Smart Experience Sampling

Code Walkthrough

Mick Thelosen,

492994

06-01-2025

Introduction

This document is for made to give the user a better understanding of the software that runs on the S.E.S. system. It will go into detail about the way we have developed the code and how user can change the code based on the environment the system is used in.

Context

What is Smart Experience Sampling? S.E.S. is a project based on traditional experience sampling meant to measure the quality of the work environment. Meaning that people are given a question about the environment during their day. In case of IMEC (Our stakeholder) this meant that the users where given an analog clicker and got prompted with an instruction such as “Click every time you get distracted.” This way the people that set up the survey will get data about the workplace. However, this data is only an accumulation of the complete day. So, there is no time and place connected to the “clicks.” This is where S.E.S. comes in where we used Radio technologies to trilaterate the indoor location of the clicker the moment it gets clicked.

Code Specifications

- Code Language C
- STM32WB0x HAL driver
- DW1000 API Rev. 2

The code is developed using STM32CubeIDE and is programmed to run on the STM32WB07CCV6 development board (mb2119a). This development is coupled with a DWS1000 shield that carries the UWB chips used for the ranging. Using CubeIDE we generated the pinout configuration.

Processes

When the clicker is pressed messages are transmitted over both UWB and 2.4Ghz. The messages sent over 2.4Ghz are done using the proprietary radio drive by ST. The implementation goes as follows the Beacon (Listener) starts listening on its specified channel between 1-39. Using this code.

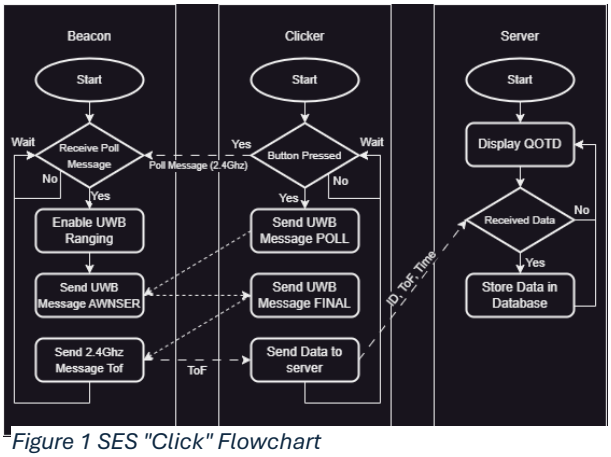

Figure 1 SES "Click" Flowchart

```
1. HAL_RADIO_ReceivePacketWithAck(BEACON_CHANNEL, RX_WAKEUP_TIME, receivedData,
2.                                sendAckData, RX_TIMEOUT, MAX_LL_PACKET_LENGTH, HAL_RADIO_Callback);
```

When there is no reception within the defined RX_TIMEOUT the radio driver will call this function.

```
1. void HAL_RADIO_CallbackRcvTimeout(RxStats_t *rxPacketStats) {
2.             HAL_RADIO_ReceivePacketWithAck(BEACON_CHANNEL, RX_WAKEUP_TIME, receivedData,
3.                                            sendAckData, RX_TIMEOUT, MAX_LL_PACKET_LENGTH, HAL_RADIO_Callback);
4. }
```

This restarts the listening process. When a message does get received, it will also restart the listening process, but it will also set the flag ranging = TRUE. The reasoning for choosing to use flags is because the radio driver code should not be interrupted so calling a function other than a new listening process in the Callback would disturb the radio driver. This way using flags ensure that the ranging only happens when it is not busy.

When ranging is set the DW1000_responder function is called this takes the beacon channel as argument to filter out incoming ranging messages meant for different beacons. The measurement is returned as a double however packages sent over the 2.4Ghz radio are limited to an array of 8-bit integers. So, we split up the meters resulting in a max of 255 meters and centimetres. This gets stored in the acknowledge data. The clicker will poll the beacon again after ranging to return this data.

```
1.  if (ranging == TRUE) {
2.          double distance = DW1000_responder(&dw1000, BEACON_CHANNEL);
3.          uint8_t meters = (uint8_t) distance;
4.          uint8_t centimeters = (uint8_t) ((distance - meters) * 100);
5.          if (meters != 0 || centimeters != 0) {
6.                  sendAckData[0] = 0xAE;
7.                  sendAckData[1] = 10;
8.                  sendAckData[10] = meters;
9.                  sendAckData[11] = centimeters;
10.                 ranging = FALSE;
11.         }
```

Each beacon uses its own 64-bit UUID as identification and the clicker stores the UUID's and their respective distance measurements. To get these UUID's we read out the address and then format it into the 8-bit array.

```
1. uint64_t ID = *((uint64_t*) UID64_BASE);
2. uint8_t i = 2;
3. sendData[1] = 8;
4. do {
5.          sendData[i++] = ID & 0xFF;
6. } while (ID >>= 8);
```

On the clickers side a similar flag structure is used. A flag to start ranging with surrounding beacons is set when the external interrupt bound to the switch on the clicker is called. This sets sendNewPacket = TRUE.

```
1. if (sendNewPacket) {
2.          sendNewPacket = FALSE;
3.          for (current_channel = 0; current_channel < MAX_BEACON_COUNT;
4.                                          current_channel++) {
5.                  sendData[0] = 0xDD;
6.                  HAL_RADIO_SendPacketWithAck(current_channel, TX_WAKEUP_TIME,
7.                                          sendData, receivedData, RX_TIMEOUT_ACK,
8.                                          MAX_LL_PACKET_LENGTH, HAL_RADIO_Callback);
9.                  HAL_Delay(40);
10.                 HAL_RADIO_TIMER_Tick();
11.                 if (beacon_ID[current_channel] != 0) {
12.                         DW1000_initiator(&dw1000, current_channel);
13.                         sendData[0] = 0xD1;
14.                         HAL_RADIO_SendPacketWithAck(current_channel,
15.                                 TX_WAKEUP_TIME, sendData, receivedData,
16.                                 RX_TIMEOUT_ACK,
17.                                 MAX_LL_PACKET_LENGTH, HAL_RADIO_Callback);
18.                         HAL_Delay(40);
19.                         HAL_RADIO_TIMER_Tick();
20.
21.                         }
22.                 }
```

This code goes and polls every channel and waits for a response from a beacon. The delay here is needed otherwise a new poll message would be tried to send using the radio drive whilst it is still trying to listen for responses. If a beacon_ID is registered at the current channel (Meaning the beacon responded) ranging will be started. And then the message to return the measured distance is set.

When an ID gets received the header of the message will be 0x1D. In the callback function of the clicker the ID is checked for duplicates and if it is not already registered it gets stored in the array of beacon_ID's and the beacon_count is recounted.

```
1. if (receivedData[0] == 0x1D) {
2.                         uint64_t ID = combine_to_uint64(receivedData, 2U);
3.                         uint8_t duplicate = 0;
4.                         for (uint8_t i = 0; i < 39; i++) {
5.                                 if (beacon_ID[i] == ID) {
6.                                         duplicate++;
7.                                 }
8.                         }
9.                         if (duplicate == 0) {
10.                                beacon_ID[current_channel] = ID;
11.                        }
12.                        beacon_count = 0;
13.                        for (uint8_t i = 0; i < 39; i++) {
14.                                if (beacon_ID[i] != 0) {
15.                                        beacon_count++;
16.                                }
17.                        }
18.            }
```

If a distance measurement is returned the distance gets combined to centimeters meaning the meters will be multiplied by 100 and the remaining centimetres will be added. This measurement will then be corrected using a polynomial error correction formula previously calibrated by testing different known measurements.

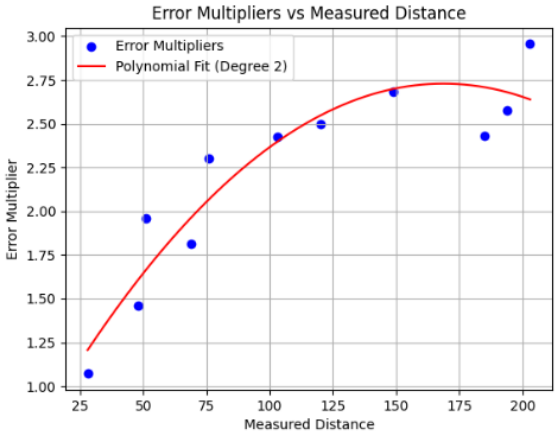

Figure 2 Polynomial Error correction.

```
1. else if (receivedData[0] == 0xAE) {
2.                         uint32_t measured_distance = (receivedData[10] * 100)
3.                                                 + receivedData[11];
4.                         uint32_t distance =
measured_distance * (0.5316 + 0.0259 * measured_distance + -0.0001 * (measured_distance * measured_distance));
5.                         distance_to_beacon[current_channel] = distance;
6.             }
```

When the clicker has polled all beacons channels 1-39. It checks if any beacons have responded with a distance measurement and if so, it gets added to the message that will be sent to the server.

```
1. if (beacon_count > 0) {
2.             receivedDistanceData[0] = 0xFF;
3.             receivedDistanceData[1] = 10 * beacon_count;
4.             for (uint8_t i = 2, j = 0; j + 1 < MAX_BEACON_COUNT; j++) {
5.                         if (distance_to_beacon[j] != 0 && beacon_ID[j] != 0) {
6.                                     uint8_t uuid[8];
7.                                     split_from_uint64(beacon_ID[j], uuid);
8.                                                 for (uint8_t k = i, l = 0; k < i + 8; k++, l++) {
9.                                                             receivedDistanceData[k] = uuid[l];
10.                                                                         }
11.                                                 receivedDistanceData[i + 8] = (uint8_t) distance_to_beacon[j] / 100;
12.                                                 receivedDistanceData[i + 9] = distance_to_beacon[j]
13.                                                                         - (receivedDistanceData[i + 8] * 100);
14.                                                                         i += 10;
15.                                                             }
16.                                                 }
17.                                                 HAL_RADIO_SendPacketWithAck(SERVER_CHANNEL,
19.                                                 TX_WAKEUP_TIME, receivedDistanceData, receivedData,
20.                                                 RX_TIMEOUT_ACK,
21.                                                 MAX_LL_PACKET_LENGTH, HAL_RADIO_Callback);
22.                                                 HAL_Delay(40);
23.                                                 HAL_RADIO_TIMER_Tick();
24.                                     }
```

The data is formatted where the first byte is the header the second byte is the data size which is the beacon_count * 10 because the data per beacon consists of the UUID which is 8 bytes and the distance measurements and those are 2 bytes. This data is then sent to the server over 2.4Ghz. This is always done over channel 0. The server also acts a beacon.

DW1000 Driver

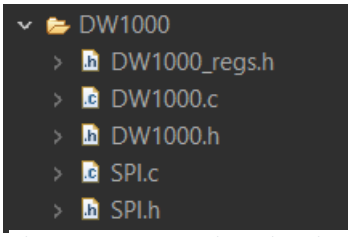

Figure 4 DW1000 Driver File Hierarchy

The API provided with the DW1000 chip did not work consistently in our use case. To fix this we started debugging using a logic analyzer and found out that the SPI functions from the HAL library did not work correctly and resulted in the DW1000 chip not responding.

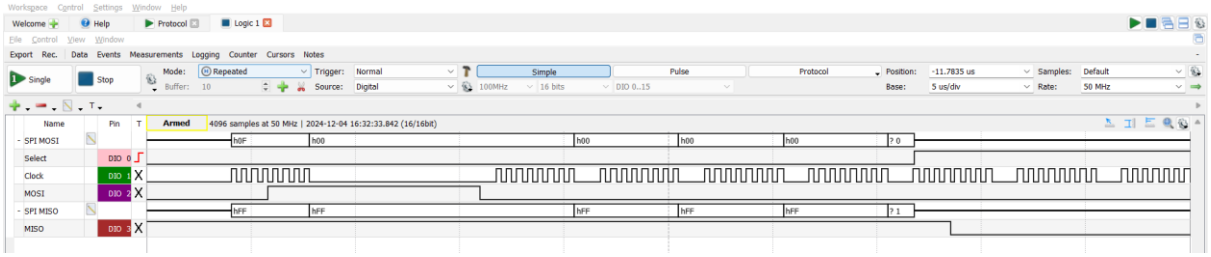

Figure 3 No response on MISO DW1000

To fix this we implemented our own version of a SPI read and write function which is also located in this driver.

```
1. uint32_t SPI_write8(SPI_HandleTypeDef* spi, const uint8_t* buffer, uint32_t size, uint32_t timeout) {
2.              spi->Instance->CR1 |= 0x00000040UL;
3.              uint64_t start = uwTick;
4.              uint32_t i = 0;
5.              for (; i < size; i++) {
6.                              while (!(spi->Instance->SR & 0x00000002UL))    { if ( uwTick - start > timeout) { goto
SPI_master_write8_end; } }
7.                              *((volatile uint8_t*)&spi->Instance->DR) = buffer[i];
8.              }
9.              while (spi->Instance->SR & 0x00000080UL)       { if ( uwTick - start > timeout) { goto
SPI_master_write8_end; } }
10.             i++; SPI_master_write8_end:
11.             spi->Instance->CR1 &= ~0x00000040UL;
12.             return i;
13. }
```

To make this work with the existing API, we created a function that wraps the SPI write function and formats the SPI message according to the message format specified by Decawave.

Bit number:	7	6	5	4	3	2	1	0	
Meaning:	Operation: 0 = Read 1 = Write	Bit = 1, says sub-index is present	Register file ID – Range 0x00 to 0x3F (64 locations)						Transaction Header Octet 1
	Extended Address: 0 = no	7-bit Register File sub-address, range 0x00 to 0x7F (128 byte locations)							Octet 2

Figure 5 SPI Message format.

```
1. static inline void DW1000_read_reg(DW1000_t *dw1000, uint32_t reg,
2.                                     uint16_t offset, uint8_t *buffer, uint32_t size) {
3.             __disable_irq();
4.             reg &= 0x3FU;
5.             HAL_GPIO_WritePin(dw1000->NSS_port, dw1000->NSS_pin, 0);
6.             if (offset & 0xFF00) {
7.                             reg |= 0x8040U | (offset << 8);
8.                             SPI_write8(dw1000->spi, (void*) ®, 3, DW1000_TIMEOUT);
9.             } else if (offset) {
10.                            reg |= 0x40U | (offset << 8);
11.                            SPI_write8(dw1000->spi, (void*) ®, 2, DW1000_TIMEOUT);
12.            } else {
13.                            SPI_write8(dw1000->spi, (void*) ®, 1, DW1000_TIMEOUT);
14.            }
15.            SPI_read8(dw1000->spi, buffer, size, DW1000_TIMEOUT);
16.            HAL_GPIO_WritePin(dw1000->NSS_port, dw1000->NSS_pin, 1);
17.            __enable_irq();
18. }
```

This has resulted in consistent responses from the DW1000 chip, and we have not had any issues.

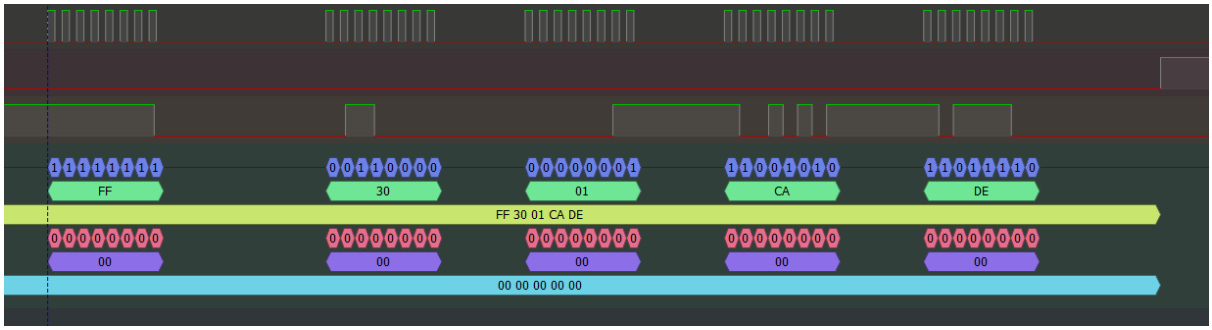

Figure 6 Correct Response "DECA0130".

Figures