DWM1002 PDoA Node Source Code Guide

Version 1.2

This document is subject to change without notice.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This document, “DWM1002 PDoA Node Source Code Guide” is a guide for an embedded software developers to the application source code of Decawave’s Phase Difference of Arrival (PDoA) evaluation kit node application, running on the Nordic Semiconductor ARM Cortex M4 MCU on the PDoA node hardware platform. The code and the document is organized on the same manner as the “PDoA Tag Source Code Guide” [3]. The Node project is supplied as Segger Embedded studio project with Nordic SDK16.0.0.

The application source code employs a real-time operating system (FreeRTOS), 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 PDoA node, acts as a responder and performs double-sided two-way ranging with tags, and then calculates the range and phase difference of the arriving signals on two antennae using two DW1000 receivers and reports the calculated X and Y coordinates 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.

On start-up the PDoA node listens for *Blink* messages from tags and when a *Blink* message has received, the PDoA node responds to the tag with a *Ranging Config* message. The *Ranging Config* message provides information to the tag describing how to perform ranging with the PDoA node. This information includes the PAN ID, the short address of the PDoA node, a short address assignment for the tag, and timing parameters for the ranging phase. Upon receiving the *Ranging Config* message, the tag changes to ranging mode where it periodically initiates ranging exchanges with the PDoA node.

The PDoA node has two antennae and two DW1000 ICs (one for each antenna). Each ranging exchange starts with the tag sending *a Poll* message, while the PDoA node listens with only one DW1000 IC active. When the PDoA node receives the *Poll* message, it replies with a *Response* message, and the tag completes the ranging exchange by sending a *Final* message. The PDoA node turns on both DW1000 ICs to receive the *Final* message. This enables it to determine the phase difference (PDoA) between their arrival times at each antennae. The PDoA node then calculates X and Y coordinates of the tag which it reports via USB/UART for displaying by the PC GUI application.

1.2 This document

This document relates to the PDoA node project "dwm1002" of the Beta PDoA Kit source code package, application version ”5.0.0” and DW1000 device driver version “04.01.01”, running on PDoA node hardware revision “v2”. In the source code itself, the application version is specified in the “version.h” file, and the DW1000 driver version information may be found in the file “deca_version.h”.

- Section 2 provides a description of the architecture and structure of the PDoA node platform and application.
- Section 3 provides a description of the operation of the core tasks.
- Section 4 describes the operation of the top-level applications.
- Section 5 gives a brief overview of how to build and debug the code.
- Section 6 Appendix A details the ranging method and the ranging messages, and provides a flowchart of the interaction between the major software blocks, and gives a list of the project files.

2 DESCRIPTION OF THE PDOA NODE ARM PLATFORM

The PDoA node platform is based on Nordic Semiconductor ARM Cortex M4F MCU. The sections below discuss the architecture, structure and workflow of the software, residing in the microcontroller, that the developer can 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).

2.1 Node architecture

An overview of the architecture of the PDoA node is given by Figure 1. 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 corresponding physical interfaces. The detailed scheme of interaction between software blocks is given in Appendix A, section 6.6.

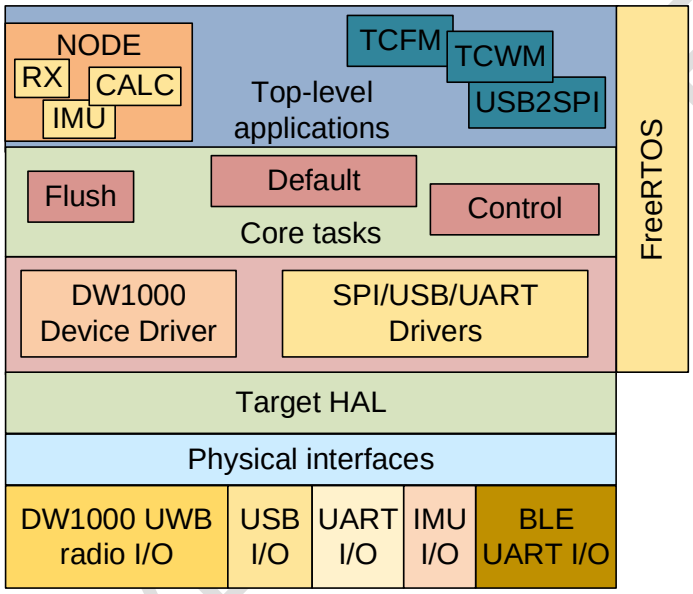


Figure 1: Architecture of the PDoA node application

2.1.1 Top-level applications layer

This is the top level of the software defining the operation of the node unit. It has four separate applications / modes of operation. The main mode, named “NODE” is the normal PDoA node functionality as introduced in section 1.1 above, while the three other operational modes are for testing purposes. These are: TCFM – Test Continuous Frame transmission Mode; TCWM – Test Continuous Wave transmission Mode; and, USB2SPI test mode, allowing external test software direct access to the DW1000 SPI’s.

The top-level applications cannot run concurrently since they use the same resources: e.g. NODE application configures the DW1000s to do two-way ranging, while TCWM application initializes selected DW1000 to run a Continuous Wave mode test.

2.1.2 Core tasks

The core tasks are always running once the RTOS kernel has been started. These core tasks are described in section 3.2. 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: 3.2.1
- The *Control* core task is responsible for reception and execution of *commands* from external IO interfaces USB and 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: 0, 3.2.3.
- The *Flush* core task is responsible for transmitting of any output data to the external I/O interfaces USB and UART. This is described in detail in section 3.2.4.

2.1.3 Drivers

The drivers are responsible for translating of higher-layer requests and the specific sequences to control particular peripherals.

To control the DW1000 UWB radio transceivers the DW1000 API and the device driver is incorporated as a library. Other physical interfaces have corresponding drivers, namely UART, USB, etc.

2.1.4 FreeRTOS

The PDoA node platform runs under the FreeRTOS operating system control. This is a CMSIS compatible RTOS, thus the PDoA node 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.

The PDoA node 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.

2.2 PDoA Node source code - folder structure

The project folder structure can be seen on Figure 2 below.

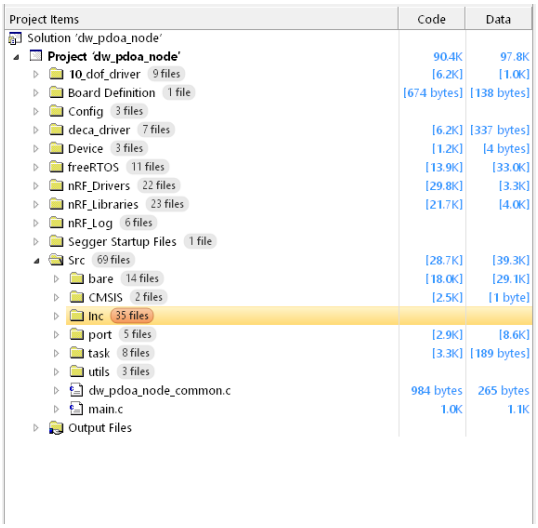


Figure 2: Node source code folder structure

Folder name	Description
Output files	The target build directory.
SDK	This folder contains the HAL drivers and runtime support for nRF52840
10_dof_driver	This folder contains 10 degrees of freedom IMU driver
deca_driver	This folder contains DW1000 driver
Src	Tag's main project folder, where: • **"bare"** – collection of bare-metal functions and "processes", which holds the actual functionality to make a non-RTOS based application; • **"inc"** – include files; • **"port"** – folder with platform-dependent files; • **"task"** – collection of core and user-application tasks, needed to implement the actual functionality, using RTOS; • **"utils"** – collection of support utilities.

3 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.

3.1 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 of the configuration parameters, the *main()* code initialises the core tasks and enables the Real-Time kernel (scheduler), see Figure 3. After starting of the RTOS kernel, the core tasks will begin to run "in parallel", each executing its dedicated role.

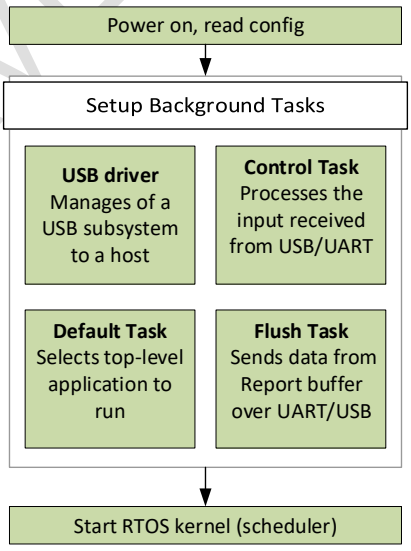


Figure 3: Initial startup workflow

3.2 Core tasks

There are few logically different core functions, which are running “in parallel” as core tasks all the time. All core tasks have lower priority 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 4.1).

The following are the core tasks.

- 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 DW1000’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 3.2.1.
- 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 3.2.4.

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 is implemented as a part of this task.

3.2.1 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 0 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.

For more details about events and tasks interconnection mechanisms see section 3.3.

3.2.2 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 4, 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 3.2.3.

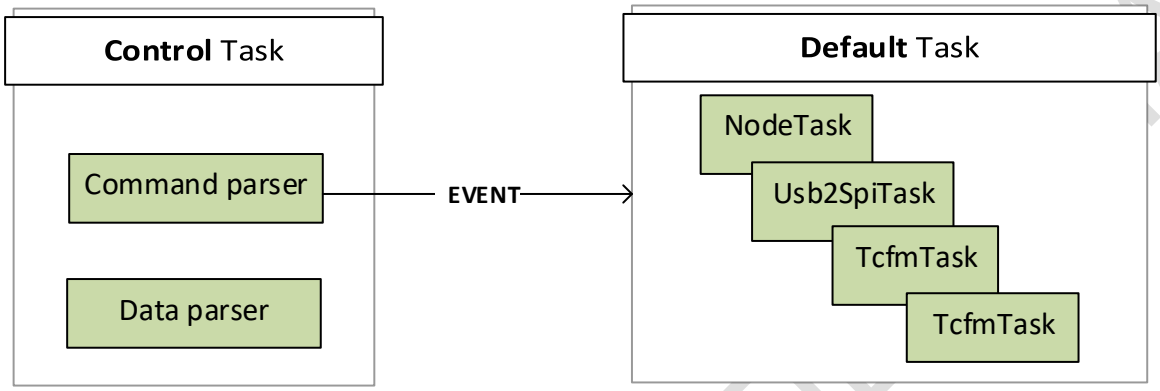


Figure 4 Control task (in command mode) sends EVENT to the Default task

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).

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 5 below. For more information about *Usb2Spi* top-level application see 4.2.

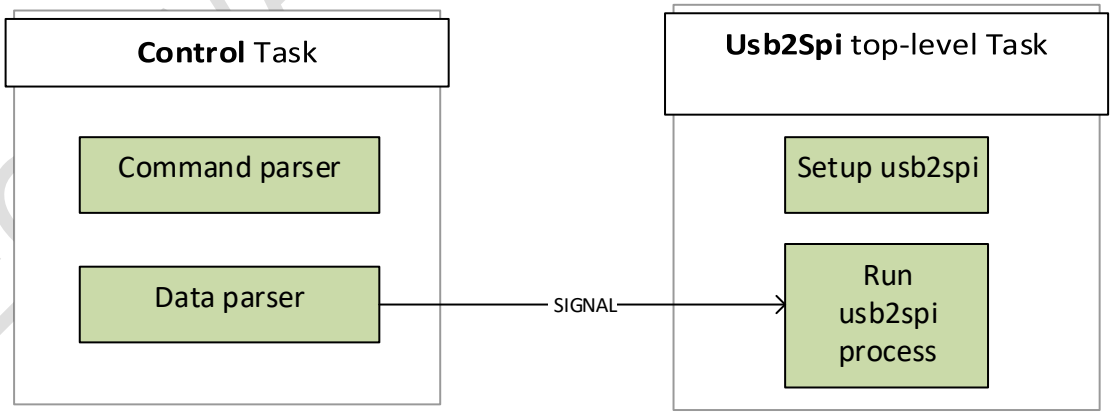


Figure 5 Control Task (in data mode) sends signal to the Usb2Spi application

3.2.3 Command mode of Control task

The command mode of *Control* task includes parsing and execution of a number of 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 corresponding reply is sent for output. The *Flush* task is then actually sending of the output to the USB/UART.

There are three types of commands, which can be parsed in the node: anytime commands, state changing commands and run-time parameters access commands, which all can be named as “generic commands” set, and a set of commands to control the PDoA Node top-level application.

The generic commands set described in subsections 3.2.3.1, 3.2.3.2, 3.2.3.3, and Node’s application set - in subsections 3.2.3.4 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>).

3.2.3.1 Anytime commands

The anytime commands listed in Table 1 can be executed anytime except of the *USPI* mode of operation, i.e. when binary data parser is running (see 0). 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 ?	Outputs a list of all known commands.
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’s top-level application commands set	See section 3.2.3.4 below.

3.2.3.2 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 4 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. Chip A will act as a master and chip B will act as a slave
TCWM <val>	Run the Test Continuous Wave transmission Mode top-level application. The DW1000 chip can be selected by optional <val> variable, which can be specified to "0" (default) or to "1". If <val> is not specified, the default chip will be used.
TCFM <val>	Run the Test Continuous Frame transmission Mode top-level application. The DW1000 chip can be selected by optional <val> variable, which can be specified to "0" (default) or to "1". If <val> is not specified, the default chip will be used.
USPI <val>	Run the USB-to-2SPI conversion top-level application. The DW1000 chip can be selected using optional <val> variable, which can be specified to "0" (default) or to "1". If <val> is not specified, the default chip will be used. The USB-to-SPI conversion mode gives an external host/PC direct access to the SPI bus of DW1000 (one from two ICs). Can be used for testing of DW1000 ICs and their RF performance.

3.2.3.3 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 PDoA node 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	Definition of functionality
ADDR <val>	Set the PDoA node's short address to decimal <val>. The default is 1 (which is decimal for PDoA node's address 0x0001).
PANID <val>	Set the PDoA node's PAN ID to decimal <val>. The default is 57034 (which is decimal for 0xDECA).
NUMSLOT <val>	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 4.1.1.
SLOTPER <val>	Set the slot's window period to decimal <val>, milliseconds (ms). The 5 ms slot window is used in the system.

Command	Definition of functionality
SFPER <val>	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.
REPDEL <val>	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>	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	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>	"0" (default) - Disables the UART. The PDoA node will not output data to UART. "1" - Enables the UART. This enables the PDoA 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).
AUTO <val>	Used to start default Node top-level application automatically after Reset/Power on. May be configured to "0" to stay in STOP mode of operation. By default is configured to "1", which after reset leads to start the Node top-level application with first DW1000 chip in a "master" role. Also can be configured to "2", which will enable another DW1000 chip to act as a "master" in the Node top-level application.
ANTTXA <val>	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 currently configured "master" chip.
ANTRXA <val>	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 currently configured "master" chip.
PDOFF <val>	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.

Command	Definition of functionality
RNGOFF <val>	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.
PCREP <val>	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	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.

3.2.3.4 The PC to the PDoA Node application commands

When PDoA 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 PDoA 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 3.2.3.5.

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,

Command	Definition of functionality
	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 PDoA 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 PDoA 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.
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”.

3.2.3.5 The Node top-level application output

The PDoA 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":"PDoA Node", "Version":"1.0.0", "Build":"Sep 18 2017 14:06:47", "Driver":"DW1000 Device Driver Version 04.00.07"}}
PDoA 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 IMU task is running, it may report the “SN” object anytime node is moving	SN	Node’s Stationary Object	JSxxxx{"SN": <stationary_obj>} Where <stationary_obj> is: {"a16":<string>,"V":<int>,"X":<int>,"Y":<int>,"Z":<int>} **"a16"** - is node’s short address, hex **"V"** - is a service data wrt to the node, bitfields, dec: bit 0 indicates 1 if node is stationary and 0 if node is moving. **"X"** - is the node’s accelerometer X axis data, in milli-G, dec **"Y"** - is the node’s accelerometer Y axis data, in milli-G, dec **"Z"** - is the node’s accelerometer Z axis data, in milli-G, dec *Example:* JS0034{"SN": {"a16":"0001","V":0,"X":-15,"Y":-5,"Z":-985}}
When the PDoA 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 hundreds of ppm (float as int), dec **"V"** - is a service data wrt to the tag, bitfields, dec:

Action	JSON object	Type	Format of JSON object with TLV wrapper
			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}}

3.2.4 Flush task

Any functions, including ISR functions, or RTOS tasks can produce and request to send data to the terminal 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 on the Figure 6 below.

The *Flush* task is coded as FlushTask() in the task_flush.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 REPORT_BUFSIZE, which is defined to 0x2000 bytes. 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 6 below.

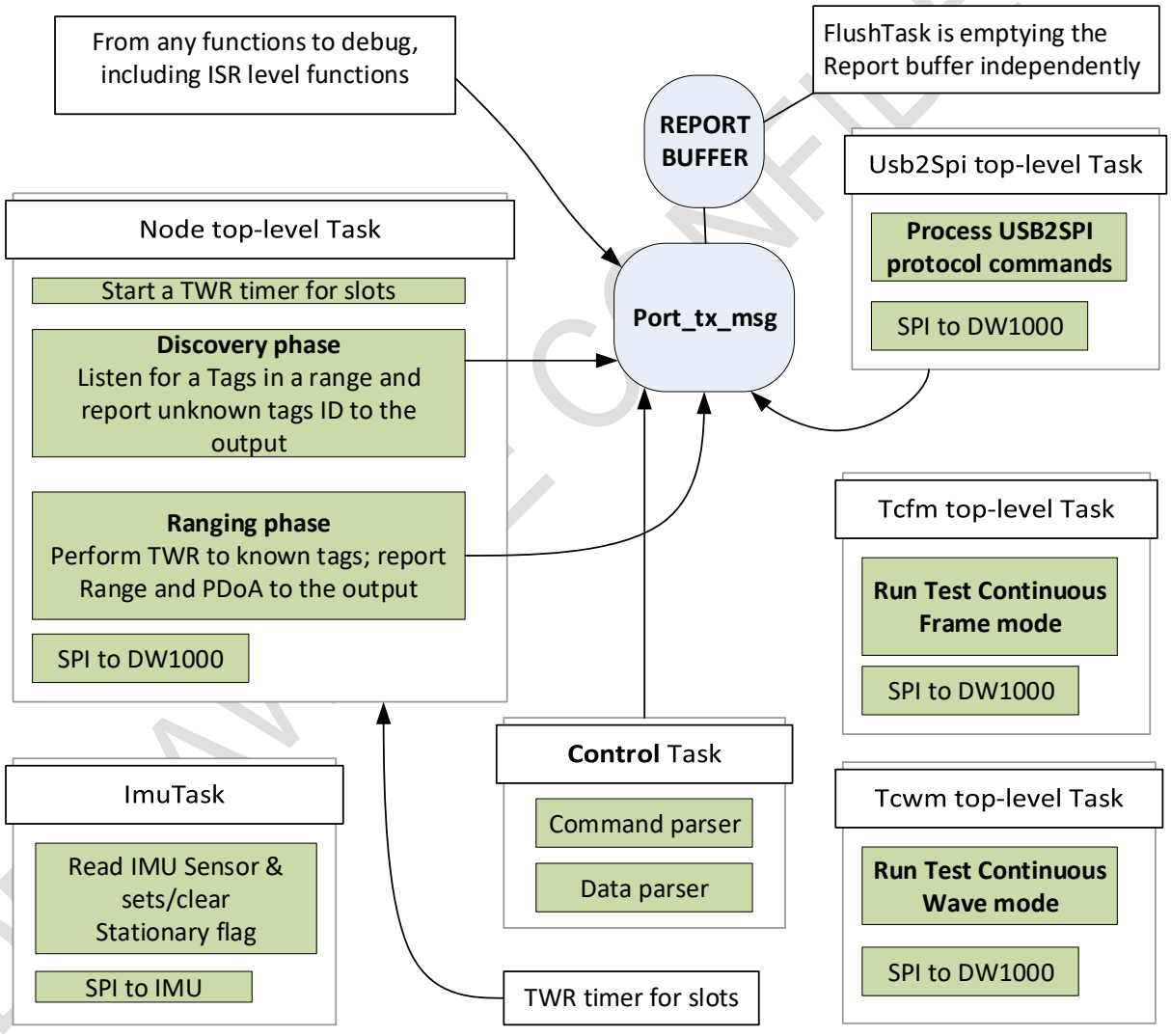


Figure 6 Output data using shared Report buffer

3.3 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 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. 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 & corresponded 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  imuTask;      /* Tag/Node */
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 */
```

4 TOP-LEVEL APPLICATIONS

There are four top-level applications, listed in the Table 6, which can run in the dedicated mode on the node 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 4.1	NODE <val>	Main PDoA two-way ranging Node top-level application, which use <val> to specify the chip, acting as a "Master" in node application.
Usb2Spi, section 4.2	USPI <val>	USB (or UART) to SPI converter to <val> chip. Used for testing.
TCFM, section 4.3	TCFM <val>	Test Continuous Frame transmission to <val> chip. Used for testing.
TCWM, section 4.3	TCWM <val>	Test Continuous Wave transmission to <val> chip. Used for testing

4.1 PDoA Node top-level application

On startup of PDoA node, it will setup and execute the two-way-ranging *Node* top-level application, if this is configured in `pConfig->s.auto` parameter.

As described in section 1.1, the PDoA node hardware employs two DW1000 IC. The same physical clock is provided to both DW1000 to enable their receivers to operate using the synchronous timings. One of the DW1000 is used in the role of "*Master*" where it is in the reception state most of the time. The *"Master"* chip is also used to transmit, as "Responder" in the Two-Way-Ranging process, see 6.1. The other DW1000, the *"Slave"*, is in the reception state only for the reception of the *Final* message in TWR exchange process, otherwise it is in the idle state.

The PDoA node supports more than one tag ranging to it. 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 PDoA node, see 4.1.3.

Initially each tag sends only blink messages to advertise itself and be discovered by the PDoA node. For the tag, see [3], this is called the *Discovery* phase. After sending the blink, the tag awaits a *Ranging Config* response from the PDoA node, and upon its successful reception, the tag configures itself, as instructed by the *Ranging Config* response, to range to the PDoA node in a designated slot.

The PDoA 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 PDoA node.

When the PDoA 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 using the "ADDTAG" command. When a blink is received from a tag that is in the KList, the PDoA 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 PDoA node configuration also saves the current *KList*. The "DELTAG" command, can be used to remove a 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 PDoA node address, specified in the *Ranging Config* message), awaits of a *Response* message from the PDoA node, and upon its successful reception, replies with a *Final* message to complete the ranging sequence.

For reception of the *Final* message from the tag, the PDoA node configures both *"Master"* and *"Slave"* chips for a reception. On successful reception of the *Final*, the node reads data from both chips, calculates distance and phase difference between received *Final* messages, and reports result to the output. Figure 7 below shows a high-level view to the *Node* application flow and more detailed description is giving in the section 4.1.2.

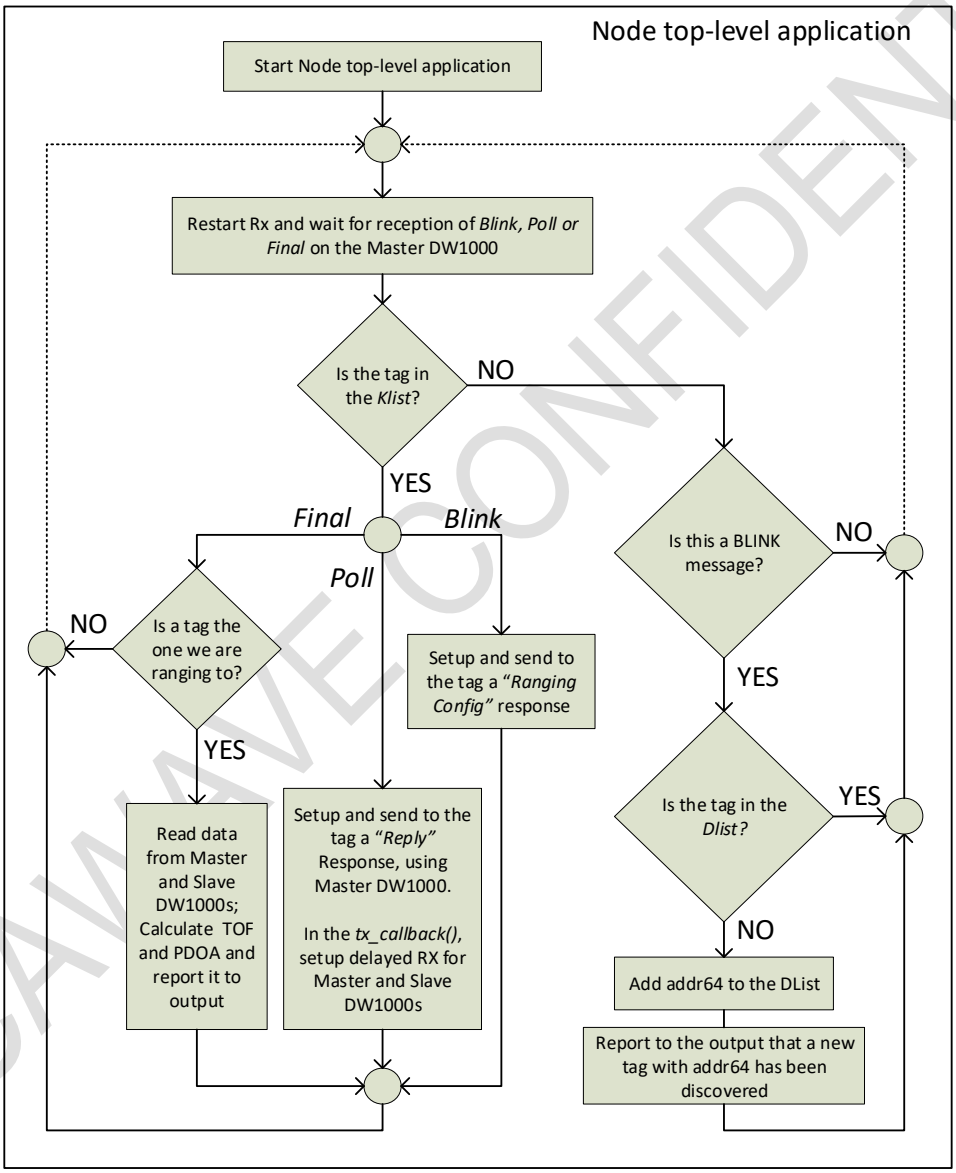


Figure 7 *Node* top-level application

4.1.1 Concept of Discovered and Known tags lists

Before the PDoA 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 that the PDoA node sends in response to receiving 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, are used to add and remove tag information to/from the *KList*. The *“*GETKLIST*”* command is used to retrieve the *KList* information, e.g. by the PC GUI application so that it can update the list of known tags in the system on its start up.

When the PDoA 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 PDoA 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)
```

4.1.2 Discovery and ranging to tags

There is a set of RTOS tasks (threads) to implement the PDoA *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 DW1000s chips to run with configured UWB parameters and starts following sub-tasks: *RxTask* and *CalckTask*.

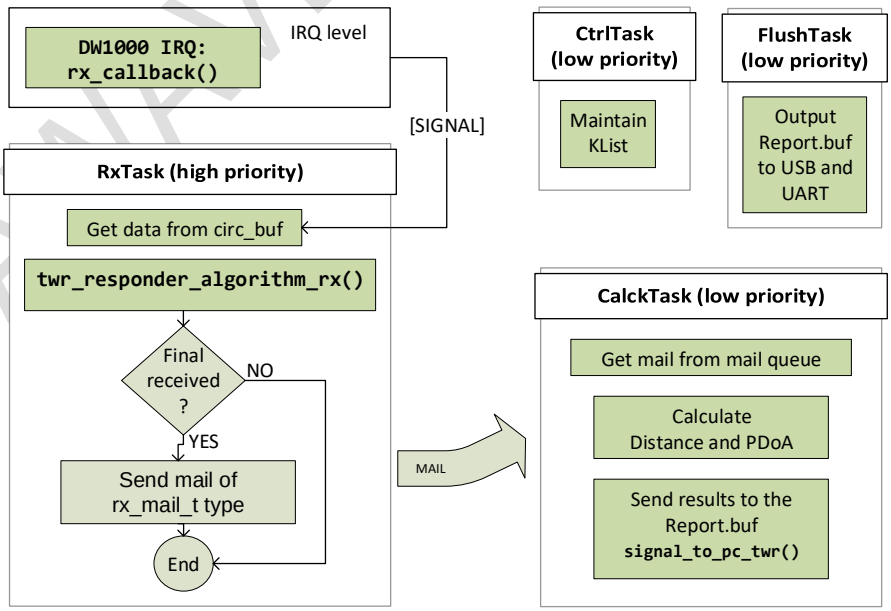


Figure 8 Tasks used in the PDoA Node application

Please note, the core tasks, i.e. *Control*, for input handling, and *Flush,* for output handling, are always running, see sections: 2.1.2, 3.2.3, 3.2.4.

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 expect that it will start ranging to the PDoA 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 PDoA node begins range to that particular tag, also node send 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 4.1.3 and 6.5.

On reception of *Final* message from the tag, the Node's *rxTask* sends the mail using mailbox RTOS mechanism to the lower priority *calcTask*, which will calculate and report the estimated distance, PDoA and X-Y coordinates of the tag with respect to the Node.

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

To ensure non-overlapping ranging exchanges for multiple tags, the PDoA node uses Time-Division Multiple Access method (TDMA), to assign to every tag its own dedicated slot, of T_Slot duration, within the PDoA node's superframe period, see Figure 9.

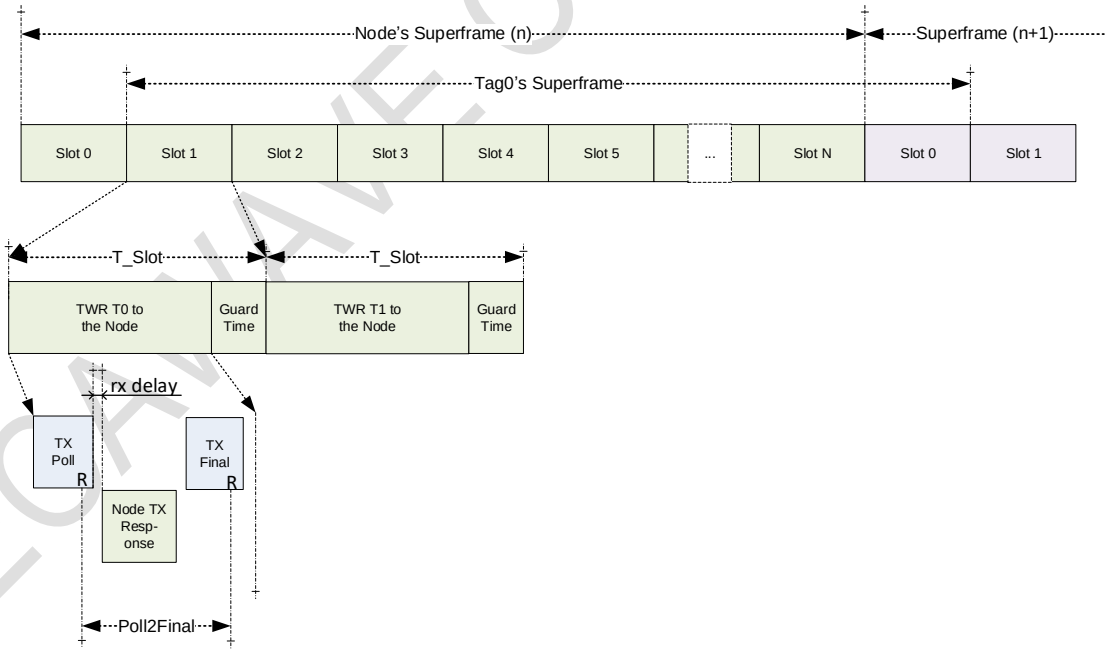


Figure 9: Superframe structure and ranging exchange time profile

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 6.1, 6.2, 6.3, [4].

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 reseption of a Response from a Node, see P2FDEL and RCDEL parameters in Table 3 above.

Note, the slot number zero is reserved and may 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 PDoA node specifies the superframe period in the *Ranging Config* message which the tag saves in its `framePeriod_ms` variable The PDoA 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 PDoA 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 6.5.

4.2 Usb2Spi top-level application

When the *Control* task receives the command "*USPI <val>"*, see Table 2, it sets the *Ev_Usb2Spi_A_Task* or *Ev_Usb2Spi_B_Task* to the xStartTaskEvent. The *Default* task consumes the event, safely ends all running tasks and starts *Usb2Spi* top-level application. There are two DW1000 chip on the PDoA node hardware. The optional <val> parameter can be empty or "0", or be set to "1", and specifies the chip to run the Usb2Spi application.

Once *Usb2Spi* top-level application has started, the *Control* task is switched to the "Data parser" mode, see 0, where it passes the whole incoming USB/UART stream into the *Usb2Spi* task, which has the implementation of the Usb2Spi protocol to control the DW1000 from an external application. This illustrated on the Figure 10 below.

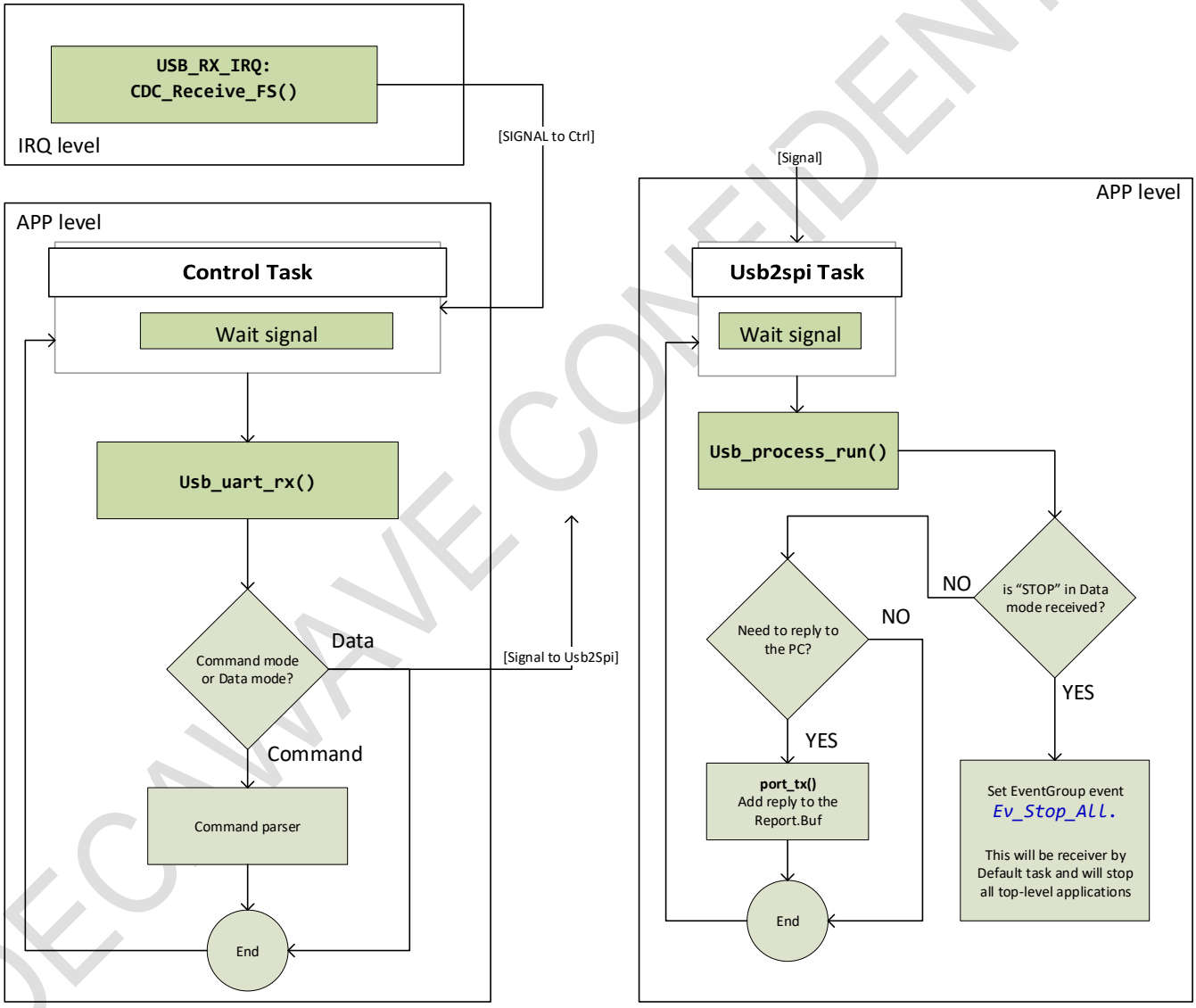


Figure 10 Usb2Spi top-level application

4.3 TCFM and TCWM top-level applications

These tasks configure one of the DW1000, indicated in <val> field, to run in the particular operational test mode (Test Continuous Wave or Test Continuous Frame) and awaiting to be stopped from *Control* task. Both of them are an interface to the corresponding bare-metal production test functions, located in files *tcfm.c* and *tcwm.c* respectively.

4.4 Low power mode

The PDoA node does not use a low power mode for DW1000's when it is not actively ranging. This is because the PDoA system is currently designed to keep one "Master" receiver on all the time to allow for the discovery of new tags.

It would be possible to modify the system to include power save features and put DW1000 to sleep for the periods of time when there no tag ranging is expected. This is out of the scope of current document.

4.5 IMU task

The IMU task, see Figure 11, is started by *node_helper()* function and it uses a third SPI (i.e. different to others, used to talk to the DW1000 ICs) to read the IMU sensor's state periodically and determine whether the PDoA node is moving or stationary. This task does not conflict with any top-level applications and can run at the same time as any of them, if needed. This illustrated on the Figure 11.

If the PCREP configuration parameter is set to "1", the PDoA node is reporting to the output the "SN" JSON object, indicating whether the device is stationary or moving as well as accelerometer normalized values per all 3 axis. The accelerometer readings are normalized to 1000 milli-g per axis. If PDoA node is not moving, it sends last report, indicated PDoA node is stationary and after that it does not send reports to the output unless the IMU task will detect movement again.

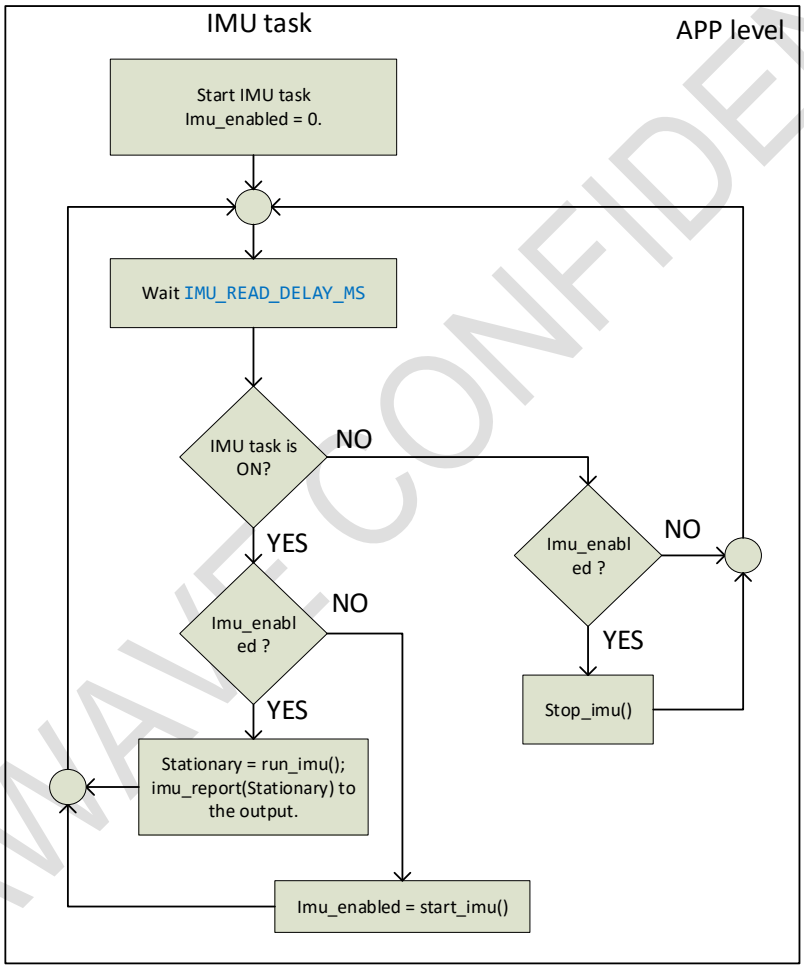


Figure 11 IMU task

5 BUILDING AND RUNNING THE CODE

5.1 Building the code

This project is created using the Segger Embedded Studio IDE with Nordic SDK16.0.0. The Segger IDE includes a full license to develop, debug and run applications targeting nRF52832 ARM MCU, which is the MCU used onto the Decawave's DWM1001 hardware platform.

5.2 Installing of the Segger IDE

Download and install the Segger Embedded Studio IDE from the following web site:
https://www.segger.com/products/development-tools/embedded-studio/

5.3 Loading of the project to the IDE

Unpack the source code to the, for example, "dw_pdoa_tag" folder. In the Segger IDE, choose File->Open Solution and select the project dw_pdoa_tag.emProject, located in the examples\dw_pdoa_tag\ folder.

5.4 Connecting, Building and Running of the application

Connect the board to the PC. You may require to install J-Link drivers, which could be found on the Segger web site:

https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack

Install J-Link Software and Documentation pack, which includes drivers and J-Flash Lite tool needed for reprogramming new FW binary into tag.

To check if the target board is working select Target->Connect J-Link from the menu. To start the TDOA Tag application select Build->Build and Run from menu.

5.5 Get a license for Nordic MCU

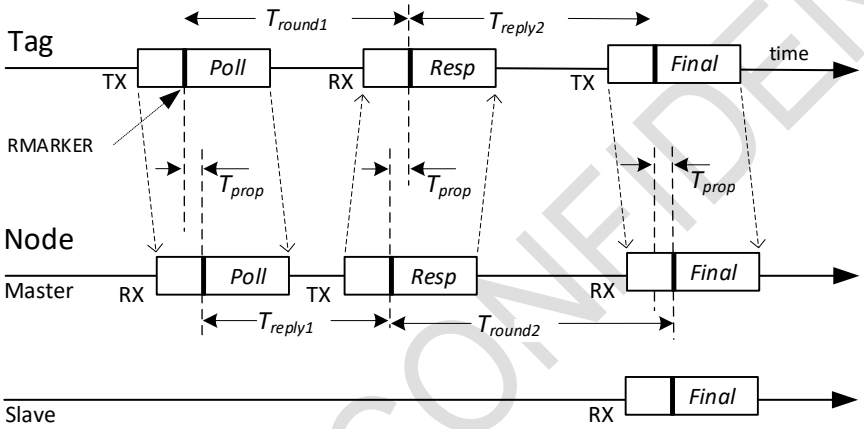
If it was the first time you start your app you shall be asking for the licence. License is free for Nordic MCU'S, please refer to the Appendix A – Installing a license for Nordic MCU for details.

6 APPENDIX A

6.1 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 DW1000. These transmit and receive timestamps are used to work out a round trip delay and calculate the range, see [4].

In the ranging scheme shown in Figure 12 below, the tag sends a *Poll* message which is received by the PDoA node. The PDoA 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's Master, 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 12: Distance calculation in TWR

Since the PDoA node also calculates the phase difference on the reception of the *Final* message between Master and Slave chip, 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.

6.2 UWB configuration and TWR timing profile used in the PDoA system

The PDoA 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 PDoA system

UWB Config	Channel	Data Rate	Preamble Length	PRF	Preamble Code	SFD	PHR mode	PAC	Smart Tx power
Value	5	6.81 Mbit/s	128	64 MHz	9	Non-Standard	Standard	8	Disabled

In the PDoA TWR timing profile, see Figure 13, 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 6.1. 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 PDoA node, see Figure 13.

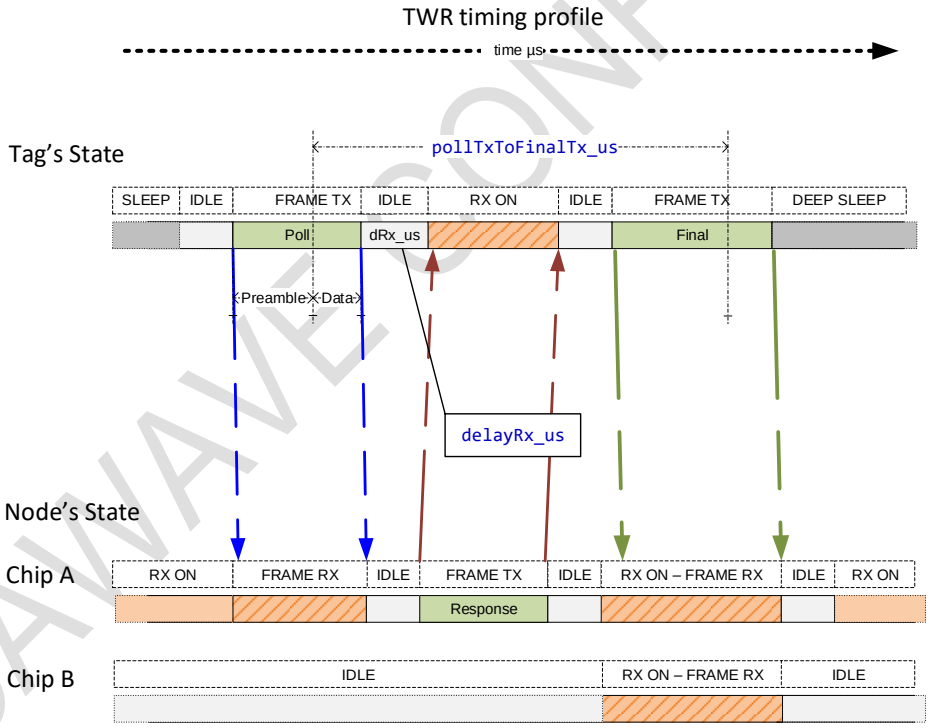


Figure 13: PDoA TWR timing profile

6.3 Frame time adjustments

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

The significant event making the TX and RX times is specified in IEEE 802.15.4 [4] 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 DW1000, as long as the TX and RX antenna delays are configured.

The PDoA 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 is the measured distance. This can be re-programmed in the *FConfig* by using a corresponding control command interface, see 3.2.3.3.

6.4 UWB messages, used in the PDoA TWR

There are following messages addressing modes employed in the PDoA system: the *Blink* message uses long (64-bits) addressing mode, the *Ranging Config* message uses short-long (16 to 64-bits) addressing mode, and the *Poll*, the *Response*, and the *Final* messages are uses short-short (16 to 16-bit) addressing 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.

6.4.1 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 14 below.

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

Figure 14 Encoding of Tag's 12-bytes blink message

6.4.2 Ranging Config message

During initial blinking the tag has only long 64-bit address. To maintain shortest timings, the PDoA node assign 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 15 below.

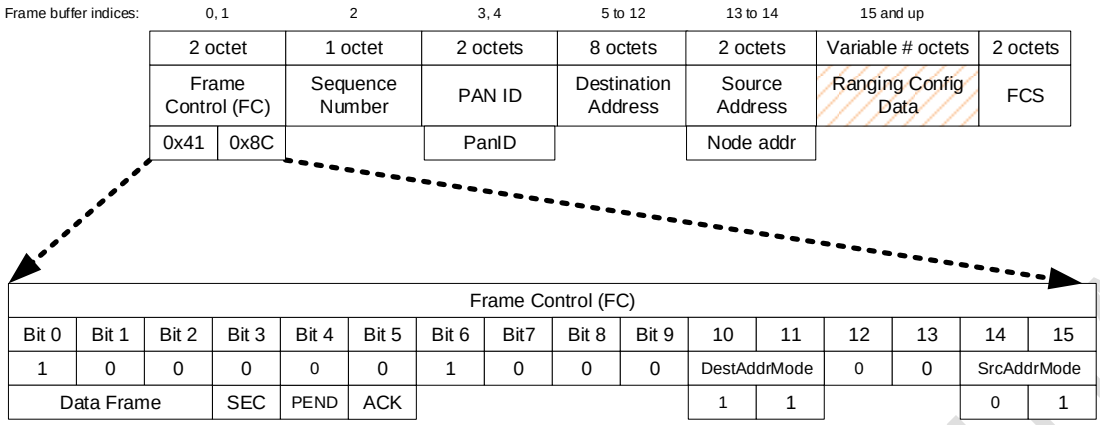


Figure 15 Frame format of Ranging Config message

The PAN ID of the PDoA system and the PDoA 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: PDoA 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.

6.4.3 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 16 below.

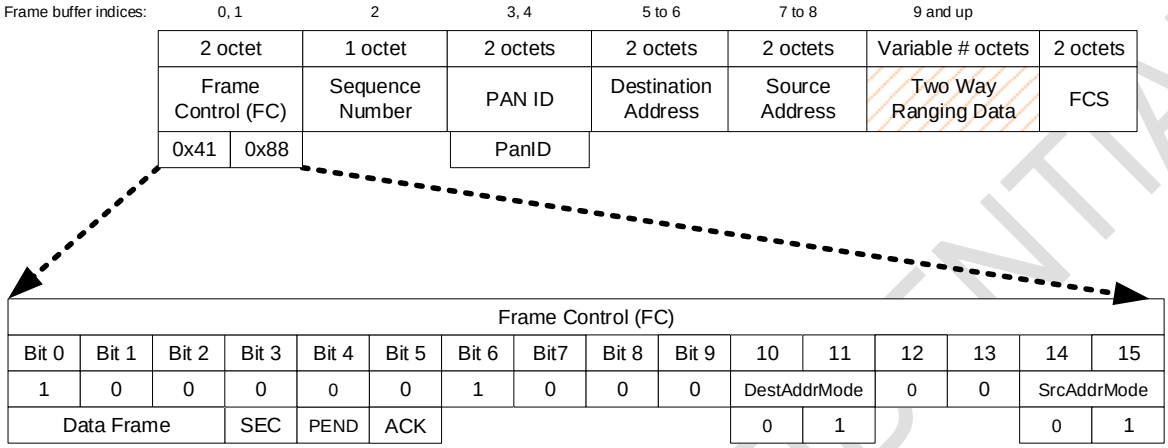


Figure 16 Frame format used for Ranging

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 PDoA TWR exchange

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

6.4.3.1 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, on each range this number is incremented (by modulo 256).

6.4.3.2 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 is 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.

6.4.3.3 Final message

The *Final* message is a structure of type final_msg_t and it sent by the tag after receiving the PDoA 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 PDoA node, i.e. the time the tag received the *Response* frame from the PDoA 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, where bit0, whether "0" indicates the Tag is moving and "1" when it is stationary.
acc_x	2	IMU data, accelerometer, normalized value for X-axis, in milli-G
acc_y	2	IMU data, accelerometer, normalized value for Y-axis, in milli-G
acc_z	2	IMU data, accelerometer, normalized value for Z-axis, in milli-G

6.5 Slot Time correction method

The PDoA node and tag both use RTC timers to implement the slotted TDMA access method. The PDoA 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 PDoA node's time domain (the value is close to the PDoA 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 PDoA node, the tag receives the `slotCorr_us`, which specifies the time where the PDoA node has expected the reception of the tag's *Poll* with respect to the start of the node's superframe (gRtcNode), see Figure 17. Upon reception of the Response, the `nextWakeUpPeriod_ns` is calculated as follows:

```
nextWakeUpPeriod_ns  = 1e6* sframePeriod_ms;
nextWakeUpPeriod_ns -= WKUP_RESOLUTION_NS * (rtcNow - gRtcSFrameZeroCnt );
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.

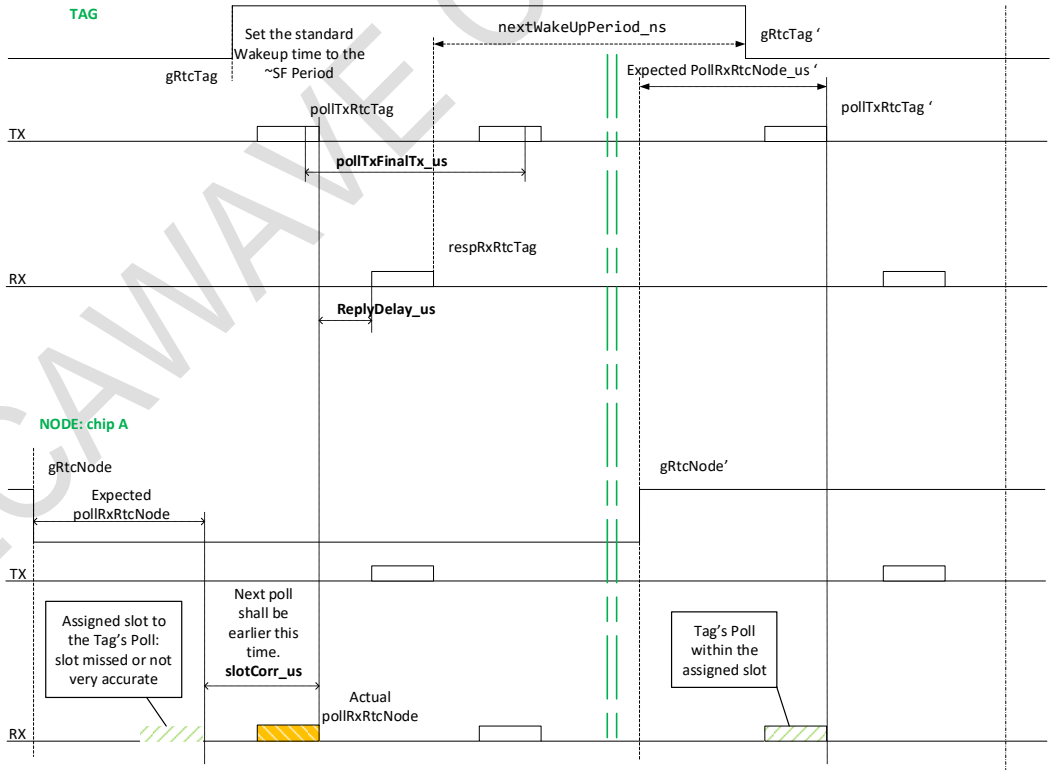


Figure 17 Node-Tag Slot Time correction method

6.6 The application architecture in the flowchart

The flowcharts on Figure 18 and Figure 19 shows the interaction of the software blocks with each other and shows where the main functionality of the node can be found in the sources.

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

1. Initialization of the hardware and peripherals. This includes initialization of MCU peripherals (like UART, SPI, I2C etc), initialization of DW1000 chips, loading configuration data from NVM into RAM.
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.
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 2.2 and section 6.7.

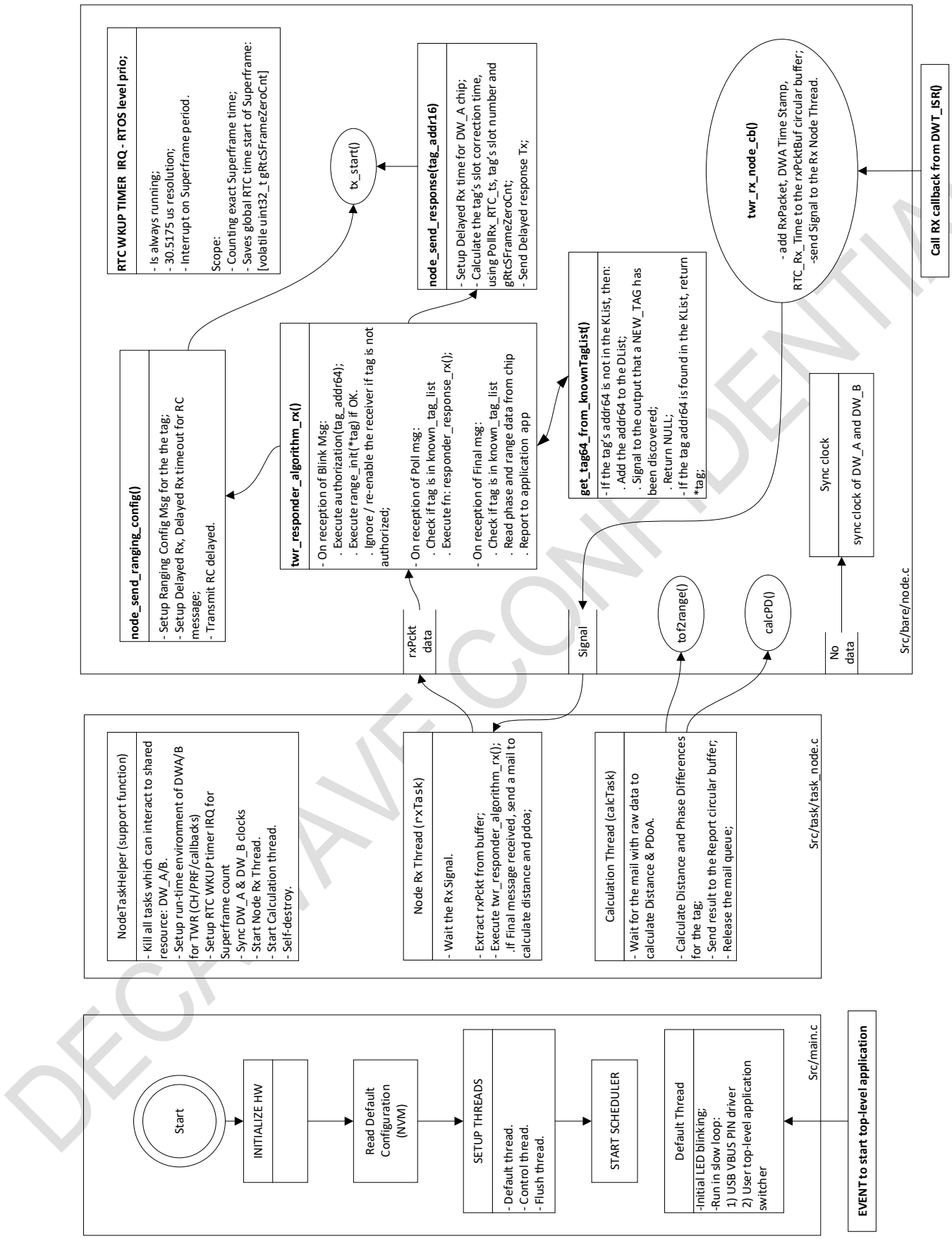


Figure 18 Application flowchart part 1

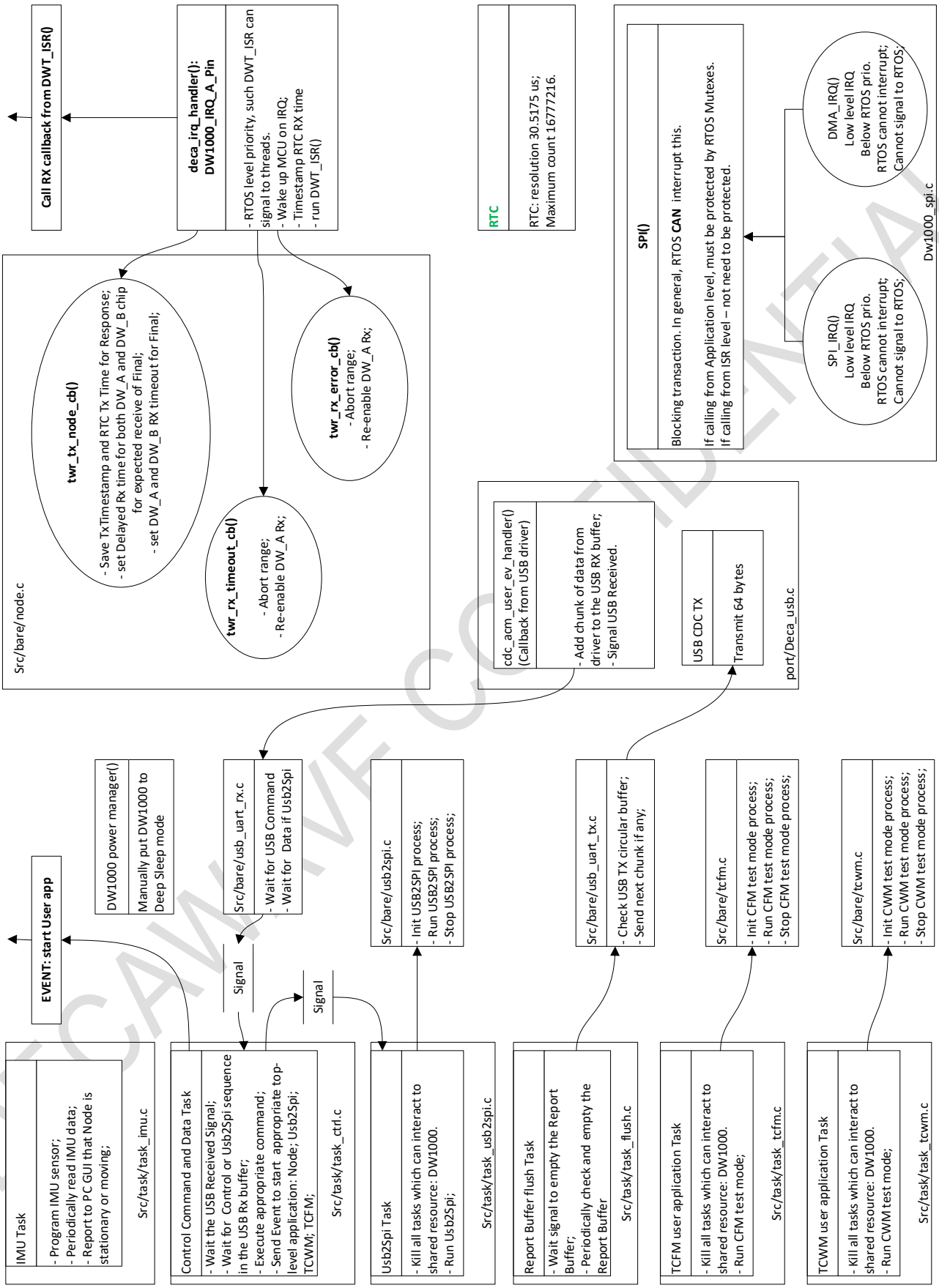


Figure 19 Application flowchart part 2

6.7 List of source code files

Table 13 gives a list of files (excluding nRF SDK files) that make up the source code of the Node ARM application.

Table 13 List of core files and directories in the node ARM application

Filename	Brief description
decadriver	
deca_device.c	DW1000 device library API functions – source code
deca_device_api.h	DW1000 device library API interface header
deca_params_init.c	Device library configuration parameters tables
deca_param_types.h	Device library configuration parameters interface header
deca_regs.h	DW1000 Register Definitions
deca_types.h	Device library types definition
deca_version.h	DW1000 device library version
10_dof_driver	10 degrees of freedom IMU drivers
10_dof_driver\lis2mdl	3DoF magnetometer/compass
lis2mdl.c	
lis2mdl.h	
10_dof_driver\lps22hb	Barometer
lps22hb.c	
lps22hb.h	
lps22hb_reg.h	
10_dof_driver\lsm6dsl	3DoF accelerometer + 3DoF gyro
lsm6dsl.c	
lsm6dsl.h	
bsp	
boards.h	
pca10056.h	Definitions for the board – pins, LEDs, etc.
dw_pdoa_node	
dw_pdoa_node.emProject	Project file
dw_pdoa_node.emSession	
flash_placement.xml	Linker config file (stack, heap, code, data, etc.)
dw_pdoa_node\Src	
freertos.c	application-interface to the FreeRTOS
main.c	Entry point to the application. Has the *Default* task in the StartDefaultTask()
dw_pdoa_node_common.c	
dw_pdoa_node\Src\bare	
cmd.c	Command parser
cmd_fn.c	Command parser's functions for execution
config.c	Interface to NVM and .bss configurations
json_2pc.c	
tag_list.c	
node.c	Collection of bare-metal functions to implement Node application.
tcfm.c	Collection of bare-metal functions to implement TCFM application.

Filename	Brief description
tcwm.c	Collection of bare-metal functions to implement TCWM application.
usb_uart_rx.c	Bare-metal receive from USB/UART
usb_uart_tx.c	Bare-metal transmit via to USB/UART
usb2spi.c	
dw_pdoa_node\Src\Inc	
circ_buf.h	Macros for circular buffer
cmd.h	Header for cmd.c
cmd_fn.h	-
config.h	-
dw_pdoa_node_common.h	-
default_config.h	Default configuration header. All default parameters are here.
error.h	Error codes
msg_time.h	-
node.h	Header for node.c
task_list.h	Header for task_list.c
task_flush.h	-
task_imu.h	-
task_node.h	-
task_tcfm.h	-
task_tcwm.h	-
task_usb2spi.h	-
tcfm.h	-
tcwm.h	-
translate.h	-
usb2spi.h	-
usb_uart_rx.h	-
usb_uart_tx.h	-
util.h	-
uwb_frames.h	Defines the common frames structures used in Tag and Node
version.h	Version of the application
dw_pdoa_node\Src\port	
deca_uart.c	Platform-specific UART functions
deca_usb.c	Platform-specific USB functions
dw1000_rbct.c	DW1000 range correction tables
port_platform.c	Other platform-specific functions
dw_pdoa_node\Src\port\Inc	
port_platform.h	-
dw_pdoa_node\Src\task	
task_ctrl.c	Core task: Command Control and Data task
task_flush.c	Core task: Flush Report Buffer
task_imu.c	Service task for IMU sensor, sets stationary flag
task_node.c	Collection of top-level application tasks for Node
task_tcfm.c	top-level application task: TCFM
task_tcwm.c	top-level application task: TCFM
task_usb2spi.c	top-level application task: USB2SPI
dw_pdoa_node\Src\utils	

Filename	Brief description
msg_time.c	Calculates frames durations
translate.c	Translates some parameters to human-readable format and back
util.c	Collection of helpful utilities
dw_pdoa_node\Output	Build files for the project
dw_pdoa_node\config	
sdk_config.h	Configuration of peripherals used by SDK and application
SDK	nRF SDK source files

7 BIBLIOGRAPHY

Ref	Author	Title
[1]	Decawave	DW1000 Data Sheet
[2]	Decawave	DW1000 User Manual
[3]	Decawave	PDoA Tag Source Code Guide
[4]	IEEE	IEEE 802.15.4-2011 or "IEEE Std 802.15.4™-2011" (Revision of IEEE Std 802.15.4-2006). IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Computer Society Sponsored by the LAN/MAN Standards Committee. Available from http://standards.ieee.org/

8 DOCUMENT HISTORY

Table 14: Document History

Revision	Date	Description
1.2	2nd of April 2020	Firmware 5.0.0
1.1	27th of August 2019	Firmware 4.2.0
1.0	19th of October 2018	First release

9 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.

4