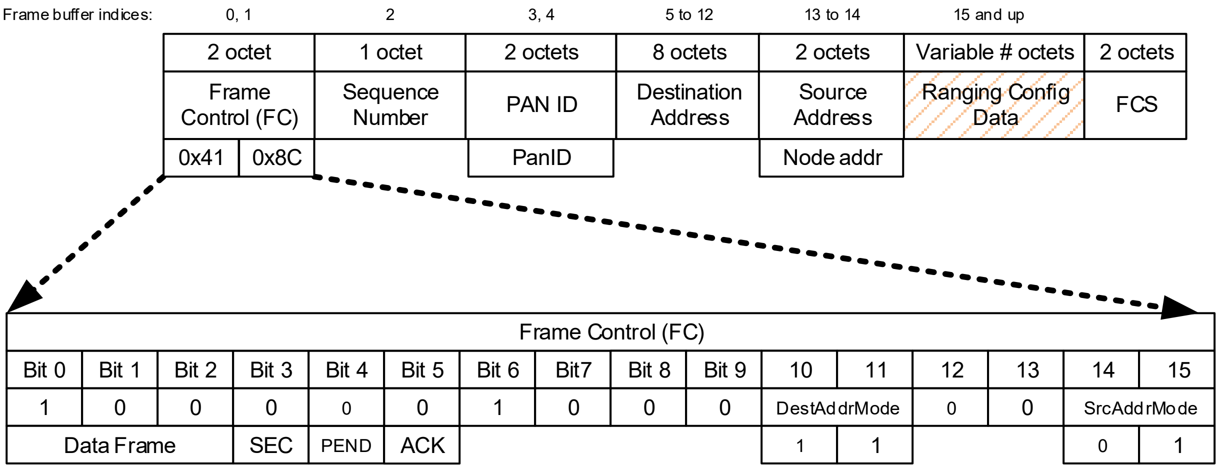
Initially a tag transmits *Blink* messages using the shortest IEEE blink message format. This is an optimized blink message which also can be used for TDOA (Time Difference of Arrival) location methods. The encoding of the blink message is as per Figure 18 below.

1 octet	1 octet	8 octets	2 octets
Frame Ctrl	Seq Number	Tag’s 64-bit Address	FCS
0xC5		-	-

Figure 18: Encoding of Tag's 12-bytes blink message

5.4.2 Ranging Config message

During initial blinking the tag has only the long 64-bit address. To support short timings, the node assigns to the tag a temporary short address using *Ranging Config* response. Thus, for *Ranging Config* message to be delivered to the tag, the short-long addressing mode is used. This is shown on the Figure 19 below.



Frame buffer indices:	0, 1	2	3, 4	5 to 12	13 to 14	15 and up	
	2 octet	1 octet	2 octets	8 octets	2 octets	Variable # octets	2 octets
	Frame Control (FC)	Sequence Number	PAN ID	Destination Address	Source Address	Ranging Config Data	FCS
	0x41 0x8C		PanID		Node addr		

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			0	1

Figure 19: Frame format of Ranging Config message

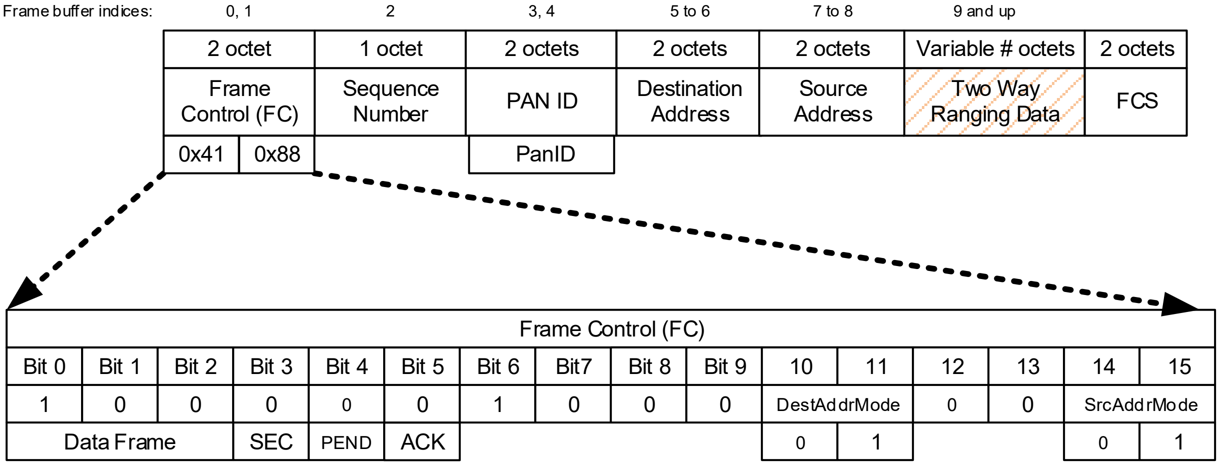
The PAN ID of the system and the node's 16-bit addresses are in the MAC header, and the rest is in the *Ranging Config Data* field: tag's new 16-bit address and its configuration parameters. The description of *Ranging Config Data* part of a range_init_msg_t structure is given in the Table 8.

Table 8: Fields within the Ranging Config message

Parameter	Size, octets	Value	Description
fCode	1	0x20	Function code: This octet 0x20 identifies this as node's Ranging Config message
tagAddr	2	-	Tag's short address to be used in the Ranging phase
reserved	4	-	Reserved for compatibility with Decawave's TREK-1000 project
version	1	0x02	Version of *Ranging Config* message. Version 0x02: TWR to one node.
sframePeriod_ms	2	-	Super Frame period, ms
slotCorr_us	4	-	Slot correction from reception of *Blink* to the dedicated slot, µs
pollTxToFinalTx_us	2	-	The rough delay to be used by the tag, when it ranging to the node: from the RMARKER of the *Poll* to the RMARKER of the *Final*, µs
delayRx_us	2	-	The tag shall start reception of *Response* with this delay after the end of transmission of the *Poll*, µs
pollMultFast	2	-	The multiplier factor for Fast ranging (when tag is moving), in number of superframes, e.g. 1 means "range every 1 superframe"
pollMultSlow	2	-	The multiplier factor for Slow ranging (when tag is stationary), in number of superframes, e.g. 10 means "range every 10th superframe"
Mode	2	-	Bit fields for tag mode of operation: bit 0: The Tag should use its IMU to detect its stationary mode. bit 1-15: reserved for future application enhancement.

5.4.3 Ranging messages

During the Two Way Ranging, the tag and node use short (16-bit) addressing modes. The format of the ranging frames, which are *Poll*, *Response* and *Final* is shown in Figure 20 below.



Frame buffer indices:	0, 1	2	3, 4	5 to 6	7 to 8	9 and up	
	2 octet	1 octet	2 octets	2 octets	2 octets	Variable # octets	2 octets
	Frame Control (FC)	Sequence Number	PAN ID	Destination Address	Source Address	Two Way Ranging Data	FCS
	0x41 0x88		PanID				

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					0	1			0	1

Figure 20: Frame format used for Ranging

The content of the *Two Way Ranging Data* portion of the frame, defined by a first octet, identifies the type of the Ranging message: *Poll*, *Response* or *Final*, as per Table 9 below.

Table 9: List of Function Codes in the TWR exchange

Function Code		Description
Twr_Fcode_Tag_Poll	0x84	Initiator (Tag) Poll message
Twr_Fcode__Resp	0x72	Responder (Node) extended Response
Twr_Fcode_Tag_Accel_Final	0x89	Initiator (Tag) Final message back to responder (Node)

5.4.4 The Poll message

The *Poll* message structure defined in the code as a structure of type poll_msg_t. This sent by the tag to initiate a Ranging sequence. Table 10 describes the individual fields within the *Poll* message.

Table 10: Fields within the ranging Poll message

Parameter	Size, octets	Description
fCode	1	Function code: This octet 0x84 identifies this as a tag Poll message
rNum	1	Range number: This is a range sequence number; after each range attempt this number is incremented (by modulo 256).

5.4.4.1 Response message

The *Response* message structure is defined in the code as a structure of type resp_msg_t. This sent by the node as a *Response* to a *Poll* from the tag. Table 11 describes the individual fields within the *Response* message.

Table 11: Fields within the ranging Response message

Parameter	Size, octets	Description
fCode	1	Function code: This octet identifies this as the extended Response message
slotCorr_us	4	Tag's correction in microseconds, Least Significant Byte First. This four octets are a correction factor that adjusts the Tag's next wakeup duration so that the Tag's ranging activity can be assigned and aligned into its dedicated slot.
rNum	1	Range number: This is a range sequence number, corresponding to the range number as sent in the Poll.
x_cm	2	Last measurement of the X coordinate reported back to the tag from the node. If no previous measurements, the 0xDEAD reported (-8531 dec).
y_cm	2	Last measurement of the Y coordinate reported back to the tag from the node. If no previous measurements, the 0xDEAD reported (-8531 dec).
offset_ppmh	2	The Tag's crystal offset value with respect to the Nodes' master TCXO, reported back to the tag from the node. If no previous measurements, the 0xDEAD reported (-8531 dec). This can be used in the automatic crystal trimming process with respect to the node on the tag's side.

5.4.4.2 Final message

The *Final* message is a structure of type final_msg_t and it sent by the tag after receiving the node's *Response* message. Table 12 lists and describes the individual fields within the *Final* message.

Table 12: Fields within the ranging Final message

Octet #'s	Size, octets	Description
fCode	1	Function code: This octet identifies the message as the tag Final message
rNum	1	Range number: This is a range sequence number, corresponding to the range number as sent in the Poll.
pollTx_ts	5	Tag Poll TX time: This 5-octet field is the TX timestamp for the tag's poll message, i.e. the precise time the *Poll* frame was transmitted.
responseRx_ts	5	Tag Response RX time: This 5-octet field is the RX timestamp for the response from the node, i.e. the time the tag received the *Response* frame from the node.
finalTx_ts	5	Final TX time: This 5-octet field is the TX timestamp of the final message, i.e. the time the *Final* frame will be transmitted, (this is pre-calculated by the tag).
flag	1	User flag, not in use.
acc_x	2	Used to supply X absolute location to the Trilat function when Tag acts as the Fixed infrastructure element.
acc_y	2	Used to supply Y absolute location to the Trilat function when Tag acts as the Fixed infrastructure element.
acc_z	2	Used to supply Z absolute location to the Trilat function when Tag acts as the Fixed infrastructure element.

5.5 *Slot Time correction method in between Node and Tag*

The node and tag both use RTC timers to implement the slotted TDMA access method. The node's RTC timer is used to trigger the start of node's superframe, which absolute timestamp in MCU RTC time units is saved in the `gRtcNode` timestamp variable (in the Node's application it is `gRtcSFrameZeroCnt).`
The tag's RTC timer is intended to keep the tag transmitting in its assigned TDMA slot. Every time the tag is starting a *Poll* frame, the timer is configured to expire every `wakeUpPeriodCorrected_ns`, which holds the value, of the duration of the tag's superframe – note this is in the tag's time and not in the node's time domain (the value is close to the node's superframe period, but may vary). If the tag is not intended to range on the next expiration of the timer (i.e. when the tag is configured to range less frequently than every superframe), the timer will keep expiring every `wakeUpPeriodCorrected_ns` until tag decides it is time for the next ranging exchange.
On the reception of the *Response* message from the node, the tag receives the `slotCorr_us`, which specifies the time where the node has expected the reception of the tag's *Poll* with respect to the start of the node's superframe (gRtcNode), see Figure 21. Upon reception of the Response, the `nextWakeUpPeriod_ns` is calculated as follows:

```
nextWakeUpPeriod_ns  = 1e6* sframePeriod_ms;
nextWakeUpPeriod_ns -= WKUP_RESOLUTION_NS * (gRtcTag - respRxRtcTag);
nextWakeUpPeriod_ns -= 1e3* slotCorr_us;
```

The method above includes the correction of the time needed for the tag to wake up and initiate transmit the *Poll* message, so that for the next transmission the RTC should wake up the MCU at the correct time for the *pollTask* to execute and transmit the *Poll* message.

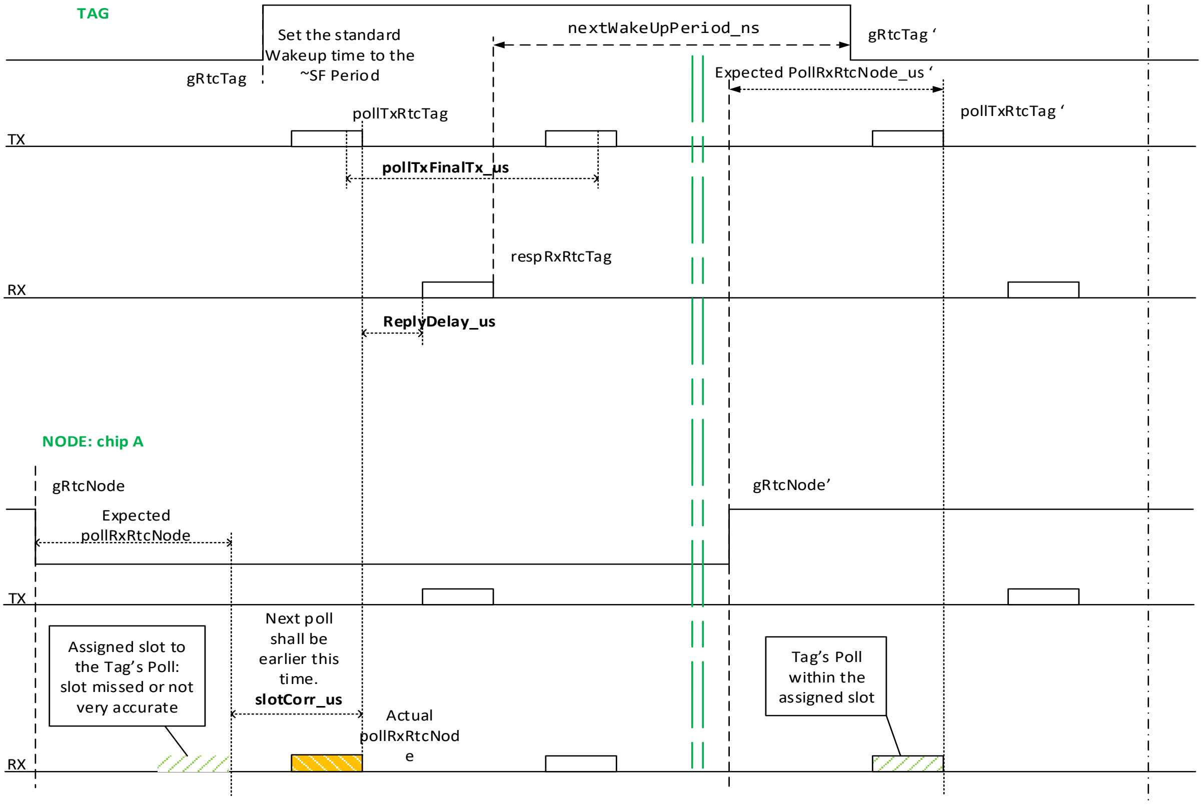

TAG
Set the standard Wakeup time to the ~SF Period
nextWakeUpPeriod_ns
gRtcTag '
gRtcTag
Expected PollRxRtcNode_us '
pollTxRtcTag
pollTxRtcTag '
TX
pollTxFinalTx_us
respRxRtcTag
RX
ReplyDelay_us
NODE: chip A
gRtcNode
gRtcNode'
Expected pollRxRtcNode
TX
Next poll shall be earlier this time.
slotCorr_us
Assigned slot to the Tag's Poll: slot missed or not very accurate
Actual pollRxRtcNode
Tag's Poll within the assigned slot
RX

Figure 21: Node-Tag Slot Time correction method

5.6 *The application architecture in the flowchart*

The application structure is present on block-diagrams on Figure 22, Figure 23 and Figure 24.

The flowchart on Figure 22 shows common platform blocks, on top of which a particular top-level application is running and utilizing the common infrastructure (core tasks).

The flowchart on Figure 23 shows the interaction of the software blocks with each other on the example of the **tag** top-level can be found in the sources.

The flowchart on Figure 24 shows the interaction of the software blocks with each other on the example of the **node** top-level can be found in the sources.

The application of tag and node are very similar structured. A new application can be added on the same manner to the platform.

With the reference to the figures, the main blocks are as follow:

1. Initialization of the hardware and peripherals. This is initially generated by CubeMX HAL software, by providing the initialization of the target MCU and simplifying the physical interface driver development.
2. RTOS based functionality. This includes the FreeRTOS kernel, and all files with prefix “task_”. This separates the functionality of the applications, and aids in the management of MCU time more effectively and makes the code design cleaner at the expense of a small increase in latency. This adds an “application” layer, sitting on the top of bare-metal implementations, i.e. Tag, Node, Trilat, TCFM, TCWM and Usb2Spi.

3. Bare-metal implementation of functionality: callbacks, tx_start, usb_uart_rx, usb_uart_tx, etc.

The source files are organized to match the architecture of the application, see section 1.2.

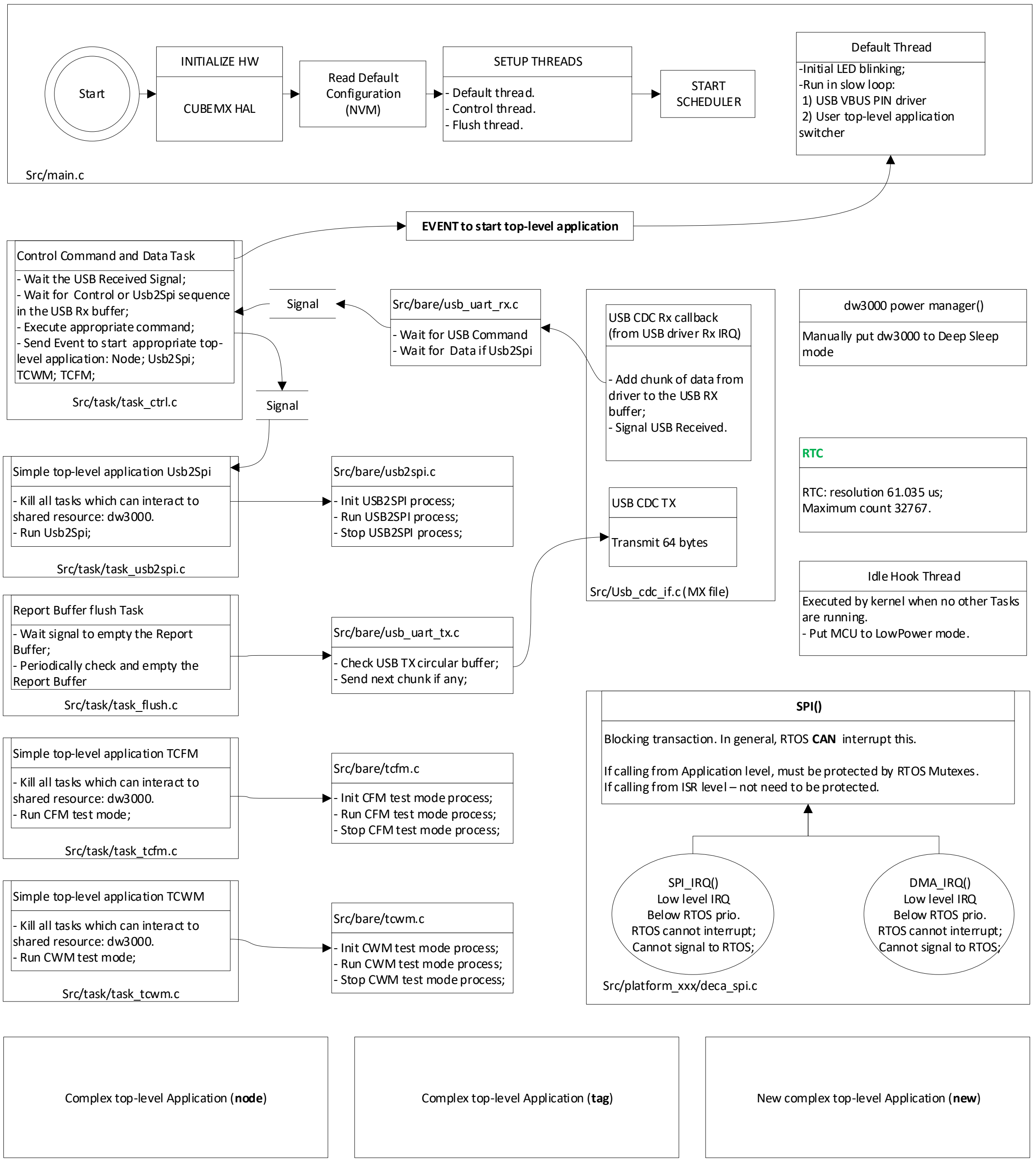

Start
INITIALIZE HW
CUBEMX HAL
Read Default Configuration (NVM)
SETUP THREADS
- Default thread.
- Control thread.
- Flush thread.
START SCHEDULER
Default Thread
-Initial LED blinking;
-Run in slow loop:
1) USB VBUS PIN driver
2) User top-level application switcher
Src/main.c
EVENT to start top-level application
Control Command and Data Task
- Wait the USB Received Signal;
- Wait for Control or Usb2Spi sequence in the USB Rx buffer;
- Execute appropriate command;
- Send Event to start appropriate top-level application: Node; Usb2Spi; TCWM; TCFM;
Src/task/task_ctrl.c
Signal
Signal
Src/bare/usb_uart_rx.c
- Wait for USB Command
- Wait for Data if Usb2Spi
USB CDC Rx callback (from USB driver Rx IRQ)
- Add chunk of data from driver to the USB RX buffer;
- Signal USB Received.
USB CDC TX
Transmit 64 bytes
Src/Usb_cdc_if.c (MX file)
dw3000 power manager()
Manually put dw3000 to Deep Sleep mode
RTC
RTC: resolution 61.035 us;
Maximum count 32767.
Idle Hook Thread
Executed by kernel when no other Tasks are running.
- Put MCU to LowPower mode.
Simple top-level application Usb2Spi
- Kill all tasks which can interact to shared resource: dw3000.
- Run Usb2Spi;
Src/task/task_usb2spi.c
Src/bare/usb2spi.c
- Init USB2SPI process;
- Run USB2SPI process;
- Stop USB2SPI process;
Report Buffer flush Task
- Wait signal to empty the Report Buffer;
- Periodically check and empty the Report Buffer
Src/task/task_flush.c
Src/bare/usb_uart_tx.c
- Check USB TX circular buffer;
- Send next chunk if any;
SPI()
Blocking transaction. In general, RTOS CAN interrupt this.
If calling from Application level, must be protected by RTOS Mutexes.
If calling from ISR level – not need to be protected.
SPI_IRQ()
Low level IRQ
Below RTOS prio.
RTOS cannot interrupt;
Cannot signal to RTOS;
DMA_IRQ()
Low level IRQ
Below RTOS prio.
RTOS cannot interrupt;
Cannot signal to RTOS;
Src/platform_xxx/deca_spi.c
Simple top-level application TCFM
- Kill all tasks which can interact to shared resource: dw3000.
- Run CFM test mode;
Src/task/task_tcfm.c
Src/bare/tcfm.c
- Init CFM test mode process;
- Run CFM test mode process;
- Stop CFM test mode process;
Simple top-level application TCWM
- Kill all tasks which can interact to shared resource: dw3000.
- Run CWM test mode;
Src/task/task_tcwm.c
Src/bare/tcwm.c
- Init CWM test mode process;
- Run CWM test mode process;
- Stop CWM test mode process;
Complex top-level Application (node)
Complex top-level Application (tag)
New complex top-level Application (new)

Figure 22 Common platform blocks

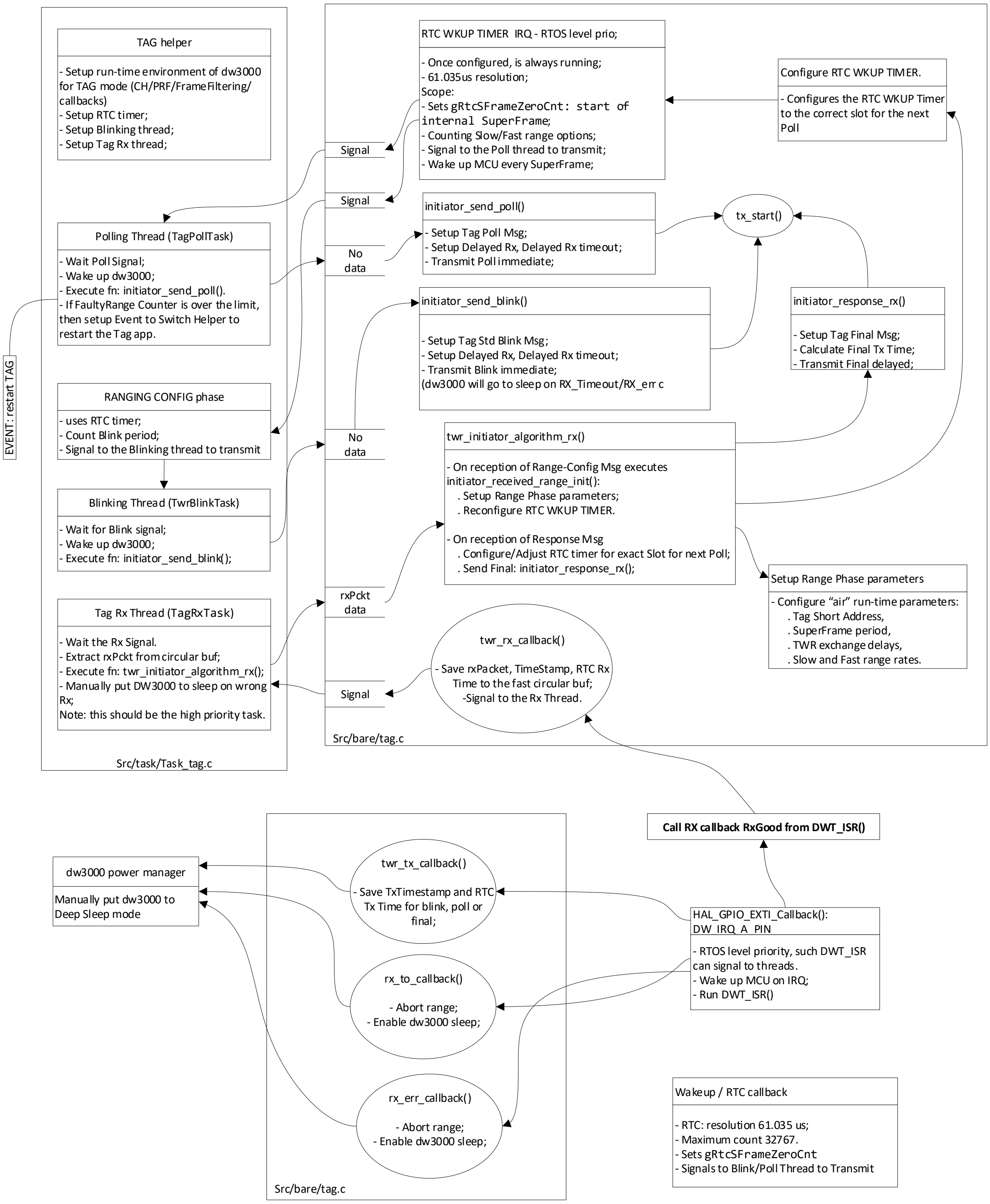

TAG helper
- Setup run-time environment of dw3000 for TAG mode (CH/PRF/FrameFiltering/ callbacks)
- Setup RTC timer;
- Setup Blinking thread;
- Setup Tag Rx thread;
Polling Thread (TagPollTask)
- Wait Poll Signal;
- Wake up dw3000;
- Execute fn: initiator_send_poll().
- If FaultyRange Counter is over the limit, then setup Event to Switch Helper to restart the Tag app.
EVENT: restart TAG
RANGING CONFIG phase
- uses RTC timer;
- Count Blink period;
- Signal to the Blinking thread to transmit
Blinking Thread (TwrBlinkTask)
- Wait for Blink signal;
- Wake up dw3000;
- Execute fn: initiator_send_blink();
Tag Rx Thread (TagRxTask)
- Wait the Rx Signal.
- Extract rxPckt from circular buf;
- Execute fn: twr_initiator_algorithm_rx();
- Manually put DW3000 to sleep on wrong Rx;
Note: this should be the high priority task.
Src/task/Task_tag.c
Signal
Signal
No data
No data
rxPckt data
Signal
RTC WKUP TIMER IRQ - RTOS level prio;
- Once configured, is always running;
- 61.035us resolution;
Scope:
- Sets gRtcSFrameZeroCnt: start of internal SuperFrame;
- Counting Slow/Fast range options;
- Signal to the Poll thread to transmit;
- Wake up MCU every SuperFrame;
Configure RTC WKUP TIMER.
- Configures the RTC WKUP Timer to the correct slot for the next Poll
initiator_send_poll()
- Setup Tag Poll Msg;
- Setup Delayed Rx, Delayed Rx timeout;
- Transmit Poll immediate;
tx_start()
initiator_send_blink()
- Setup Tag Std Blink Msg;
- Setup Delayed Rx, Delayed Rx timeout;
- Transmit Blink immediate;
(dw3000 will go to sleep on RX_Timeout/RX_err c
initiator_response_rx()
- Setup Tag Final Msg;
- Calculate Final Tx Time;
- Transmit Final delayed;
twr_initiator_algorithm_rx()
- On reception of Range-Config Msg executes initiator_received_range_init():
. Setup Range Phase parameters;
. Reconfigure RTC WKUP TIMER.
- On reception of Response Msg
. Configure/Adjust RTC timer for exact Slot for next Poll;
. Send Final: initiator_response_rx();
Setup Range Phase parameters
- Configure "air" run-time parameters:
. Tag Short Address,
. SuperFrame period,
. TWR exchange delays,
. Slow and Fast range rates.
twr_rx_callback()
- Save rxPacket, TimeStamp, RTC Rx Time to the fast circular buf;
-Signal to the Rx Thread.
Src/bare/tag.c
Call RX callback RxGood from DWT_ISR()
dw3000 power manager
Manually put dw3000 to Deep Sleep mode
twr_tx_callback()
- Save TxTimestamp and RTC Tx Time for blink, poll or final;
rx_to_callback()
- Abort range;
- Enable dw3000 sleep;
rx_err_callback()
- Abort range;
- Enable dw3000 sleep;
Src/bare/tag.c
HAL_GPIO_EXTI_Callback(): DW_IRQ_A_PIN
- RTOS level priority, such DWT_ISR can signal to threads.
- Wake up MCU on IRQ;
- Run DWT_ISR()
Wakeup / RTC callback
- RTC: resolution 61.035 us;
- Maximum count 32767.
- Sets gRtcSFrameZeroCnt
- Signals to Blink/Poll Thread to Transmit

Figure 23 Operational flow on the "tag" top-level application

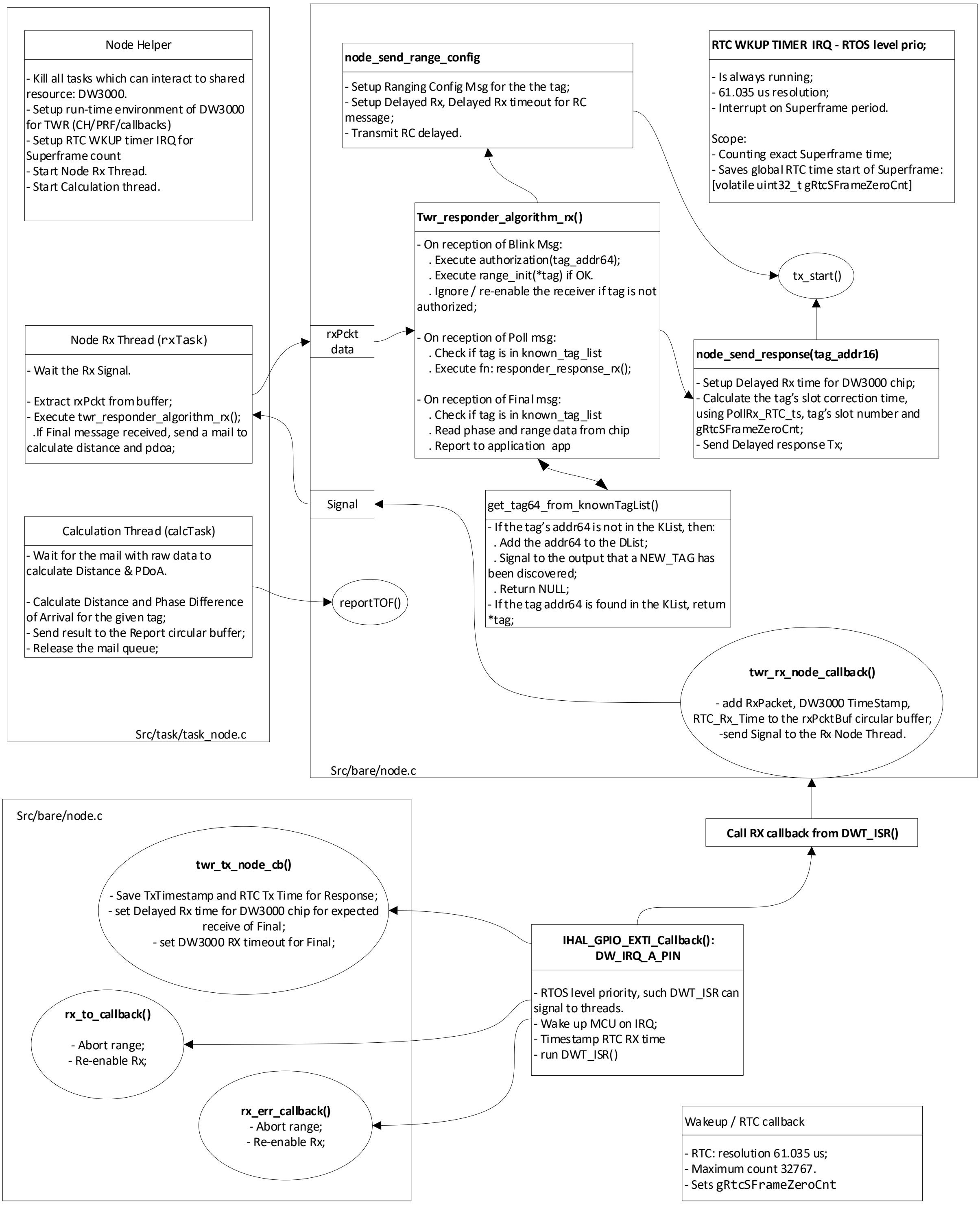

Node Helper
- Kill all tasks which can interact to shared resource: DW3000.
- Setup run-time environment of DW3000 for TWR (CH/PRF/callbacks)
- Setup RTC WKUP timer IRQ for Superframe count
- Start Node Rx Thread.
- Start Calculation thread.
Node Rx Thread (rxTask)
- Wait the Rx Signal.
- Extract rxPckt from buffer;
- Execute twr_responder_algorithm_rx();
.If Final message received, send a mail to calculate distance and pdoa;
Calculation Thread (calcTask)
- Wait for the mail with raw data to calculate Distance & PDoA.
- Calculate Distance and Phase Difference of Arrival for the given tag;
- Send result to the Report circular buffer;
- Release the mail queue;
Src/task/task_node.c
node_send_range_config
- Setup Ranging Config Msg for the the tag;
- Setup Delayed Rx, Delayed Rx timeout for RC message;
- Transmit RC delayed.
RTC WKUP TIMER IRQ - RTOS level prio;
- Is always running;
- 61.035 us resolution;
- Interrupt on Superframe period.
Scope:
- Counting exact Superframe time;
- Saves global RTC time start of Superframe: [volatile uint32_t gRtcSFrameZeroCnt]
Twr_responder_algorithm_rx()
- On reception of Blink Msg:
. Execute authorization(tag_addr64);
. Execute range_init(*tag) if OK.
. Ignore / re-enable the receiver if tag is not authorized;
- On reception of Poll msg:
. Check if tag is in known_tag_list
. Execute fn: responder_response_rx();
- On reception of Final msg:
. Check if tag is in known_tag_list
. Read phase and range data from chip
. Report to application app
tx_start()
rxPckt data
node_send_response(tag_addr16)
- Setup Delayed Rx time for DW3000 chip;
- Calculate the tag's slot correction time, using PollRx_RTC_ts, tag's slot number and gRtcSFrameZeroCnt;
- Send Delayed response Tx;
Signal
get_tag64_from_knownTagList()
- If the tag's addr64 is not in the KList, then:
. Add the addr64 to the DList;
. Signal to the output that a NEW_TAG has been discovered;
. Return NULL;
- If the tag addr64 is found in the KList, return *tag;
reportTOF()
twr_rx_node_callback()
- add RxPacket, DW3000 TimeStamp, RTC_Rx_Time to the rxPcktBuf circular buffer;
-send Signal to the Rx Node Thread.
Src/bare/node.c
Src/bare/node.c
twr_tx_node_cb()
- Save TxTimestamp and RTC Tx Time for Response;
- set Delayed Rx time for DW3000 chip for expected receive of Final;
- set DW3000 RX timeout for Final;
rx_to_callback()
- Abort range;
- Re-enable Rx;
rx_err_callback()
- Abort range;
- Re-enable Rx;
Call RX callback from DWT_ISR()
IHAL_GPIO_EXTI_Callback(): DW_IRQ_A_PIN
- RTOS level priority, such DWT_ISR can signal to threads.
- Wake up MCU on IRQ;
- Timestamp RTC RX time
- run DWT_ISR()
Wakeup / RTC callback
- RTC: resolution 61.035 us;
- Maximum count 32767.
- Sets gRtcSFrameZeroCnt

Figure 24 Operational flow on the "node" top-level application

6 BIBLIOGRAPHY

Ref	Author	Title
[1]	IEEE	IEEE 802.15.4-2011 or “IEEE Std 802.15.4™-2015” (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/
[2]	Decawave	DecaRanging PC application, available from https://www.decawave.com/

7 REVISION HISTORY

Revision	Date	Description
1.2	06-Dec-2021	Public release
1.1	25-Jul-2020	Updated to include new commands
1.0	16-Apr-2020	Updated to include Listener application
0.2	03-Oct-2019	Revised to match actual
0.1	24-Jan-2019	Initial draft

8 FURTHER INFORMATION

Decawave develops semiconductors solutions, software, modules, reference designs - that enable real-time, ultra-accurate, ultra-reliable local area micro-location services. Decawave's technology enables an entirely new class of easy to implement, highly secure, intelligent location functionality and services for IoT and smart consumer products and applications.

For further information on this or any other Decawave product, please refer to our website www.decawave.com.