Assembly Test

User Guide

Assembly Test

User Guide

Author: Acconeer AB

Version:v2.0.0

Acconeer AB December 2, 2019

Contents

1 Assembly Test

The assembly test can be used for two different purposes which both aim to verify the assembly of the sensor into a custom PCB:

- Flexible and accurate production test to maintain good quality in PCB assembly of the sensor
- A tool for easy bring-up of HW/SW to decrease the time-to-market

The assembly test contains several tests, where each test is focused on testing a particular functionality. The test purpose and interpretation of the test result is described in this document. The tests are structured to provide clear feedback on what in the assembly, implementation, and/or integration is failing.

Acconeer provides an example of how to use the assembly test: example_assembly_test.c

1.1 Disclaimer

Assembly test should not be used on A111 with batch number 10467, 10457 or 10178 (also when mounted on XR111 and XR112).

2 Setting up the Assembly Test

2.1 Initializing the System

The Radar System Software (RSS) must be activated before any other calls are done. The activation requires a pointer to an acc_hal_t struct which contains information on the hardware integration and function pointers to hardware driver functions that are needed by RSS. See chapter 4 in the document "HAL Integration User Guide" for more information on how to integrate to the driver layer and populate the hal struct.

In Acconeer's example integration towards STM32 and the drivers generated by the STM32Cube tool, there is a function acc_hal_integration_get_implementation to obtain the hal struct.

```
acc_hal_t hal = acc_hal_integration_get_implementation();

if (!acc_rss_activate(&hal))
{
    /* Handle error */
}
```

The corresponding code looks slightly different in software packages for the Raspberry Pi and other software packages from Acconeer where peripheral drivers for the host are included. The driver layer is first initialized by calling acc_driver_hal_init. The hal struct is then obtained with the function acc_driver_hal_get_implementation.

```
if (!acc_driver_hal_init())
{
    /* Handle error */
}

acc_hal_t hal = acc_driver_hal_get_implementation();

if (!acc_rss_activate(&hal))
{
    /* Handle error */
}
```

2.2 Assembly Test Configuration

Before the assembly test can be executed, we must prepare a test configuration. First a configuration is created.

```
acc_rss_assembly_test_configuration_t assembly_test_configuration =
    acc_rss_assembly_test_configuration_create();

if (assembly_test_configuration == NULL)
{
    /* Handle error */
}
```

All tests are enabled by default but the application can choose to run a subset of the tests by disabling tests in the configuration. acc_rss_assembly_test.h provides functions to disable individual tests

3 Running the Assembly Test

```
acc_rss_assembly_test_result_t test_results[
    ACC_RSS_ASSEMBLY_TEST_MAX_NUMBER_OF_TESTS];
uint16_t                       nr_of_test_results =
    ACC_RSS_ASSEMBLY_TEST_MAX_NUMBER_OF_TESTS;
bool                           success;

success = acc_rss_assembly_test(assembly_test_configuration, test_results, &
    nr_of_test_results);

if (success)
{
    for (uint16_t i = 0; i < nr_of_test_results; i++)
    {
        const bool test_passed = test_results[i].test_passed;
        printf("Name: %s, result: %s\n", test_results[i].test_name,
            test_passed ? "Pass" : "Fail");
    }
}
else
{
    fprintf(stderr, "Assembly test: Failed to complete\n");
}
```

4 Deactivation and Destroy

After the test is finished, the configuration needs to be destroyed to release memory

```
acc_rss_assembly_test_configuration_destroy(&configuration);
```

The system also needs to be deactivated to release its resources

```
acc_rss_deactivate();
```

5 Interpret the Assembly Test Result

The different sub tests in the Assembly test aim to verify the integration of A111 into a custom PCB. The assembly test does not test the actual sensor but focus on testing the interaction. The assembly test returns an array with the result for each individual test and a variable containing the number of tests. The array should be iterated to check the result of each test and all tests should pass to confirm that the sensor and HAL has been integrated properly.

```
acc_rss_assembly_test_result_t test_results[
    ACC_RSS_ASSEMBLY_TEST_MAX_NUMBER_OF_TESTS];
uint16_t                       nr_of_test_results =
    ACC_RSS_ASSEMBLY_TEST_MAX_NUMBER_OF_TESTS;

for (uint16_t i = 0; i < nr_of_test_results; i++)
```

```
{
    const bool test_passed = test_results[i].test_passed;
    printf("Name: %s, result: %s\n", test_results[i].test_name, test_passed
       ? "Pass" : "Fail");
}
```

The assembly test returns a pass or fail verdict and this can be used to find failing units. The debugging of failing tests is done with the assembly test's log, which provides details on each test to understand the root cause of why the test fails.

5.1 Communication Test

The first group of the assembly test checks that the communication with the sensor is functioning. The communication test is divided into three subtests:

5.1.1 Read Test

SPI bus communication is tested by reading data from the sensor. This basic test is useful in bring-up to reassure that the SPI driver and HAL registration is working properly. If the SPI bus is monitored using a logic analyzer, this is the pattern that should be observed (first 16 MISO bits may be integration dependent).

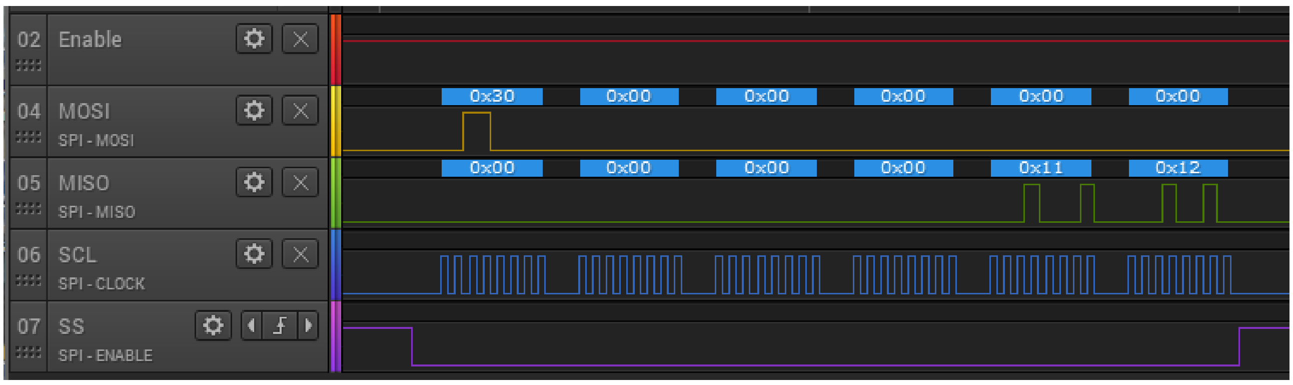

Figure 1: Assembly test, communication read test SPI pattern

5.1.2 Write and read Test

SPI bus communication is tested by writing and reading data on the sensor. The test contains a longer sequence of random bit pattern which is good to use for detecting glitches and runt pulses in the HW implementation that could increase the risk of communication failure. This test is also useful in bring-up to investigate the maximum stable SPI bit rate. The log also contains information on effective bit-rate and SPI frame overhead, which can be used as a guide on that the SPI driver is working properly.

5.1.3 Interrupt Test

The Interrupt pin connectivity is tested. This test is useful in bring-up to ensure that the Interrupt connectivity and pin selection in the HAL is correct. It also tests that the HAL function: acc_integration_wait_for_sensor_interrupt is working. In addition, the test also verifies the bandwidth requirement on the Interrupt path to ensure that no sensor Interrupts are lost.

5.2 Supply Test

The second group of the assembly test checks that the sensor is properly powered.

This test can be used in bring-up to ensure that all the power supplies (VIO_1-3) are valid during operation. If a power switch is implemented for VIO_1 and VIO_2, the test also ensures that the power switch control connectivity and pin selection in HAL is correct.

5.3 Clock Test

The third group of the assembly test checks that the reference clock for the sensor is valid. If the HW implementation includes an externally generated reference clock, the test also ensures that the clock control connectivity and pin selection in HAL is correct. If this test fails it could be that the specified reference frequency in the HAL does not match in actual frequency in the HW implementation.

6 Disclaimer