|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | | | **Protokół konstrukcyjny układu wykonawczego-software** | | | | | | Data wystawienia: | |
|  | 27/12/2021 | |
|  | Doc# | 1/CZB/006 |
|  | Nr wniosku NCBR: | | POIR.01.01.01-00-0196/19 | | | Nazwa projektu: | | Smart Yacht |
|  | Rozpoczęcie testów: | | 01-12-2021 | | Zakończenie testów: | | 27-12-2021 | |

#### Cel programu

Celem jest napisanie software obsługującego inicjator(opaskę na rękę) oraz responder(układ stacjonarny na łodzi), oraz wyznaczającego dystans metodą ToF kóry następnie zostanie wysłany magistralą CAN do mastera.

Docelowo ma on obsługiwać 6 responderów wymaganych do wykonania trilateracji 6 punktowej, oraz 9 inicjatorów do śledzenia 9 członków załogi.

#### **Program respondera**

/\* USER CODE BEGIN Header \*/

/\* USER CODE END Header \*/

/\* Includes ------------------------------------------------------------------\*/

**#include** "main.h"

/\* Private includes ----------------------------------------------------------\*/

/\* USER CODE BEGIN Includes \*/

//main ex

**#include** "platform/port.h"

app\_t app;

//end of main ex

**#include** <string.h>

**#include** "decadriver/deca\_device\_api.h"

**#include** "decadriver/deca\_regs.h"

**#include** "platform/stdio.h"

**#include** "platform/deca\_spi.h"

**#include** "platform/port.h"

/\* USER CODE END Includes \*/

/\* Private typedef -----------------------------------------------------------\*/

/\* USER CODE BEGIN PTD \*/

/\* USER CODE END PTD \*/

/\* Private define ------------------------------------------------------------\*/

/\* USER CODE BEGIN PD \*/

/\* USER CODE END PD \*/

/\* Private macro -------------------------------------------------------------\*/

/\* USER CODE BEGIN PM \*/

/\* USER CODE END PM \*/

/\* Private variables ---------------------------------------------------------\*/

ADC\_HandleTypeDef hadc1;

CAN\_HandleTypeDef hcan1;

SPI\_HandleTypeDef hspi1;

DMA\_HandleTypeDef hdma\_spi1\_rx;

DMA\_HandleTypeDef hdma\_spi1\_tx;

TIM\_HandleTypeDef htim2;

UART\_HandleTypeDef huart2;

/\* USER CODE BEGIN PV \*/

/\* USER CODE END PV \*/

/\* Private function prototypes -----------------------------------------------\*/

**void** SystemClock\_Config(**void**);

**static** **void** MX\_GPIO\_Init(**void**);

**static** **void** MX\_SPI1\_Init(**void**);

**static** **void** MX\_USART2\_UART\_Init(**void**);

**static** **void** MX\_ADC1\_Init(**void**);

**static** **void** MX\_CAN1\_Init(**void**);

**static** **void** MX\_DMA\_Init(**void**);

**static** **void** MX\_TIM2\_Init(**void**);

/\* USER CODE BEGIN PFP \*/

/\* USER CODE END PFP \*/

/\* Private user code ---------------------------------------------------------\*/

/\* USER CODE BEGIN 0 \*/

/\* Example application name and version to display. \*/

**#define** APP\_NAME "DS TWR RESP v1.2\r\n"

/\* Default communication configuration. We use here EVK1000's default mode (mode 3). \*/

**static** dwt\_config\_t config = { 2, /\* Channel number. \*/

DWT\_PRF\_64M, /\* Pulse repetition frequency. \*/

DWT\_PLEN\_1024, /\* Preamble length. Used in TX only. \*/

DWT\_PAC32, /\* Preamble acquisition chunk size. Used in RX only. \*/

9, /\* TX preamble code. Used in TX only. \*/

9, /\* RX preamble code. Used in RX only. \*/

1, /\* 0 to use standard SFD, 1 to use non-standard SFD. \*/

DWT\_BR\_110K, /\* Data rate. \*/

DWT\_PHRMODE\_STD, /\* PHY header mode. \*/

(1025 + 64 - 32) /\* SFD timeout (preamble length + 1 + SFD length - PAC size). Used in RX only. \*/

};

/\* Default antenna delay values for 64 MHz PRF. See NOTE 1 below. \*/

**#define** TX\_ANT\_DLY 16505

**#define** RX\_ANT\_DLY 16505

//#define TX\_ANT\_DLY 65500

//#define RX\_ANT\_DLY 65500

/\* Frames used in the ranging process. See NOTE 2 below. \*/

**static** uint8 rx\_poll\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, 'C', 'Z', '0', '0',

0x21, 0, 0 };

**static** uint8 tx\_resp\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, '0', '0', 'Z', 'C',

0x10, 0x02, 0, 0, 0, 0 };

**static** uint8 rx\_final\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, 'C', 'Z', '0', '0',

0x23, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

uint8\_t responderNumber = '1';

uint8\_t actualInitiatorNumber = '0';

/\* Length of the common part of the message (up to and including the function code, see NOTE 2 below). \*/

**#define** ALL\_MSG\_COMMON\_LEN 10

/\* Index to access some of the fields in the frames involved in the process. \*/

**#define** ALL\_MSG\_SN\_IDX 2

**#define** FINAL\_MSG\_POLL\_TX\_TS\_IDX 10

**#define** FINAL\_MSG\_RESP\_RX\_TS\_IDX 14

**#define** FINAL\_MSG\_FINAL\_TX\_TS\_IDX 18

**#define** FINAL\_MSG\_TS\_LEN 4

/\* Frame sequence number, incremented after each transmission. \*/

**static** uint8 frame\_seq\_nb = 0;

/\* Buffer to store received messages.

\* Its size is adjusted to longest frame that this example code is supposed to handle. \*/

**#define** RX\_BUF\_LEN 24

//#define RX\_BUF\_LEN 36//changed

**static** uint8 rx\_buffer[RX\_BUF\_LEN];

/\* Hold copy of status register state here for reference so that it can be examined at a debug breakpoint. \*/

**static** uint32 status\_reg = 0;

/\* UWB microsecond (uus) to device time unit (dtu, around 15.65 ps) conversion factor.

\* 1 uus = 512 / 499.2 µs and 1 µs = 499.2 \* 128 dtu. \*/

//#define UUS\_TO\_DWT\_TIME 65536

**#define** UUS\_TO\_DWT\_TIME 158875//changed

/\* Delay between frames, in UWB microseconds. See NOTE 4 below. \*/

/\* This is the delay from Frame RX timestamp to TX reply timestamp used for calculating/setting the DW1000's delayed TX function. This includes the

\* frame length of approximately 2.46 ms with above configuration. \*/

**#define** POLL\_RX\_TO\_RESP\_TX\_DLY\_UUS 2750

//#define POLL\_RX\_TO\_RESP\_TX\_DLY\_UUS 10000//changed

/\* This is the delay from the end of the frame transmission to the enable of the receiver, as programmed for the DW1000's wait for response feature. \*/

**#define** RESP\_TX\_TO\_FINAL\_RX\_DLY\_UUS 500

//#define RESP\_TX\_TO\_FINAL\_RX\_DLY\_UUS 2000//changed

/\* Receive final timeout. See NOTE 5 below. \*/

//#define FINAL\_RX\_TIMEOUT\_UUS 3300

**#define** FINAL\_RX\_TIMEOUT\_UUS 10000

//#define FINAL\_RX\_TIMEOUT\_UUS 7550//changed

/\* Preamble timeout, in multiple of PAC size. See NOTE 6 below. \*/

//#define PRE\_TIMEOUT 8

**#define** PRE\_TIMEOUT 0//changed

/\* Timestamps of frames transmission/reception.

\* As they are 40-bit wide, we need to define a 64-bit int type to handle them. \*/

**typedef** **signed** **long** **long** int64;

**typedef** **unsigned** **long** **long** uint64;

**static** uint64 poll\_rx\_ts;

**static** uint64 resp\_tx\_ts;

**static** uint64 final\_rx\_ts;

/\* Speed of light in air, in metres per second. \*/

**#define** SPEED\_OF\_LIGHT 299702547

/\* Hold copies of computed time of flight and distance here for reference so that it can be examined at a debug breakpoint. \*/

**static** **double** tof;

**static** **double** distance;

//test

//CAN communication

//create can buffer and headers

CAN\_TxHeaderTypeDef TxHeader;

CAN\_RxHeaderTypeDef RxHeader;

uint32\_t TxMailbox;

uint8\_t RxData[2];

uint8\_t TxData[3];

//anonymous union to store distance to send via CAN

**union** {

uint8\_t array[2];

int16\_t integer;

} distanceCm;

/\* String used to display measured distance on UART. \*/

**char** dist\_str[20] = { 0 };

/\* Declaration of static functions. \*/

**static** uint64 get\_tx\_timestamp\_u64(**void**);

**static** uint64 get\_rx\_timestamp\_u64(**void**);

**static** **void** final\_msg\_get\_ts(**const** uint8 \*ts\_field, uint32 \*ts);

/\* USER CODE END 0 \*/

/\*\*

\* @brief The application entry point.

\* @retval int

\*/

**int** main(**void**) {

/\* USER CODE BEGIN 1 \*/

/\* USER CODE END 1 \*/

/\* MCU Configuration--------------------------------------------------------\*/

/\* Reset of all peripherals, Initializes the Flash interface and the Systick. \*/

HAL\_Init();

/\* USER CODE BEGIN Init \*/

/\* USER CODE END Init \*/

/\* Configure the system clock \*/

SystemClock\_Config();

/\* USER CODE BEGIN SysInit \*/

/\* USER CODE END SysInit \*/

/\* Initialize all configured peripherals \*/

MX\_GPIO\_Init();

MX\_SPI1\_Init();

MX\_USART2\_UART\_Init();

MX\_ADC1\_Init();

MX\_CAN1\_Init();

MX\_DMA\_Init();

MX\_TIM2\_Init();

/\* USER CODE BEGIN 2 \*/

//turn on watchdog

// HAL\_IWDG\_Init(&hiwdg);

//give responder its own number

rx\_final\_msg[7] = responderNumber;

rx\_poll\_msg[7] = responderNumber;

tx\_resp\_msg[6] = responderNumber;

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, 1);

//main ex

setup\_DW1000RSTnIRQ(0);

stdio\_init(&huart2);

// HAL\_TIM\_Base\_Init(&htim2);

//end of my ex

/\* Display application name. \*/

stdio\_write(APP\_NAME);

stdio\_write("\033[s"); // Save cursor position

/\* Reset and initialise DW1000.

\* For initialisation, DW1000 clocks must be temporarily set to crystal speed. After initialisation SPI rate can be increased for optimum

\* performance. \*/

reset\_DW1000(); /\* Target specific drive of RSTn line into DW1000 low for a period. \*/

port\_set\_dw1000\_slowrate();

**if** (dwt\_initialise(DWT\_LOADUCODE) == DWT\_ERROR) {

stdio\_write("INIT FAILED");

**while** (1) {

};

}

HAL\_Delay(10);

port\_set\_dw1000\_fastrate();

/\* Configure DW1000. See NOTE 7 below. \*/

dwt\_configure(&config);

/\* Apply default antenna delay value. See NOTE 1 below. \*/

dwt\_setrxantennadelay(RX\_ANT\_DLY);

dwt\_settxantennadelay(TX\_ANT\_DLY);

/\* Set preamble timeout for expected frames. See NOTE 6 below. \*/

dwt\_setpreambledetecttimeout(PRE\_TIMEOUT);

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, 0);

**if** (HAL\_CAN\_Init(&hcan1) != HAL\_OK) {

// Error\_Handler();

}

**if** (HAL\_CAN\_Start(&hcan1) != HAL\_OK) {

// Error\_Handler();

}

/\* USER CODE END 2 \*/

/\* Infinite loop \*/

/\* USER CODE BEGIN WHILE \*/

**while** (1) {

/\* Clear reception timeout to start next ranging process. \*/

dwt\_setrxtimeout(0);

/\* Activate reception immediately. \*/

dwt\_rxenable(DWT\_START\_RX\_IMMEDIATE);

/\* Poll for reception of a frame or error/timeout. See NOTE 8 below. \*/

**while** (!((status\_reg = dwt\_read32bitreg(SYS\_STATUS\_ID))

& (SYS\_STATUS\_RXFCG | SYS\_STATUS\_ALL\_RX\_TO

| SYS\_STATUS\_ALL\_RX\_ERR))) {

};

**if** (status\_reg & SYS\_STATUS\_RXFCG) {

uint32 frame\_len;

/\* Clear good RX frame event in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID, SYS\_STATUS\_RXFCG);

/\* A frame has been received, read it into the local buffer. \*/

frame\_len = dwt\_read32bitreg(RX\_FINFO\_ID) & RX\_FINFO\_RXFL\_MASK\_1023;

**if** (frame\_len <= RX\_BUFFER\_LEN) {

dwt\_readrxdata(rx\_buffer, frame\_len, 0);

}

/\* Check that the frame is a poll sent by "DS TWR initiator" example.

\* As the sequence number field of the frame is not relevant, it is cleared to simplify the validation of the frame. \*/

rx\_buffer[ALL\_MSG\_SN\_IDX] = 0;

//get initiator number

actualInitiatorNumber = rx\_buffer[8];

//write initiator number to frames for correct cheking

rx\_final\_msg[8] = actualInitiatorNumber;

rx\_poll\_msg[8] = actualInitiatorNumber;

tx\_resp\_msg[5] = actualInitiatorNumber;

**if** (memcmp(rx\_buffer, rx\_poll\_msg, ALL\_MSG\_COMMON\_LEN) == 0) {

uint32 resp\_tx\_time;

**int** ret;

/\* Retrieve poll reception timestamp. \*/

poll\_rx\_ts = get\_rx\_timestamp\_u64();

/\* Set send time for response. See NOTE 9 below. \*/

resp\_tx\_time = (poll\_rx\_ts

+ (POLL\_RX\_TO\_RESP\_TX\_DLY\_UUS \* UUS\_TO\_DWT\_TIME)) >> 8;

dwt\_setdelayedtrxtime(resp\_tx\_time);

/\* Set expected delay and timeout for final message reception. See NOTE 4 and 5 below. \*/

dwt\_setrxaftertxdelay(RESP\_TX\_TO\_FINAL\_RX\_DLY\_UUS);

dwt\_setrxtimeout(FINAL\_RX\_TIMEOUT\_UUS);

/\* Write and send the response message. See NOTE 10 below.\*/

tx\_resp\_msg[ALL\_MSG\_SN\_IDX] = frame\_seq\_nb;

dwt\_writetxdata(**sizeof**(tx\_resp\_msg), tx\_resp\_msg, 0); /\* Zero offset in TX buffer. \*/

dwt\_writetxfctrl(**sizeof**(tx\_resp\_msg), 0, 1); /\* Zero offset in TX buffer, ranging. \*/

ret = dwt\_starttx(DWT\_START\_TX\_DELAYED | DWT\_RESPONSE\_EXPECTED);

/\* If dwt\_starttx() returns an error, abandon this ranging exchange and proceed to the next one. See NOTE 11 below. \*/

**if** (ret == DWT\_ERROR) {

**continue**;

}

/\* Poll for reception of expected "final" frame or error/timeout. See NOTE 8 below. \*/

**while** (!((status\_reg = dwt\_read32bitreg(SYS\_STATUS\_ID))

& (SYS\_STATUS\_RXFCG | SYS\_STATUS\_ALL\_RX\_TO

| SYS\_STATUS\_ALL\_RX\_ERR))) {

};

/\* Increment frame sequence number after transmission of the response message (modulo 256). \*/

frame\_seq\_nb++;

**if** (status\_reg & SYS\_STATUS\_RXFCG) {

// stdio\_write("Received final frame\r\n");

/\* Clear good RX frame event and TX frame sent in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID,

SYS\_STATUS\_RXFCG | SYS\_STATUS\_TXFRS);

/\* A frame has been received, read it into the local buffer. \*/

frame\_len = dwt\_read32bitreg(

RX\_FINFO\_ID) & RX\_FINFO\_RXFLEN\_MASK;

**if** (frame\_len <= RX\_BUF\_LEN) {

dwt\_readrxdata(rx\_buffer, frame\_len, 0);

}

/\* Check that the frame is a final message sent by "DS TWR initiator" example.

\* As the sequence number field of the frame is not used in this example, it can be zeroed to ease the validation of the frame. \*/

rx\_buffer[ALL\_MSG\_SN\_IDX] = 0;

//Pull Initiator number

uint8\_t initiatorNumberInt = atoi((**char**\*) &rx\_buffer[8]);

**if** (memcmp(rx\_buffer, rx\_final\_msg, ALL\_MSG\_COMMON\_LEN)

== 0) {

uint32 poll\_tx\_ts, resp\_rx\_ts, final\_tx\_ts;

uint32 poll\_rx\_ts\_32, resp\_tx\_ts\_32, final\_rx\_ts\_32;

**double** Ra, Rb, Da, Db;

int64 tof\_dtu;

/\* Retrieve response transmission and final reception timestamps. \*/

resp\_tx\_ts = get\_tx\_timestamp\_u64();

final\_rx\_ts = get\_rx\_timestamp\_u64();

/\* Get timestamps embedded in the final message. \*/

final\_msg\_get\_ts(&rx\_buffer[FINAL\_MSG\_POLL\_TX\_TS\_IDX],

&poll\_tx\_ts);

final\_msg\_get\_ts(&rx\_buffer[FINAL\_MSG\_RESP\_RX\_TS\_IDX],

&resp\_rx\_ts);

final\_msg\_get\_ts(&rx\_buffer[FINAL\_MSG\_FINAL\_TX\_TS\_IDX],

&final\_tx\_ts);

/\* Compute time of flight. 32-bit subtractions give correct answers even if clock has wrapped. See NOTE 12 below. \*/

poll\_rx\_ts\_32 = (uint32) poll\_rx\_ts;

resp\_tx\_ts\_32 = (uint32) resp\_tx\_ts;

final\_rx\_ts\_32 = (uint32) final\_rx\_ts;

Ra = (**double**) (resp\_rx\_ts - poll\_tx\_ts);

Rb = (**double**) (final\_rx\_ts\_32 - resp\_tx\_ts\_32);

Da = (**double**) (final\_tx\_ts - resp\_rx\_ts);

Db = (**double**) (resp\_tx\_ts\_32 - poll\_rx\_ts\_32);

tof\_dtu = (int64) ((Ra \* Rb - Da \* Db)

/ (Ra + Rb + Da + Db));

tof = tof\_dtu \* DWT\_TIME\_UNITS;

distance = (tof \* SPEED\_OF\_LIGHT);

distanceCm.integer = distance \*100;

/\* Display computed distance. \*/

HAL\_GPIO\_WritePin(LED\_STANDBY\_GPIO\_Port,

LED\_STANDBY\_Pin, 1);

memset(dist\_str, ' ', **sizeof**(dist\_str));

stdio\_write("\r\n");

sprintf(dist\_str, "R%c I%d: %d cm", responderNumber,

initiatorNumberInt, distanceCm.integer);

stdio\_write(dist\_str);

//convert responderNumber to int

uint8\_t responderNumberInt = responderNumber - 48;

TxData[0] = initiatorNumberInt;

TxData[1] = distanceCm.array[0];

TxData[2] = distanceCm.array[1];

TxHeader.StdId = 300 + responderNumberInt;

TxHeader.DLC = **sizeof**(TxData);

TxHeader.RTR = CAN\_RTR\_DATA;

TxHeader.IDE = CAN\_ID\_STD;

TxHeader.TransmitGlobalTime = DISABLE;

HAL\_CAN\_AbortTxRequest(&hcan1, TxMailbox);

**if** (HAL\_CAN\_AddTxMessage(&hcan1, &TxHeader, TxData,

&TxMailbox) != HAL\_OK) {

// Error\_Handler();

}

// HAL\_IWDG\_Refresh(&hiwdg);

HAL\_GPIO\_WritePin(LED\_STANDBY\_GPIO\_Port,

LED\_STANDBY\_Pin, 0);

}

} **else** {

/\* Clear RX error/timeout events in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID,

SYS\_STATUS\_ALL\_RX\_TO | SYS\_STATUS\_ALL\_RX\_ERR);

/\* Reset RX to properly reinitialise LDE operation. \*/

dwt\_rxreset();

}

}

} **else** {

/\* Clear RX error/timeout events in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID,

SYS\_STATUS\_ALL\_RX\_TO | SYS\_STATUS\_ALL\_RX\_ERR);

/\* Reset RX to properly reinitialise LDE operation. \*/

dwt\_rxreset();

}

stdio\_write("..");

}

/\* USER CODE END WHILE \*/

/\* USER CODE BEGIN 3 \*/

/\* USER CODE END 3 \*/

}

/\*\*

\* @brief System Clock Configuration

\* @retval None

\*/

**void** SystemClock\_Config(**void**) {

RCC\_OscInitTypeDef RCC\_OscInitStruct = { 0 };

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = { 0 };

/\*\* Configure the main internal regulator output voltage

\*/

**if** (HAL\_PWREx\_ControlVoltageScaling(PWR\_REGULATOR\_VOLTAGE\_SCALE1)

!= HAL\_OK) {

Error\_Handler();

}

/\*\* Initializes the RCC Oscillators according to the specified parameters

\* in the RCC\_OscInitTypeDef structure.

\*/

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSI;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.HSICalibrationValue = RCC\_HSICALIBRATION\_DEFAULT;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_ON;

RCC\_OscInitStruct.PLL.PLLSource = RCC\_PLLSOURCE\_HSI;

RCC\_OscInitStruct.PLL.PLLM = 2;

RCC\_OscInitStruct.PLL.PLLN = 16;

RCC\_OscInitStruct.PLL.PLLP = RCC\_PLLP\_DIV7;

RCC\_OscInitStruct.PLL.PLLQ = RCC\_PLLQ\_DIV2;

RCC\_OscInitStruct.PLL.PLLR = RCC\_PLLR\_DIV4;

**if** (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != HAL\_OK) {

Error\_Handler();

}

/\*\* Initializes the CPU, AHB and APB buses clocks

\*/

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK | RCC\_CLOCKTYPE\_SYSCLK

| RCC\_CLOCKTYPE\_PCLK1 | RCC\_CLOCKTYPE\_PCLK2;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_PLLCLK;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV1;

RCC\_ClkInitStruct.APB2CLKDivider = RCC\_HCLK\_DIV1;

**if** (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct, FLASH\_LATENCY\_1) != HAL\_OK) {

Error\_Handler();

}

}

/\*\*

\* @brief ADC1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_ADC1\_Init(**void**) {

/\* USER CODE BEGIN ADC1\_Init 0 \*/

/\* USER CODE END ADC1\_Init 0 \*/

ADC\_ChannelConfTypeDef sConfig = { 0 };

/\* USER CODE BEGIN ADC1\_Init 1 \*/

/\* USER CODE END ADC1\_Init 1 \*/

/\*\* Common config

\*/

hadc1.Instance = ADC1;

hadc1.Init.ClockPrescaler = ADC\_CLOCK\_ASYNC\_DIV1;

hadc1.Init.Resolution = ADC\_RESOLUTION\_12B;

hadc1.Init.DataAlign = ADC\_DATAALIGN\_RIGHT;

hadc1.Init.ScanConvMode = ADC\_SCAN\_DISABLE;

hadc1.Init.EOCSelection = ADC\_EOC\_SINGLE\_CONV;

hadc1.Init.LowPowerAutoWait = DISABLE;

hadc1.Init.ContinuousConvMode = DISABLE;

hadc1.Init.NbrOfConversion = 1;

hadc1.Init.DiscontinuousConvMode = DISABLE;

hadc1.Init.ExternalTrigConv = ADC\_SOFTWARE\_START;

hadc1.Init.ExternalTrigConvEdge = ADC\_EXTERNALTRIGCONVEDGE\_NONE;

hadc1.Init.DMAContinuousRequests = DISABLE;

hadc1.Init.Overrun = ADC\_OVR\_DATA\_PRESERVED;

hadc1.Init.OversamplingMode = DISABLE;

**if** (HAL\_ADC\_Init(&hadc1) != HAL\_OK) {

Error\_Handler();

}

/\*\* Configure Regular Channel

\*/

sConfig.Channel = ADC\_CHANNEL\_6;

sConfig.Rank = ADC\_REGULAR\_RANK\_1;

sConfig.SamplingTime = ADC\_SAMPLETIME\_2CYCLES\_5;

sConfig.SingleDiff = ADC\_SINGLE\_ENDED;

sConfig.OffsetNumber = ADC\_OFFSET\_NONE;

sConfig.Offset = 0;

**if** (HAL\_ADC\_ConfigChannel(&hadc1, &sConfig) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN ADC1\_Init 2 \*/

/\* USER CODE END ADC1\_Init 2 \*/

}

/\*\*

\* @brief CAN1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_CAN1\_Init(**void**) {

/\* USER CODE BEGIN CAN1\_Init 0 \*/

/\* USER CODE END CAN1\_Init 0 \*/

/\* USER CODE BEGIN CAN1\_Init 1 \*/

/\* USER CODE END CAN1\_Init 1 \*/

hcan1.Instance = CAN1;

hcan1.Init.Prescaler = 16;

hcan1.Init.Mode = CAN\_MODE\_NORMAL;

hcan1.Init.SyncJumpWidth = CAN\_SJW\_1TQ;

hcan1.Init.TimeSeg1 = CAN\_BS1\_13TQ;

hcan1.Init.TimeSeg2 = CAN\_BS2\_2TQ;

hcan1.Init.TimeTriggeredMode = DISABLE;

hcan1.Init.AutoBusOff = DISABLE;

hcan1.Init.AutoWakeUp = ENABLE;

hcan1.Init.AutoRetransmission = ENABLE;

hcan1.Init.ReceiveFifoLocked = DISABLE;

hcan1.Init.TransmitFifoPriority = DISABLE;

**if** (HAL\_CAN\_Init(&hcan1) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN CAN1\_Init 2 \*/

/\* USER CODE END CAN1\_Init 2 \*/

}

/\*\*

\* @brief SPI1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_SPI1\_Init(**void**) {

/\* USER CODE BEGIN SPI1\_Init 0 \*/

/\* USER CODE END SPI1\_Init 0 \*/

/\* USER CODE BEGIN SPI1\_Init 1 \*/

/\* USER CODE END SPI1\_Init 1 \*/

/\* SPI1 parameter configuration\*/

hspi1.Instance = SPI1;

hspi1.Init.Mode = SPI\_MODE\_MASTER;

hspi1.Init.Direction = SPI\_DIRECTION\_2LINES;

hspi1.Init.DataSize = SPI\_DATASIZE\_8BIT;

hspi1.Init.CLKPolarity = SPI\_POLARITY\_LOW;

hspi1.Init.CLKPhase = SPI\_PHASE\_1EDGE;

hspi1.Init.NSS = SPI\_NSS\_SOFT;

hspi1.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_4;

hspi1.Init.FirstBit = SPI\_FIRSTBIT\_MSB;

hspi1.Init.TIMode = SPI\_TIMODE\_DISABLE;

hspi1.Init.CRCCalculation = SPI\_CRCCALCULATION\_DISABLE;

hspi1.Init.CRCPolynomial = 7;

hspi1.Init.CRCLength = SPI\_CRC\_LENGTH\_DATASIZE;

hspi1.Init.NSSPMode = SPI\_NSS\_PULSE\_ENABLE;

**if** (HAL\_SPI\_Init(&hspi1) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN SPI1\_Init 2 \*/

/\* USER CODE END SPI1\_Init 2 \*/

}

/\*\*

\* @brief TIM2 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_TIM2\_Init(**void**) {

/\* USER CODE BEGIN TIM2\_Init 0 \*/

/\* USER CODE END TIM2\_Init 0 \*/

TIM\_ClockConfigTypeDef sClockSourceConfig = { 0 };

TIM\_MasterConfigTypeDef sMasterConfig = { 0 };

/\* USER CODE BEGIN TIM2\_Init 1 \*/

/\* USER CODE END TIM2\_Init 1 \*/

htim2.Instance = TIM2;

htim2.Init.Prescaler = 0;

htim2.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim2.Init.Period = 4294967295;

htim2.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

htim2.Init.AutoReloadPreload = TIM\_AUTORELOAD\_PRELOAD\_DISABLE;

**if** (HAL\_TIM\_Base\_Init(&htim2) != HAL\_OK) {

Error\_Handler();

}

sClockSourceConfig.ClockSource = TIM\_CLOCKSOURCE\_INTERNAL;

**if** (HAL\_TIM\_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL\_OK) {

Error\_Handler();

}

sMasterConfig.MasterOutputTrigger = TIM\_TRGO\_RESET;

sMasterConfig.MasterSlaveMode = TIM\_MASTERSLAVEMODE\_DISABLE;

**if** (HAL\_TIMEx\_MasterConfigSynchronization(&htim2, &sMasterConfig)

!= HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN TIM2\_Init 2 \*/

/\* USER CODE END TIM2\_Init 2 \*/

}

/\*\*

\* @brief USART2 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_USART2\_UART\_Init(**void**) {

/\* USER CODE BEGIN USART2\_Init 0 \*/

/\* USER CODE END USART2\_Init 0 \*/

/\* USER CODE BEGIN USART2\_Init 1 \*/

/\* USER CODE END USART2\_Init 1 \*/

huart2.Instance = USART2;

huart2.Init.BaudRate = 115200;

huart2.Init.WordLength = UART\_WORDLENGTH\_8B;

huart2.Init.StopBits = UART\_STOPBITS\_1;

huart2.Init.Parity = UART\_PARITY\_NONE;

huart2.Init.Mode = UART\_MODE\_TX\_RX;

huart2.Init.HwFlowCtl = UART\_HWCONTROL\_NONE;

huart2.Init.OverSampling = UART\_OVERSAMPLING\_16;

huart2.Init.OneBitSampling = UART\_ONE\_BIT\_SAMPLE\_DISABLE;

huart2.AdvancedInit.AdvFeatureInit = UART\_ADVFEATURE\_NO\_INIT;

**if** (HAL\_UART\_Init(&huart2) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN USART2\_Init 2 \*/

/\* USER CODE END USART2\_Init 2 \*/

}

/\*\*

\* Enable DMA controller clock

\*/

**static** **void** MX\_DMA\_Init(**void**) {

/\* DMA controller clock enable \*/

\_\_HAL\_RCC\_DMA1\_CLK\_ENABLE();

/\* DMA interrupt init \*/

/\* DMA1\_Channel2\_IRQn interrupt configuration \*/

HAL\_NVIC\_SetPriority(DMA1\_Channel2\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(DMA1\_Channel2\_IRQn);

/\* DMA1\_Channel3\_IRQn interrupt configuration \*/

HAL\_NVIC\_SetPriority(DMA1\_Channel3\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(DMA1\_Channel3\_IRQn);

}

/\*\*

\* @brief GPIO Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_GPIO\_Init(**void**) {

GPIO\_InitTypeDef GPIO\_InitStruct = { 0 };

/\* GPIO Ports Clock Enable \*/

\_\_HAL\_RCC\_GPIOC\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOH\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

/\*Configure GPIO pin Output Level \*/

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, GPIO\_PIN\_RESET);

/\*Configure GPIO pin Output Level \*/

HAL\_GPIO\_WritePin(GPIOB, DW\_NSS\_Pin | DW\_RESET\_Pin | LED\_STANDBY\_Pin,

GPIO\_PIN\_RESET);

/\*Configure GPIO pin Output Level \*/

HAL\_GPIO\_WritePin(DW\_WKUP\_GPIO\_Port, DW\_WKUP\_Pin, GPIO\_PIN\_SET);

/\*Configure GPIO pin : LED\_ERROR\_Pin \*/

GPIO\_InitStruct.Pin = LED\_ERROR\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(LED\_ERROR\_GPIO\_Port, &GPIO\_InitStruct);

/\*Configure GPIO pins : DW\_NSS\_Pin LED\_STANDBY\_Pin \*/

GPIO\_InitStruct.Pin = DW\_NSS\_Pin | LED\_STANDBY\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

/\*Configure GPIO pins : DW\_RESET\_Pin DW\_WKUP\_Pin \*/

GPIO\_InitStruct.Pin = DW\_RESET\_Pin | DW\_WKUP\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_OD;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

/\*Configure GPIO pin : DW\_IRQn\_Pin \*/

GPIO\_InitStruct.Pin = DW\_IRQn\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_RISING;

GPIO\_InitStruct.Pull = GPIO\_PULLDOWN;

HAL\_GPIO\_Init(DW\_IRQn\_GPIO\_Port, &GPIO\_InitStruct);

/\* EXTI interrupt init\*/

HAL\_NVIC\_SetPriority(EXTI2\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(EXTI2\_IRQn);

}

/\* USER CODE BEGIN 4 \*/

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn get\_tx\_timestamp\_u64()

\*

\* @brief Get the TX time-stamp in a 64-bit variable.

\* /!\ This function assumes that length of time-stamps is 40 bits, for both TX and RX!

\*

\* @param none

\*

\* @return 64-bit value of the read time-stamp.

\*/

**static** uint64 get\_tx\_timestamp\_u64(**void**) {

uint8 ts\_tab[5];

uint64 ts = 0;

**int** i;

dwt\_readtxtimestamp(ts\_tab);

**for** (i = 4; i >= 0; i--) {

ts <<= 8;

ts |= ts\_tab[i];

}

**return** ts;

}

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn get\_rx\_timestamp\_u64()

\*

\* @brief Get the RX time-stamp in a 64-bit variable.

\* /!\ This function assumes that length of time-stamps is 40 bits, for both TX and RX!

\*

\* @param none

\*

\* @return 64-bit value of the read time-stamp.

\*/

**static** uint64 get\_rx\_timestamp\_u64(**void**) {

uint8 ts\_tab[5];

uint64 ts = 0;

**int** i;

dwt\_readrxtimestamp(ts\_tab);

**for** (i = 4; i >= 0; i--) {

ts <<= 8;

ts |= ts\_tab[i];

}

**return** ts;

}

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn final\_msg\_get\_ts()

\*

\* @brief Read a given timestamp value from the final message. In the timestamp fields of the final message, the least

\* significant byte is at the lower address.

\*

\* @param ts\_field pointer on the first byte of the timestamp field to read

\* ts timestamp value

\*

\* @return none

\*/

**static** **void** final\_msg\_get\_ts(**const** uint8 \*ts\_field, uint32 \*ts) {

**int** i;

\*ts = 0;

**for** (i = 0; i < FINAL\_MSG\_TS\_LEN; i++) {

\*ts += ts\_field[i] << (i \* 8);

}

}

/\* USER CODE END 4 \*/

/\*\*

\* @brief This function is executed in case of error occurrence.

\* @retval None

\*/

**void** Error\_Handler(**void**) {

/\* USER CODE BEGIN Error\_Handler\_Debug \*/

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, GPIO\_PIN\_SET);

**while** (1)

;

/\* USER CODE END Error\_Handler\_Debug \*/

}

**#ifdef** USE\_FULL\_ASSERT

/\*\*

\* @brief Reports the name of the source file and the source line number

\* where the assert\_param error has occurred.

\* @param file: pointer to the source file name

\* @param line: assert\_param error line source number

\* @retval None

\*/

**void** assert\_failed(uint8\_t \*file, uint32\_t line)

{

/\* USER CODE BEGIN 6 \*/

/\* USER CODE END 6 \*/

}

**#endif** /\* USE\_FULL\_ASSERT \*/

Kod jest bazowany na przykładzie dystansowania UWB.

Na początku zostaje załączone API czujników UWB.

W strukturze typu dwt\_config\_t o nazwie config zostają ustawione parametry transmisji danych.

Tablice rx\_poll\_msg[], tx\_resp\_msg[] oraz rx\_final\_msg[] zawierają wzory ramek, zmienna responderNumber przechowuje wstępnie zaprogramowany numer reposndera, a actualInitiatorNumber przechowuje numer aktualnie dystansowanego inicjatora.

Baza czasu UUS\_TO\_DWT\_TIME została zmieniona ze względu na niższe taktowanie docelowego MCU.

W pętli głównej responder oczekuje na sygnał odebrany od inicjatora. Jeżeli taki zostaje odebrany, z ramki zostaje odczytany oraz przypisany do actualInitiatorNumber numer inicjatora. Po zweryfikowaniu poprawności ramki zostaje zapisany timestamp, który jest następnie odsyłany do inicjatora. Reponder oczekuje na kolejną ramkę. Po jej odebraniu wylicza dystans na podstawie różnicy pomiędzy zapisanym poprzednim TS a aktualnie odebranym od inicjatora. Wyliczony dystans zostaje przesłany przez CAN, podając numer respondera(zawięrający się w adresie CAN) oraz 3 bajtową ramkę zawierającą numer zdystansowanego inicjatora, oraz 2 bajty zawierające dystans w centrymetrach.

#### Program inicjatora

Treść programu inicjatora:

/\* USER CODE BEGIN Header \*/

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : main.c

\* @brief : Main program body

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @attention

\*

\* <h2><center>&copy; Copyright (c) 2021 STMicroelectronics.

\* All rights reserved.</center></h2>

\*

\* This software component is licensed by ST under BSD 3-Clause license,

\* the "License"; You may not use this file except in compliance with the

\* License. You may obtain a copy of the License at:

\* opensource.org/licenses/BSD-3-Clause

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\* USER CODE END Header \*/

/\* Includes ------------------------------------------------------------------\*/

**#include** "main.h"

/\* Private includes ----------------------------------------------------------\*/

/\* USER CODE BEGIN Includes \*/

//main ex

**#include** "platform/port.h"

app\_t app;

//end of main ex

**#include** <stdio.h>

**#include** <string.h>

**#include** "decadriver/deca\_device\_api.h"

**#include** "decadriver/deca\_regs.h"

**#include** "platform/stdio.h"

**#include** "platform/deca\_spi.h"

**#include** "platform/port.h"

//#include "../Display/an\_disp.h"

/\* USER CODE END Includes \*/

/\* Private typedef -----------------------------------------------------------\*/

/\* USER CODE BEGIN PTD \*/

/\* USER CODE END PTD \*/

/\* Private define ------------------------------------------------------------\*/

/\* USER CODE BEGIN PD \*/

/\* USER CODE END PD \*/

/\* Private macro -------------------------------------------------------------\*/

/\* USER CODE BEGIN PM \*/

/\* USER CODE END PM \*/

/\* Private variables ---------------------------------------------------------\*/

ADC\_HandleTypeDef hadc1;

IWDG\_HandleTypeDef hiwdg;

RTC\_HandleTypeDef hrtc;

SPI\_HandleTypeDef hspi1;

UART\_HandleTypeDef huart2;

/\* USER CODE BEGIN PV \*/

/\* USER CODE END PV \*/

/\* Private function prototypes -----------------------------------------------\*/

**void** SystemClock\_Config(**void**);

**static** **void** MX\_GPIO\_Init(**void**);

**static** **void** MX\_USART2\_UART\_Init(**void**);

**static** **void** MX\_ADC1\_Init(**void**);

**static** **void** MX\_SPI1\_Init(**void**);

**static** **void** MX\_RTC\_Init(**void**);

**static** **void** MX\_IWDG\_Init(**void**);

/\* USER CODE BEGIN PFP \*/

/\* USER CODE END PFP \*/

/\* Private user code ---------------------------------------------------------\*/

/\* USER CODE BEGIN 0 \*/

/\* Example application name and version to display. \*/

**#define** APP\_NAME "DS TWR INIT v1.2"

/\* Inter-ranging delay period, in milliseconds. \*/

**#define** RNG\_DELAY\_MS 900

/\* Default communication configuration. We use here EVK1000's default mode (mode 3). \*/

**static** dwt\_config\_t config = { 2, /\* Channel number. \*/

DWT\_PRF\_64M, /\* Pulse repetition frequency. \*/

DWT\_PLEN\_1024, /\* Preamble length. Used in TX only. \*/

DWT\_PAC32, /\* Preamble acquisition chunk size. Used in RX only. \*/

9, /\* TX preamble code. Used in TX only. \*/

9, /\* RX preamble code. Used in RX only. \*/

1, /\* 0 to use standard SFD, 1 to use non-standard SFD. \*/

DWT\_BR\_110K, /\* Data rate. \*/

DWT\_PHRMODE\_STD, /\* PHY header mode. \*/

(1025 + 64 - 32) /\* SFD timeout (preamble length + 1 + SFD length - PAC size). Used in RX only. \*/

};

/\* Default antenna delay values for 64 MHz PRF. See NOTE 1 below. \*/

**#define** TX\_ANT\_DLY 16505

**#define** RX\_ANT\_DLY 16505

/\* Frames used in the ranging process. See NOTE 2 below. \*/

**static** uint8 tx\_poll\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, 'C', 'Z', '0', '0',

0x21, 0, 0 };

**static** uint8 rx\_resp\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, '0', '0', 'Z', 'C',

0x10, 0x02, 0, 0, 0, 0 };

uint8 tx\_final\_msg[] = { 0x41, 0x88, 0, 0xCA, 0xDE, 'C', 'Z', '0', '0', 0x23, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

//Initiator Number

uint8\_t initiatorNumber = '3';

uint8\_t actualResponderNumber = '0';

/\* Length of the common part of the message (up to and including the function code, see NOTE 2 below). \*/

**#define** ALL\_MSG\_COMMON\_LEN 10

/\* Indexes to access some of the fields in the frames defined above. \*/

**#define** ALL\_MSG\_SN\_IDX 2

**#define** FINAL\_MSG\_POLL\_TX\_TS\_IDX 10

**#define** FINAL\_MSG\_RESP\_RX\_TS\_IDX 14

**#define** FINAL\_MSG\_FINAL\_TX\_TS\_IDX 18

**#define** FINAL\_MSG\_TS\_LEN 4

/\* Frame sequence number, incremented after each transmission. \*/

**static** uint8 frame\_seq\_nb = 0;

/\* Buffer to store received response message.

\* Its size is adjusted to longest frame that this example code is supposed to handle. \*/

**#define** RX\_BUF\_LEN 20

**static** uint8 rx\_buffer[RX\_BUF\_LEN];

/\* Hold copy of status register state here for reference so that it can be examined at a debug breakpoint. \*/

**static** uint32 status\_reg = 0;

/\* UWB microsecond (uus) to device time unit (dtu, around 15.65 ps) conversion factor.

\* 1 uus = 512 / 499.2 µs and 1 µs = 499.2 \* 128 dtu. \*/

//#define UUS\_TO\_DWT\_TIME 65536 //changed

**#define** UUS\_TO\_DWT\_TIME 158875

/\* Delay between frames, in UWB microseconds. See NOTE 4 below. \*/

/\* This is the delay from the end of the frame transmission to the enable of the receiver, as programmed for the DW1000's wait for response feature. \*/

**#define** POLL\_TX\_TO\_RESP\_RX\_DLY\_UUS 300 //changed

//#define POLL\_TX\_TO\_RESP\_RX\_DLY\_UUS 1200

/\* This is the delay from Frame RX timestamp to TX reply timestamp used for calculating/setting the DW1000's delayed TX function. This includes the

\* frame length of approximately 2.66 ms with above configuration. \*/

**#define** RESP\_RX\_TO\_FINAL\_TX\_DLY\_UUS 3100 //changed

//#define RESP\_RX\_TO\_FINAL\_TX\_DLY\_UUS 12000

/\* Receive response timeout. See NOTE 5 below. \*/

//#define RESP\_RX\_TIMEOUT\_UUS 2700 //changed

**#define** RESP\_RX\_TIMEOUT\_UUS 15000

/\* Preamble timeout, in multiple of PAC size. See NOTE 6 below. \*/

//#define PRE\_TIMEOUT 8 //changed

**#define** PRE\_TIMEOUT 0

/\* Dummy buffer for DW1000 wake-up SPI read. See NOTE 2 below. \*/

**#define** DUMMY\_BUFFER\_LEN 2000

//static uint8 dummy\_buffer[DUMMY\_BUFFER\_LEN];

/\* Time-stamps of frames transmission/reception, expressed in device time units.

\* As they are 40-bit wide, we need to define a 64-bit int type to handle them. \*/

**typedef** **unsigned** **long** **long** uint64;

**static** uint64 poll\_tx\_ts;

**static** uint64 resp\_rx\_ts;

**static** uint64 final\_tx\_ts;

/\* Declaration of static functions. \*/

**static** uint64 get\_tx\_timestamp\_u64(**void**);

**static** uint64 get\_rx\_timestamp\_u64(**void**);

**static** **void** final\_msg\_set\_ts(uint8 \*ts\_field, uint64 ts);

/\* USER CODE END 0 \*/

/\*\*

\* @brief The application entry point.

\* @retval int

\*/

**int** main(**void**) {

/\* USER CODE BEGIN 1 \*/

/\* USER CODE END 1 \*/

/\* MCU Configuration--------------------------------------------------------\*/

/\* Reset of all peripherals, Initializes the Flash interface and the Systick. \*/

HAL\_Init();

/\* USER CODE BEGIN Init \*/

/\* USER CODE END Init \*/

/\* Configure the system clock \*/

SystemClock\_Config();

/\* USER CODE BEGIN SysInit \*/

/\* USER CODE END SysInit \*/

/\* Initialize all configured peripherals \*/

MX\_GPIO\_Init();

MX\_USART2\_UART\_Init();

MX\_ADC1\_Init();

MX\_SPI1\_Init();

MX\_RTC\_Init();

MX\_IWDG\_Init();

/\* USER CODE BEGIN 2 \*/

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, 0);

//turn on watchdog

HAL\_IWDG\_Init(&hiwdg);

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, 1);

//give initiator its own number

tx\_final\_msg[8] = initiatorNumber;

tx\_poll\_msg[8] = initiatorNumber;

rx\_resp\_msg[5] = initiatorNumber;

//wake up timer different in each initiator number

uint16\_t wkupTimer = 1800 + (atoi((**char**\*) &initiatorNumber) \* 5);

HAL\_RTCEx\_SetWakeUpTimer\_IT(&hrtc, wkupTimer, RTC\_WAKEUPCLOCK\_RTCCLK\_DIV16);

//main ex

setup\_DW1000RSTnIRQ(0);

stdio\_init(&huart2);

// HAL\_TIM\_Base\_Init(&htim1);

//end of my ex

/\* Display application name. \*/

stdio\_write(APP\_NAME);

/\* Reset and initialise DW1000.

\* For initialisation, DW1000 clocks must be temporarily set to crystal speed. After initialisation SPI rate can be increased for optimum

\* performance. \*/

reset\_DW1000(); /\* Target specific drive of RSTn line into DW1000 low for a period. \*/

/\* Dummy buffer for DW1000 wake-up SPI read. See NOTE 2 below. \*/

**static** uint8 dummy\_buffer[DUMMY\_BUFFER\_LEN];

dwt\_spicswakeup(dummy\_buffer, DUMMY\_BUFFER\_LEN);

port\_set\_dw1000\_slowrate();

**if** (dwt\_initialise(DWT\_LOADUCODE) == DWT\_ERROR) {

stdio\_write("INIT FAILED");

**while** (1) {

};

}

HAL\_Delay(10);

port\_set\_dw1000\_fastrate();

/\* Configure DW1000. See NOTE 7 below. \*/

dwt\_configure(&config);

/\* Apply default antenna delay value. See NOTE 1 below. \*/

dwt\_setrxantennadelay(RX\_ANT\_DLY);

dwt\_settxantennadelay(TX\_ANT\_DLY);

/\* Set expected response's delay and timeout. See NOTE 4, 5 and 6 below.

\* As this example only handles one incoming frame with always the same delay and timeout, those values can be set here once for all. \*/

dwt\_setrxaftertxdelay(POLL\_TX\_TO\_RESP\_RX\_DLY\_UUS); // changed

// dwt\_setrxaftertxdelay(0);

dwt\_setrxtimeout(RESP\_RX\_TIMEOUT\_UUS);

dwt\_setpreambledetecttimeout(PRE\_TIMEOUT);

//configure radio sleep

// dwt\_configuresleep(DWT\_PRESRV\_SLEEP | DWT\_CONFIG, DWT\_WAKE\_CS | DWT\_SLP\_EN);

// dwt\_entersleepaftertx(1);

HAL\_GPIO\_WritePin(LED\_ERROR\_GPIO\_Port, LED\_ERROR\_Pin, 0);

dwt\_configuresleep(DWT\_PRESRV\_SLEEP | DWT\_CONFIG, DWT\_WAKE\_CS | DWT\_SLP\_EN);

/\* USER CODE END 2 \*/

/\* Infinite loop \*/

/\* USER CODE BEGIN WHILE \*/

**while** (1) {

**for** (uint8\_t i = 1; i <= 6; i++) {

//switch responder

**switch** (actualResponderNumber) {

**case** '0':

actualResponderNumber = '1';

**break**;

**case** '1':

actualResponderNumber = '2';

**break**;

**case** '2':

actualResponderNumber = '3';

**break**;

**case** '3':

actualResponderNumber = '4';

**break**;

**case** '4':

actualResponderNumber = '5';

**break**;

**case** '5':

actualResponderNumber = '6';

**break**;

**case** '6':

actualResponderNumber = '1';

**break**;

**default**:

actualResponderNumber = '1';

**break**;

}

tx\_final\_msg[7] = actualResponderNumber;

tx\_poll\_msg[7] = actualResponderNumber;

rx\_resp\_msg[6] = actualResponderNumber;

HAL\_Delay(1);

/\* Write frame data to DW1000 and prepare transmission. See NOTE 8 below. \*/

tx\_poll\_msg[ALL\_MSG\_SN\_IDX] = frame\_seq\_nb;

dwt\_writetxdata(**sizeof**(tx\_poll\_msg), tx\_poll\_msg, 0); /\* Zero offset in TX buffer. \*/

dwt\_writetxfctrl(**sizeof**(tx\_poll\_msg), 0, 1); /\* Zero offset in TX buffer, ranging. \*/

/\* Start transmission, indicating that a response is expected so that reception is enabled automatically after the frame is sent and the delay

\* set by dwt\_setrxaftertxdelay() has elapsed. \*/

dwt\_starttx(DWT\_START\_TX\_IMMEDIATE | DWT\_RESPONSE\_EXPECTED);

/\* We assume that the transmission is achieved correctly, poll for reception of a frame or error/timeout. See NOTE 9 below. \*/

**while** (!((status\_reg = dwt\_read32bitreg(SYS\_STATUS\_ID))

& (SYS\_STATUS\_RXFCG | SYS\_STATUS\_ALL\_RX\_TO

| SYS\_STATUS\_ALL\_RX\_ERR))) {

};

// stdio\_write("Request sent\r\n");

/\* Increment frame sequence number after transmission of the poll message (modulo 256). \*/

frame\_seq\_nb++;

**if** (status\_reg & SYS\_STATUS\_RXFCG) {

// stdio\_write("Response received\r\n");

uint32 frame\_len;

/\* Clear good RX frame event and TX frame sent in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID,

SYS\_STATUS\_RXFCG | SYS\_STATUS\_TXFRS);

/\* A frame has been received, read it into the local buffer. \*/

frame\_len =

dwt\_read32bitreg(RX\_FINFO\_ID) & RX\_FINFO\_RXFLEN\_MASK;

**if** (frame\_len <= RX\_BUF\_LEN) {

dwt\_readrxdata(rx\_buffer, frame\_len, 0);

}

/\* Check that the frame is the expected response from the companion "DS TWR responder" example.

\* As the sequence number field of the frame is not relevant, it is cleared to simplify the validation of the frame. \*/

rx\_buffer[ALL\_MSG\_SN\_IDX] = 0;

**if** (memcmp(rx\_buffer, rx\_resp\_msg, ALL\_MSG\_COMMON\_LEN) == 0) {

// stdio\_write("Data correct\r\n");

uint32 final\_tx\_time;

**int** ret;

/\* Retrieve poll transmission and response reception timestamp. \*/

poll\_tx\_ts = get\_tx\_timestamp\_u64();

resp\_rx\_ts = get\_rx\_timestamp\_u64();

/\* Compute final message transmission time. See NOTE 10 below. \*/

final\_tx\_time = (resp\_rx\_ts

+ (RESP\_RX\_TO\_FINAL\_TX\_DLY\_UUS \* UUS\_TO\_DWT\_TIME))

>> 8;

dwt\_setdelayedtrxtime(final\_tx\_time);

/\* Final TX timestamp is the transmission time we programmed plus the TX antenna delay. \*/

final\_tx\_ts = (((uint64) (final\_tx\_time & 0xFFFFFFFEUL))

<< 8) + TX\_ANT\_DLY;

/\* Write all timestamps in the final message. See NOTE 11 below. \*/

final\_msg\_set\_ts(&tx\_final\_msg[FINAL\_MSG\_POLL\_TX\_TS\_IDX],

poll\_tx\_ts);

final\_msg\_set\_ts(&tx\_final\_msg[FINAL\_MSG\_RESP\_RX\_TS\_IDX],

resp\_rx\_ts);

final\_msg\_set\_ts(&tx\_final\_msg[FINAL\_MSG\_FINAL\_TX\_TS\_IDX],

final\_tx\_ts);

/\* Write and send final message. See NOTE 8 below. \*/

tx\_final\_msg[ALL\_MSG\_SN\_IDX] = frame\_seq\_nb;

dwt\_writetxdata(**sizeof**(tx\_final\_msg), tx\_final\_msg, 0); /\* Zero offset in TX buffer. \*/

dwt\_writetxfctrl(**sizeof**(tx\_final\_msg), 0, 1); /\* Zero offset in TX buffer, ranging. \*/

ret = dwt\_starttx(DWT\_START\_TX\_DELAYED);

HAL\_IWDG\_Refresh(&hiwdg);

/\* If dwt\_starttx() returns an error, abandon this ranging exchange and proceed to the next one. See NOTE 12 below. \*/

**if** (ret == DWT\_SUCCESS) {

/\* Poll DW1000 until TX frame sent event set. See NOTE 9 below. \*/

**while** (!(dwt\_read32bitreg(SYS\_STATUS\_ID)

& SYS\_STATUS\_TXFRS)) {

};

// stdio\_write("Final response sent\r\n");

/\* Clear TXFRS event. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID, SYS\_STATUS\_TXFRS);

/\* Increment frame sequence number after transmission of the final message (modulo 256). \*/

frame\_seq\_nb++;

}

}

} **else** {

/\* Clear RX error/timeout events in the DW1000 status register. \*/

dwt\_write32bitreg(SYS\_STATUS\_ID,

SYS\_STATUS\_ALL\_RX\_TO | SYS\_STATUS\_ALL\_RX\_ERR);

/\* Reset RX to properly reinitialise LDE operation. \*/

dwt\_rxreset();

}

}

//before sleep

HAL\_GPIO\_WritePin(LED\_STANDBY\_GPIO\_Port, LED\_STANDBY\_Pin, 0);

//sleep

dwt\_entersleep();

HAL\_PWR\_EnterSTOPMode(PWR\_LOWPOWERREGULATOR\_ON, PWR\_STOPENTRY\_WFI);

//after sleep

SystemClock\_Config();

HAL\_GPIO\_WritePin(LED\_STANDBY\_GPIO\_Port, LED\_STANDBY\_Pin, 1);

//wake up radio

dwt\_spicswakeup(dummy\_buffer, DUMMY\_BUFFER\_LEN);

dwt\_setrxantennadelay(RX\_ANT\_DLY);

dwt\_settxantennadelay(TX\_ANT\_DLY);

dwt\_setrxaftertxdelay(POLL\_TX\_TO\_RESP\_RX\_DLY\_UUS);

dwt\_setrxtimeout(RESP\_RX\_TIMEOUT\_UUS);

dwt\_setpreambledetecttimeout(PRE\_TIMEOUT);

/\* USER CODE END WHILE \*/

/\* USER CODE BEGIN 3 \*/

}

/\* USER CODE END 3 \*/

}

/\*\*

\* @brief System Clock Configuration

\* @retval None

\*/

**void** SystemClock\_Config(**void**) {

RCC\_OscInitTypeDef RCC\_OscInitStruct = { 0 };

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = { 0 };

RCC\_PeriphCLKInitTypeDef PeriphClkInit = { 0 };

/\*\* Initializes the RCC Oscillators according to the specified parameters

\* in the RCC\_OscInitTypeDef structure.

\*/

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSI

| RCC\_OSCILLATORTYPE\_LSI;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.HSICalibrationValue = RCC\_HSICALIBRATION\_DEFAULT;

RCC\_OscInitStruct.LSIState = RCC\_LSI\_ON;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_ON;

RCC\_OscInitStruct.PLL.PLLSource = RCC\_PLLSOURCE\_HSI;

RCC\_OscInitStruct.PLL.PLLM = 1;

RCC\_OscInitStruct.PLL.PLLN = 8;

RCC\_OscInitStruct.PLL.PLLP = RCC\_PLLP\_DIV7;

RCC\_OscInitStruct.PLL.PLLQ = RCC\_PLLQ\_DIV2;

RCC\_OscInitStruct.PLL.PLLR = RCC\_PLLR\_DIV4;

**if** (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != HAL\_OK) {

Error\_Handler();

}

/\*\* Initializes the CPU, AHB and APB buses clocks

\*/

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK | RCC\_CLOCKTYPE\_SYSCLK

| RCC\_CLOCKTYPE\_PCLK1 | RCC\_CLOCKTYPE\_PCLK2;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_PLLCLK;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV1;

RCC\_ClkInitStruct.APB2CLKDivider = RCC\_HCLK\_DIV1;

**if** (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct, FLASH\_LATENCY\_1) != HAL\_OK) {

Error\_Handler();

}

PeriphClkInit.PeriphClockSelection = RCC\_PERIPHCLK\_RTC

| RCC\_PERIPHCLK\_USART2 | RCC\_PERIPHCLK\_ADC;

PeriphClkInit.Usart2ClockSelection = RCC\_USART2CLKSOURCE\_PCLK1;

PeriphClkInit.AdcClockSelection = RCC\_ADCCLKSOURCE\_SYSCLK;

PeriphClkInit.RTCClockSelection = RCC\_RTCCLKSOURCE\_LSI;

**if** (HAL\_RCCEx\_PeriphCLKConfig(&PeriphClkInit) != HAL\_OK) {

Error\_Handler();

}

/\*\* Configure the main internal regulator output voltage

\*/

**if** (HAL\_PWREx\_ControlVoltageScaling(PWR\_REGULATOR\_VOLTAGE\_SCALE1)

!= HAL\_OK) {

Error\_Handler();

}

}

/\*\*

\* @brief ADC1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_ADC1\_Init(**void**) {

/\* USER CODE BEGIN ADC1\_Init 0 \*/

/\* USER CODE END ADC1\_Init 0 \*/

ADC\_ChannelConfTypeDef sConfig = { 0 };

/\* USER CODE BEGIN ADC1\_Init 1 \*/

/\* USER CODE END ADC1\_Init 1 \*/

/\*\* Common config

\*/

hadc1.Instance = ADC1;

hadc1.Init.ClockPrescaler = ADC\_CLOCK\_ASYNC\_DIV1;

hadc1.Init.Resolution = ADC\_RESOLUTION\_12B;

hadc1.Init.DataAlign = ADC\_DATAALIGN\_RIGHT;

hadc1.Init.ScanConvMode = ADC\_SCAN\_DISABLE;

hadc1.Init.EOCSelection = ADC\_EOC\_SINGLE\_CONV;

hadc1.Init.LowPowerAutoWait = DISABLE;

hadc1.Init.ContinuousConvMode = DISABLE;

hadc1.Init.NbrOfConversion = 1;

hadc1.Init.DiscontinuousConvMode = DISABLE;

hadc1.Init.ExternalTrigConv = ADC\_SOFTWARE\_START;

hadc1.Init.ExternalTrigConvEdge = ADC\_EXTERNALTRIGCONVEDGE\_NONE;

hadc1.Init.DMAContinuousRequests = DISABLE;

hadc1.Init.Overrun = ADC\_OVR\_DATA\_PRESERVED;

hadc1.Init.OversamplingMode = DISABLE;

**if** (HAL\_ADC\_Init(&hadc1) != HAL\_OK) {

Error\_Handler();

}

/\*\* Configure Regular Channel

\*/

sConfig.Channel = ADC\_CHANNEL\_6;

sConfig.Rank = ADC\_REGULAR\_RANK\_1;

sConfig.SamplingTime = ADC\_SAMPLETIME\_2CYCLES\_5;

sConfig.SingleDiff = ADC\_SINGLE\_ENDED;

sConfig.OffsetNumber = ADC\_OFFSET\_NONE;

sConfig.Offset = 0;

**if** (HAL\_ADC\_ConfigChannel(&hadc1, &sConfig) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN ADC1\_Init 2 \*/

/\* USER CODE END ADC1\_Init 2 \*/

}

/\*\*

\* @brief IWDG Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_IWDG\_Init(**void**) {

/\* USER CODE BEGIN IWDG\_Init 0 \*/

/\* USER CODE END IWDG\_Init 0 \*/

/\* USER CODE BEGIN IWDG\_Init 1 \*/

/\* USER CODE END IWDG\_Init 1 \*/

hiwdg.Instance = IWDG;

hiwdg.Init.Prescaler = IWDG\_PRESCALER\_16;

hiwdg.Init.Window = 4095;

hiwdg.Init.Reload = 4095;

**if** (HAL\_IWDG\_Init(&hiwdg) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN IWDG\_Init 2 \*/

/\* USER CODE END IWDG\_Init 2 \*/

}

/\*\*

\* @brief RTC Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_RTC\_Init(**void**) {

/\* USER CODE BEGIN RTC\_Init 0 \*/

/\* USER CODE END RTC\_Init 0 \*/

RTC\_TimeTypeDef sTime = { 0 };

RTC\_DateTypeDef sDate = { 0 };

/\* USER CODE BEGIN RTC\_Init 1 \*/

/\* USER CODE END RTC\_Init 1 \*/

/\*\* Initialize RTC Only

\*/

hrtc.Instance = RTC;

hrtc.Init.HourFormat = RTC\_HOURFORMAT\_24;

hrtc.Init.AsynchPrediv = 127;

hrtc.Init.SynchPrediv = 255;

hrtc.Init.OutPut = RTC\_OUTPUT\_DISABLE;

hrtc.Init.OutPutRemap = RTC\_OUTPUT\_REMAP\_NONE;

hrtc.Init.OutPutPolarity = RTC\_OUTPUT\_POLARITY\_HIGH;

hrtc.Init.OutPutType = RTC\_OUTPUT\_TYPE\_OPENDRAIN;

**if** (HAL\_RTC\_Init(&hrtc) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN Check\_RTC\_BKUP \*/

/\* USER CODE END Check\_RTC\_BKUP \*/

/\*\* Initialize RTC and set the Time and Date

\*/

sTime.Hours = 0x0;

sTime.Minutes = 0x0;

sTime.Seconds = 0x0;

sTime.DayLightSaving = RTC\_DAYLIGHTSAVING\_NONE;

sTime.StoreOperation = RTC\_STOREOPERATION\_RESET;

**if** (HAL\_RTC\_SetTime(&hrtc, &sTime, RTC\_FORMAT\_BCD) != HAL\_OK) {

Error\_Handler();

}

sDate.WeekDay = RTC\_WEEKDAY\_MONDAY;

sDate.Month = RTC\_MONTH\_JANUARY;

sDate.Date = 0x1;

sDate.Year = 0x0;

**if** (HAL\_RTC\_SetDate(&hrtc, &sDate, RTC\_FORMAT\_BCD) != HAL\_OK) {

Error\_Handler();

}

/\*\* Enable the WakeUp

\*/

**if** (HAL\_RTCEx\_SetWakeUpTimer\_IT(&hrtc, 2000, RTC\_WAKEUPCLOCK\_RTCCLK\_DIV16)

!= HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN RTC\_Init 2 \*/

/\* USER CODE END RTC\_Init 2 \*/

}

/\*\*

\* @brief SPI1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_SPI1\_Init(**void**) {

/\* USER CODE BEGIN SPI1\_Init 0 \*/

/\* USER CODE END SPI1\_Init 0 \*/

/\* USER CODE BEGIN SPI1\_Init 1 \*/

/\* USER CODE END SPI1\_Init 1 \*/

/\* SPI1 parameter configuration\*/

hspi1.Instance = SPI1;

hspi1.Init.Mode = SPI\_MODE\_MASTER;

hspi1.Init.Direction = SPI\_DIRECTION\_2LINES;

hspi1.Init.DataSize = SPI\_DATASIZE\_8BIT;

hspi1.Init.CLKPolarity = SPI\_POLARITY\_LOW;

hspi1.Init.CLKPhase = SPI\_PHASE\_1EDGE;

hspi1.Init.NSS = SPI\_NSS\_SOFT;

hspi1.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_4;

hspi1.Init.FirstBit = SPI\_FIRSTBIT\_MSB;

hspi1.Init.TIMode = SPI\_TIMODE\_DISABLE;

hspi1.Init.CRCCalculation = SPI\_CRCCALCULATION\_DISABLE;

hspi1.Init.CRCPolynomial = 7;

hspi1.Init.CRCLength = SPI\_CRC\_LENGTH\_DATASIZE;

hspi1.Init.NSSPMode = SPI\_NSS\_PULSE\_ENABLE;

**if** (HAL\_SPI\_Init(&hspi1) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN SPI1\_Init 2 \*/

/\* USER CODE END SPI1\_Init 2 \*/

}

/\*\*

\* @brief USART2 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_USART2\_UART\_Init(**void**) {

/\* USER CODE BEGIN USART2\_Init 0 \*/

/\* USER CODE END USART2\_Init 0 \*/

/\* USER CODE BEGIN USART2\_Init 1 \*/

/\* USER CODE END USART2\_Init 1 \*/

huart2.Instance = USART2;

huart2.Init.BaudRate = 115200;

huart2.Init.WordLength = UART\_WORDLENGTH\_8B;

huart2.Init.StopBits = UART\_STOPBITS\_1;

huart2.Init.Parity = UART\_PARITY\_NONE;

huart2.Init.Mode = UART\_MODE\_TX\_RX;

huart2.Init.HwFlowCtl = UART\_HWCONTROL\_NONE;

huart2.Init.OverSampling = UART\_OVERSAMPLING\_16;

huart2.Init.OneBitSampling = UART\_ONE\_BIT\_SAMPLE\_DISABLE;

huart2.AdvancedInit.AdvFeatureInit = UART\_ADVFEATURE\_NO\_INIT;

**if** (HAL\_UART\_Init(&huart2) != HAL\_OK) {

Error\_Handler();

}

/\* USER CODE BEGIN USART2\_Init 2 \*/

/\* USER CODE END USART2\_Init 2 \*/

}

/\*\*

\* @brief GPIO Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** MX\_GPIO\_Init(**void**) {

GPIO\_InitTypeDef GPIO\_InitStruct = { 0 };

/\* GPIO Ports Clock Enable \*/

\_\_HAL\_RCC\_GPIOH\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

/\*Configure GPIO pin Output Level \*/

HAL\_GPIO\_WritePin(GPIOB, DW\_NSS\_Pin | DW\_WKUP\_Pin, GPIO\_PIN\_SET);

/\*Configure GPIO pin Output Level \*/

HAL\_GPIO\_WritePin(GPIOB, DW\_RESET\_Pin | LED\_STANDBY\_Pin | LED\_ERROR\_Pin,

GPIO\_PIN\_RESET);

/\*Configure GPIO pins : DW\_NSS\_Pin LED\_STANDBY\_Pin LED\_ERROR\_Pin \*/

GPIO\_InitStruct.Pin = DW\_NSS\_Pin | LED\_STANDBY\_Pin | LED\_ERROR\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

/\*Configure GPIO pins : DW\_RESET\_Pin DW\_WKUP\_Pin \*/

GPIO\_InitStruct.Pin = DW\_RESET\_Pin | DW\_WKUP\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_OD;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

/\*Configure GPIO pin : DW\_IRQn\_Pin \*/

GPIO\_InitStruct.Pin = DW\_IRQn\_Pin;

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_RISING;

GPIO\_InitStruct.Pull = GPIO\_PULLDOWN;

HAL\_GPIO\_Init(DW\_IRQn\_GPIO\_Port, &GPIO\_InitStruct);

/\* EXTI interrupt init\*/

HAL\_NVIC\_SetPriority(EXTI2\_IRQn, 1, 0);

HAL\_NVIC\_EnableIRQ(EXTI2\_IRQn);

}

/\* USER CODE BEGIN 4 \*/

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn get\_tx\_timestamp\_u64()

\*

\* @brief Get the TX time-stamp in a 64-bit variable.

\* /!\ This function assumes that length of time-stamps is 40 bits, for both TX and RX!

\*

\* @param none

\*

\* @return 64-bit value of the read time-stamp.

\*/

**static** uint64 get\_tx\_timestamp\_u64(**void**) {

uint8 ts\_tab[5];

uint64 ts = 0;

**int** i;

dwt\_readtxtimestamp(ts\_tab);

**for** (i = 4; i >= 0; i--) {

ts <<= 8;

ts |= ts\_tab[i];

}

**return** ts;

}

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn get\_rx\_timestamp\_u64()

\*

\* @brief Get the RX time-stamp in a 64-bit variable.

\* /!\ This function assumes that length of time-stamps is 40 bits, for both TX and RX!

\*

\* @param none

\*

\* @return 64-bit value of the read time-stamp.

\*/

**static** uint64 get\_rx\_timestamp\_u64(**void**) {

uint8 ts\_tab[5];

uint64 ts = 0;

**int** i;

dwt\_readrxtimestamp(ts\_tab);

**for** (i = 4; i >= 0; i--) {

ts <<= 8;

ts |= ts\_tab[i];

}

**return** ts;

}

/\*! ------------------------------------------------------------------------------------------------------------------

\* @fn final\_msg\_set\_ts()

\*

\* @brief Fill a given timestamp field in the final message with the given value. In the timestamp fields of the final

\* message, the least significant byte is at the lower address.

\*

\* @param ts\_field pointer on the first byte of the timestamp field to fill

\* ts timestamp value

\*

\* @return none

\*/

**static** **void** final\_msg\_set\_ts(uint8 \*ts\_field, uint64 ts) {

**int** i;

**for** (i = 0; i < FINAL\_MSG\_TS\_LEN; i++) {

ts\_field[i] = (uint8) ts;

ts >>= 8;

}

}

/\* USER CODE END 4 \*/

/\*\*

\* @brief This function is executed in case of error occurrence.

\* @retval None

\*/

**void** Error\_Handler(**void**) {

/\* USER CODE BEGIN Error\_Handler\_Debug \*/

/\* User can add his own implementation to report the HAL error return state \*/

\_\_disable\_irq();

**while** (1) {

}

/\* USER CODE END Error\_Handler\_Debug \*/

}

**#ifdef** USE\_FULL\_ASSERT

/\*\*

\* @brief Reports the name of the source file and the source line number

\* where the assert\_param error has occurred.

\* @param file: pointer to the source file name

\* @param line: assert\_param error line source number

\* @retval None

\*/

**void** assert\_failed(uint8\_t \*file, uint32\_t line)

{

/\* USER CODE BEGIN 6 \*/

/\* User can add his own implementation to report the file name and line number,

ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) \*/

/\* USER CODE END 6 \*/

}

**#endif** /\* USE\_FULL\_ASSERT \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* (C) COPYRIGHT STMicroelectronics \*\*\*\*\*END OF FILE\*\*\*\*/

Konfiguracja parametrów transmisji oraz wzory ramek zostały ustalone podobnie jak w responderze.

Program został skonstruowany w taki sposób, żeby inicjator usypiał się na około 1s a następnie wybudzał się i wykonywał procedurę skanowania po kolei wszystkich responderów.

Jest to działanie niezbędne w celu utrzymania niskiego poboru prądu układu- będzie on zasilany z niewielkiego akumulatora litowo-jonowego.

Dokładny czas spania układu różni się w zalezności od numeru inicjatora- działanie to ma na celu rozsynchronizowywanie się poszczególnych inicjatorów pomiędzy sobą, ponieważ transmisja nie może pracować jednocześnie na tym samym kanale pomiędzy dowolnymi urządzeniami w zasiągu. W przypadku kiedy mimo to nastąpi jednoczesna komunikacja więcej niż jednej pary inicjator-responder jedna z nich nie dojdzie do skutku. Jeżeli sytuacja taka powtórzy się 2-3 krotnie- inicjator zostaje zresetowany przez układ niezależnego watchdoga. O wybudzania MCU dba wbudowany weń zegar czasu rzeczywistego.

#### Wnioski

Układ został przetestowany przy użyciu 2 responderów oraz 3 inicjatorów. Umożliwia każdy z responderów jest w stanie w ciągu minimum 5 sekund zdystansować wszystkie 3 inicjatory.

Wykonał:

Bartosz Pracz