XM125 I²C Ref App Breathing

User Guide

XM125 I²C Ref App Breathing

User Guide

Author: Acconeer AB

Version:a121-v1.5.0

Acconeer AB January 19, 2024

Contents

1 Acconeer SDK Documentation Overview

To better understand what SDK document to use, a summary of the documents are shown in the table below.

Table 1: SDK document overview.

Name	Description	When to use
	RSS API documentation (html)	
rss_api	The complete C API documentation.	- RSS application implementation - Understanding RSS API functions
	User guides (PDF)	
A121 Assembly Test	Describes the Acconeer assembly test functionality.	- Bring-up of HW/SW - Production test implementation
A121 Breathing Reference Application	Describes the functionality of the Breathing Reference Application.	- Working with the Breathing Reference Application
A121 Distance Detector	Describes usage and algorithms of the Distance Detector.	- Working with the Distance Detector
A121 SW Integration	Describes how to implement each integration function needed to use the Acconeer sensor.	- SW implementation of custom HW integration
A121 Presence Detector	Describes usage and algorithms of the Presence Detector.	- Working with the Presence Detector
A121 Smart Presence Reference Application	Describes the functionality of the Smart Presence Reference Application.	- Working with the Smart Presence Reference Application
A121 Sparse IQ Service	Describes usage of the Sparse IQ Service.	- Working with the Sparse IQ Service
A121 Tank Level Reference Application	Describes the functionality of the Tank Level Reference Application.	- Working with the Tank Level Reference Application
A121 Touchless Button Reference Application	Describes the functionality of the Touchless Button Reference Application.	- Working with the Touchless Button Reference Application
A121 STM32CubeIDE	Describes the flow of taking an Acconeer SDK and integrate into STM32CubeIDE.	- Using STM32CubeIDE
A121 Raspberry Pi Software	Describes how to develop for Raspberry Pi.	- Working with Raspberry Pi
A121 Ripple	Describes how to develop for Ripple.	- Working with Ripple on Raspberry Pi
XM125 Software	Describes how to develop for XM125.	- Working with XM125
XM126 Software	Describes how to develop for XM126.	- Working with XM126
I2C Distance Detector	Describes the functionality of the I2C Distance Detector Application.	- Working with the I2C Distance Detector Application
I2C Presence Detector	Describes the functionality of the I2C Presence Detector Application.	- Working with the I2C Presence Detector Application
I2C Breathing Reference Application	Describes the functionality of the I2C Breathing Reference Application.	- Working with the I2C Breathing Reference Application
	Handbook (PDF)	
Handbook	Describes different aspects of the Acconeer offer, for example radar principles and how to configure	- To understand the Acconeer sensor - Use case evaluation
	Readme (txt)	
README	Various target specific information and links	- After SDK download

2 I²C Ref App Breathing

The I²C Ref App Breathing is an application that implements the Acconeer Ref App Breathing with a register based I²C interface.

The functionality of the ref app breathing is described in *A121 Breathing Reference Application User Guide.pdf* or in Acconeer Docs.

Note: Some of the registers like **start** and **end** have a different unit in the I²C Ref App Breathing, millimeters instead of meters, to make it easier to handle the register values as integers.

2.1 Usage

The module must be ready before the host starts I²C communication.

The module will enter ready state by following this procedure.

- Set **WAKE_UP** pin of the module HIGH.
- Wait for module to be ready, this is indicated by the **MCU_INT** pin being HIGH.
- Start I²C communication.

The module will enter a low power state by following this procedure.

- Wait for module to be ready, this is indicated by the **MCU_INT** pin being HIGH.
- Set the **WAKE_UP** pin of the module LOW.
- Wait for ready signal, the **MCU_INT** pin, to become LOW.

2.1.1 Read App Status

The status of the module can be acquired by reading the *App Status* register, The most important bits are the **Busy** and **Error** bits.

The **Busy** bit must not be set when a new command is written. If any of the **Error** bits are set the module will not accept any commands except the **RESET_MODULE** command.

2.1.2 Writing a command

A command is written to the *Command* register. When a command is written the **Busy** bit in the *App Status* register is set and it will be cleared automatically when the command has finished.

2.1.3 Setup and Start Application

Before the module can perform breathing detection it must be configured. The following steps is an example of how this can be achieved.

Note: The configuration parameters can not be changed after a **APPLY_CONFIGURATION** command. If reconfiguration is needed the module must be restarted by writing **RESET_MODULE** to the *Command* register.

- Power on module
- Read *App Status* register and verify that neither **Busy** nor **Error** bits are set.
- Write configuration to configuration registers, for example *Start* register and *End* register.
- Write **APPLY_CONFIGURATION** to *Command* register.
- Poll *App Status* until **Busy** bit is cleared.
- Verify that no **Error** bits are set in the *App Status* register.
- Write **START_APP** to *Command* register.
- Poll *App Status* until **Busy** bit is cleared.
- Verify that no **Error** bits are set in the *App Status* register.
- Read *App Result* register
  - If **RESULT_READY** is set a new breathing result is provided.

  - If **APP_ERROR** is set an error has occurred, restart module with the **RESET_MODULE** command.
  - If result was ready, the breathing rate can be read in the *Breathing Rate* register. In any state the app state can be read in the *App State* register.

2.1.4 Stop and Restart Application

The application can be stopped and restarted.

The following steps is an example of how to stop the application.

- Read *App Status* register and verify that neither **Busy** nor **Error** bits are set.
- Write **STOP_APP** to *Command* register.
- Poll *App Status* until **Busy** bit is cleared.
- Verify that no **Error** bits are set in the *App Status* register.

The following steps is an example of how to re-start the application.

- Read *App Status* register and verify that neither **Busy** nor **Error** bits are set.
- Write **START_APP** to *Command* register.
- Poll *App Status* until **Busy** bit is cleared.
- Verify that no **Error** bits are set in the *App Status* register.

2.2 Advanced Usage

2.2.1 Debug UART logs

UART logging can be enabled on the DEBUG UART by writing **ENABLE_UART_LOGS** to the *Command* register.

The application configuration can be logged on the UART by writing **LOG_CONFIGURATION** to the *Command* register.

UART logging can be disabled by writing **DISABLE_UART_LOGS** to the *Command* register.

2.2.2 Reset Module

The module can be restarted by writing **RESET_MODULE** to the *Command* register.

After the restart the application must be configured again.

3 Register Protocol

3.1 I²C Slave Address

The default slave address is 0x52.

3.2 Protocol Byte Order

Both register address, 16-bit, and register data, 32-bit, are sent in big endian byte order.

3.2.1 I²C Write Register(s)

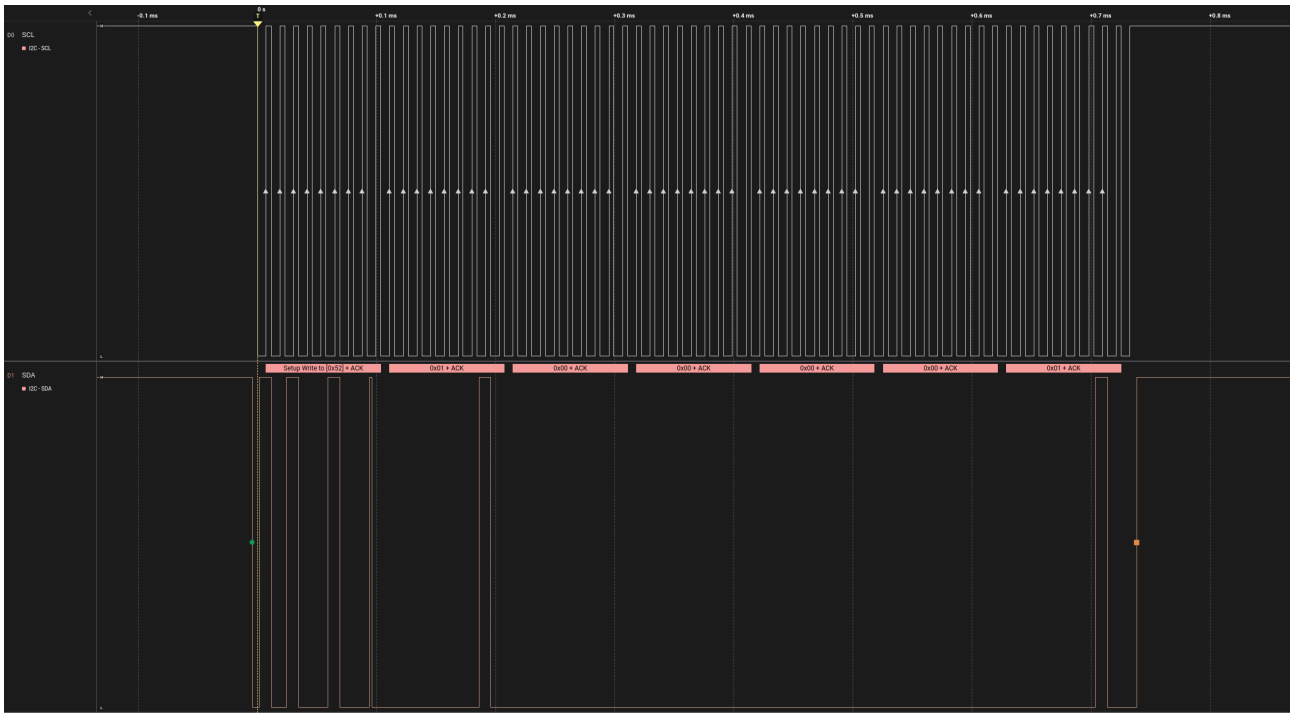
A write register operation consists of an I²C write of two address bytes and four data bytes for each register to write. Several registers can be written in the same I²C transaction, the register address will be incremented by one for each four data bytes.

Example 1: Writing six bytes will write one register, two address bytes and four data bytes.

Example 2: Writing 18 bytes will write four registers, two address bytes and 16 data bytes.

Example operation, write 0x11223344 to address 0x0025.

Description	Data
I²C Start Condition	
Slave Address + Write	0x52 + W
Address to slave [15:8]	0x00
Address to slave [7:0]	0x25
Data to slave [31:24]	0x11
Data to slave [23:16]	0x22
Data to slave [15:8]	0x33
Data to slave [7:0]	0x44
I²C Stop Condition	



Example Waveform: Write register with address 0x0100, the data sent from the master to the slave is 0x00000001

3.2.2 I²C Read Register(s)

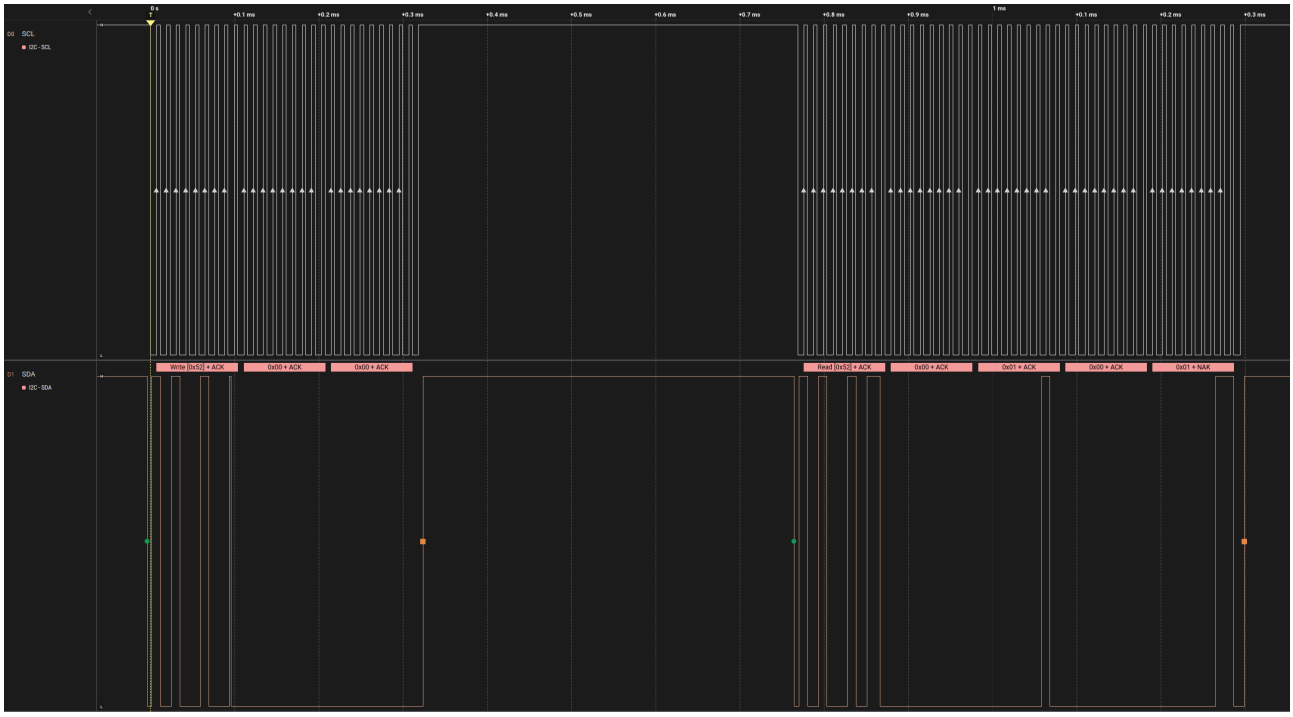
A read register operation consists of an I²C write of two address bytes followed by an I²C read of four data bytes for each register to read. Several registers can be read in the same I²C transaction, the register address will be incremented by one for each four data bytes.

Example 1: Writing two bytes and reading four bytes will read one register.

Example 2: Writing two bytes and reading 16 bytes will read four registers.

Example operation, read 0x12345678 from address 0x0003.

Description	Data
I²C Start Condition	
Slave Address + Write	0x52 + W
Address to slave [15:8]	0x00
Address to slave [7:0]	0x03
I²C Stop Condition	
I²C Start Condition	
Slave Address + Read	0x52 + R
Data from slave [31:24]	0x12
Data from slave [23:16]	0x34
Data from slave [15:8]	0x56
Data from slave [7:0]	0x78
I²C Stop Condition	



Example Waveform: Read register with address 0, the data sent from the slave to the master is 0x00010001

3.3 Register Protocol - Low Power Mode

3.3.1 I2C Communication with Low Power Mode

Low power example

Low Power Example: Magnification of Wake up, Setup Ref App Breathing, Power down

4 File Structure

The I²C Ref App Breathing application consists of the following files.

```
├── Src
│   ├── applications
│   │   └── i2c
│   │       ├── acc_reg_protocol.c
│   │       ├── ref_app_breathing_reg_protocol_access.c
│   │       ├── ref_app_breathing_reg_protocol.c
│   │       ├── i2c_application_system_stm32.c
│   │       └── i2c_ref_app_breathing.c
│   └── use_cases
│       └── reference_apps
│           └── ref_app_breathing.c
└── Inc
    ├── acc_reg_protocol.h
    ├── ref_app_breathing_reg_protocol.h
    ├── i2c_application_system.h
    └── i2c_ref_app_breathing.h
```

- **acc_reg_protocol.c** A generic protocol handler implementation.
- **ref_app_breathing_reg_protocol.c** The specific register protocol setup for the I²C Ref App Breathing.
- **ref_app_breathing_reg_protocol_access.c** The register read and write access functions for the I²C Ref App Breathing.
- **i2c_application_system_stm32.c** System functions, such as I²C handling, GPIO control and low power state
- **i2c_ref_app_breathing.c** The I²C Ref App Breathing application.
- **ref_app_breathing.c** The Ref App Breathing application.

5 Embedded Host Example

This is an example implementation of the host read and write register functions using the STM32 SDK.

5.1 Register Read/Write functions

```
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>

#include "ref_app_breathing_reg_protocol.h"

// Use 1000ms timeout
#define I2C_TIMEOUT_MS 1000

// The STM32 uses the i2c address shifted one position
// to the left (0x52 becomes 0xa4)
#define I2C_ADDR 0xa4

// The register address length is two bytes
#define REG_ADDRESS_LENGTH 2

// The register data length is four bytes
#define REG_DATA_LENGTH 4


/**
 * @brief Read register value over I2C
 *
```

```
 * @param[in] reg_addr The register address to read
 * @param[out] reg_data The read register data
 * @returns true if successful
 */
bool read_register(uint16_t reg_addr, uint32_t *reg_data)
{
    HAL_StatusTypeDef status = HAL_OK;

    uint8_t transmit_data[REG_ADDRESS_LENGTH];

    transmit_data[0] = (reg_addr >> 8) & 0xff;
    transmit_data[1] = (reg_addr >> 0) & 0xff;

    status = HAL_I2C_Master_Transmit(&STM32_I2C_HANDLE, I2C_ADDR,
                                     transmit_data, REG_ADDRESS_LENGTH,
                                     I2C_TIMEOUT_MS);
    if (status != HAL_OK)
    {
        return false;
    }

    uint8_t receive_data[REG_DATA_LENGTH];

    status = HAL_I2C_Master_Receive(&STM32_I2C_HANDLE, I2C_ADDR,
                                    receive_data, REG_DATA_LENGTH,
                                    I2C_TIMEOUT_MS);
    if (status != HAL_OK)
    {
        return false;
    }


    // Convert bytes to uint32_t
    uint32_t val = receive_data[0];
    val = val << 8;
    val |= receive_data[1];
    val = val << 8;
    val |= receive_data[2];
    val = val << 8;
    val |= receive_data[3];
    *reg_data = val;

    return true;
}


/**
 * @brief Write register value over I2C
 *
 * @param[in] reg_addr The register address to write
 * @param[in] reg_data The register data to write
 * @returns true if successful
 */
bool write_register(uint16_t reg_addr, uint32_t reg_data)
{
    HAL_StatusTypeDef status = HAL_OK;

    uint8_t transmit_data[REG_ADDRESS_LENGTH + REG_DATA_LENGTH];

    // Convert uint16_t address to bytes
    transmit_data[0] = (reg_addr >> 8) & 0xff;
```

```
    transmit_data[1] = (reg_addr >> 0) & 0xff;
    // Convert uint32_t reg_data to bytes
    transmit_data[2] = (reg_data >> 24) & 0xff;
    transmit_data[3] = (reg_data >> 16) & 0xff;
    transmit_data[4] = (reg_data >> 8) & 0xff;
    transmit_data[5] = (reg_data >> 0) & 0xff;

    status = HAL_I2C_Master_Transmit(&STM32_I2C_HANDLE, I2C_ADDR,
                                     transmit_data,
                                     REG_ADDRESS_LENGTH + REG_DATA_LENGTH,
                                     I2C_TIMEOUT_MS);
    if (status != HAL_OK)
    {
        return false;
    }

    return true;
}
```

5.2 Application setup functions

```
#include "ref_app_breathing_reg_protocol.h"

/**
 * @brief Test if configuration of application is OK
 *
 * @returns true if successful
 */
bool configuration_ok(void)
{
    uint32_t status = 0
    if (!read_register(REF_APP_BREATHING_REG_APP_STATUS_ADDRESS, &status))
    {
        //ERROR
        return false;
    }

    uint32_t config_ok_mask =
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_RSS_REGISTER_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_CONFIG_CREATE_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_SENSOR_CREATE_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_SENSOR_CALIBRATE_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_APP_CREATE_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_APP_BUFFER_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_SENSOR_BUFFER_OK_MASK |
         REF_APP_BREATHING_REG_APP_STATUS_FIELD_CONFIG_APPLY_OK_MASK;

   if (status != config_ok_mask)
   {
        //ERROR
       return false;
   }

   return true;
}


/**
 * @brief Wait for application not busy
 *
```

```
 * @returns true if successful
 */
bool wait_not_busy(void)
{
    uint32_t status = 0
    do
    {
        if (!read_register(REF_APP_BREATHING_REG_APP_STATUS_ADDRESS, &status
           ))
        {
            //ERROR
            return false;
        }
    } while((status & REF_APP_BREATHING_REG_APP_STATUS_FIELD_BUSY_MASK) !=
       0);

    return true;
}


bool example_setup_and_start(void)
{
    // Set start at 1000mm
    if (!write_register(REF_APP_BREATHING_REG_START_ADDRESS, 1000))
    {
        //ERROR
        return false;
    }
    // Set end at 5000mm
    if (!write_register(REF_APP_BREATHING_REG_END_ADDRESS, 5000))
    {
        //ERROR
        return false;
    }

    // Apply configuration
    if (!write_register(
            REF_APP_BREATHING_REG_COMMAND_ADDRESS,
            REF_APP_BREATHING_REG_COMMAND_ENUM_APPLY_CONFIGURATION))
    {
        //ERROR
        return false;
    }

    // Wait for the configuration to be done
    if (!wait_not_busy())
    {
        //ERROR
        return false;
    }

    // Test if configration of application was OK
    if (!configuration_ok())
    {
        //ERROR
        return false;
    }

    // Start application
    if (!write_register(REF_APP_BREATHING_REG_COMMAND_ADDRESS,
                        REF_APP_BREATHING_REG_COMMAND_ENUM_START_APP))
```

```
    {
        //ERROR
        return false;
    }

    // Wait for command be done
    if (!wait_not_busy())
    {
        //ERROR
        return false;
    }

    // Read application result
    uint32_t result;
    if (!read_register(REF_APP_BREATHING_REG_BREATHING_RESULT_ADDRESS, &
       result))
    {
        //ERROR
        return false;
    }

    // Was result ready?

    bool result_ready = (result &
       REF_APP_BREATHING_REG_BREATHING_RESULT_FIELD_RESULT_READY_MASK) != 0;

    // Print peak if found
    if (result_ready)
    {
        uint32_t breathing_rate;
        if (read_register(REF_APP_BREATHING_REG_BREATHING_RATE_ADDRESS, &
           breathing_rate))
        {
            printf("Breathing rate: %" PRIu32 " bpm\n", breathing_rate);
        }
        else
        {
            //ERROR
            return false;
        }
    }

    return true;
}
```

6 Registers

6.1 Register Map

Address	Register Name	Type
0x0000	Version	Read Only
0x0001	Protocol Status	Read Only
0x0002	Measure Counter	Read Only
0x0003	App Status	Read Only
0x0010	Breathing Result	Read Only
0x0011	Breathing Rate	Read Only
0x0012	App State	Read Only
0x0040	Start	Read / Write
0x0041	End	Read / Write
0x0042	Num Distances To Analyze	Read / Write
0x0043	Distance Determination Duration S	Read / Write
0x0044	Use Presence Processor	Read / Write
0x0045	Lowest Breathing Rate	Read / Write
0x0046	Highest Breathing Rate	Read / Write
0x0047	Time Series Length S	Read / Write
0x0048	Frame Rate	Read / Write
0x0049	Sweeps Per Frame	Read / Write
0x004a	Hwaas	Read / Write
0x004b	Profile	Read / Write
0x004c	Intra Detection Threshold	Read / Write
0x0100	Command	Write Only

6.2 Register Descriptions

6.2.1 Version

Address	0x0000
Access	Read Only
Register Type	field
Description	Get the RSS version.

Bitfield	Pos	Width	Mask
MAJOR	16	16	0xffff0000
MINOR	8	8	0x0000ff00
PATCH	0	8	0x000000ff

MAJOR - Major version number

MINOR - Minor version number

PATCH - Patch version number

6.2.2 Protocol Status

Address	0x0001
Access	Read Only
Register Type	field
Description	Get protocol error flags.

Bitfield	Pos	Width	Mask
PROTOCOL_STATE_ERROR	0	1	0x00000001
PACKET_LENGTH_ERROR	1	1	0x00000002

ADDRESS_ERROR	2	1	0x00000004
WRITE_FAILED	3	1	0x00000008
WRITE_TO_READ_ONLY	4	1	0x00000010

PROTOCOL_STATE_ERROR - Protocol state error

PACKET_LENGTH_ERROR - Packet length error

ADDRESS_ERROR - Register address error

WRITE_FAILED - Write register failed

WRITE_TO_READ_ONLY - Write to read only register

6.2.3 Measure Counter

Address	0x0002
Access	Read Only
Register Type	uint
Description	Get the measure counter, the number of measurements performed since restart.

6.2.4 App Status

Address	0x0003
Access	Read Only
Register Type	field
Description	Get application status flags.

Bitfield	Pos	Width	Mask
RSS_REGISTER_OK	0	1	0x00000001
CONFIG_CREATE_OK	1	1	0x00000002
SENSOR_CREATE_OK	2	1	0x00000004
SENSOR_CALIBRATE_OK	3	1	0x00000008
APP_CREATE_OK	4	1	0x00000010
APP_BUFFER_OK	5	1	0x00000020
SENSOR_BUFFER_OK	6	1	0x00000040
CONFIG_APPLY_OK	7	1	0x00000080
RSS_REGISTER_ERROR	16	1	0x00010000
CONFIG_CREATE_ERROR	17	1	0x00020000
SENSOR_CREATE_ERROR	18	1	0x00040000
SENSOR_CALIBRATE_ERROR	19	1	0x00080000
APP_CREATE_ERROR	20	1	0x00100000
APP_BUFFER_ERROR	21	1	0x00200000
SENSOR_BUFFER_ERROR	22	1	0x00400000
CONFIG_APPLY_ERROR	23	1	0x00800000
APP_ERROR	28	1	0x10000000
BUSY	31	1	0x80000000

RSS_REGISTER_OK - RSS register OK

CONFIG_CREATE_OK - Configuration create OK

SENSOR_CREATE_OK - Sensor create OK

SENSOR_CALIBRATE_OK - Sensor calibrate OK

APP_CREATE_OK - Application create OK

APP_BUFFER_OK - Application get buffer size OK

SENSOR_BUFFER_OK - Memory allocation of buffer OK

CONFIG_APPLY_OK - Application configuration apply OK

RSS_REGISTER_ERROR - RSS register error

CONFIG_CREATE_ERROR - Configuration create error

SENSOR_CREATE_ERROR - Sensor create error

SENSOR_CALIBRATE_ERROR - Sensor calibrate error

APP_CREATE_ERROR - Application create error

APP_BUFFER_ERROR - Application get buffer size error

SENSOR_BUFFER_ERROR - Memory allocation of sensor buffer error

CONFIG_APPLY_ERROR - Application configuration apply error

APP_ERROR - Application error occured, restart necessary

BUSY - Application busy

6.2.5 Breathing Result

Address	0x0010
Access	Read Only
Register Type	field
Description	The result from the breathing reference application.

Bitfield	Pos	Width	Mask
RESULT_READY	0	1	0x00000001
RESULT_READY_STICKY	1	1	0x00000002
TEMPERATURE	16	16	0xffff0000

RESULT_READY - Indication when a new breathing rate result is produced

RESULT_READY_STICKY - Indication when a new breathing rate result is produced, sticky bit with clear on read

TEMPERATURE - Temperature in sensor during measurement (in degree Celsius). Note that it has poor absolute accuracy and should only be used for relative temperature measurements.

6.2.6 Breathing Rate

Address	0x0011
Access	Read Only
Register Type	uint
Unit	bpm
Description	The breathing rate. 0 if no breathing rate available. Note: This value is a factor 1000 larger than the RSS value.

6.2.7 App State

Address	0x0012
Access	Read Only
Register Type	enum
Description	The current state of the application.

Enum	Value
INIT	0

NO_PRESENCE	1
INTRA_PRESENCE	2
DETERMINE_DISTANCE	3
ESTIMATE_BREATHING_RATE	4

INIT - Initiating

NO_PRESENCE - No presence detected

INTRA_PRESENCE - Too high intra presence detected

DETERMINE_DISTANCE - Determine distance to presence

ESTIMATE_BREATHING_RATE - Estimate breathing rate

6.2.8 Start

Address	0x0040
Access	Read / Write
Register Type	uint
Unit	mm
Description	The start point of measurement interval in millimeters. Note: This value is a factor 1000 larger than the RSS value.
Default Value	300

6.2.9 End

Address	0x0041
Access	Read / Write
Register Type	uint
Unit	mm
Description	The end point of measurement interval in millimeters. Note: This value is a factor 1000 larger than the RSS value.
Default Value	1500

6.2.10 Num Distances To Analyze

Address	0x0042
Access	Read / Write
Register Type	uint
Description	Number of distance points to analyze in breathing.
Default Value	3

6.2.11 Distance Determination Duration S

Address	0x0043
Access	Read / Write
Register Type	uint
Description	Time to determine distance to presence in seconds.
Default Value	5

6.2.12 Use Presence Processor

Address	0x0044
Access	Read / Write
Register Type	bool
Description	Use presence detector to determine distance to motion.

Default Value	True

6.2.13 Lowest Breathing Rate

Address	0x0045
Access	Read / Write
Register Type	uint
Description	Lowest anticipated breathing rate in breaths per minute.
Default Value	6

6.2.14 Highest Breathing Rate

Address	0x0046
Access	Read / Write
Register Type	uint
Description	Highest anticipated breathing rate in breaths per minute.
Default Value	60

6.2.15 Time Series Length S

Address	0x0047
Access	Read / Write
Register Type	uint
Description	Length of time series in seconds.
Default Value	20

6.2.16 Frame Rate

Address	0x0048
Access	Read / Write
Register Type	uint
Unit	mHz
Description	The presence detector frame rate. Note: This value is a factor 1000 larger than the RSS value.
Default Value	10000

6.2.17 Sweeps Per Frame

Address	0x0049
Access	Read / Write
Register Type	uint
Description	The number of sweeps that will be captured in each frame (measurement).
Default Value	16

6.2.18 Hwaas

Address	0x004a
Access	Read / Write
Register Type	uint
Description	The hardware accelerated average samples (HWAAS).
Default Value	32

6.2.19 Profile

Address	0x004b
Access	Read / Write
Register Type	enum
Description	The profile to use.
Default Value	PROFILE3

Enum	Value
PROFILE1	1
PROFILE2	2
PROFILE3	3
PROFILE4	4
PROFILE5	5

PROFILE1 - Profile 1

PROFILE2 - Profile 2

PROFILE3 - Profile 3

PROFILE4 - Profile 4

PROFILE5 - Profile 5

6.2.20 Intra Detection Threshold

Address	0x004c
Access	Read / Write
Register Type	uint
Description	The threshold for detecting faster movements inside frames. Note: This value is a factor 1000 larger than the RSS value.
Default Value	6000

6.2.21 Command

Address	0x0100
Access	Write Only
Register Type	enum
Description	Execute command.

Enum	Value
APPLY_CONFIGURATION	1
START_APP	2
STOP_APP	3
ENABLE_UART_LOGS	32
DISABLE_UART_LOGS	33
LOG_CONFIGURATION	34
RESET_MODULE	1381192737

APPLY_CONFIGURATION - Apply the configuration

START_APP - Start the breathing application

STOP_APP - Stop the breathing application

ENABLE_UART_LOGS - DEBUG: Enable UART Logs

DISABLE_UART_LOGS - DEBUG: Disable UART Logs

LOG_CONFIGURATION - DEBUG: Print application configuration to UART

RESET_MODULE - Reset module, needed to make a new configuration

7 Disclaimer

The information herein is believed to be correct as of the date issued. Acconeer AB (“Acconeer”) will not be responsible for damages of any nature resulting from the use or reliance upon the information contained herein. Acconeer makes no warranties, expressed or implied, of merchantability or fitness for a particular purpose or course of performance or usage of trade. Therefore, it is the user’s responsibility to thoroughly test the product in their particular application to determine its performance, efficacy and safety. Users should obtain the latest relevant information before placing orders.

Unless Acconeer has explicitly designated an individual Acconeer product as meeting the requirement of a particular industry standard, Acconeer is not responsible for any failure to meet such industry standard requirements.

Unless explicitly stated herein this document Acconeer has not performed any regulatory conformity test. It is the user’s responsibility to assure that necessary regulatory conditions are met and approvals have been obtained when using the product. Regardless of whether the product has passed any conformity test, this document does not constitute any regulatory approval of the user’s product or application using Acconeer’s product.

Nothing contained herein is to be considered as permission or a recommendation to infringe any patent or any other intellectual property right. No license, express or implied, to any intellectual property right is granted by Acconeer herein.

Acconeer reserves the right to at any time correct, change, amend, enhance, modify, and improve this document and/or Acconeer products without notice.

This document supersedes and replaces all information supplied prior to the publication hereof.