Interuniversity Microelectronics Center
Kapeldreef 75
B-3001 Leuven

**CRR-BB**

**CRR-FE**

# Sensing Engine User Manual

| | |
|---|---|
| Authors | Mattias Desmet |
| Date | 22/10/2013 |

## Table of Contents

CRR-BB – CRR-FE

## List of abbreviations

| | |
|---|---|
| SE | Sensing Engine |
| FE | front-end |
| API | Application Programming Interface |
| PCB | Printed Circuit Board |
| SPIDER | Sensing Platform for Integration and Demonstration of the DIFFS |
| DIFFS | DIgital Frontend For spectrum Sensing |
| WARP | Wireless Open-Access Research Platform |
| SINAD | SIgnal to Noise And Distortion |
| SDK | Software Development Kit |
| BMP | Babel Macro Processor |

# 1 Introduction

This document is a reference for anyone who wants to use imec's Sensing Engine for demonstrations, measurements or development. Throughout the text the term "Sensing Engine" (or its abbreviation "SE") will be used to refer to the combination of both hardware and software. The whole of the different hardware components, excluding the host PC, is referred to as the "platform"; the part of the software running on the host PC that is controlling the platform will be addressed as the "software API" or briefly "SW API". The SW API does not include the main application software however; a number of applications will be available, but the user can create its own applications as well, using the SW API.

To be able to use the SE, two components or parts will be provided: one or more platforms and an 'SE package'. This SE package is an archive (file extension '.tar.bz2') and can come in different forms. If a so called 'fixed version' of the SE package is provided, this means the archive contains a limited set of functionality for the platform. This functionality will be available in the form of a set of libraries that contain a part of the SW API, which can be used to build an application. This implies that the user will not be able to change the delivered functionality, but will only be able to add or ignore some functionality in his application. If the whole SE package is delivered however, all functionality of the SE will be available, including all source files that are used to generate this functionality. This means the complete SW API, including its source-files, but also the source-files for the FPGA-code and scripts to create bitfiles, libraries, and so on.

The purpose of this document is threefold. First, it gives an overview of the capabilities of the SE at the time of writing. Please note that this document will be updated regularly to include updates in the SW API, newly developed applications or to describe the integration of new hardware in the platform. Second, a detailed description of the Sensing Engine is given by introducing the different hardware components, the software needed for the SE to run and how to install a Sensing Engine from scratch. Third, the user is explained how to actually use the Sensing Engine. As mentioned before, this will include some reference applications that have been developed, but also how the user can create its own applications.

## 2 Overview

The first version of the Sensing Engine was a lab-based integrated demonstrator, although at that time it was not referred to as Sensing Engine. This platform consisted of a HAPS-32 FPGA board [1], on which both an analog front-end (Scaldio 2b) and a digital front-end (DIFFS) where plugged. The term “Sensing Engine” was first adopted when a portable setup was created that can be controlled from a standard laptop.[1] This is explained in the deliverable on the multichip demonstrator with DIFFS [2]. Accordingly, the capabilities of this first version of the SE were limited to what was verified and tested on the lab-based demonstrator. When the SE was thoroughly tested, it was decided to use it for other purposes apart from demonstrations, such as measurements. As can be concluded from this paragraph, the platform can take on different forms. This is discussed more into details in section 3.

When looking at the capabilities, attention should be paid to the component that is enabling a piece of functionality. Some functionality is enabled by the DIFFS-chip, other functionality is enabled (and thus depending on) the analog front-end that is connected to the SPIDER PCB. It is also possible to include functionality in the SW API or even in the application itself. The following list describes the capabilities that have been verified; this does not mean however that all functionality is available on each possible configuration of the SE.

- Reception and logging of RF-data
- Transmission of RF-data
- Fast Fourier Transform of RF-data
- DVB-T sensing on RF-data
- power measurements in the ISM band

As mentioned before, this list is limited to the capabilities that been verified at the time of writing. When additional capabilities are tested and found stable, these will be included in this document. The capabilities of the different components are discussed in section 4.2, as the SW API controls all functionality not enabled by the application. The functionality enabled by the application is discussed in section 6.

[1] To be able to control the SE, there are a number of requirements for the “standard” laptop. It has to run a Linux OS with some tweaks. This is explained in section 5.

# 3 Hardware

This section describes the different hardware components of the platform, thus excluding the host PC that is used to control the system. The requirements of this PC and how to install it is explained in section 5.1. The components used for the lab-based integrated setup are not included in this section, since this setup will never be distributed. For more information on this setup, one can refer to [2].

The portable SE platform consists of a Spider PCB (see section 3.1) connected to an analog front-end, possibly via an interconnect PCB. Currently there are two analog FE solutions available: imec's Scaldio 2b (see section 3.2) and Rice University's WARP (see section 3.3).

## 3.1 Spider PCB

The Spider PCB is a mixed signal design: it houses both a digital and an analog part. Its purpose is to provide an interface between the host PC and the rest of the platform, to provide functionality in the SE and to provide power and clocking for the analog FE. Two versions exist of the Spider PCB; revision 2 mainly fixes some bugs from revision 1. A picture of both revisions is shown in Figure 3-1 and Figure 3-2.

### 3.1.1 Description

The Spider PCB can be fed in two different ways. The most straightforward way is to use the USB power supply. This way, the total power consumption of the platform is limited to 500 mA at 5 V, thus 2.5 W. If more power is required, a power jack can be plugged in. It should output 5 V DC, but this way the supply current is off course "unlimited".

The analog part of the PCB can be summarized by looking at the in- and outputs. There is only one input, namely a socket for an oscillator. The oscillator is fed with 4.8 V; its output frequency should be lower than 166 MHz. There are five outputs: three connectors for clocking and two for power. The connectors for clocking provide high-impedant clocks which are routed to two SMB-jacks and one MMCX-jack. The frequency is equal to the frequency of the oscillator; the output voltage-swing is adjustable between 2.5 V and 3.3 V. The power connectors provide a clean supply for the analog FE. On one hand, there is a two pins connector: one pin for ground and one for 5 V. On the other hand, there is a four pins connector: one pin for ground, one for 1.2 V, one for 1.4 V and one for 2.5 V.

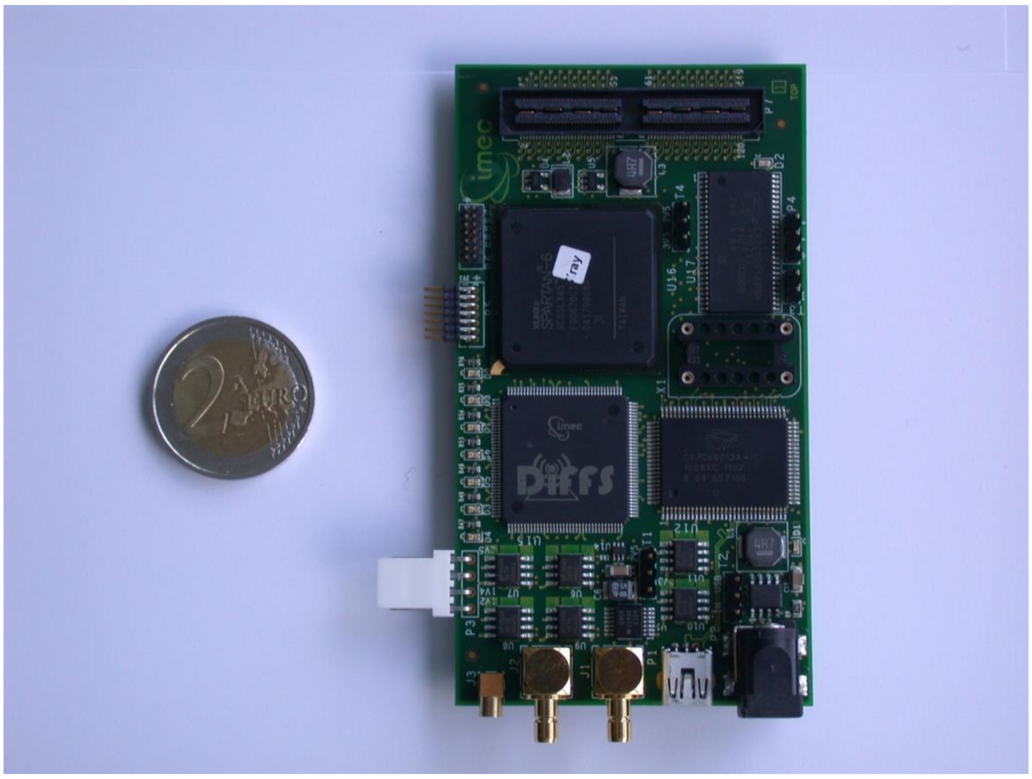

Figure 3-1: Spider (v1) PCB

The digital part can be divided in four parts. First, it foresees the connection to the host PC through a USB-connection, the same that is providing the power supply as mentioned before. The chipset used is Cypress' EZ-USB FX2LP. Second, a high speed 120-pins connector foresees an interface to connect the analog FE to. One pin is reserved to indicate the output voltage of the connector (see below). Two other pins are reserved to output a differential clock, which is a LVPECL-version of the output of the oscillator mentioned in the previous paragraph. All other 117 pins are connected to the FPGA. Third, the PCB contains imec's DIFFS chip. The DIFFS chip is a digital FE, the link between the analog FE and the baseband platform. This baseband platform is not included in the SE. The DIFFS is described more into detail in [3] and [4]. Fourth, there are some general purpose components, such as an FPGA, memory and some general purpose IO's. The FPGA is a Spartan 6 LX45 [5]. It connects all digital components on the PCB on the hardware level. Most IO voltages are fixed, except for the pins connected to the analog FE connector as mentioned before. Five of the 117 pins are connected to bank 0 and therefore fixed at 2.5 V. The other 112 pins are connected to bank 1 with a selectable IO voltage. 2.5 V and 3.3 V are foreseen on the PCB and can be selected by means of a jumper switch. In principle, it is possible to connect other IO voltages as well, as long as it is supported by the FPGA. The middle pin of the jumper should then be connected to an external power supply. This has not been tested however. The design on the FPGA is depending on the analog FE connected, since the connections are different for every FE, and every FE has its own specific functionality and interface. Furthermore, the FPGA-code is different for the two different revisions of the Spider PCB. The FPGA-code is discussed more into detail in section 3.1.2. Next to the FPGA, important for the SE is the memory present on the Spider PCB. For revision 1, this is a 16 Mbit SRAM component; for revision 2, this is a 64 MByte SDRAM component.

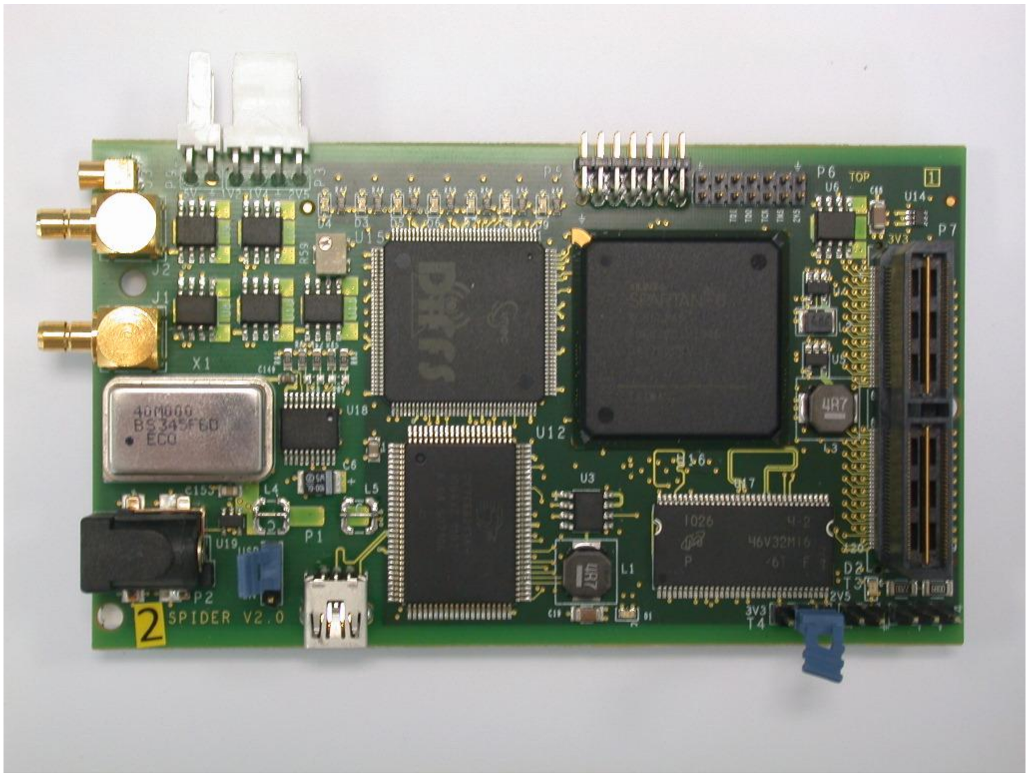

Figure 3-2: Spider v2 PCB

### 3.1.2 FPGA code

As mentioned in the previous section, there are a number of designs of the FPGA code. The design is depending on the version of the Spider board and which FPGA it has on board, the analog FE connected to the Spider board and possibly depending on the desired functionality. This last dependency will not be covered here, since it can be very specific. For example, it is possible to exclude e.g. transmit functionality from a design, or to change the size of the transmit- or receive-buffer. If the whole SE package is provided, the source files are available to finetune the design.

The purpose of the FPGA is to connect all different components on the physical level. This means that for different components, different FPGA code will be needed, since this means the pin-connections will be different. Also, every component has its own specific interface running at different clock speeds, meaning the FPGA design can differ for different components. This is shown in Figure 3-3. Examples of interfaces are the DIFFS host interface, DIFFS control interface, DIFFS front-end interface, SRAM interface, SDRAM interface, RX interface, TX interface, WARP interface, etc. All interfaces are controlled by the Program interface, and use a FIFO system to handle data streams. The translation between the different interfaces and the USB-chipset is done by the Spider Back End, or “spiderback”.

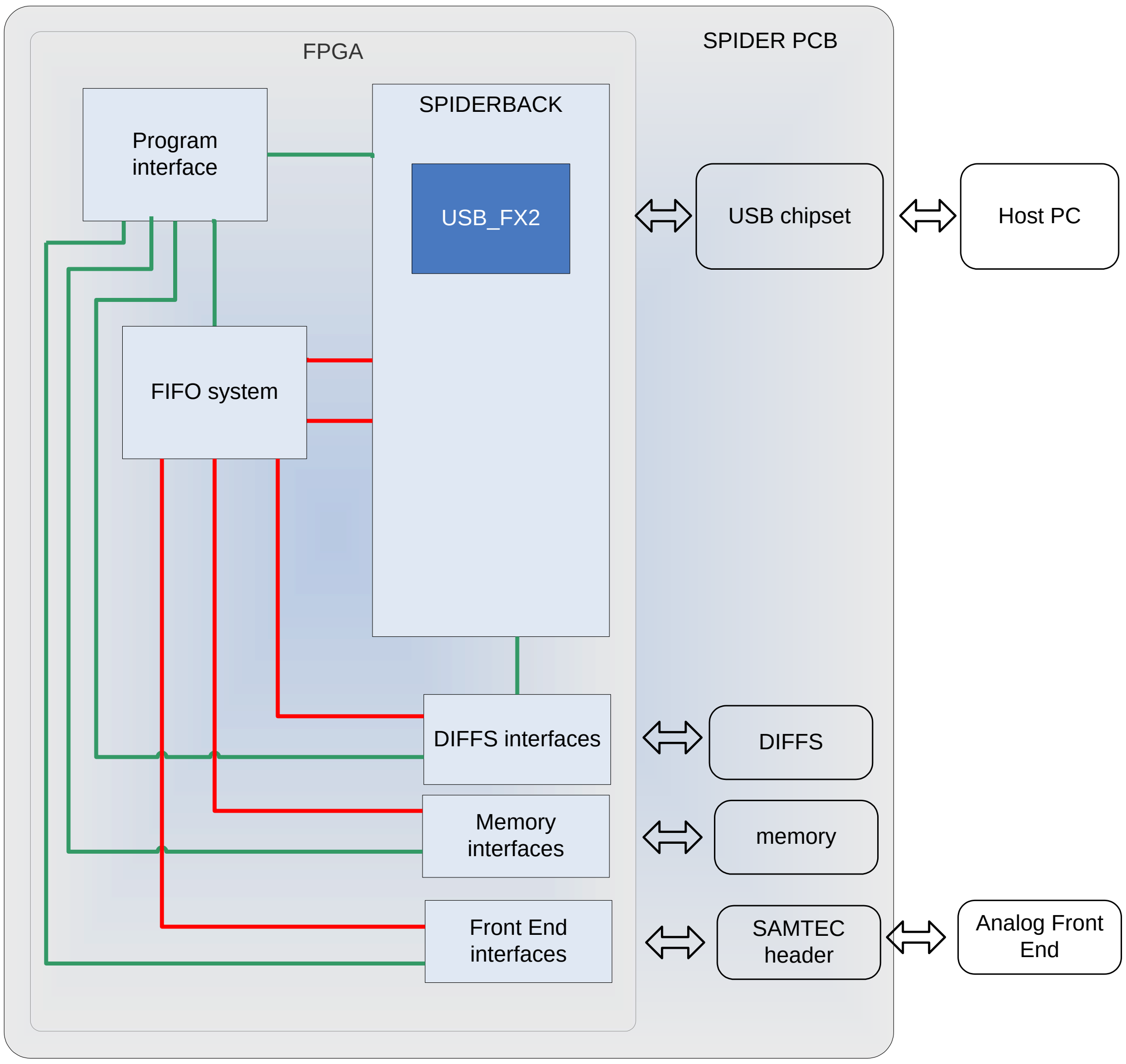

**Figure 3-3: Simplified look of the FPGA design on the Spider board**

For every FPGA design a bitfile is created. This bitfile can be uploaded to the FPGA via the USB chipset, using the ZTEX EZ-USB FX2 SDK (see section 4.1).

## 3.2 Scaldio 2b PCB

This board (depicted in Figure 3-4) houses the Scaldio 2b chip. The Scaldio chip is in the center of the PCB, under a protective acrylic glass shield. It is a flexible RF front-end: a single-chip reconfigurable receiver, transmitter and 2 frequency synthesizers in 40 nm-CMOS technology. The flexible receiver, including analog-to-digital converter, is fully software-configurable across all channels in the frequency bands between 100 MHz and 6 GHz. Its properties can be adapted to the requirements of the standards that are used. For more information, see [6].

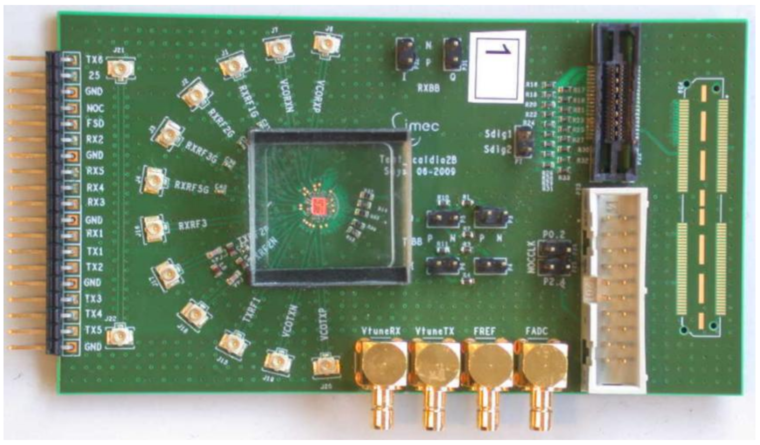

Figure 3-4: Picture of the SCALDIO 2b test PCB

For using the receiver section of the chip, the following connections are available on the PCB:

- 4 UFL connections to connect the antenna to the corresponding LNA;
- 120 pin QSH Samtec connector. This connector contains the digital output signals from the ADC, including a clock signal on which the data can be sampled. The data signal consists of 11 bits;
- 1 SMB jack to input the reference clock for the receiver PLL (FREF);
- 1 SMB jack to input the ADC sampling clock (FADC).

Power is supplied to the board using the 19 pins header at the bottom of the PCB; the other connectors are not used for normal receiver operation of the chip. The Scaldio has a Network-On-Chip (NOC) for configuration. This interface consists of 5 signals that can be accessed through the Samtec connector.

## 3.3 WARP TRX PCB and WARP Radio Board

The third analog FE solution is the WARP Radio Board, designed by Rice University [8]. The Radio Board is “a daughtercard which provides a single RF transceiver with a digital baseband interface”, which is commercially available for everyone [9]. The RF transceiver on the Radio Board is Maxim’s MAX2829, which is designed for dual-band 802.11 a/g applications, covering both the 2.4 GHz to 2.5 GHz band and the 4.9 GHz to 5.875 GHz band. The received analog baseband signal is sampled by an AD9248, while the RSSI output is sampled by an AD9200. The analog baseband signal for transmission is generated by an AD9777. Both the ADC’s and the DAC are designed by Analog Devices. The digital IQ and RSSI signals, as well as the digital control signals for the different components on the Radio Board, are available through two Hirose headers. Power is supplied to the WARP Radio Board through these connectors as well. The RF part, consisting of a power amplifier (PA), an antenna switch, bandpass filters (BPF) and baluns, connects to two SMA jacks to connect an antenna to. Three clocks have to be provided to the board: the RF reference clock is fed through an MMCX plug and must be 20 MHz or 40 MHz; the IQ sampling clock can be fed both differential and single-ended using a dedicated clock header or using the Hirose connectors and must be 40 MHz; the RSSI sampling clock is fed trough the Hirose connectors and should be 20 MHz or lower. A picture of the WARP Radio Board is shown in Figure 3-5.

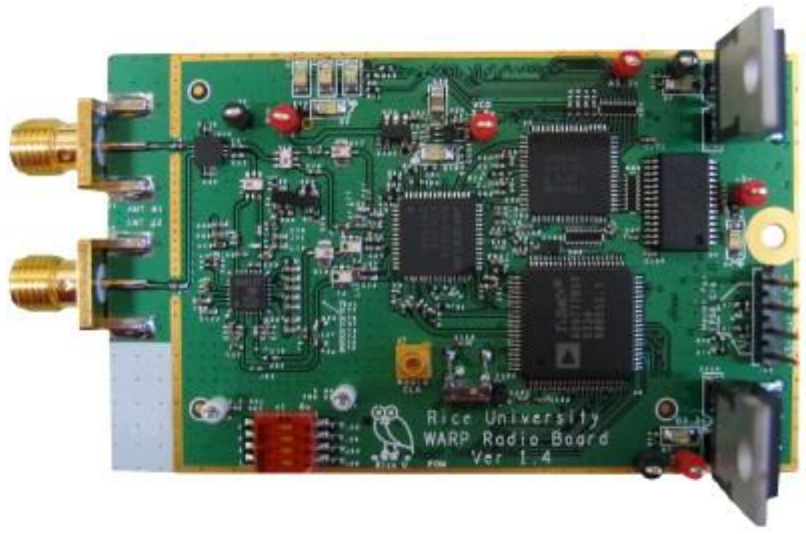

**Figure 3-5: Rice University WARP Radio Board**

To be able to connect the WARP Radio Board to the digital part of the SE platform, an interface board is needed. This WARP TRX PCB is designed especially for interfacing between the WARP Radio Board and the Spider v2 board. Two Hirose connectors are foreseen on the TRX board to connect the WARP Radio Board to, and a SAMTEC connector is foreseen for the Spider board. The signals from the Hirose connectors are routed to the Samtec connector, some after level-shifting. A two pin header enables the connection of a 5 V power supply, which is routed to the supply pins on the Hirose connectors. A picture of a WARP TRX board connected to a Spider v2 board is shown in Figure 3-6. One can distinguish the red-black power supply cable coming from the Spider board, providing the 5 V power supply for the WARP Radio Board.

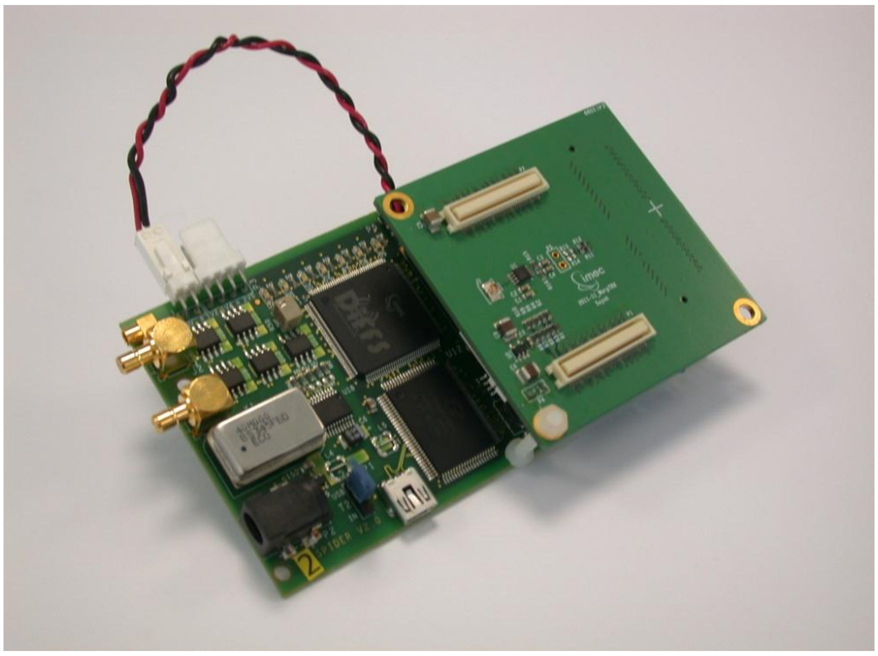

**Figure 3-6: WARP TRX board connected to a Spider v2 board**

# 4 Software

This section describes the different software components of the SE. All these components are included in the SE package that is made available to the user. If a fixed version of the SE package is provided, this package will contain an application folder. The application can be C-based or Matlab-based. In any case, the folder will contain firmware for the platform (bitfiles for the FPGA, firmware for the DIFFS, configuration-files for the analog FE and runtime firmware for the USB-chipset), libraries to control the platform, build-scripts, software to initialize the platform and one or more example applications. If the application is C-based, some header-files are available for creating an application. The structure of the provided C-based application folder would look something like this:

```
> tree sensing_engine
sensing_engine
|-- firmware
|    |-- diffs
|    |    |-- fft_test
|    |    |-- fft_test_prog_mem
|    |    |-- fft_test_vmem_lifo
|    |    `-- fft_test_xcor_prog_seq
|    |-- gain_files
|    |    `  ...
|    |-- main_firmware.ihx
|    |-- scaldio_01
|    |    `  ...
|    `-- scaldio_05
|         `  ...
|-- fpga
|    |-- spider_fpga_lx45_2C.bit
|    `-- spider_fpga_lx45_3I.bit
|-- includes
|    |-- main.h
|    |-- sensing.h
|    `-- types.h
|-- library
|    |-- libsensing.a
|    `-- libssi.a
|-- Makefile
|-- results
|-- setup.csh
|-- sources
|    `-- main.c
`-- ztex
     |-- FWLoader
     |-- FWLoader.jar
     `-- FWLoader.java
```

The structure of the provided Matlab application folder would look something like this:

```
> tree sensing_engine
sensing_engine
|-- firmware
|    |-- diffs
|    |    `  ...
|   |-- gain_files
|       ` ...
|   |-- main_firmware.ihx
|   |-- scaldio_01
|   |   ` ...
|   `-- scaldio_05
|       ` ...
|-- fpga
|   `-- spider_fpga_lx45_2C.bit
|-- library
|   ` ...
|-- mexusb
|   |-- Makefile
|   `-- sources
|       |-- mexusb.c
|       |-- spiderback_close.m
|       |-- spiderback_init.m
|       `-- ...
|-- results
|-- setup.csh
|-- sources
|   `-- example_app.m
`-- ztex
    |-- FWLoader
    |-- FWLoader.jar
    `-- FWLoader.java
```

On the other hand, if the complete SE package is provided, not only these application folders will be available, but also the whole SW API, the source-files for the FPGA-code and all necessary scripts to build what is provided in a fixed version of the SE package. Then the SE package would look something like this:

```
> tree .
.
|-- software
|   |-- interfaces
|   |   |-- haps
|   |   |-- host
|   |   `-- usb
|   |-- libs
|   |   `-- sensing
|   |-- modules
|   |   |-- diffs
|   |   |-- scaldio
|   |   `-- warp
|   `-- platforms
|       |-- host
|       |-- ssi
|       `-- swi
`-- spider
    |-- software
    |   |-- c_interface
    |   |   |-- sensing_engine
    |   |   `-- set_serial_number
    |   |-- matlab_interface
    |   |   `-- sensing_engine
    |   `-- ztex
    `-- vhdl
```

```
        |-- coregen
        |-- simulation
        |-- sources
        `-- synthesis
```

The ‘software’-folder, which is the SW API, will be discussed in section 4.2. As one can see, the ‘spider’-folder consists of two subfolders. For the ‘vhdl’-folder we refer to section 3.1.2. The ‘software’-folder contains three subfolders: the ‘ztex’-folder is discussed in section 4.1. The other two folders are application folders, which will be discussed in section 6.

## 4.1 ZTEX EZ-USB FX2 SDK

The EZ-USB FX2 Software Development Kit (SDK), developed by ZTEX [10], is an open source firmware development kit with a host software API. It is developed especially for use with the chipset used on the Spider board, Cypress’ EZ-USB FX2LP.

The firmware kit can be used for writing and building the firmware for the 8051 microprocessor on the EZ-USB device. For writing the firmware, a library with include-files is available. The firmware for the Sensing Engine is written in C. For more info on how to compile firmware, see section 4.2.2. For the compilation of firmware, the firmware kit contains the Babel Macro Processor (BMP), a powerful general purpose macro processor that can be used with many languages, e.g. Pascal or C. The BMP is written in Freepascal and is known to run on Linux and Windows, but should run an every platform that is supported by Freepascal. Pre-build binaries and scripts required for compiling the firmware are included in the firmware kit. The firmware kit also contains a firmware and bitstream upload utility, called FWLoader. FWLoader is a batch script with many possibilities, such as scanning for compatible devices, uploading firmware to an internal 16 kB RAM on the EZ-USB device, uploading a bitstream to an FPGA via the EZ-USB device, accessing EEPROM memory, etc. All accesses are done via USB, so no JTAG-connection is required.

The host software API is written in Java and allows platform independent host software. However, it is not used in the Sensing Engine. Instead, a specific SE API was written in C, based on the libusb C library. The SE API is covered in section 4.2.

The EZ-USB FX2 SDK is available as an archive from the website of ZTEX [10], and should be extracted to enable its functionality. When the user receives a SE API package however, all necessary files will be included in this package, so no extra downloads are required.

## 4.2 Software API

The Software Application Programming Interface (SW API) is a software library built to use the Sensing Engine. Since there are several possible configurations of the platform, there are also several versions of the SW API available. There are a number of predefined, fixed versions of the SW API that can be distributed; these will be a part of the fixed SE packages. It is off course also possible for the user to add specific functionality to the SW API if the whole SE package is provided. However, care should be taken when altering the SW API, and no stable functioning of the SE is guaranteed. Either way, the SW API yields a number of files bundled in one or more library-files that can be used to create and build applications for the SE. How to use these for creating applications is explained in section 6. The content of these library-files and how they are generated is explained in the remainder of this section. Also, a short overview is given of what the structure of the SW API looks like.

### 4.2.1 The SW API as part of the SE package

The SW API can come in two forms as explained earlier, depending on how the application that will use the SW API will be built. The application can be written in C (or a similar programming language), or in Matlab. Depending on the platform the SW API is intended for, the content of the package will (slightly) defer. This can be a different bitfile, different USB chipset firmware, different libraries, etc. It is possible that in the future additional SE packages will be created when new functionality is tested and added to the SE.

If the user wants to add specific functionality to the SW API, the whole SW API can be provided as well. In this case, all source-files will be added to the SW API, as well as all scripts needed for building the libraries and the applications.

### 4.2.2 Structure of the SW API

The SW API consists of a number of elements, or layers if you will. The lowest layer foresees an interface between the platform and the remainder of the SW API. This means that it enables communication with the host PC running the application. There are three different interfaces developed in the SW API: the HAPS interface, the HOST interface and the USB interface. The HAPS interface is intended for the lab-based setup with the HAPS FPGA board. For the same reason as explained in section 3, this interface is never included in the SW API. The HOST interface is a functional model; this means no platform has to be connected to the PC. The applications built with a package based on this interface will functionally behave exactly like the SE would. The intent is to allow the user to get familiar with the functionality of the SE. This interface offers no exact timing behavior of the SE however. Also, this interface is outputting dummy data, but in the same form as the ‘real’ SE would. The USB interface will be used for platforms in which the Spider board is coupled to the PC. This interface provides basic functionality to the rest of the API, in the form of a special designed data structure (struct) and 10 functions. The struct holds information on the state of the interface and which USB-device is coupled to it. Note that it is possible for multiple instances of the struct to be active at the same time, since it is possible that multiple Spider boards are connected to a single PC. Four of the functions are used for data transfer to and from the Spider board, respectively for reading and writing single or multiple values. The six other functions are used for controlling the interface, namely for opening, initializing, resetting, testing and closing an interface and getting its status. The same 10 functions are also available in the Host interface, as is the struct. This means no difference is perceived between these interfaces by the higher layers of the SW API. Next to the source-files for the interface, each directory contains the script needed for building the corresponding library. Additionally, the USB interface directory contains the firmware source files and the scripts to build the firmware.

The second layer is the so called platform layer. Every platform uses the same functionality (provided by the interface layer), but can offer different functionality to the higher layers depending on the platform. Every platform also has its own specific memory and register mapping, which is included in the platform by the use of include-files. Furthermore, the SW API must be built for every platform; therefore, every directory contains the necessary scripts, which are in their turn invoking other build scripts if necessary. Examples of platforms are the host platform, the Spider Scaldio Integration (ssi) platform and the Spider Warp Integration (swi) platform.

The third layer contains all functionality related to the different components of the platform, or modules as they are called here. The modules covered in the SW API are the DIFFS chip, the Scaldio FE and the WARP FE. For most users, this functions and everything related to it will not be visible. Moreover, the content of this layer covers all required functionality of the modules, so modifying it should be limited to extending functionality by adding new firmware for the DIFFS for example. However, care should be taken when adding new functionality as this may lead to instability or unwanted behavior, and the SW API should be thoroughly tested before use. For every module, the available functionality is more or less the same: there is a struct similar to the one of the interface layer, which keeps track of the state the module is in. These structs also hold another struct in which the active configuration of the module is saved. Furthermore, there are functions to open, initialize or close a module, to change the configuration, to activate it and to get results. An overview can be found in the respective include files.

The fourth and last layer contains the toplevel SE functionality. We call this layer the library as it foresees a library of functions that can be used in applications. For now, only one library is present, namely the sensing library. Similar to the content of the modules layer, the sensing library contains a struct, which is in fact an instance of the library, and thus actually *is* a 'Sensing Engine'. This means each SE is a struct that keeps track of the state of the SE, the configuration of the SE, and also holds the structs of the different modules that belong to the SE. This way, all information on the SE is bundled in one (instance of a) struct. Next to this struct, there are again a number of functions that in this particular case are available to the user. These functions can be used to open, initialize and close a SE, to configure it, to start or stop it and to get results from it. A prototype of the struct holding the configuration of the SE and of all available functions is available in the sensing library header file. This header file is included in every SE package.

# 5 Installation

This section describes how the Sensing Engine has to be installed. This includes the setup of the platform, the configuration of the host PC and the handling of the SE package.

## 5.1 Host PC

### 5.1.1 Installation

The SE can only be used on a PC with Ubuntu as Operating System for now. Therefore, Ubuntu 10.04.2 LTS (or a later version) is installed from CD media. Apply following settings:

- Welcome screen: Select English and click Install Ubuntu 10.04.2 LTS
- Select time zone
- Keyboard Layout: USA (default)
- Prepare Disk Space: Erase and use entire disk
- Create User Account: Name: Administrator; login: administrator; password: [choose any]; Computer name: [choose any; in this example, "picard-xps-02" is used]. Require password to log in.
- Review settings
- Start installation

After installation, the OS is updated using (Settings > Administration >) Update Manager. At the time of writing, the system reads:

```
> cat /etc/issue
Ubuntu 10.04.3 LTS \n \l

> uname -a
Linux picard-xps-02 2.6.32-37-generic #81-Ubuntu SMP Fri Dec 2
20:35:14 UTC 2011 i686 GNU/Linux
```

Following additional packages have been added to the system (and all required depencies):

- nedit
- vim
- emacs
- openssh-server
- openssh-client
- sshfs
- tcsh
- sdcc

- openjdk-6-jre
- libusb-dev

If the user wants to write his application in Matlab, Matlab has to be installed. We are using Matlab r2011a on the system. Additional users can off course be added next to the administrator account.

### 5.1.2 Spider USB access

By default, (unknown) USB devices are only `rw` for `root:root`. *Others* only have read permissions. To fix this, a rule has to be created for the `udev` driver so it knows what to do when a Spider is connected to the host PC. The Spider board is a bit special in that sense that it has 2 USB identification IDs. When plugged in, it identifies itself as `04b4:8613 Cypress Semiconductor Corp. CY7C68013 EZ-USB FX2 USB 2.0 Development Kit` on the USB bus. When it is configured, firmware is downloaded to reconfigure the USB interface. From then on, it identifies itself as `221a:0100`. So both devices will have to be added to the udev-rule in order to get it to work. This is done as follows:

1. Plug in the Spider USB device
2. Check the Vendor and Product ID through `lsusb`

```
>lsusb

Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 002: ID 046d:c00e Logitech, Inc. M-BJ58/M-BJ69 Optical
Wheel Mouse
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 024: ID 04b4:8613 Cypress Semiconductor Corp. CY7C68013
EZ-USB FX2 USB 2.0 Development Kit
Bus 001 Device 023: ID 04b4:8613 Cypress Semiconductor Corp. CY7C68013
EZ-USB FX2 USB 2.0 Development Kit
Bus 001 Device 004: ID 3923:709b National Instruments Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Note that 2 Spiders are connected to this system, both on USB bus 001, devices 23 and 24.

3. Check/verify the permissions on the USB device using the USB bus and device info read from the above `lsusb` command. To do so, execute:

```
> ls /dev/bus/usb/001/

crw-rw-r-- 1 root root 189,  0 2012-01-18 15:54 001
crw-rw-r-- 1 root root 189,  3 2012-01-18 15:54 004
crw-rw-r-- 1 root root 189, 24 2012-01-18 16:31 023
crw-rw-r-- 1 root root 189, 25 2012-01-18 16:31 024
```

Note: The `001` is the bus id. The permissions on devices 23 and 24 are read only for *others*.

4. Create the udev file `/etc/udev/rules.d/91-spider.rules`. It should contain following lines of code:

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="04b4", ATTRS{idProduct}=="8613",
MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="221a", ATTRS{idProduct}=="0100",
MODE="0666"
```

If desired, an example file can be provided. Verify the content of the udev file:

```
> cat /etc/udev/rules.d/91-spider.rules

[..]
SUBSYSTEMS=="usb", ATTRS{idVendor}=="04b4", ATTRS{idProduct}=="8613",
MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="221a", ATTRS{idProduct}=="0100",
MODE="0666"
```

5. Unplug the Spider USB board(s) and reconnect it/them again
6. Verify the permissions

```
> ls /dev/bus/usb/001/

crw-rw-r-- 1 root root 189,  0 2012-01-18 15:54 001
crw-rw-r-- 1 root root 189,  3 2012-01-18 15:54 004
crw-rw-rw- 1 root root 189, 24 2012-01-18 16:31 023
crw-rw-rw- 1 root root 189, 25 2012-01-18 16:31 024
```

The devices now have `rw` set to them. Spider boards connected in the future will also have user read/write permissions.

### 5.1.3 SE package

As mentioned before, there are several versions of the SW API, but in all cases a package will be provided. This package will have the '.tar.bz2' – file extension. The user should extract this package in the folder where the SE should be. It is assumed here that the working directory is the user's home folder and that the package is located here as well. Use the following command to extract the package:

```
> pwd

/home/[username]

> tar -xjf SensingEngine[_ddmmyyyy].tar.bz2
```

If the user has received the complete SE package, two folders will be created: "software" and "spider". The software-folder contains the SW API, the spider-folder contains a folder called "vhdl" with the FPGA-code for the Spider and a folder called "software" for building and running applications. A number of example applications will be provided. If the user has received a fixed version of the SE package, only a modified version of the last mentioned folder "software" will be provided.

In case the complete SW API is provided, the environment variables "USB_DIR" and "PLATFORM_DIR" should be set to enable building and running all applications. To avoid having to set these variables each time a new terminal is opened, these can be added to the user's runtime configuration script.

```
> setenv USB_DIR /home/[username]/software/interfaces/usb
> setenv PLATFORM_DIR /home/[username]/software/platforms/swi
```

## 5.2 Spider PCB

Before connecting power to a Spider board, either by connecting the USB cable or the DC power jack, it has to be prepared for proper operation. Both versions of the Spider board have to be prepared in a different way. In this section we are starting from PCB's coming straight from the PCB assembly company.

### 5.2.1 Spider v1

When using a Spider v1 PCB, please make sure that the necessary hardware fixes are executed: pin 5 of U6 should be shorted to pin 6 of U6, pin 5 of U17 should be connected to pin 13 of U17 and pin 14 of U17 should be connected to pin 16 of U17.

- First determine the serial number of the PCB and print it on a white sticker using font face Calibri and font size 16, then place the label on the Spider PCB. As soon as the Spider is connected to the host PC (see below), the serial number must be written in the Spider EEPROM, to enable the distinction between different Spider PCBs connected to a single host PC.
- Place a cap on jumper T2 over pins "USB" and "T2". This way, we are using power from the USB port, which has proven to be sufficient. If in some case more power would be required than the 2.5 Watts available from the USB port, the user will have to place the cap over pins "IN" and "T2". This way, power from an external 5 V DC power supply can be used. This power supply has to be connected to power jack header P2. Obviously, the Spider PCB should be able to draw more than 500 mA current from the power supply.
- The serial number can now be programmed. Therefore, connect the USB mini-B header P2 to a USB-port of the previously installed host PC and run the specific application. This application will only be available if the complete SW API is provided, which means that if a fixed version of the SW API is provided, the serial number will already be programmed in the Spider EEPROM. More info on how to use this application can be found in section 6. When the serial number is programmed, disconnect the USB cable again.

The Spider PCB is now ready to be used and to be connected to the rest of the platform.

### 5.2.2 Spider v2

Preparation of the Spider v2 PCB requires the same steps as for the Spider v1 PCB, but no hardware fixes are required. The only difference is that a yellow sticker should be used instead of a white one. Also, one additional step is required: before disconnecting the USB cable, the clock buffer chip U19 supply voltage should be regulated to approximately 3.1 V. This supply voltage corresponds to the output of voltage regulator U11, which can be measured on pin 5. To change the output voltage, use potentiometer R59. When the output voltage is set, disconnect the USB cable again. The Spider v2 PCB is now ready to be used and to be connected to the rest of the platform.

## 5.3 WARP

### 5.3.1 WARP PCB

As the WARP Radio Board is a complete analog FE solution, only one item has to be adressed. To connect the sampling clock coming from the Spider PCB to the WARP Radio Board, two resistors have to be mounted: both R77 and R78 require a 0 Ohm resistor.

### 5.3.2 WARP TRX PCB

Since the WARP TRX PCB is custom made, no preparation is needed. R13 and R14 should not be mounted, since these resistors connect an alternative clock signal to the WARP Radio Board, which is not needed.

## 5.4 Sensing Engine

There are currently three different configurations of the SE platform: Spider v1 connected to Scaldio 2b, Spider v2 connected to Scaldio 2b and Spider v2 connected to the WARP TRX PCB and a WARP Radio Board. If desired multiple SE platforms can be connected to the same host PC.

### 5.4.1 Spider v1 – Scaldio 2b

- Since the Scaldio 2b IO voltage is 2.5 V, a cap must be placed on jumper T4 over pins “2V5” and “VCCO”. This ensures that the FPGA bank connected to the Samtec header P7 is supplied with 2.5 V.
- Connect the Scaldio 2b to the Spider v1 using the Samtec connectors, with the Scaldio 2b on top of the Spider v1.
- Connect the Scaldio 2b supply to header P3.
- Place a 40 MHz oscillator on the socket X1.
- Connect the Scaldio 2b reference clock “FREF” and ADC clock “FADC” to connectors J1 and J2.
- Connect 4 antennas to the 4 UFL antenna connections.
- Finally, connect the USB mini-B header P2 to a USB-port of the host PC.

Since the Spider v2 PCB is used, the use of the Spider v1 – Scaldio 2b platform is no longer supported.

### 5.4.2 Spider v2 – Scaldio 2b

- Since the Scaldio 2b IO voltage is 2.5 V, a cap must be placed on jumper T4 over pins “2V5” and “VCCO”. This ensures that the FPGA bank connected to the Samtec header P7 is supplied with 2.5 V.
- Connect the Scaldio 2b to the Spider v2 using the Samtec connectors, with the Scaldio 2b on top of the Spider v2.
- Connect the Scaldio 2b supply to header P3.
- Place a 40 MHz oscillator on the socket X1.
- Connect the Scaldio 2b reference clock “FREF” and ADC clock “FADC” to connectors J1 and J2.
- Connect 4 antennas to the 4 UFL antenna connections.
- Finally, connect the USB mini-B header P2 to a USB-port of the host PC.

### 5.4.3 Spider v2 – WARP

- Since the WARP IO voltage is 3.3 V, a cap must be placed on jumper T4 over pins “3V3” and “VCCO”. This ensures that the FPGA bank connected to the Samtec header P7 is supplied with 3.3 V.
- Connect the WARP TRX PCB and the WARP Radio Board to the Spider v2 using the Samtec connectors, with the WARP TRX PCB on top of the Spider v2.
- Connect the WARP TRX PCB supply header P4 to Spider v2 header P9.
- Place a 40 MHz oscillator on the socket X1.
- Connect the reference clock from the Spider v2 connector J3 to MMCX plug J7 on the WARP Radio Board.
- Connect one or two antennas to the antenna connectors J1 and J2 on the WARP Radio Board.
- Finally, connect the USB mini-B header P2 to a USB-port of the host PC.

# 6 Applications

This section describes how to create and run an application. This includes which files are needed, how to create and build an application, what output to expect, etc. As mentioned before there are two sorts of applications: C-based applications and Matlab-based applications. Both have their own separate folder. In any case, the platform has to be initialized before the application can be run. This is also covered in this section.

## 6.1 Programming a serial number

This application is only available if the complete SE package is provided. No specific initialization is needed; in fact by initializing the platform the programming of the serial number is performed. In the directory software/interfaces/usb, there is the file "set_serial_number.c". Open this file and change the serial number on line 42 in the one that has to be programmed. If the serial number is for a Spider v1 board, the offset is 0; if the serial number is for a Spider v2 board, the offset is 0x80 or 128. For example, Spider v1 number 1 will have serial number 0x01; Spider v1 number 4 will have serial number 0x04; Spider v2 number 3 will have serial number 0x83; Spider v2 number 18 will have serial number 0x92. Save and close the file and build the application. Now go to the folder containing the script to run the application and run it.

Following unix-commands are needed:

```
> pwd
/home/[username]
> cd software/interfaces/usb
> nedit set_serial_number.c
> make –f Makefile.setup
> cd home/[username]/
> cd spider/software/c_interface/set_serial_number
> ./run.csh
```

## 6.2 Initialization of the SE

Before running an application, the platform has to be initialized. Therefore, a c-shell script is located in both application-folders, called 'setup.csh'. This script will load the runtime-firmware in the USB-chipset, initialize the Spider PCB and program the bitfile in the FPGA. An argument has to be provided when calling the script to indicate how much platforms have to be initialized. As mentioned before, every platform needs its own specific bitfile, depending on the configuration of the platform. All bitfiles are stored in the 'fpga'-folder. The firmware is loaded from the 'firmware'-folder. If the complete SE package is provided, both firmware and bitfile(s) will be a symbolic link to the corresponding file in the folder where it can be created. If a fixed SE package is provided, the folders will contain the necessary firmware and bitfile(s).

```
> cd /home/[username]/spider/software/c_interface/sensing_engine
> ./setup.csh 1

> cd /home/[username]/spider/software/matlab_interface
> ./setup.csh 1
```

## 6.3 C-based applications

The C-based applications are based on a set of libraries that can be created in the SW API. When an application is built, the build-script will use the source-files from the 'sources'-folder and refer to the libraries in the 'libraries'-folder. The source-files in their turn are based on the header-files in the 'includes'-folder. If the complete SE package is provided, these libraries will be a symbolic link to the corresponding library-files in the corresponding platform-folder where it can be created. If a fixed SW API is provided, the folder will contain the actual libraries. The same goes for the include-files. The libraries can be static or shared. As explained before, these libraries enable the use of specific functionality on a specific platform. All available functionality, consisting of a number of toplevel functions and structs, is collected in the header file specific for this library. This means that by including this header-file in his source-file, the user can use all available functionality without having to know about underlying specifications. The available functionality is covered in section 6.3.1.

To build an application, a Makefile is foreseen. This Makefile will compile the selected source-files in the 'Objects'-folder and build an executable in the 'Output'-folder. The default name of the executable is 'MAIN', but this can be changed in the Makefile as well. By typing 'make', the Makefile will build the application; by typing 'make clean', all intermediate files and executables will be erased, by typing 'make cleanall' all intermediate files and executables and all folders will be erased.

```
> make cleanall
cleaning...
... done.

cleaning all...
... done.

> make
creating Objects/ dir...
... done.
creating Output/ dir...
... done.

compiling...
... done.
building...

... done.
```

### 6.3.1 The sensing-library

As explained in section 4.2.2, the sensing library provides a library of functionality. We will now take a closer look to the content of the header-file belongs to the sensing library. It can be found in the 'includes'-folder and is called "sensing.h". First, let's take a look at the different structs and types; afterwards we will examine the functions more into detail.

```
> pwd
/home/[username]
> cd software/libs/sensing
> cat ./sensing.h
[..]
typedef struct se_s *se_t;

typedef enum {
    FFT_SWEEP = 0,
    WLAN_G = 1,
    WLAN_A = 2,
    BLUETOOTH = 3,
    ZIGBEE = 4,
    LTE = 5,
    DVB_T = 6,
    ISM_POWER_DETECT = 7,
    TRANSMIT = 97,
    ADC_LOG1 = 98,
    ADC_LOG2 = 99,
    STDBY = 100
    } se_mode_t;

struct se_config_s {
    se_mode_t se_mode;
    int16_t fe_gain;
    uint32_t first_channel;
    uint32_t last_channel;
    uint32_t bandwidth;
    uint16_t fft_points;
    uint16_t dvb_nr_carriers;
    float dvb_guard_interval;
    float threshold_power;
};
[..]
```

The struct “se_s” is the main struct of the SE. Every instance of this struct *is* a SE. As can be seen here, we define a type “se_t” which is a pointer the struct. Another type is defined, namely the type “se_mode_t”. This type contains an enumeration of the different modes the SE can be in. Last but not least, there is the struct “se_config_s”. This struct describes the configuration of a SE. A detailed description of each mode with the corresponding parameters is given in Appendix A.

```
> pwd
/home/[username]/software/libs/sensing
> cat ./sensing.h
[..]
se_t se_open(int spider, int frontend);
int se_init(se_t se_h, struct se_config_s *se_config);
void se_close(se_t se_h);
int se_configure(se_t se_h, struct se_config_s se_config, uint16_t
mode);
int se_check_config(se_t se_h, struct se_config_s se_config);
int se_start_measurement(se_t se_h);
int se_stop_measurement(se_t se_h);
int se_location(se_t se_h, float *current_loc);
int se_get_result(se_t se_h, float *destination, ...);
int se_get_status(se_t se_h);
[...]
```

The functions can be divided in three categories: there are a number of functions for general handling of the SE, there are a number of functions to control the SE and there are a number of functions to get results out of the SE. A detailed description of the functions, their parameters and their output is given in Appendix B.

### 6.3.2 Example applications

As a reference, we take a look at how the example application looks.

```
> pwd
/home/[username]/software/libs/sensing
> cd
/home/[username]/spider/software/c_interface/sensing_engine/sources
> cat ./main.c
#include "main.h"

int main (int argc, char* argv[])
{
    se_t se_1_h, se_2_h;
    struct se_config_s my_se_config;
    int result = 0;
    float fft_result[6*128];
    int i;

    se_1_h = se_open(0x81, 24);
    se_2_h = se_open(0x85, 0);
    result = se_init(se_1_h, &my_se_config);
    printf("Initialization [0 invalid, 1 valid]: %i\n", result);
    my_se_config.first_channel = 96;
    my_se_config.last_channel = 101;
    result = se_check_config(se_1_h, my_se_config);
    printf("Configuration check [0 invalid, 1 valid]: %i\n", result);
    result = se_configure(se_1_h, my_se_config, 0);
    printf("Configuration setup [0 invalid, 1 valid]: %i\n", result);
    se_start_measurement(se_1_h);
    se_get_result(se_1_h, fft_result);
    for (i=0;i<(my_se_config.last_channel - my_se_config.first_channel
+ 1)*my_se_config.fft_points;i++) {
        printf("fft-result[%i]: %f\n", i, fft_result[i]);
    }
    if (se_get_status(se_1_h)) se_close(se_1_h);
    if (se_get_status(se_2_h)) se_close(se_2_h);
    return 0;
}
```

First, the “main.h”-include-files is included. This file will in its turn include all other include-files. Then we declare two instances of the “se_t”-type, which will serve as instances of the SE. Furthermore, we have a number of floats and integers for control within the application. Second, we start the application by opening the two instances of the SE; the numbers provided indicate which platforms to use. In this case, Spider v2 number 1 will be coupled to Scaldio2b number 24, and Spider v1 number 5 will be coupled to a WARP Radio Board. Then we initialize one of the SE’s, alter the configuration, check this configuration, reconfigure the SE, start the measurement, extract the result and print it to the screen. After each step, we give some feedback if the previous step was completed successfully or not. In the end, we close the instances of the SE and return from the application.

## 6.4 Matlab applications

Applications written in Matlab follow a different approach then C-base applications. First of all, the library is more or less limited to the lowest layer of the SW API. This means basic connectivity is provided by the library, but there are some additional functions available that require a specific sequence of commands. This is due to the fact that Matlab applications up until now have only been used as a mean for connecting to or testing other hardware. The functionality on the Spider PCB was always used by means of the SW API. It is however possible that in the future a more extended library of Matlab functionality will be added to the SE.

Basic functionality consists of the following Matlab functions:

- spiderback_open
- spiderback_init
- spiderback_reset
- spiderback_close
- spiderback_write_single_reg
- spiderback_read_single_reg
- spiderback_write_burst
- spiderback_read_burst

We refer to the documentation added to these functions for more information. By typing “help [function name]” in Matlab, this documentation can be visualized. The coupling of these functions to the hardware is done by a file called ‘mexusb.c’. This is a C-file based on the ‘spider.c’-file from the SW API. To be able to address it from within Matlab a kind of executable has to be built. This is done by typing make in the ‘mexusb’-folder. This command will also add the basic functions to the library.

```
> pwd
/home/[username]
> cd spider/software/matlab_interface/mexusb
> make
```

The additional functionality consists of a number of functions mainly designed to control or interface with the specific interfaces on the FPGA. Each function has documentation in its file, to which we refer to for more detailed information on the specific functions. These functions are:

- adc_spi_write
- clk_spi_write

- dac_spi_write
- mcp4822_spi_write
- fill_tx_buffer
- i2c_read
- i2c_write
- NoCCtrl
- NOCReset
- NOCWrite
- program_scaldio
- read_full_rx_buffer
- read_rx_buffer
- sram_read
- sram_write
- start_rx
- stop_rx
- start_tx
- stop_tx
- stream_tx_data
- stop_tx_streaming

A basic example application is included in the application folder and can be found in the 'sources'-folder.

```
> pwd
/home/[username]/spider/software/matlab_interface/mexusb
> cd /home/[username]/spider/software/matlab_interface/sources
> cat main.m
addpath ../lib

% change the serial number hereunder to the correct one if needed
spiderback_init('05');
spiderback_reset;

disp(['FPGA bitfile version = ' spiderback_read_single_reg('4140')
spiderback_read_single_reg('4141')]);

% toggling the status leds of the DACADC board
spiderback_write_single_reg('5ff8','001e');
pause(1);
spiderback_write_single_reg('5ff8','0000');
disp('Closing interface...');
spiderback_close;
```

As can be seen, it is not possible to select the analog FE during initialization. The user can do this himself for example by creating a global variable that indicates which analog FE is connected to the platform and include a check for this global variable in his functions and scripts. In this script, only the Spider board is selected. Next, the platform is reset and the FPGA version number is read from the FPGA. After toggling on and off all LED's on the DACADC board, the interface is closed. If the user wants to control multiple platforms at the same time, multiple Matlab windows have to be opened, since the interface communicating with the active platform can only be linked to one Matlab window.

## 7 References

[1] Hardi ASIC Prototyping System (HAPS), Synopsys, http://www.synopsys.com/Systems/FPGABasedPrototyping/Pages/HAPS.aspx

[2] M. Desmet, L. Hollevoet, *'DIFFS Multichip Demonstrator'*, imec 2011, SSET-WL-PROT5.doc

[3] L. Hollevoet, S.Pollin, *'DIFFS specification'*, imec 2009, SSET-WL-DFE1.doc

[4] A. Couvreur et al., '*DIFFS hardware implementation*', imec 2010, SSET-WL-DFE3-DIFFS_implementation.doc

[5] Xilinx Spartan 6 LX45 PFGA, http://www.xilinx.com/products/silicon-devices/fpga/spartan-6/lx.htm

[6] M. Ingels et al., *'A 5mm2 40nm LP CMOS 0.1-to-3GHz Multistandard Transceiver'*, accepted for ISSCC 2010

[7] V. Giannini, M. Ingels, T. Sano, B. Debaillie, J. Borremans, J. Craninckx, *'A multiband LTE SAW-less modulator with ?160dBc/Hz RX-band noise in 40nm LP CMOS'*, accepted for ISSCC 2011

[8] Rice University WARP Project, http://warp.rice.edu

[9] Mango Communications, Inc., http://www.mangocomm.com/

[10] ZTEX GmbH, http://www.ztex.de

# Appendix A

## List of all configurations of the SE and their output

1. FFT_SWEEP

short description: sweep a part of the spectrum and perform an FFT calculation on each subband. There are 291 subbands available for Scaldio2b, numbered from 1 to 291; there are 28 subbands available for WARP, numbered from 1 to 28. Each subband is 20 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to sweep. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel.

**Scaldio2b:** the channel-frequency equals 500 MHz + 20 MHz * (channel number-1). Must be an integer number between 1 and 291.

**WARP:** the channel-frequency equals

- 2.412 GHz + 20 MHz * (channel number - 1) for channel 1 to 4;
- 2.84 GHz for channel 5;
- 5.18 GHz + 20 MHz * (channel number - 6) for channel 6 to 13;
- 5.5 GHz + 20 MHz * (channel number - 14) for channel 14 to 24;
- 5.745 GHz + 20 MHz * (channel number - 25) for channel 25 to 28.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 10.45 MHz for WARP.

fft_points: number of bins in the FFT-calculation. Must be 16, 32, 64 or 128. Currently only 128 supported.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each sweep yields an array with the number of channels times the number of FFT-points values. Every value is a logarithmic power value for the corresponding FFT-bin. Note: take into account an 8 MHz overlap between channel 4 and 5 in case the WARP analog FE is used.

2. WLAN_G

short description: determine the instantaneous power in a number of IEEE 802.11g channels. There are 14 channels available, numbered from 1 to 14, following the IEEE 802.11g standard. Each channel is 20 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.412 GHz + 5 MHz * (channel number - 1) for channel 1 to 13 and equals 2.484 GHz for channel 14.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 6.75 MHz for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

3. WLAN_A

short description: determine the instantaneous power in a number of IEEE 802.11a channels. There are 23 channels available, numbered from 36 to 64 in steps of 4, from 100 to 140 in steps of 4 and from 149 to 161 in steps of 4, following the IEEE 802.11a standard. Each channel is 20 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 5 GHz + 5 MHz * channel number.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 10.45 MHz for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

4. BLUETOOTH

short description: determine the instantaneous power in a number of IEEE 802.15.1 channels. There are 79 channels available, numbered from 1 to 79, following the IEEE 802.15.1 standard. Each channel is 1 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.402 GHz + 1 MHz * (channel number - 1).

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and automatically calculated for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

5. ZIGBEE

short description: determine the instantaneous power in a number of IEEE 802.15.4 Zigbee channels. There are 16 channels available, numbered from 11 to 26, following the IEEE 802.15.4 standard. Each channel is 2 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.35 GHz + 5 MHz * channel number.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and automatically calculated for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

6. LTE

short description: not defined yet.

7. DVB_T

short description: cyclostationary detection for channels of the DVB-T standard. There are 51 channels available, numbered from 16 to 66. Each channel is 8 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for Scaldio2b. This mode is not available for WARP front-ends.

fe_gain: gain setting of the analog front-end; between 27 and 100 (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 434 MHz + 8 MHz * (channel number - 16).

bandwidth: not applicable. Fixed at 10 MHz.

fft_points: not applicable in this mode.

dvb_nr_carriers: the number of carriers per symbol. Set to 2048 for 2k mode or 8192 for 8k mode.

dvb_guard_interval: portion of the symbol that is copied in front of the signal, in order to create a cyclic signal and hence avoid ISI. Floating point value which must be set to 1/4, 1/8, 1/16 or 1/32.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a qualifier for the cyclostationary property of the received signal for the corresponding channel; every second value is 0 if the first value is lower than the threshold or 1 otherwise.

8. ISM_POWER_DETECT

short description: determine the instantaneous power in a number of ISM band channels. There are 89 channels available, numbered from 1 to 89. Each channel is 2 or 4 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.404 GHz + 1 MHz * (channel number - 1).

bandwidth: 1 MHz or 2 MHz, depending on the width of the channel. Enter a value in Hz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

9. TRANSMIT

short description: not defined yet.

10. ADC_LOG1

short description: log the time-domain samples coming out of the ADC of Scaldio2b. There are 901 channels available, numbered from 1 to 901, which corresponds to every possible Scaldio2b carrier frequency.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals

- 93.75 MHz + 625 kHz * (channel number - 1) for channel 1 to 151;
- 187.5 MHz + 1.25 MHz * (channel number - 151) for channel 151 to 301;
- 375 Hz + 2.5 MHz * (channel number - 301) for channel 301 to 451;
- 750 MHz + 5 MHz * (channel number - 451) for channel 451 to 601;
- 1.5 GHz + 10 MHz * (channel number - 601) for channel 601 to 751;
- 3 GHz + 20 MHz * (channel number - 751) for channel 751 to 901;

bandwidth: not applicable. Fixed at 10 MHz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each run yields an array of samples with the number of channels times two times 1023 floats. Every first value is the real part of the sample, every second value the imaginary part. Only the number of samples indicated by the function call that fetches the result (see Appendix B) is valid.

## 11. ADC_LOG2

short description: log the time-domain samples coming out of the ADC of WARP. There are 37 channels available, numbered from 1 to 37, which corresponds to every possible WARP carrier frequency.

fe_gain: gain setting of the analog front-end; between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals

- 2.412 GHz + 5 MHz * (channel number - 1) for channel 1 to 13;
- 2.484 GHz for channel 14;
- 5.18 GHz + 20 MHz * (channel number - 15) for channel 15 to 22;
- 5.5 GHz + 20 MHz * (channel number - 23) for channel 23 to 33;
- 5.745 GHz + 20 MHz * (channel number - 34) for channel 34 to 37;

bandwidth: selectable from following values: 6.75 MHz, 7.125 MHz, 7.5 MHz, 7.875 MHz, 8.25 MHz, 8.55 MHz, 9.025 MHz, 9.5 MHz, 9.975 MHz, 10.45 MHz, 12.6 MHz, 13.3 MHz, 14 MHz, 14.7 MHz, 15.4 MHz, 16.2 MHz, 17.1 MHz, 18 MHz, 18.9 MHz and 19.8 MHz. Enter a value in Hz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each run yields an array of samples with the number of channels times two times 24575 floats. Every first value is the real part of the sample, every second value the imaginary part. Only the number of samples indicated by the function call that fetches the result (see Appendix B) is valid.

12. STDBY

short description: stand-by mode. Not implemented yet.

# Appendix B

## List of all functions in the sensing library

1. se_t se_open(int spider, int frontend);

short description: opens a new sensing engine (if available).
return value: the pointer to the SE-handler struct of the opened sensing engine.
parameters:

- spider: integer that equals the Spider-board serial number; this determines also the SE- and DIFFS-handler;
- frontend: integer which represents the front-end serial number; 0 stands for any WARP radio board module, any other stands for the Scaldio-board serial number.

2. int se_init(se_t se_h, struct se_config_s *se_config);

short description: initializes a given sensing engine.
return value: 1 on success, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the sensing engine to be initialized
- *se_config: pointer to a se-configuration struct; the (default) configuration after initialization will be copied to this pointer.

3. void se_close(se_t se_h);

short description: closes the given sensing engine.
return value: none.
parameters:

- se_h: pointer to the se-handler struct of the sensing engine to be closed.

4. int se_configure(se_t se_h, struct se_config_s se_config, uint16_t mode);

short description: configures the given sensing engine with the given configuration and mode.
return value: 1 on success, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the sensing engine to be configured;
- se_config: se-configuration struct to be configured;
- mode: mode of operation: 0 for single measurement, 1 for continuous operation.

5. int se_check_config(se_t se_h, struct se_config_s se_config);

short description: checks the given configuration.
return value: 1 if valid configuration, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine;
- se_config: the se-configuration struct that has to be checked.

6. int se_start_measurement(se_t se_h);

short description: starts the sensing engine.
return value: 1 if sensing thread was started successfully, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine.

7. int se_stop_measurement(se_t se_h);

short description: stops the active sensing engine activities
return value: 1 if sensing was stopped successfully, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine.

8. int se_location(se_t se_h, float *current_loc);

short description: returns the location of the sensing engine.
return value: 1 on success, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine;
- *current_loc: pointer to an array of floats where the location (x, y, z values) will be copied.

9. int se_get_result(se_t se_h, float *destination, ...);

short description: fetches the result of the last sensing engine measurement.
return value: 1 on success, 0 otherwise.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine;
- *destination: pointer where the sensing results will be copied to. The user has to make sure that enough memory space is allocated;
- optional parameter: in case of ADC-samples logging, an integer number that represents the number of IQ couples to fetch.

10. int se_get_status(se_t se_h);

short description: returns the status of the sensing engine.
return value: 1 or more if in use, 0 if free.
parameters:

- se_h: pointer to the se-handler struct of the active sensing engine.