SACRA

SEVENTH FRAMEWORK PROGRAMME

Project acronym:	**SACRA**
Project title:	Spectrum and Energy Efficiency through multi-band Cognitive Radio
Project number:	European Commission – 249060
Call identifier:	FP7-ICT-2007-1.1
Start date of project:	01/01/2010 **Duration:** 36 months

Document reference number:	**D6.3**
Document title:	Report on the Implementation of selected algorithms
Version:	1.0
Due date of document:	30th of June 2012
Actual submission date:	17th of July 2012
Lead beneficiary:	IT
Participants:	Dorin PANAITOPOL (NTUK), Tapio RAUTIO (VTT), Wael GUIBENE (EURE), Renaud PACALET (IT); Reviewers: Alexander JASCHKE (IIS), Djamal ZEGHLACHE (IT)

Project co-funded by the European Commission within the 7th Framework Programme		
DISSEMINATION LEVEL		
PU	Public	**X**
PCA	Public with confidential annex	
CO	Confidential, only for members of the consortium (including Commission Services)	

EXECUTIVE SUMMARY

This deliverable presents the first results of task WP6.2: « Implementation of selected algorithms ». Three algorithms have been selected, two for sensing and one for classification. They have been fully specified in terms of functionality, interfaces with their environment, type and range of their parameters. They have been designed using the synchronous software library and functionally validated against Matlab® references. They have then been enhanced for parallelization in order to take the maximum benefit of the target processor and thus reach the required performance. These parallel versions have been validated in terms of functionally against the synchronous version and in terms of performance using the SystemC virtual prototype of the processor. Finally the same parallel versions have been run on a Field Programmable Gate Array (FPGA)-based hardware implementation of the processor and validated.

Before presenting the implementation results we first describe the environment and constraints that developers and integrators had to deal with. This will ease the understanding of the implementation choices, the obtained results and performance. In Chapter 2 we first recall the main characteristics of the target baseband processor, its memory organization, the digital signal processing functions that can be hardware-accelerated and the performance of the processor when executing these functions. We then present the different target technologies on which the processor can be mapped. For each of the target technologies we also provide information about the maximum reachable clock frequency (the raw processing power) and the estimated power consumption.

Then, in Chapter 3, we briefly present the software design flow that has been used and that is one of the outcomes of WP5.

From the contribution of WP2 (and more precisely D2.5 [12]) we have selected two sensing algorithms. The two sensing algorithms and their implementations are presented in Chapter 4. The Energy Detector (ED) and the Welch Periodogram Detector (WPD) run on the target processor with very good performance and low power consumption.

The classification algorithm and its implementation is detailed in Chapter 5. Priority has been given to the Higher Order Cumulants (HOC) method which runs on the target processor with very good performance and low power consumption. However, this chapter also presents in detail another classification algorithm based on cyclostationary properties. The goal of this second section was to investigate the feasibility of this algorithm with respect to the operations provided by the processor. However, for the time being, this algorithm seems to be very computationally intensive. One possible way of improvement, that must still be investigated, would be to reformulate the algorithm using the Fast Fourier Transforms (FFT) supported by the baseband processor.

Please also note that this deliverable is in agreement with the Message Sequence Chart (MSC) provided in D3.3 [4] from WP3, where it is shown how complexity and computational load can be alleviated between different functional blocks. Therefore, it has been also considered that the normalization, averaging and the decision is made with the help of a Personal Computer (PC) used for visualization and control.

We then conclude with several remarks about these first implementation experiments.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LISTINGS

1 INTRODUCTION

1.1 Purpose of the Document

This document is deliverable D6.3 of the project. It is a technical report about the implementation of the selected sensing and classification algorithms. It briefly recalls the main characteristics of the target baseband processor, of its companion Software Design Kit (SDK) and of the validation framework. For each algorithm it then:

- presents the algorithm on a pure functional point of view,
- specifies its input parameters, their types, ranges and inter-dependencies,
- reports how it has been implemented on the baseband processor, what memory management strategy has been used and what degree of parallelization has been achieved,
- provides accurate performance figures in terms of computation load and estimates of power consumption for a realistic target manufacturing technology.

We conclude with several remarks and lessons about the target processor and the proposed software development flow.

2 THE TARGET DIGITAL SIGNAL PROCESSOR

2.1 General Overview

The Digital Signal Processor (DSP) of the SACRA platform is an instance of a generic architecture named EMBB[1] (Figure 2.1). When used in the base station side of WP6 demonstrations, it is mapped in the FPGA-based ExpressMIMO I board (Figure 2.2).

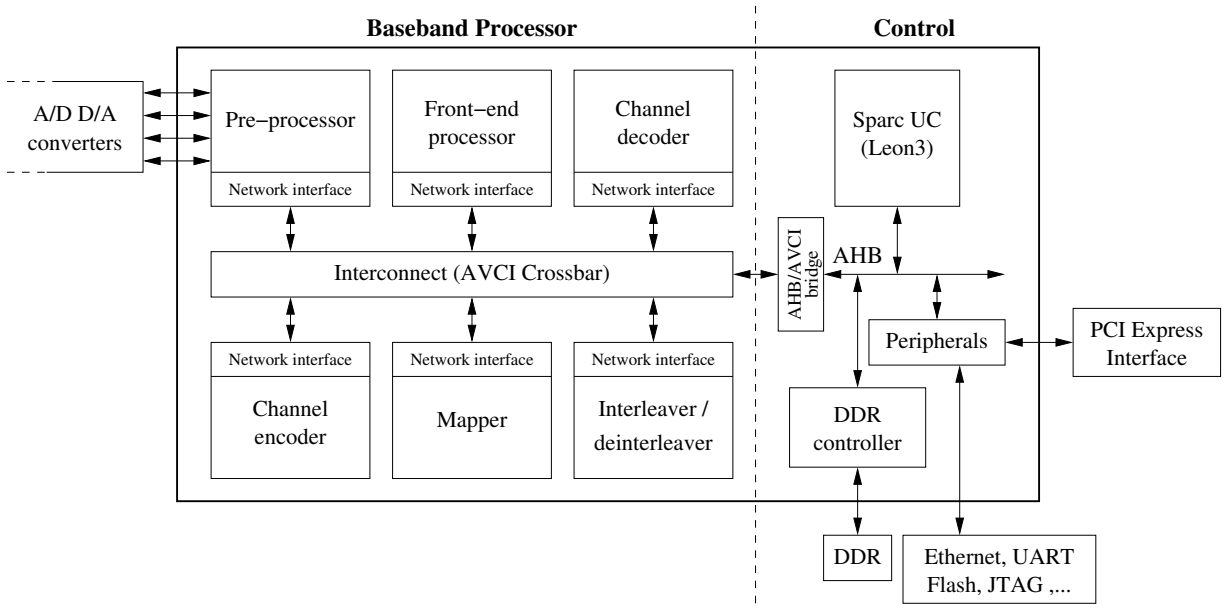

Figure 2.1: The generic EMBB architecture for SACRA

The three algorithms studied in this document operate on raw received samples and process them with pure vector operations and/or Fourier transforms. The involved EMBB units are thus:

- The Pre-Processor (PP) which interfaces directly with the converters. It operates on up to four transmit and four receive chains in parallel. It is responsible for compensation of phase and amplitude imbalance in In-phase/Quadrature-phase (I/Q) modulation schemes, carrier frequency adjustment, sample rate conversion and generation of signals and events to trigger baseband processing on received samples and data transmission.
- The Front-End Processor (FEP) which is a DSP dedicated to vector processing and time-frequency domain conversions.

[1] EMBB is an historical denomination that became a proper name and is not an acronym any more.

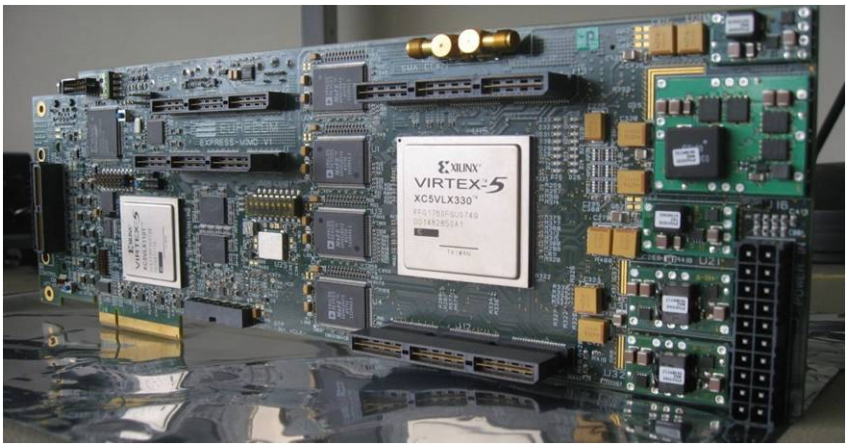

Figure 2.2: The FPGA based ExpressMIMO-1 prototyping board

- The main micro-processor (CPU), a Leon3 32 bits Sparc V8 processor by Gaisler-Aeroflex [1], which runs the low level software application controlling all processing and data transfers.

PP is configured to transparently store the acquired samples in its receiver (RX) output First In First Out (FIFO) buffer corresponding to the RX chain that has been selected as input of the sensing or classification application. The PP does not compensate for I/Q imbalance, carrier frequency offset nor does it re-sample the input stream because it does not make sense for the considered applications and, of course, in order to save power. FEP fetches the incoming samples directly from the PP output FIFO and moves them in its own internal working memory. It does so thanks to its own Direct Memory Access (DMA) engine which is programmed periodically to transfer a chunk of samples (a segment) from PP to FEP. FEP processes the samples and stores the results in its working memory. FEP is responsible for the most regular and computation-demanding parts of the application (Fourier transforms, vector operations). Finally, the raw results are read by the Leon processor out from FEP working memory and they are post-processed in software before being transmitted to the upper application layers. The Leon core is in charge of controlling the PP and FEP operations and the DMA transfers. It is also running low rate computations that cannot easily and efficiently be computed by FEP. On a realistic target technology, the upper application layers would run on the same CPU but in the context of the SACRA demonstration activities they run on a host PC. The reason for this is threefold:

- While the main CPU would run at a clock frequency between 500 MHz and 1 GHz on an Integrated Circuit (IC) baseband processor, it only runs at about 100 MHz on the naturally slower FPGA-based prototype[2]. Its processing power is thus very limited.
- In order to save the limited hardware resources of the FPGA, in which the Leon core is mapped, it is not equipped with all the peripherals that would be found in an IC: it has no floating point unit, no Memory Management Unit (MMU), and its hardware multipliers and dividers are slow multi-cycles ones.
- Designing the upper layers of the sensing and classification applications is much more comfortable on a regular PC than on an embedded processor. The control, the computation and the visualization parts of the applications can, for instance, be entirely designed under Matlab® or a similar environment, which would of course be impossible with the embedded processor.

[2]FPGAs are reconfigurable but this flexibility comes with a price in terms of power consumption and speed

The partitioning of the three candidate applications on the different computing nodes of the SACRA platform can be summarized as follows:

- EMBB PP DSP unit: samples acquisition, framing, FIFO
- EMBB FEP DSP unit: Fourier transforms and other vector operations on vectors of samples
- Leon embedded main CPU: control of the processing and of the data transfers, interface with the upper application layers, low rate, simple, post processing of the raw results from FEP (e.g. accumulation of results over a series of consecutive segments)
- Host PC: upper layers of application (control, visualization, thresholds computation, decisions making, etc.), finalization of the data processing (floating point operations, normalizations, etc.)

This partitioning is a very logical and classical one: highly regular and demanding operations run on dedicated hardware nodes (FEP) while less regular and less demanding operations run on general purpose CPUs (Leon and host PC). The only significant difference between this set-up and a realistic one is the host PC which would be merged with the embedded processor in a high end IC baseband processor.

2.2 The Front-End Processor

2.2.1 Front-End Processor (FEP) Supported Operations

FEP is a key element for the considered applications. More information about its supported operations can be found in the first appendix of the SACRA D5.2 deliverable [8]. Here is a brief reminder of its main characteristics. FEP implements Discrete Fourier Transform (DFT) and vector processing. Operations take one or two input vectors, X and Y, and output one vector Z. A set of five scalar results, collectively referred to as the « SMA values », can optionally be also computed: the sum, max, $argmax$, min and $argmin$ of the output vector Z. Different data types are supported: the components of the input and output vectors can be 8- or 16-bits signed integers (`int8`, `int16`), or complex numbers which real and imaginary parts are 8- or 16-bits signed integers (`cpx16`, `cpx32`). Table 2.1 lists the different supported operations. X is the first input vector, Y is the second (if any) and Z is the output vector. Computations are performed with limited accuracy and the actual outputs are integer approximations of the exact results. The Component-Wise Lookup (`CWL`) operation is used to approximate non-linear arbitrary functions like square roots, sine, cosine, inverse, etc. It can optionally be computed with linear interpolation between consecutive entries of the tabulated function. Many parameters are used to alter the default behaviour of the basic operations and offer many variations. Thanks to this, FEP operations are extremely rich and complex DSP primitives and one single FEP instructions usually replaces hundreds of instructions of a more classical DSP.

Mapping the selected applications on FEP consists in reformulating them in FEP dialect, that is, restrict the processing to the supported operations. Of course, unsupported operations cannot always be replaced by supported ones. If the missing operation are used at a sufficiently low rate they can be emulated in software, either on the Leon CPU or on the host PC. If they are very demanding, high rate, operations, a rework of the hardware architecture must be considered[3].

2.2.2 FEP Working Memory

Another essential task when porting applications on FEP (and more generally on EMBB) is the memory management: each DSP unit embeds a pool of working memory which size and organization is unit-dependant.

[3]During the implementation of the signal classification based on HOC, it has been noticed that the `CWS` operation, while not mandatory, would be beneficial. FEP has thus been reworked, both the hardware and the software, to support this extra operation.

Operation	Z	Comments
`CWA`	$Z \approx X + Y$	Component Wise Addition
`CWP`	$Z \approx (X \times Y)/2^{15}$	Component Wise Product
`CWM`	$Z \approx \lvert X\rvert^2/2^{15}$	Component Wise square of Modulus
`CWS`	$Z \approx X^2/2^{15}$	Component Wise Square
`MOV`	$Z \approx X$	Component wise copy
`CWL`	$Z[i] \approx Y[X[i]]$	Component Wise filter by a Lookup table
`FT`	$Z \approx DFT(X)$	Fourier Transform

Table 2.1: FEP operations

Input data must be stored in the unit's working memory prior processing and the output results are stored back in the unit's working memory, from which they can be either read again for further processing by the same unit or transferred to another unit. FEP working memory is a 64k-Bytes memory, split in four 16k-Bytes banks. FEP input and output vectors must be stored in different banks and cannot span over several banks. Fourier transforms are an exception: the input and output vectors can be stored in the same bank (but cannot span over several banks neither). The number of components of the largest possible vector that can be stored in a FEP bank depend on the components' type and ranges from 4096 for `cpx32` vectors to 16384 for `int8` vectors. Processing larger vectors is possible but must be split in smaller sub-vectors. This principle is used for the ED sensing application and for the HOC classification application. Figure 2.3 illustrates the organization of FEP working memory. Programming FEP mainly consists in programming a DMA transfer to move data in, launch one of FEP operations with the right set of parameters (the FEP instruction), and programming a DMA transfer to move the result out. Interrupts signal the end of each of these steps and can be routed either to the main CPU or to the local 8-bits micro-controller.

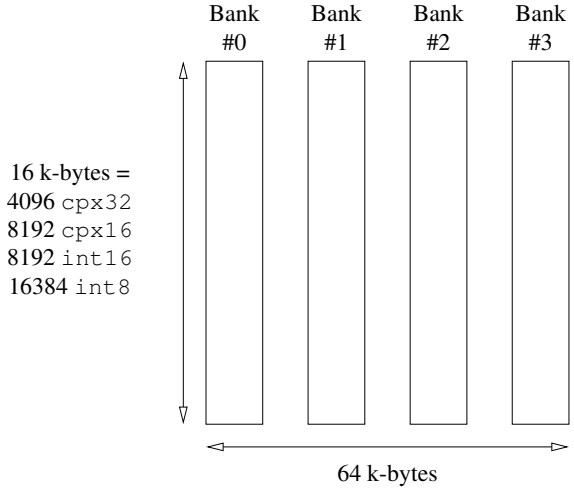

Figure 2.3: FEP working memory

2.3 EMBB Performance

The SACRA prototype is based on FPGAs and shall not be considered as a realistic target for a baseband processor in a user equipment: its power consumption and limited processing power would be orders of magnitude from what is required. Even in base stations, the power budget and required processing power make FPGA-based solutions expensive and sub-optimal. In this document the performance evaluations are done for 3 different target technologies:

- The SACRA FPGA-based prototype, running at a 100 MHz maximum clock frequency. This target is shown only for information and to state whether the SACRA demonstrations are feasible or not.
- A 65 nano-meters (nm) technology on which an EMBB instance runs at a 500 MHz clock frequency (result of static timing analysis after logic synthesis but with estimated routing parasitics). This target is a good representative of existing commercial products.
- A 22 nm technology on which EMBB would run at a 1 GHz clock frequency (result of static timing analysis after logic synthesis on a 28 nm process, with estimated routing parasitics, followed by an extrapolation to 22 nm). This target is a good representative of next generation of commercial products, as they should be available around 2015.

Estimating the power consumption of an application on EMBB is very difficult because firstly power simulations on high level models are so inaccurate that they cannot really provide absolute figures and secondly FPGA-based prototypes are orders of magnitude from realistic integrated targets. Low level power simulations on placed and routed standard cells-based designs provide much better estimates but require a complete back-end design which is out of scope of the SACRA project. However, the power budget of user equipments is the most constrained by thermal limits. In recent products the power budget of the baseband processing (analogue and digital, including external dynamic memories) is usually estimated as 300 mW at most, and even less (see for instance [7]). Logic synthesis of the SACRA baseband processor for different target technologies (Xilinx Virtex 5 FPGAs, 40 and 28 nm Application Specific Integrated Circuit (ASIC) standard cells libraries) show that FEP resources usage is about 15% of the complete EMBB processor. An alternate way of very roughly estimating the power consumption thus consists in applying a power factor to the processing load of FEP for a given application. Of course, this makes sense only if EMBB is at least as power-efficient as other baseband processors. This is a strong assumption that cannot be verified yet for the above mentioned reasons, but we consider it as very reasonable. EMBB is significantly more dedicated than most similar processors and it is universally admitted that the more a hardware operator is dedicated, the more it is energy-efficient. Moreover, if the considered application is the only one running on the baseband processor, Dynamic Voltage and Frequency Scaling (DVFS) can be used to further reduce the power consumption. According the International Technology Roadmap for Semiconductors (ITRS) 2011 annual report [9], in 2012, a 22 nm process would operate in a voltage range from $0.7V$ (low operating power, low V_{dd} transistors) to 0.87 (high performance). Applying a DVFS strategy and operating at the minimum power would thus further reduce the power by $(0.87/0.7)^2 \approx 1.54$. In the following, power estimates are provided for FEP, which is the main computing node, for the 22-nm target and for the two different operating voltages. Power consumptions of the main CPU and of the other elements (interconnect, DMA engines) are not reported because there is no reliable way to estimate them in our context, but as will be shown the load of the main CPU is very low. In order to be as conservative as possible we considered a total power budget of 300 mW, at high power supply voltage, for the sole digital baseband processing. This leads to a FEP consuming 45 mW when fully loaded. Using the more aggressive figures from [7] would further reduce this down to 130 and 15 mW respectively.

3 THE SOFTWARE DESIGN FLOW

The software development flow has already been discussed in the SACRA D5.2 deliverable [8]. This chapter is a short reminder and a description of the way it has been used when designing the sensing and classification algorithms.

The design flow of baseband digital signal processing applications for the SACRA baseband processor, EMBB, comprises 4 steps:

1. High level algorithm design and validation
2. Refinement in `libembb` dialect for synchronous execution and validation
3. Refinement for parallelization, asynchronous execution and validation on virtual prototype
4. Execution and validation on target hardware

3.1 High Level Algorithm Design and Validation

This step is conducted using the standard tools of the field, Matlab®, Octave or the like. It can be further split in a preliminary phase in which no constraints related to the target processor are considered and a second one in which it is taken into account. Vector operations, for instance, can be used on arbitrary large vectors in the first phase, while in the second phase they are restricted to the supported vector lengths of the processor. The main characteristics of this step with respect to the global design flow are:

- The double-precision, floating point, computations
- The synchronous programming model in which operations take place sequentially, one after the other
- The absence of memory management

Once validated by simulation, the application is ready for refinement towards the `libembb` dialect.

3.2 `libembb` Synchronous Design and Validation

The application is reworked and re-coded using the C language. The emulation software library (`libembb`) is written in C++ but it has a 100% C Application Programming Interface (API). It offers all digital signal processing primitives that are supported by the EMBB processor and it is 100% bit accurate. The first transformation thus consists in identifying which DSP primitive to use as a replacement for the high level ones. These primitives are specialized through a set of parameters that must be assigned according the intended behaviour. The reader interested in a detailed description can refer to the SACRA D5.2 deliverable [8] and its two appendixes.

Memory management must also be added: in the EMBB processor the hardware-accelerated DSP processing is handled by DSP units which all embed a local working memory. The working memories are mapped in the global processor's address space but DSP units cannot take their input data outside their working memories

and cannot store their computation results elsewhere. So, in order to compute a DSP operation on a data set, the set must first be moved to the right working memory, the DSP unit must be launched and when the computation ends the result must be moved from the working memory to its destination. The data transfers are handled by DMA engines that must be properly configured through their interface registers. One of the most challenging tasks when programming EMBB is the management of these working memories. They are limited in size and the programmer must carefully allocate them to data sets in order to never exceed the available memory size and to always have the right data set in the right memory before launching a DSP operation. `libembb` abstract all this up to a certain point but it does not hide all the details. The DMA transfers are explicitly represented by function calls which parameters include source and destination addresses. Similarly, DSP operations are abstracted by function calls which parameters also refer to source and destination addresses.

The Matlab® listing 3.1 implements a direct DFT on a 1024-points vector (`X` is an already declared and initialized Matlab® 1024-components vector variable). Listing 3.2 represents the same operation re-coded in C on top of `libembb` with the assumption that the input and output vectors are stored in the main CPU memory (`X` and `Y` arrays) and must thus be moved around between this memory and the working memory of the DSP unit responsible for Fourier transforms, the FEP.

```
1 Y = fft(X);
```

Listing 3.1: A Matlab® DFT

```
1  uint32_t X[1024], Y[1024]; // Input and output vectors in CPU memory
2  FEP_CONTEXT ctx;  // Parameters (context) of FEP operations
3  // ...
4  fep_ctx_init(&ctx, (uintptr_t) fep_mss); // Initialize context, fep_mss points to FEP
       working memory
5  embb_mem2ip((EMBB_CONTEXT*) &ctx, 0, X, 1024 * 4); // Move data set X in FEP working
       memory
6  fep_set_op(&ctx, FEP_OP_FT); // Set FEP operation to Fourier transform
7  fep_set_l(&ctx, 1024);       // Set DFT length
8  fep_set_i(&ctx, 0);          // Set DFT direction (direct)
9  fep_set_r(&ctx, 6);          // Set output rescaling factor to 1/2^6 (saturation
       avoidance)
10 fep_set_qx(&ctx, 0);         // Set source bank to FEP bank #0
11 fep_set_bx(&ctx, 0);         // Set index of first component of source vector in bank
12 fep_set_wx(&ctx, 3);         // Set wrapping section to whole source bank (4096 points)
13 fep_set_qz(&ctx, 0);         // Set destination bank to FEP bank #0
14 fep_set_bz(&ctx, 0);         // Set index of first component of result vector in bank
15 fep_set_wz(&ctx, 3);         // Set wrapping section to whole destination bank (4096
       points)
16 fep_do((FEP_CONTEXT*) &ctx); // Launch operation
17 embb_ip2mem(Y, (EMBB_CONTEXT*) &ctx, 0, 1024 * 4); // Move result from FEP working memory
       to CPU memory (Y)
18 fep_ctx_cleanup(&ctx);       // Deallocate context
```

Listing 3.2: A `libembb` DFT

Of course, once configured, a DSP context can be re-used unchanged for identical operations. The only parameters that must be re-assigned from one operation to another are the modified ones. For instance, after the direct DFT, the inverse operation could have been computed by adding the two lines of listing 3.3 after line 16 of listing 3.2. Note that due to limited accuracy the result `Y` would not be exactly the original vector `X`.

```
1 fep_set_i(&ctx, 1);          // Set DFT direction (inverse)
2 fep_do((FEP_CONTEXT*) &ctx); // Launch operation
```

Listing 3.3: A `libembb` inverse DFT

Once the porting to `libembb` is done, the functional validation is done by compiling, linking against the `libembb` library, running experiments and comparing the results against the original high level model. The `libembb` version can be debugged with classical software design tools (debuggers, memory checkers, etc.). Two potential sources of mismatches between the high level model and the `libembb`-based one must be taken into account:

- Some basic DSP primitives may have different definitions. Matlab® direct and inverse DFTs, for instance, are defined as:

$$Y(k) = \sum_{l=0}^{l=N-1} X(l) \times e^{-2j\pi kl/N} \tag{3.1}$$

$$X(l) = \frac{1}{N} \times \sum_{k=0}^{k=N-1} Y(k) \times e^{2j\pi kl/N} \tag{3.2}$$

  while in `libembb` they are defined as:

$$Y(k) = \frac{1}{\sqrt{N}} \times \sum_{l=0}^{l=N-1} X(l) \times e^{-2j\pi kl/N} \tag{3.3}$$

$$X(l) = \frac{1}{\sqrt{N}} \times \sum_{k=0}^{k=N-1} Y(k) \times e^{2j\pi kl/N} \tag{3.4}$$

- The fixed point computations with limited accuracy usually add some computation noise, due to quantization effects. Another challenging design task consists in ordering the DSP operations and in selecting their parameters in order to limit this computation noise and keep acceptable algorithmic performance. An example of a detailed analysis of quantization noise is given in Section 4.2.1 dedicated to the WPD.

Once validated, this version of the application is much closer to the final one. Computations are now fixed point and the memory management of the working memories has been designed. Indeed the application can already be cross-compiled and run on the virtual prototype or on the hardware target processor. One important aspect is however still missing: parallelization for asynchronous execution.

3.3 Parallel Design and Asynchronous Validation on the Virtual Prototype

The main CPU of the EMBB processor runs a Real Time Operating System (RTOS) which is a customized version of the MutekH [2] exo-kernel. Thanks to this RTOS the synchronous application is once more refined in order to take the maximum benefit of the highly parallel EMBB processor. Data transfers between the main memory and the working memories or between working memories can be launched during a DSP processing. As each DSP unit is equipped with a local DMA engine, several data transfers can also take place simultaneously. The DSP units are independent and can run their own DSP primitives in parallel with others. Thanks to their local 8-bits micro-controller they can even run small local applications including DMA transfers, DSP operations and control flow.

Parallelizing a synchronous application is probably the most challenging design task because it requires a deep knowledge of the EMBB hardware architecture and also because parallel programming is much more difficult and error prone than sequential programming. The RTOS offers all the classical basic functions for multi-threaded programming (semaphores, mutex, etc.). It has been specialized with hardware device drivers for the DSP units and the DMA engines and with dedicated Interrupt Service Routines (ISR).

In order to parallelize the synchronous application all function calls dedicated to data transfers or DSP processing must be split in a `start` call and a `wait` call which do not necessarily immediately follow the `start` call. Listing 3.4 illustrates this with a simple interleaving of DMA transfer and DSP processing.

```
1 embb_dma_start(&fep_dma_ctx, 0, 0, 4096); // Launch DMA transfer
2 embb_fep_start(&fep_dft_ctx);             // Launch DSP processing
3 embb_dma_wait(&fep_dma_ctx);              // Wait until end of DMA transfer
4 embb_fep_wait(&fep_dft_ctx);              // Wait until end of DSP processing
```

Listing 3.4: EMBB parallel programming

In most cases this transform is not sufficient to fully exploit the parallelism of the EMBB processor. The synchronous application must then be split in several independent threads (MutekH implements POSIX threads), usually corresponding to activities that can run concurrently like, for instance:

- Periodically transfer data from one DSP unit to another using one of the DMA engines
- Chain computations on the newly transferred data in the destination DSP unit
- Read computation results from the working memory of the DSP unit and post-process them in software on the main CPU

The synchronization between these threads is designed on top of the classical parallel programming primitives of the RTOS (semaphores, mutex, etc.).

Once parallelized the application altogether the RTOS are cross-compiled for the main CPU of EMBB. Function calls to DMA transfers or DSP processing are no longer emulated in software but replaced by actual calls to the different hardware device drivers. The application can be run on the hardware target processor but as the debugging capabilities are always limited on embedded systems, the software design kit comprises a SystemC virtual prototype built on top of the SoCLib [3] open source library. The virtual prototype runs on a regular desktop PC. The EMBB main CPU is simulated by the SoCLib Sparc Instruction Set Simulator (ISS) and the DSP units are simulated by custom SystemC designs which are mainly cycle-approximate wrappers around calls to the synchronous `libembb` library. The virtual prototype can be connected to the GNU gdb debugger for step by step execution, registers and memories examination, etc. It also offers an integrated Valgrind-like memory checker that catches accesses to uninitialized memory locations, memory leaks, etc. Finally, the virtual prototype can be used to generate waveforms representing the timed behaviour of the whole processor (Figure 3.1). These waveforms are analysed to understand the parallel interleaving between activities, track bugs due to the parallelization, understand where performance limitations come from and improve the global performance.

The three implemented applications have been reworked for parallelization as described and cross-compiled. Extensive simulations have been run on the virtual prototype in order to debug them and optimize their performance.

3.4 Execution and Validation on Target Hardware

Once validated on the virtual prototype the application is ready to run on the hardware target. It must be recompiled because the SoCLib-based virtual prototype has subtle differences with the real EMBB processor but exactly the same source code is used. The produced binary is downloaded on the prototyping board and the main CPU is requested to run the application. In most cases the behaviour is the same as on the virtual prototype (but the speed which is orders of magnitude faster than in simulation). In some rare cases mismatches can be encountered and, in case they lead to undesirable results, investigated. The source of these mismatches is the cycle inaccuracy of the virtual prototype: as it is optimized for simulation speed it does not reflect 100%

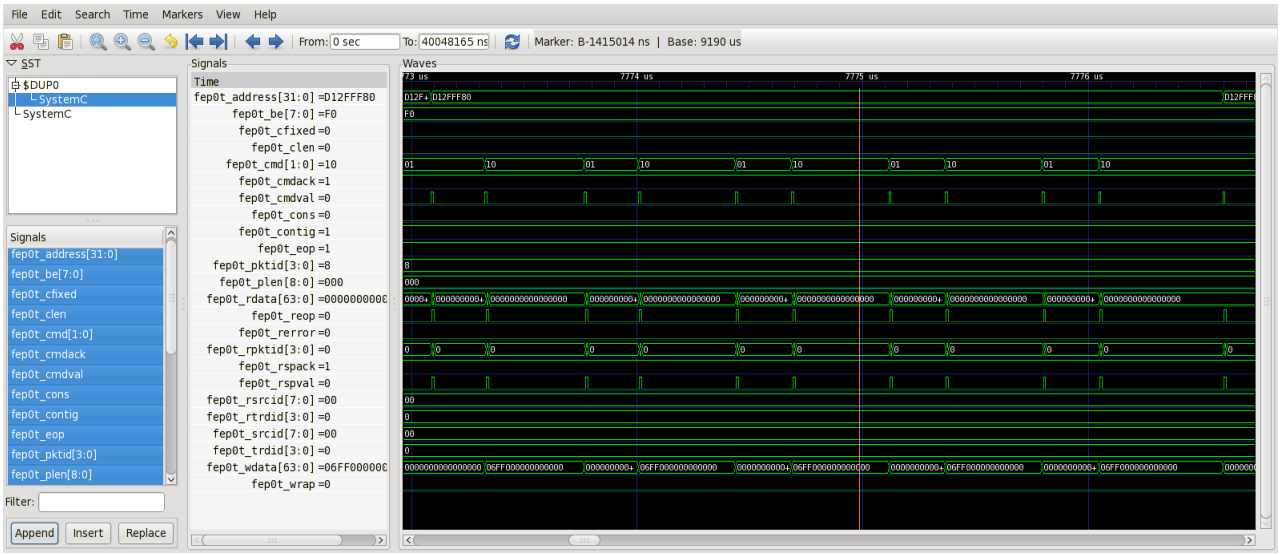

Figure 3.1: A simulation waveform from the EMBB virtual prototype

exactly what happens on the bare metal. It can thus be that an application runs as expected on the virtual prototype but not on the hardware. This is always an indication that some parallelization-related bugs have not been caught yet. Typically, a race condition still exists that, by chance, did not show up during simulations. These bugs are the most difficult to identify. It is sometimes necessary to run the same (usually very long) sequence of operations on a Hardware Description Language (HDL) simulator using the HDL model of the EMBB processor. The simulation speed is dramatically reduced and the simulation run time until the bug is encountered can be extremely long (hours or even days). Debugging tools are available on the hardware target too and can be used to identify and fix these bugs but as the main CPU of EMBB has no Memory Management Unit (MMU), at least not yet, bugs must be caught before they crash the main CPU, which is sometimes not trivial. For all these reasons (slow HDL simulations and limited on-target debugging capabilities) it is thus extremely important that the parallelization is done with the most aggressive programming techniques and extensively validated on the virtual prototype in order to avoid such situations. Up to now, for the SACRA activities, this situation has never been encountered.

The activities of task WP5.4 (software generation) are an attempt to solve as much as possible these issues by relying on automatic software generation from very high level abstract models of the applications, the target processor and the mapping of the former to the later. The results are encouraging but the tool chain is not yet mature enough to be considered as a definite answer.

4 IMPLEMENTATION OF SENSING ALGORITHMS

This chapter presents the two implemented sensing algorithms, the way they have been implemented and gives resulting performance figures (memory usage, processor load, estimated power consumption).

4.1 Energy Detection

4.1.1 Description of the Algorithm

The ED is a sensing algorithm, probably the simplest of all. It operates on the discrete received signal $y(k)$, computes its energy on a given number N of samples $y(1), y(2), \ldots, y(N)$, and compares this score with a precomputed threshold. In the context of the SACRA project it has been decided to implement a continuous time domain energy detection. The different parameters have been restricted to meaningful ranges in order to simplify the implementation on the target platform. The sensing bandwidth is restricted to 3 most relevant values (5, 10 or 20 MHz). These bandwidths are typical LTE ones and they fully cover the requirements of the SACRA demonstrations. Allowing the other LTE bandwidths (1.4, 3 and 15 MHz) would not be a problem but the added value would be limited which does not justify the extra design effort. The corresponding sampling frequencies F_s are 7.68, 15.36 and 30.72 mega-samples per second (Ms/s) respectively. The incoming samples stream is split in segments, L_s-samples each, and each segment is processed independently (Figure 4.1).

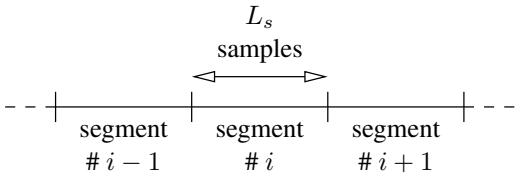

Figure 4.1: Split of input stream in L_s-samples segments

For ED L_s is fixed to 2048, which optimizes the data transfers and the required processing power. The sensing centre frequency is between 300 MHz and 6 GHz. The sensing periods T_s are defined by the total number N of sensed samples which is restricted to be a multiple of $L_s = 2048$ between $1 \times L_s$ and $16383 \times L_s$: $T_s = N/F_s = L_s \times M/F_s$, with $1 \leq M \leq 16383$. The constraints on the sampling periods T_s are thus:

$$266.67\mu s \leq \quad Ts \quad \leq 4.37s \text{ (5 MHz bandwidth)} \tag{4.1}$$

$$133.34\mu s \leq \quad Ts \quad \leq 2.18s \text{ (10 MHz bandwidth)} \tag{4.2}$$

$$66.67\mu s \leq \quad Ts \quad \leq 1.09s \text{ (20 MHz bandwidth)} \tag{4.3}$$

Table 4.1 summarizes the input parameters of ED. $L_s = 2048$ being constant is not considered an input parameter.

Name	Type	Range	Description
BW	Integer	$BW \in \{0, 1, 2\}$	Bandwidth selector (5, 10, 20 MHz)
F_c	Integer	$300 \times 10^6 \leq F_c \leq 6 \times 10^9$	Centre frequency (Hz)
M	Integer	$1 \leq M \leq 16383$	Sensing period (L_s-samples segments)

Table 4.1: ED input parameters

When launched with parameters BW, F_c and M, ED configures the Radio Frequency (RF) chain for the reception of the specified band (BW, F_c) and the corresponding sampling frequency. N acquired samples are then processed and the final result of ED is:

$$ED(BW, F_c, M) \approx \frac{\sum_{k=1}^{k=N} |y(k)|^2}{2^{15}}, \text{ with } N = M \times L_s \tag{4.4}$$

The result is returned to the higher levels of the application running on the host PC and is used for noise estimation and signal detection after comparison with a threshold that depends on the noise level and on the input parameters.

4.1.2 Implementation of the Algorithm on the Target Processor

On the SACRA baseband platform the implementation of the ED is straightforward. The interface with the Analogue/Digital (A/D) converters is configured to continuously acquire input samples from the selected RX chain.

The FEP DSP unit fetches the input samples with DMA transfers between the RX FIFO of PP and its own internal memory and computes the energies on L_s-samples segments with CWM instructions. The parameters of the CWM instructions are set so that the result vector is discarded and only the « SMA-values » (sum, max, argmax, min, argmin) are retained (32 bytes) and, among them, only the real part[1] of the sum (32 bits, signed integer) is used in subsequent steps. By definition of the CWM operation and because of the selected parameters, the real part of the sum on segment $0 \leq i < M$ is:

$$\left\lfloor \frac{\sum_{k=1}^{L_s} |y(i \times M + k)|^2}{2^{15}} \right\rfloor \leq \frac{L_s \times 2 \times \left(2^{15}\right)^2}{2^{15}} = 2^{27} \tag{4.5}$$

Each L_s-samples segment thus produces an intermediate energy value, represented as a 32 bits integer (the other SMA values are discarded). The main CPU of the baseband processor accumulates these M intermediate results with 64 bits precision. The final result is returned to the upper layers.

Memory Management

The memory management of the ED application is also straightforward. The internal memory of FEP is split into four banks, 16 k-bytes each. The topmost part of bank #0 is used to store the incoming data segments from the RX FIFO of PP. Two buffers, L_s samples each, are used in flip-flop mode in order to allow the processing of a segment in parallel with the DMA transfer of the next one. The output of the CWM operation (32 bytes of SMA values) is stored in bank #1, always at the same address. The real part of the sum (32 bits integer in the $[0 \ldots 2^{27}]$ range) is used in the subsequent processing steps (accumulation over M segments by the main

[1] Technically, a square of modulus is a real number and has no imaginary part. But in FEP, thanks to pre- and post-processing type conversions, all operations are computed on 32 bits complex numbers and produce 32 bits complex numbers. A real result is thus represented as a complex which imaginary part is null.

CPU, post-processing, normalization, threshold comparison and decision making by the host PC). Figure 4.2 illustrates the memory management. The first data segment is stored in the topmost buffer of bank #0. As soon as it is fully transferred from PP, another DMA transfer is launched for the second data segment but with the second buffer of bank #0 as destination. This way, the DMA transfer takes place in parallel with the CWM operation which input is the topmost buffer of bank #0 and output is always the top of bank #1. Finally, the main CPU reads the real part of the sum out from bank #1 and accumulates the total energy. Banks #2 and #3 are unused. All in all the bank usage is summarized in Table 4.2.

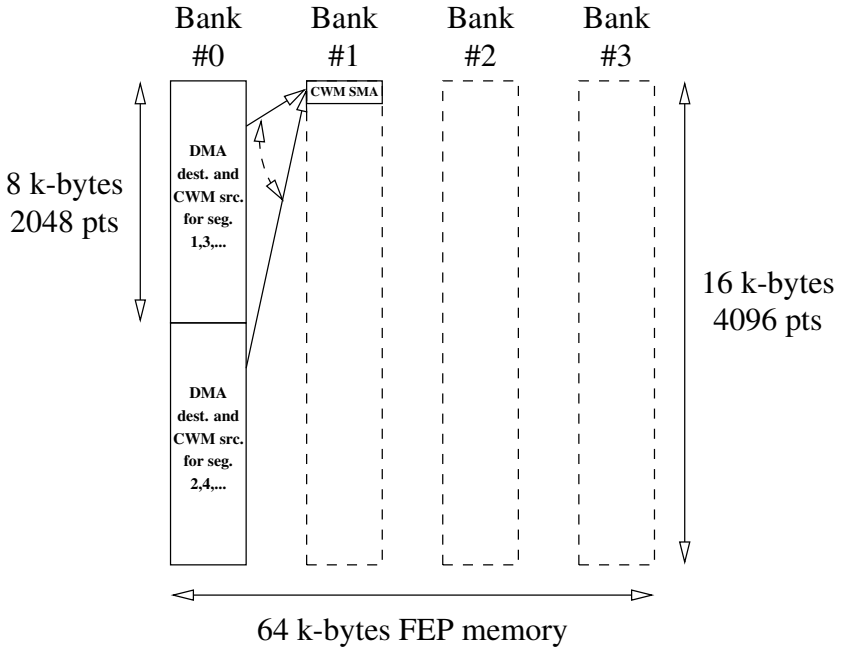

Figure 4.2: FEP internal memory management ($L_s = 2048$) for ED

L_s	Bank usage (%)				Total
	#0	#1	#2	#3	
2048	100%	0.2%	0%	0%	25.05%

Table 4.2: FEP memory usage for ED

Parallelization

FEP is involved in all steps of the computation. The only potential sources of parallelism are between:

- The DMA transfers from PP to FEP
- The computations by FEP
- The control and the accumulation of the segments' energies by the main CPU

Algorithm 4.1 represents the pseudo-code of the complete application. The detection scores are pushed in a software FIFO that the host system reads periodically (not represented in the algorithm description).

Algorithm 4.1 The parallelized ED application

1: $dma.engine = FEP$; ▷ use DMA engine of FEP to transfer from PP to FEP
2: $dma.src = PPRX0$; ▷ source of PP to FEP DMA transfers is RX0 FIFO of PP
3: $dma.len = L_s$; ▷ length of PP to FEP DMA transfers
4: $cwm.len = L_s$; ▷ length of CWM
5: $cwm.sdst = FEPBANK1$; ▷ destination of CWM operations (SMA only)
6: $cwm.sma = SMAONLY$; ▷ store only sum, max, argmax, ... (SMA values) of CWM
7: $S0 = 0; S1 = 0$; ▷ semaphores
8: $idx = 0$; ▷ unique index
9: **procedure** DIRECT_MEMORY_TRANSFERS_THREAD
10: **for** $i = 0; i < M; i = i + 1$ **do**
11: **if** i even **then**
12: $dma.dst = FEPBANK0$; ▷ destination of PP to FEP DMA transfers
13: **else**
14: $dma.dst = FEPBANK0 + L_s$; ▷ destination of PP to FEP DMA transfers
15: **end if**;
16: WAIT($S0 < 2$); ▷ wait until one of the 2 destination buffers is free
17: DMA(dma); ▷ launch DMA transfer
18: WAIT(dma); ▷ wait until end of DMA transfer
19: $S0 = S0 + 1$; ▷ mark destination buffer as full
20: **end for**;
21: **end procedure**;
22: **procedure** COMPUTE_DETECTION_SCORES_THREAD
23: **for** $i = 0; i < M; i = i + 1$ **do**
24: **if** i even **then**
25: $cwm.src = FEPBANK0$; ▷ source of CWM operation
26: **else**
27: $cwm.src = FEPBANK0 + L_s$; ▷ source of CWM operation
28: **end if**;
29: WAIT($S0 > 0$); ▷ wait until one of the 2 source buffers is full
30: WAIT($S1 < 1$); ▷ wait until the destination buffer is free
31: FEP(cwm); ▷ launch CWM
32: WAIT(cwm); ▷ wait until end of CWM
33: $S0 = S0 - 1$; ▷ mark source buffer as free
34: $S1 = S1 + 1$; ▷ mark the destination buffer as full
35: **end for**
36: **end procedure**;
37: **procedure** ACCUMULATE_DETECTION_SCORES_THREAD
38: $score = 0$; ▷ initialize total energy
39: **for** $i = 0; i < M; i = i + 1$ **do**
40: WAIT($S1 > 0$); ▷ wait until the source buffer is full
41: $score = score +$ READ_INT32($FEPBANK1$); ▷ accumulate energy
42: $S1 = S1 - 1$; ▷ mark source buffer as free
43: **end for**
44: PUSH($\{idx, score\}$); ▷ push in software FIFO
45: $idx = idx + 1$; ▷ increment unique index
46: **end procedure**;

Threshold Computation

In order to compute the threshold for the ED, we proceed by a Monte Carlo algorithm. Once the target probability of false alarm P_{fa} fixed, we feed the ED with a unit variance noise input signal N_T times, say $N_T = 1000$, and store the score for each iteration in a temporary vector ($temp\_vect$), sort it, and the index giving the exact value of the threshold is the one having as index $ind = N_T - floor(P_{fa} * N_T)$. The exact threshold for the fixed P_{fa} is then given by: $\gamma = temp\_vect(ind)$.

4.1.3 Performance Analysis

The required raw processing power of ED is one square of modulus per input sample, plus their accumulation. The FEP computes the CWM operations, including accumulation, on 2048-components vectors in 1036 clock cycles (about two vector components per cycle). Table 4.3 summarizes the required processing power as a percentage of the full load of FEP for different input sampling frequencies, corresponding to three typical LTE downlink bandwidths (5, 10 and 20 MHz) and for the three different target technologies defined in Chapter 2 (100, 500 and 1000 MHz clock frequencies). The table also presents the estimated power consumption of FEP in the 22 nm version of the baseband processor, at 1 GHz, and under the two different operating voltages discussed in Chapter 2.

BW	Clock cycles CWM	F (MHz)	Load (%)	Power (mW)	
				High voltage	Low voltage
5 MHz	1036	100	3.89	-	-
		500	0.78	-	-
		1000	0.39	0.17	0.11
10 MHz	1036	100	7.77	-	-
		500	1.55	-	-
		1000	0.78	0.35	0.23
20 MHz	1036	100	15.54	-	-
		500	3.11	-	-
		1000	1.55	0.70	0.45

Table 4.3: FEP loads for different ED use cases

While the main CPU of EMBB is involved in the ED application, Table 4.3 does not present its load because it is negligible: considering the segments' length and the sampling frequency, the shortest segment duration (for a 20 MHz bandwidth) is $66.67\mu s$, that is, 66667 clock cycles at 1 GHz. The post-processing implemented in software on the main CPU consists in one 64-bits add per segment, that is, a dozen CPU instructions or clock cycles per segment, or about 0.018% of the full load at 1 GHz.

As can be seen the total required processing power is extremely small on realistic targets (500 or 1000 MHz) and even on the much slower SACRA prototype (100 MHz) the ED application fits without problem. The estimated power consumption of FEP for the ED application is almost negligible compared to the 300 mW reference power budget defined in Chapter 2.

4.1.4 Proposed Software API

The host PC controlling the sensing drives the ED application and retrieves the computed energies through a simplified C-language API, which will also be available from Matlab®. Two different modes are supported: the one-shot mode in which the application is launched, returns an energy level and quits, and the continuous

mode in which the application runs continuously and periodically stores energy levels in a software FIFO. The one-shot mode API is depicted in listing 4.1 and the continuous mode API in listing 4.2.

```
/* ED returns -1 on failure (ED already running or invalid parameters) and 0 on
 * success. When successful, ED returns after energy computation (blocking call)
 * and the energy level is stored in the 64-bits integer *e */
int ED(
    int BW, /* bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    int Fc, /* centre frequency (Hz), in [300e6 ... 6e9] */
    int M,  /* sensing period (number of 2048-samples segments), in [1 ... 16383] */
    int64_t *e /* returned energy level */
    );
```

Listing 4.1: ED API in one-shot mode

```
/* ED_start returns -1 on failure (ED already running or invalid parameters) and
 * 0 on success. ED_start returns immediately (non blocking call) and, when
 * successful, starts storing energy levels in the software FIFO, altogether
 * with a unique, self-incremented index. The index of the first energy level is
 * 0. The software FIFO and the unique index are re-initialized upon ED_start
 * call. Energy levels that were still in the FIFO are lost. */
int ED_start(
    int BW, /* bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    int Fc, /* centre frequency (Hz), in [300e6 ... 6e9] */
    int M   /* sensing period (number of 2048-samples segments), in [1 ... 16383] */
    );

/* ED_stop returns -1 on failure (ED not running) and 0 on success. ED_stop
 * returns immediately (non blocking call) and, when successful, the currently
 * running ED application is stopped at the end of the current energy
 * computation. */
int ED_stop(
    void
    );

/* ED_pop returns -1 on failure (software FIFO empty), else pops the next
 * {index, energy level} pair from the software FIFO, decrements the counter of
 * FIFO entries, and returns the current value of the counter. ED_pop returns
 * immediately (non blocking call). */
int ED_pop(
    int *idx,  /* unique index */
    int64_t *e /* energy level */
    );
```

Listing 4.2: ED API in continous mode

4.2 Welch Periodograms

4.2.1 Description of the Algorithm

The WPD can be seen as a generalisation of the ED. It is based on the detection of the energy of the received signal via DFT. It is a modification of a basic periodogram. WPD attempts to improve the statistical properties of the periodograms by dividing the received data sequence into segments and performing averaging over the segments. Using the WPD, the signal energy can be measured in the sub-bands of interest simultaneously. In addition, it enables to estimate the energy of the noise from the regions where the signal is less predominant.

More details on the WPD can be found in SACRA deliverable D2.1 [13] and in [6]. The idea of using the WPD for noise level estimation is introduced in this deliverable.

WPD operates on the discrete received signal $y(k)$, computes its energy in a given sub-band and on a given number N of samples $y(1), y(2), \ldots, y(N)$, and compares this score with a precomputed threshold. As in the ED case different parameters have been restricted to meaningful ranges in order to simplify the implementation. The sensing bandwidth is restricted to the same 3 possible values (5, 10 or 20 MHz) with the same corresponding sampling frequencies F_s: 7.68, 15.36 and 30.72 Ms/s respectively. The stream of incoming samples is again split in segments, L_s-samples each, and each segment is processed independently but L_s depends on the bandwidth: $L_s = 512, 1024, 2048$ for bandwidths of 5, 10 and 20 MHz respectively. The sensing centre frequencies are again between 300 MHz and 6 GHz. As for ED, the sensing period T_s is defined by the total number M of segments to process, with $1 \leq M \leq 16383$. The sampling period $T_s = M \times L_s/F_s$ is thus in the $[66.67\mu s \ldots 1.09s]$ range. The sub-band of interest is defined by its width $1 \leq L \leq L_s$ and the bin index $0 \leq F_{start} \leq L_s - L$ of its lower edge. A last parameter $0 \leq r \leq 7$ is used to define the downscaling factor (2^r) applied to the result of the Fourier transform. Depending on the input signal level r can be set to different values to prevent overflows and saturations during the computations. Table 4.4 summarizes the input parameters of WPD.

Name	Type	Range	Description
BW	Integer	$BW \in \{0, 1, 2\}$	Bandwidth selector (5, 10, 20 MHz)
F_c	Integer	$300 \times 10^6 \leq F_c \leq 6 \times 10^9$	Centre frequency (Hz)
M	Integer	$1 \leq M \leq 16383$	Sensing period (L_s-samples segments)
L	Integer	$1 \leq L \leq L_s - F_{start}$	Width of sub-band of interest (sub-carriers)
F_{start}	Integer	$0 \leq F_{start} \leq L_s - L$	Bin index of lower edge of sub-band of interest (sub-carriers)
r	Integer	$0 \leq r \leq 7$	Log2 of downscaling factor for FT outputs

Table 4.4: WPD input parameters

Table 4.5 presents the dependencies between the bandwidth, the sampling frequency and the segments' length. The table also suggests values for the r parameter. These values are conservative ones guaranteeing that the FT outputs are not saturated, but smaller values can be used if the input signal level is known to be less than its theoretical maximum.

BW	Bandwidth (MHz)	F_s (MHz)	L_s (samples)	Suggested r value
0	5	7.68	512	5
1	10	15.36	1024	6
2	20	30.72	2048	6

Table 4.5: Inter-dependencies between WPD parameters

When launched with parameters BW, F_c, M, L, F_{start} and r, WPD configures the RF for the reception of the specified band (BW, F_c) and the corresponding sampling frequency. $N = M \times L_s$ acquired samples are then processed and the final result of WPD is defined by:

$$\forall\, 0 \leq i < M,\ Y_i = [y(i \times L_s) + 1 \ldots y((i+1) \times L_s)] \tag{4.6}$$

$$Z_i \approx \frac{DFT_{L_s}(Y_i)}{2^r} \tag{4.7}$$

$$w_i \approx \frac{\sum_{k=F_{start}}^{k=F_{start}+L-1} |Z_i(k)|^2}{2^{15}} \tag{4.8}$$

$$WPD(BW, F_c, M, L, F_{start}) = \sum_{i=0}^{i=M-1} (w_i) \tag{4.9}$$

where $F = DFT_{L_s}(X)$ is defined as in the first appendix of the SACRA D5.2 deliverable [8]:

$$\forall\, k \in [0, L_s - 1],\ F(k) = \frac{1}{\sqrt{L_s}} \sum_{l=0}^{l=L_s-1} X(l) \times e^{\frac{-2\pi jkl}{L_s}} \tag{4.10}$$

The result is returned to the higher levels of the application running on the host PC and is used for noise estimation and signal detection after normalization and comparison with a threshold γ that depends on the noise level and on the specified parameters [10, 14]:

$$\gamma = \frac{\sqrt{2}}{\sqrt{M \times L_s}} \times \sigma_w^2 \times Q^{-1}(P_{FA}) + \sigma_w^2 \tag{4.11}$$

where σ_w^2 is the noise variance, P_{FA} is the target false alarm probability, and Q^{-1} is the inverse of the Q function defined as:

$$Q(t) \equiv 0.5 \times \left(1 - \mathrm{erf}\left(\frac{t}{\sqrt{2}}\right)\right) \tag{4.12}$$

4.2.2 Implementation of the Algorithm on the Target Processor

On the SACRA baseband platform the implementation of the WPD is straightforward. The interface with the A/D converters is configured to continuously acquire input samples from the selected RX chain.

The FEP DSP unit fetches the input samples with DMA transfers between the RX FIFO of PP and its own internal memory and computes the energies on L_s-samples segments with FT (Fourier Transform) and CWM (Component-Wise square of Modulus) instructions. The parameters of the FT instructions are set to avoid overflows (the r rescaling parameter) and CWM instructions are set so that the result vector is discarded and only the « SMA-values » (sum, max, argmax, min, argmin) are retained (32 bytes) and, among them, only the real part of the sum (32 bits, signed integer) is used in subsequent steps. By definition of the CWM operation and because of the selected parameters, the real part of the sum on segment $0 \leq i < M$ is:

$$Z_i \approx \frac{DFT_{L_s}(Y_i)}{2^r} \tag{4.13}$$

$$\Re(sum) = \left\lfloor \frac{\sum_{k=F_{start}}^{k=F_{start}+L-1} |Z_i(k)|^2}{2^{15}} \right\rfloor \leq \frac{L_s \times 2 \times \left(2^{15}\right)^2}{2^{15}} = 2^{27} \tag{4.14}$$

Each L_s-samples segment thus produces an intermediate energy value, represented as a 32 bits integer (the other SMA values are discarded). The main CPU of the baseband processor accumulates these M intermediate results with 64 bits precision. The final result is returned to the upper layers.

Memory Management

The memory management of the WPD application is also straightforward. The internal memory of FEP is split into four banks, 16 k-bytes each. The topmost part of bank #0 is used to store the incoming data segments from the RX FIFO of PP. Two buffers, L_s samples each, are used in flip-flop mode in order to allow the processing of a segment in parallel with the DMA transfer of the next one. The output of the FT operation is stored in bank #1, always at the same address. The output of the CWM operation (32 bytes of SMA values) is stored in bank #2, always at the same address. The real part of the sum (32 bits integer in the $[0 \dots 2^{27}]$ range) is used in the subsequent processing steps. Figure 4.3 illustrates the memory management in the $BW = 2$ case ($F_s = 20$ MHz, $L_s = 2048$). The first data segment is stored in the topmost buffer of bank #0. As soon as it is fully transferred from PP, another DMA transfer is launched for the second data segment but with the second buffer of bank #0 as destination. This way, the DMA transfer takes place in parallel with the FT operation which input is the topmost buffer of bank #0 and output is always the top of bank #1. The CWM operation follows with the top of bank #1 as input and the the top of bank #2 as output. Finally, the main CPU reads the real part of the sum out from bank #2 and accumulates the total energy. Bank #3 is unused. All in all the bank usage is summarized in Table 4.6.

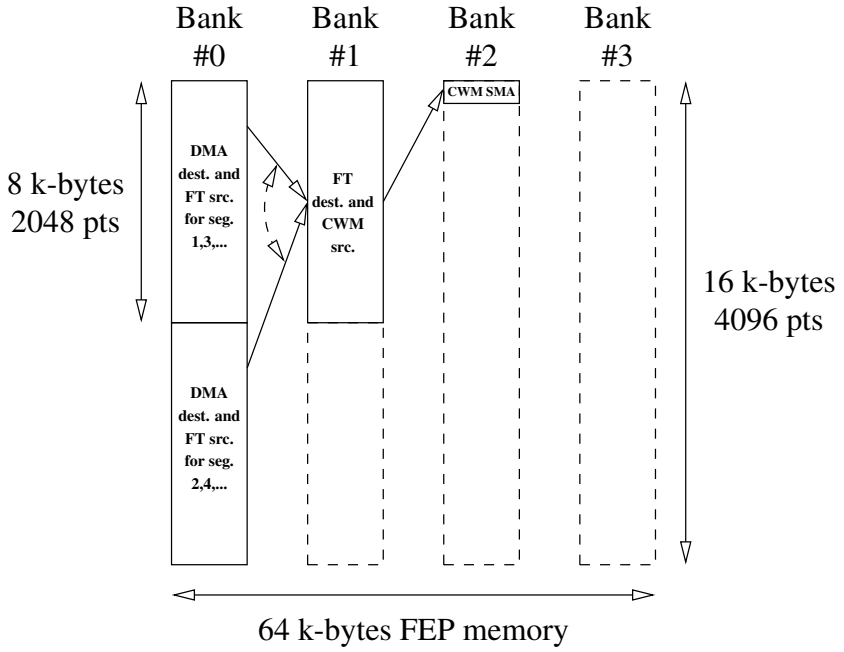

Figure 4.3: FEP internal memory management ($L_s = 2048$) for WPD

Parallelization

FEP is involved in all steps of the computation. The only potential sources of parallelism are between:

- The DMA transfers from PP to FEP
- The computations by FEP
- The control and the accumulation of segments' energies by the main CPU

L_s	Bank usage (%)				Total
	#0	#1	#2	#3	
512	25%	12.5%	0.2%	0%	9.42%
1024	50%	25%	0.2%	0%	18.80%
2048	100%	50%	0.2%	0%	37.55%

Table 4.6: FEP memory usage for WPD

Algorithm 4.2 represents the pseudo-code of the complete application in the 20 MHz bandwidth configuration. The detection scores are pushed in a software FIFO that the host system reads periodically (not represented in the algorithm description).

4.2.3 Performance Analysis

Computation Noise

The target hardware uses fixed-point processing. Due to quantization effects the actual results differ from the ideal ones. In order to estimate this quantization noise a comparison has been made between a Matlab® implementation and the fixed-point implementation using the `libembb` software library. Table 4.7 presents the set-up of the simulation.

Detection technique	Single sensor, single-antenna link, path loss and line-of-sight parameters turned off
Primary user	Terrestrial Digital Video Broadcasting (DVB-T) over a single path Rayleigh channel, 2K mode, 8 MHz channelization, 1705 sub-carriers per Orthogonal Frequency-Division Multiplexing (OFDM) symbol, approximately 4KHz apart
Modulation	16 Quadrature Amplitude Modulation (QAM)
Guard interval	$1/4$
WPD segments' length	$L_s = 512$
Oversampling	No
False alarm probability	$P_{FA} = 0.05$
Minimum Signal to Noise Ratio	SNR_{min}: $[-50 : 2 : 20]$ dB

Table 4.7: Set-up for simulation of WPD computation noise

The results of 1000 simulated iterations have been averaged. The main objective was to estimate the required sensing time to achieve Probability of Detection (PD) PD1 = 0.90 and PD2 = 0.95 with a required SNR = -18.86 dB [11, 12] for 10 MHz LTE configuration. Figures 4.4 and 4.5 compare the performance of the Matlab® and the fixed-point implementations. From Figure 4.4, we see that PD1 is achieved with a 14 ms sensing time with both implementations. Similarly, with a 20.5 ms sensing time, PD2 is achived with SNR = -18.86 dB (Figure 4.5). The performance of the fixed-point `libembb`-based implementation is practically the same as that of the Matlab® implementation in a reasonable probability of detection range.

Algorithm 4.2 The parallelized WPD application

1: $dma.engine = FEP$; ▷ use DMA engine of FEP to transfer from PP to FEP
2: $dma.src = PPRX0$; ▷ source of PP to FEP DMA transfers is RX0 FIFO of PP
3: $dma.len = L_s$; ▷ length of PP to FEP DMA transfers
4: $ft.len = L_s$; ▷ length of FT
5: $ft.r = r$; ▷ downscaling factor of FT
6: $ft.dst = FEPBANK1$; ▷ destination of FT operations
7: $cwm.len = L$; ▷ length of CWM
8: $cwm.src = FEPBANK1 + F_{start}$; ▷ source of CWM operations
9: $cwm.sdst = FEPBANK2$; ▷ destination of CWM operations (SMA only)
10: $cwm.sma = SMAONLY$; ▷ store only sum, max, argmax, ... (SMA values) of CWM
11: $S0 = 0; S1 = 0$; ▷ semaphores
12: $idx = 0$; ▷ unique index
13: **procedure** DIRECT_MEMORY_TRANSFERS_THREAD
14: **for** $i = 0; i < M; i = i + 1$ **do**
15: **if** i even **then**
16: $dma.dst = FEPBANK0$; ▷ destination of PP to FEP DMA transfers
17: **else**
18: $dma.dst = FEPBANK0 + L_s$; ▷ destination of PP to FEP DMA transfers
19: **end if**;
20: WAIT($S0 < 2$); ▷ wait until one of the 2 destination buffers is free
21: DMA(dma); ▷ launch DMA transfer
22: WAIT(dma); ▷ wait until end of DMA transfer
23: $S0 = S0 + 1$; ▷ mark destination buffer as full
24: **end for**;
25: **end procedure**;
26: **procedure** COMPUTE_DETECTION_SCORES_THREAD
27: **for** $i = 0; i < M; i = i + 1$ **do**
28: **if** i even **then**
29: $ft.src = FEPBANK0$; ▷ source of FT operation
30: **else**
31: $ft.src = FEPBANK0 + L_s$; ▷ source of FT operation
32: **end if**;
33: WAIT($S0 > 0$); ▷ wait until one of the 2 source buffers is full
34: FEP(ft); ▷ launch FT
35: WAIT(ft); ▷ wait until end of FT
36: $S0 = S0 - 1$; ▷ mark source buffer as free
37: WAIT($S1 < 1$); ▷ wait until the destination buffer is free
38: FEP(cwm); ▷ launch CWM
39: WAIT(cwm); ▷ wait until end of CWM
40: $S1 = S1 + 1$; ▷ mark the destination buffer as full
41: **end for**
42: **end procedure**;
43: **procedure** ACCUMULATE_DETECTION_SCORES_THREAD
44: $score = 0$; ▷ initialize total energy
45: **for** $i = 0; i < M; i = i + 1$ **do**
46: WAIT($S1 > 0$); ▷ wait until the source buffer is full
47: $score = score +$ READ_INT32($FEPBANK2$); ▷ accumulate energy
48: $S1 = S1 - 1$; ▷ mark source buffer as free
49: **end for**
50: PUSH($\{idx, score\}$); ▷ push in software FIFO
51: $idx = idx + 1$; ▷ increment unique index
52: **end procedure**;

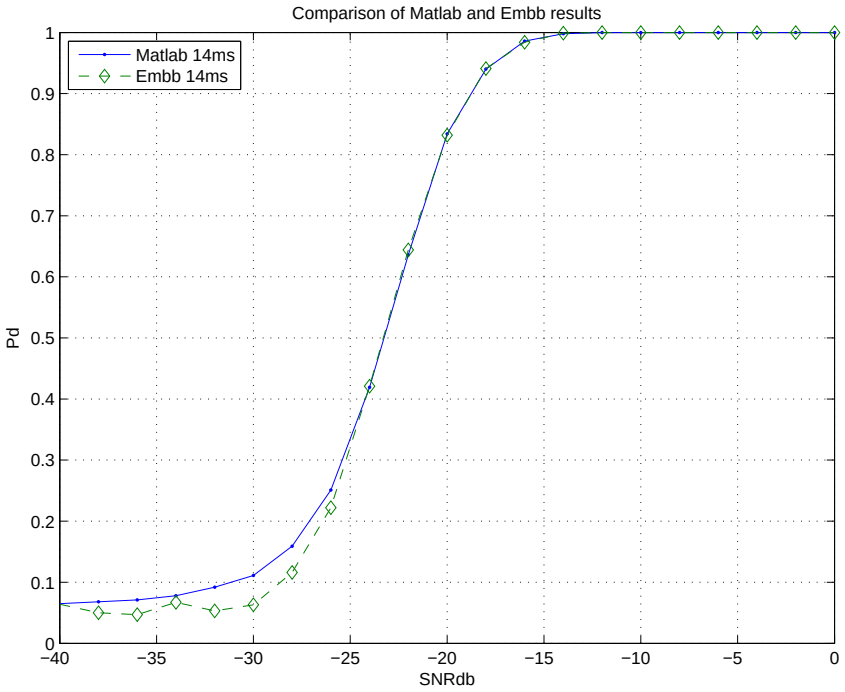

Figure 4.4: PD = 0.9 with SNR = -18,86 dB and sensing time = 14 ms

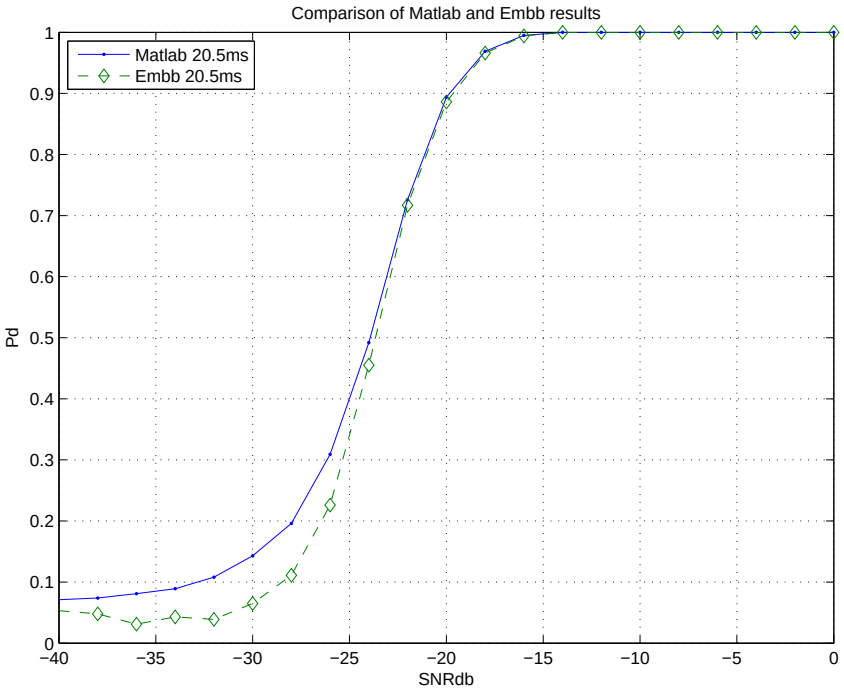

Figure 4.5: PD = 0.95 with SNR = -18,86 dB and sensing time = 20.5 ms

Computation Load and Power Consumption

The required raw processing power of WPD is one L_s-points Fourier transform per segment, one square of modulus per input sample, plus their accumulation. The FEP computes the 512-, 1024- and 2048-points Fourier transforms in 387, 707 and 1616 clock cycles respectively. It computes the CWM operations, including accumulation, in $12 + \lceil \frac{L}{2} \rceil$ (about two vector components per cycle). In the worst case execution time, when the sub-band of interest is the whole band ($L = L_s, F_{start} = 0$), the CWM operations are computed in 268, 524 and 1036 clock cycles for $L_s = 512$, $L_s = 1024$ and $L_s = 2048$ respectively. Table 4.8 summarizes the worst case required processing power as a percentage of the full load of FEP for different input sampling frequencies, corresponding to three typical LTE downlink bandwidths (5, 10 and 20 MHz) and for the three different target technologies defined in Chapter 2 (100, 500 and 1000 MHz clock frequencies). The table also presents the estimated power consumption of FEP in the 22 nm version of the baseband processor, at 1 GHz, and under the two different operating voltages discussed in Chapter 2.

BW	Clock cycles			F (MHz)	Load (%)	Power (mW)	
	CWM	FT	Total			High Vdd	Low Vdd
5 MHz	387	268	655	100	9.83	-	-
				500	1.97	-	-
				1000	0.98	0.44	0.29
10 MHz	707	524	1231	100	18.47	-	-
				500	3.69	-	-
				1000	1.85	0.83	0.54
20 MHz	1616	1036	2652	100	39.78	-	-
				500	7.96	-	-
				1000	3.98	1.79	1.16

Table 4.8: FEP loads for different WPD use cases

As for energy detection, while the main CPU of EMBB is involved in the WPD application, Table 4.8 does not present its load because it is negligible: considering the segments' length and the sampling frequency, the segment duration is $66.67\mu s$, that is, 66667 clock cycles at 1 GHz. The post-processing implemented in software on the main CPU consists in one 64-bits add per segment, that is, a dozen CPU instructions or clock cycles per segment, or about 0.018% of the full load at 1 GHz.

And again, as can be seen the total required processing power is extremely small on realistic targets (500 or 1000 MHz) and even on the much slower SACRA prototype (100 MHz) the WPD application fits without problem. The estimated power consumption of FEP for the WPD application is almost negligible compared to the 300 mW reference power budget defined in Chapter 2.

4.2.4 Proposed Software API

The host PC controlling the sensing drives the WPD application and retrieves the computed energies through a simplified C-language API, which will also be available from Matlab®. Two different modes are supported: the one-shot mode in which the application is launched, returns an energy level and quits, and the continuous mode in which the application runs continuously and periodically stores energy levels in a software FIFO. The one-shot mode API is depicted in listing 4.3 and the continuous mode API in listing 4.4.

```
1 /* WPD returns -1 on failure (WPD already running or invalid parameters) and 0
```

```
 * on success. When successful, WPD returns after energy computation (blocking
 * call) and the energy level is stored in the 64-bits integer *e */
int WPD(
    int BW, /* bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    int Fc, /* centre frequency (Hz), in [300e6 ... 6e9] */
    int M,  /* sensing period (number of 2048-samples segments), in [1 ... 16383] */
    int r,  /* log2 of FT downscaling factor, in [0 ... 7] */
    int64_t *e /* returned energy level */
    );
```

Listing 4.3: WPD API in one-shot mode

```
/* WPD_start returns -1 on failure (WPD already running or invalid parameters)
 * and 0 on success. WPD_start returns immediately (non blocking call) and, when
 * successful, starts storing energy levels in the software FIFO, altogether
 * with a unique, self-incremented index. The index of the first energy level is
 * 0. The software FIFO and the unique index are re-initialized upon WPD_start
 * call. Energy levels that were still in the FIFO are lost. */
int WPD_start(
    int BW, /* bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    int Fc, /* centre frequency (Hz), in [300e6 ... 6e9] */
    int M,  /* sensing period (number of 2048-samples segments), in [1 ... 16383] */
    int r   /* log2 of FT downscaling factor, in [0 ... 7] */
    );

/* WPD_stop returns -1 on failure (WPD not running) and 0 on success. WPD_stop
 * returns immediately (non blocking call) and, when successful, the currently
 * running WPD application is stopped at the end of the current energy
 * computation. */
int WPD_stop(
    void
    );

/* WPD_pop returns -1 on failure (software FIFO empty), else pops the next
 * {index, energy level} pair from the software FIFO, decrements the counter of
 * FIFO entries, and returns the current value of the counter. WPD_pop returns
 * immediately (non blocking call). */
int WPD_pop(
    int *idx,  /* unique index */
    int64_t *e /* energy level */
    );
```

Listing 4.4: WPD API in continous mode

5 IMPLEMENTATION OF CLASSIFICATION ALGORITHMS

This chapter is dedicated to classification algorithms and is divided in two parts:

1. Signal Classification using HOC (previously presented in D2.5 [12]) - which has been implemented (see Section 5.1) and

2. Signal Classification without Quiet Period using cyclostationary properties (previously presented in D2.2 [5]) - which has not been implemented (see Section 5.2).

Please note that the second algorithm from Section 5.2 can be used for classification of both DVB-T and Programme Making and Special Events (PMSE) signals while the first algorithm from Section 5.1 can be used only for PMSE signals.

The algorithms can be implemented on firstly User Equipments (UE) in Frequency Division Duplex (FDD) DownLink (DL) or Time Division Duplex (TDD) DL and/or secondly Evolved Node B (eNB) TDD UpLink (UL) or FDD UL for the use cases:

1. opportunistic intra-cell spectrum aggregation of non-licensed FDD or TDD Long Term Evolution (LTE) from TeleVision White Space (TVWS) band, with licensed FDD or TDD LTE from 2.6 GHz band and

2. inter-cell spectrum aggregation of non-licensed FDD or TDD LTE from TVWS band, with licensed FDD or TDD LTE from 2.6 GHz band.

Table 5.1 and Table 5.2 are further summarizing a few standardized LTE TDD and FDD bands which have been considered important with respect to the previous described use cases.

Band	Name of Band	TDD UL and DL Band	Total Allowed Bandwidth
38	TDD International Mobile Telecommunications (IMT) Extension	2570–2620 MHz	50 MHz
41	TDD 2.5 GHz	2496-2690 MHz	194 MHz

Table 5.1: Example of LTE TDD Bands (3GPP) [15]

Please note that only the first use case is going to be demonstrated in the context of the SACRA project. One of the reasons is that the UE used for the demonstrations is not based on the EMBB processor and cannot implement sensing and/or classification. Therefore, sensing and classification will be implemented on eNB. However, when implemented on eNB, the Primary User (PU) classification is not reliable when the eNB receives a high power from its own LTE system. The classification reliability has been described in [5] but in

Band	Name of Band	FDD UL Band	FDD DL Band	Total Allowed Bandwidth
5	850 MHz	869-894 MHz	824-849 MHz	25 MHz UL and 25 MHz DL
6	-	875-885 MHz	830-840 MHz	10 MHz UL and 10 MHz DL
7	2.6 GHz	2620-2690 MHz	2500-2570 MHz	70 MHz UL and 70 MHz DL
8	900 MHz	925-960 MHz	880-915 MHz	35 MHz UL and 35 MHz DL
13	Upper 700 MHz	746-756 MHz	777-787 MHz	10 MHz UL and 10 MHz DL
14	Public Safety	758-768 MHz	788-798 MHz	10 MHz UL and 10 MHz DL

Table 5.2: Example of LTE FDD Bands (3GPP) [15]

another implementation context (i.e. UE based classification in FDD DL) and [12] provided a UE exclusion-based solution for reliable classification: UEs near eNB are not used for classification. For the same reason, eNB cannot perform classification for the DL signal because its own transmission may affect the classification. Therefore, for eNB classification cannot be performed all the time. eNB classification can be implemented for:

1. TDD UL, meaning that the classification block has to sample during TDD UL and has to wait during TDD DL when the eNB transmitted power is high. In this scenario, both TDD UL and DL are configured to unlicensed band (TVWS).

2. FDD UL, meaning that the classification block has to sample only the uplink frequency band. For propagation reasons, 3rd Generation Partnership Project (3GPP) usually considers FDD UL (from UE to eNB) carrier lower than FDD DL (from eNB to UE). Therefore, still in the context of SACRA demonstration activities, FDD UL could be configured to the unlicensed band (TVWS) while FDD DL can be configured to licensed band (which is higher than TVWS). FDD UL and DL TVWS implementation (in the same time) is not useful, since in this case FDD UL and DL need classification/sensing and FDD DL classification cannot be implemented on the eNB (for reasons related to classification reliability) nor on the UE (for reasons related to EMBB processor).

However in both cases, and as explained in D2.5 [12], if the transmitters (meaning UEs according to this scenario) are located near eNB, the eNB classification is not very reliable. In D2.5 [12] it has also been shown that when FDD DL classification is performed by the UE, the classification needs around 250 ms with respect to FCC requirements. Obviously, this time increases when classification is performed on the TDD configuration since the classification block needs more time to collect the same number of samples as for the FDD case (please see Table 5.3).

Please note that the abbreviations from 5.3 are considered as follows:

1. DL is a subframe for downlink transmission,

2. UL is a subframe for uplink transmission,

3. G is the guard time used to avoid collision between DL and UL (i.e., 3GPP special subframe).

TDD UL - DL configuration	DL to UL switch periodicity	Subframe (2 slots of 0.5 ms) number									
		0	1	2	3	4	5	6	7	8	9
Type 0	5 ms	DL	G	UL	UL	UL	DL	G	UL	UL	UL
Type 1	5 ms	DL	G	UL	UL	DL	DL	G	UL	UL	DL
Type 2	5 ms	DL	G	UL	DL	DL	DL	G	UL	DL	DL
Type 3	10 ms	DL	G	UL	UL	UL	DL	DL	DL	DL	DL
Type 4	10 ms	DL	G	UL	UL	DL	DL	DL	DL	DL	DL
Type 5	10 ms	DL	G	UL	DL	DL	DL	DL	DL	DL	DL
Type 6	5 ms	DL	G	UL	UL	UL	DL	G	UL	UL	DL

Table 5.3: LTE TDD Configuration (3GPP) and allowed classification time [15]

From Table 5.3 it seems that from the TDD UL classification point of view (performed by the eNB), the most convenient configuration is Type 0, because it allows a maximum UL communication per frame and thus a maximum classification time.

However, as explained in D2.2 [5] and D2.5 [12], the 250 ms classification is necessary only with respect to FCC requirements [11]. Please note that for the SACRA demonstrator, PU and Secondary User (SU) systems can be positioned in a way that allows lower classification time.

In the context of signal classification described in Section 5.1 and Section 5.2, the received signal is not a pure signal, but a mixture of SU and PU. Therefore, the received $y(t)$ can be represented as $y(t) = x(t) + z(t) + n(t)$, with $x(t)$ the useful PU signal, $z(t)$ is the SU signal, and $n(t)$ the noise. In Section 5.1 and Section 5.2 the goal is therefore to classify the signal $x(t)$ from the received mixture $y(t)$.

5.1 High Order Cumulants

Higher Order Cumulants (HOC) have been described in D2.5 [12] and presented as a potential classification method. As explained in both D2.2 [5] and D2.5 [12], when an LTE system operates in TVWS, it must detect PMSE Transmitter (TX) while continuing transmitting in the concerned channel, so there is the need to detect PMSE signal mixed with LTE signal (and noise).

D2.1 [13], D2.2 [5] and Section 3.2 from D2.5 [12] indicate that the energy variations are not sufficient to decide if the sudden environmental change is from SU or PU. D2.5 [12] proposes to use feature variation given by HOC in order to decide if the sudden environmental change is from PMSE PU. This method can be therefore used as an alternative for the classification methods proposed in D2.2 [5], but it can only classify non-OFDM signals mixed with noise and OFDM. However, compared to the classification method proposed in Section 5.2, the advantage of this method is that it is less complex and completely blind (it does not need any knowledge of the PU signal parameters).

In the following implementation example we compare the HOC $C_{4,2}$ computed for a certain classification period with a previous HOC $C_{4,2}$ computed during another classification period. The process therefore resumes to comparing HOC$((n+1)\,T_c)$ with HOC(nT_c) where T_c is classification time, and where HOC is computed with:

$$\hat{C}_{4,2} = \frac{1}{N}\sum_{k=1}^{N}|y(k)|^4 - \left|\frac{1}{N}\sum_{k=1}^{N}y(k)^2\right|^2 - 2\frac{1}{N^2}\left(\sum_{k=1}^{N}|y(k)|^2\right)^2 \tag{5.1}$$

where $y(k)$ is the discrete (after sampling) received signal.

The classification process can be further resumed in three parts, as also explained in D2.5 [12]:

1. The time-domain (sampled) signal is normalized in order to alleviate the impact of power variation on HOC value (the variance of the normalized signal equals 1).
2. HOC is applied on time-domain normalized signals from step 1. As seen also in D2.5 [12], HOC of LTE (OFDM) is not dependent on the received power while HOC of PMSE x-Phase Shift Keying (PSK), Frequency Modulation (FM) or n-QAM depends on the received power.
3. HOC computed for a certain classification period is compared with a previous HOC computed during another classification period.

Another possibility is to normalize the HOC at the end of the process. For the SACRA demonstration this is done directly on the host PC (as explained in D3.3 [4]) which will take the decision about the presence of the primary user. This will be less precise but it will consume less computation power and resources.

5.1.1 Description of the Algorithm

The signal classification based on HOC therefore operates in time domain on the discrete received signal $y(k)$, computes its HOC on a given number N of samples $y(1), y(2), \ldots, y(N)$, and compares this score with a pre-computed threshold. The different parameters have been restricted to meaningful ranges in order to simplify the implementation on the target platform. The bandwidth is restricted to 3 possible values (5, 10 or 20 MHz). The corresponding sampling frequencies F_s are 7.68, 15.36 and 30.72 Ms/s respectively. The stream of incoming samples is split in segments, L_s-samples each, and each segment is processed independently. L_s is fixed to 2048. The centre frequencies are between 300 MHz and 6 GHz. The classification period T_s is defined by the total number M of segments to process, with $1 \leq M \leq 16383$. The classification period $T_s = M \times L_s/F_s$ is thus in the $[66.67\mu s \ldots 1.092s]$ range.

Table 5.4 summarizes the input parameters of HOC.

Name	Type	Range	Description
M	Integer	$1 \leq M \leq 16383$	Sensing period (L_s-samples segments)

Table 5.4: HOC input parameters

Please note that for HOC classification, compared with the sensing algorithms, the BW and the F_c parameters are no longer required. The reason for the absence of these parameters is that the RF chain has been already configured when communication in TVWS has started.

When launched with the parameter M, HOC acquires samples from the specified communication band (BW, F_c) and the corresponding sampling frequency. $N = M \times L_s$ acquired samples are then processed and the final result of HOC is defined by:

$$HOC(BW, F_c, M, L_s) = \{OP1, OP2, OP3\} \tag{5.2}$$

$$OP1 = \frac{\sum_{k=1}^{k=N} |y(k)|^4}{2^{47}} \tag{5.3}$$

$$OP2 = \frac{\sum_{k=1}^{k=N} y(k)^2}{2^{15}} \tag{5.4}$$

$$OP3 = \frac{\sum_{k=1}^{k=N} |y(k)|^2}{2^{16}} \tag{5.5}$$

The result is returned to the higher levels of the application running on the host PC and is used for noise estimation and signal classification after comparison with a threshold that depends on the noise level and on the specified parameters.

5.1.2 Implementation of the Algorithm on the Target Processor

On the SACRA baseband platform the implementation of the HOC is straightforward. The interface with the A/D converters is configured to continuously acquire input samples from the selected RX chain.

The FEP DSP unit fetches the input samples with DMA transfers between the RX FIFO of PP and its own internal memory and computes the 3 HOC scores on L_s-samples segments with CWS and CWM instructions. The parameters of the CWS instruction are set so that the result vector is discarded and only the « SMA-values » (sum, max, argmax, min, argmin) are retained (32 bytes) and, among them, only the sum (real and imaginary parts on 32 bits, signed integer) is used in following steps to compute $OP2$. Two CWM instructions are needed, one to compute simultaneously the vector of the squares of modulus and their sum (used for $OP3$ computation), and a second which input is the output vector of the first one and for which the parameters are set so that the result vector is discarded and only the « SMA-values » (sum, max, argmax, min, argmin) are retained (32 bytes) and, among them, only the real part of the sum (32 bits, signed integer, used to compute $OP1$). For the first CWM instruction the r parameter is set to 1 to avoid overflows.

Each L_s-samples segment thus produces 3 intermediate values, represented as 32 bits integers ($OP1$ and $OP3$) or 64 bits complex ($OP2$). The main CPU of the baseband processor accumulates these intermediate results with 64 bits precision in 3 different scalar variables. The final result is returned to the upper layers which are responsible for normalization, finalization and threshold comparison. For the SACRA demonstrations these tasks are implemented in software on the host PC:

$$HOC = \frac{1}{N} \times \left(2^{47} \times OP1\right) - \frac{1}{N^2} \times |2^{15} \times OP2|^2 - 2 \times \frac{1}{N^2} \times \left(2^{16} \times OP3\right)^2 \tag{5.6}$$

Memory Management

The memory management of the HOC application is also straightforward. The internal memory of FEP is split into four banks, 16 k-bytes each. The topmost part of bank #0 is used to store the incoming data segments from the RX FIFO of PP. Two buffers, L_s samples each, are used in flip-flop mode in order to allow the processing of a segment in parallel with the DMA transfer of the next one. The vector output of the first CWM operation (vector of 16-bits integers) is stored in bank #1, always at the same address. Its SMA outputs are stored in bank #2, always at the same address. The SMA outputs of the CWS and of the second CWM operations are stored in bank #2, always at the same address. The real part of the sum of the first CWM (32 bits integers in the $[0 \ldots 2^{26}]$ range), the sum of the CWS (64 bits complex with real and imaginary parts in the $[-2^{27} \ldots 2^{27}]$ range) and the sum of the second CWM (32 bits integers in the $[0 \ldots 2^{26}]$ range) are used in the following processing steps. Figure 5.1 illustrates the memory management. The first data segment is stored in the topmost buffer of bank #0. As soon as it is fully transferred from PP, another DMA transfer is launched for the second data segment but with the second buffer of bank #0 as destination. This way, the DMA transfer takes place in parallel with the first CWM operation which input is the topmost buffer of bank #0, output vector is always the top of bank #1 and SMA output is always the top of bank #2. The CWS operation follows with the top of bank #0 as input and the second 32-bytes buffer of bank #2 as output. The second CWM takes the top of bank #1 as input and stores its SMA output always in the third 32-bytes buffer of bank #2. Finally, the main CPU reads the real part of the two CWM sums and the complex sum of the CWS out from bank #2 and accumulates the 3 totals. Bank #3 is unused. All in all the bank usage is summarized in Table 5.5.

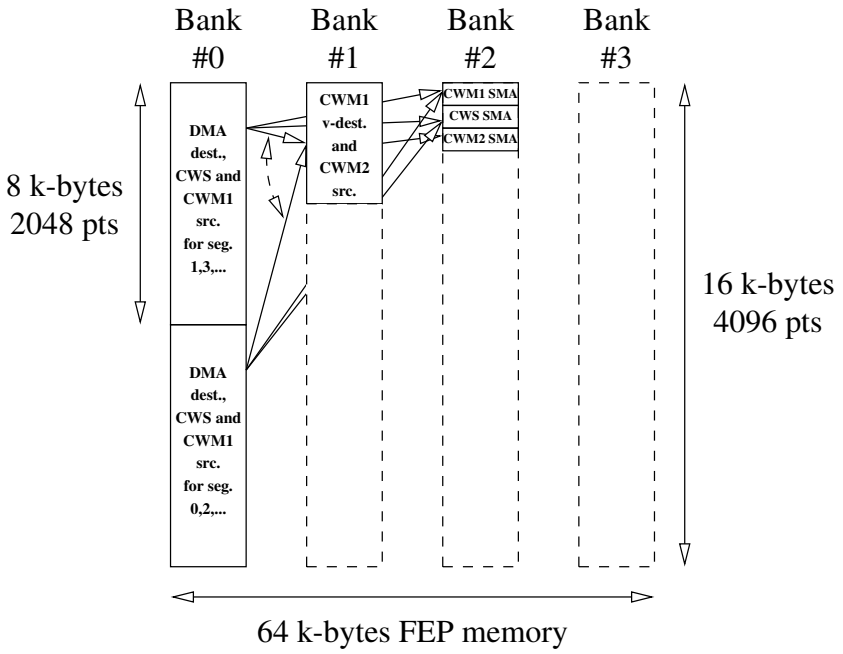

Figure 5.1: FEP internal memory management ($L_s = 2048$) for HOC

L_s	Bank usage (%)				Total
	#0	#1	#2	#3	
2048	100%	25%	0.6%	0%	31.39%

Table 5.5: FEP memory usage for HOC

Parallelization

FEP is involved in all steps of the computation. The only potential sources of parallelism are between:

- The DMA transfers from PP to FEP
- The computations by FEP
- The accumulations by the main CPU

Algorithm 5.1 represents the pseudo-code of the complete application in the 20 MHz bandwidth configuration. The HOC scores are pushed in a software FIFO that the host system reads periodically (not represented in the algorithm description).

5.1.3 Performance Analysis

The required raw processing power of HOC is 3 L_s-points CWM or CWS operations per segment, all with accumulation. The FEP computes the CWM and CWS operations, including accumulation, on 2048-components vectors in 1036 clock cycles (about two vector components per cycle). Table 5.6 summarizes the load of FEP for three different target technologies and for different input sampling frequencies, corresponding to three typical LTE downlink bandwidths (5, 10 and 20 MHz). The post-processing implemented in software on the main

Algorithm 5.1 The parallelized HOC application

```
1: dma.engine = FEP;                                   ▷ use DMA engine of FEP to transfer from PP to FEP
2: dma.src = PPRX0;                                    ▷ source of PP to FEP DMA transfers is RX0 FIFO of PP
3: dma.len = L_s;                                      ▷ length of PP to FEP DMA transfers
4: cwm1.len = L_s;                                     ▷ length of CWM1
5: cwm1.dst = FEPBANK1;                                ▷ vector destination of CWM1 operations
6: cwm1.sdst = FEPBANK2;                               ▷ SMA destination of CWM1 operations
7: cwm1.sma = BOTH;                                    ▷ store both result vector and sum, max, argmax, ... (SMA values) of CWM1
8: cws.len = L_s;                                      ▷ length of CWS
9: cws.sdst = FEPBANK2 + 32;                           ▷ destination of CWS operations (SMA only)
10: cws.sma = SMAONLY;                                 ▷ store only sum, max, argmax, ... (SMA values) of CWS
11: cwm2.len = L_s;                                    ▷ length of CWM2
12: cwm2.src = FEPBANK1;                               ▷ source of CWM2 operations
13: cwm2.sdst = FEPBANK2 + 64;                         ▷ destination of CWM2 operations (SMA only)
14: cwm2.sma = SMAONLY;                                ▷ store only sum, max, argmax, ... (SMA values) of CWM2
15: S0 = 0; S1 = 0;                                    ▷ semaphores
16: idx = 0;                                           ▷ unique index
17: procedure DIRECT_MEMORY_TRANSFERS_THREAD
18:     for i = 0; i < M; i = i + 1 do
19:         if i even then
20:             dma.dst = FEPBANK0;                    ▷ destination of PP to FEP DMA transfers
21:         else
22:             dma.dst = FEPBANK0 + L_s;              ▷ destination of PP to FEP DMA transfers
23:         end if;
24:         WAIT(S0 < 2);                              ▷ wait until one of the 2 destination buffers is free
25:         DMA(dma);                                  ▷ launch DMA transfer
26:         WAIT(dma);                                 ▷ wait until end of DMA transfer
27:         S0 = S0 + 1;                               ▷ mark destination buffer as full
28:     end for;
29: end procedure;
30: procedure COMPUTE_DETECTION_SCORES_THREAD
31:     for i = 0; i < M; i = i + 1 do
32:         if i even then
33:             cwm1.src = cws.src = FEPBANK0;         ▷ source of CWM1 and CWS operations
34:         else
35:             cwm1.src = cws.src = FEPBANK0 + L_s;   ▷ source of CWM1 and CWS operations
36:         end if;
37:         WAIT(S0 > 0);                              ▷ wait until one of the 2 source buffers is full
38:         WAIT(S1 < 1);                              ▷ wait until the destination SMA buffers are free
39:         FEP(cwm1);                                 ▷ launch CWM1
40:         WAIT(cwm1);                                ▷ wait until end of CWM1
41:         FEP(cws);                                  ▷ launch CWS
42:         WAIT(cws);                                 ▷ wait until end of CWS
43:         S0 = S0 − 1;                               ▷ mark source buffer as free
44:         FEP(cwm2);                                 ▷ launch CWM2
45:         WAIT(cwm2);                                ▷ wait until end of CWM2
46:         FEP(cwm);                                  ▷ launch CWM
47:         WAIT(cwm);                                 ▷ wait until end of CWM
48:         S1 = S1 + 1;                               ▷ mark the destination SMA buffers as full
49:     end for
50: end procedure;
51: procedure ACCUMULATE_DETECTION_SCORES_THREAD
52:     OP1 = OP2 = OP3 = 0;                           ▷ initialize scores
53:     for i = 0; i < M; i = i + 1 do
54:         WAIT(S1 > 0);                              ▷ wait until the source buffers are full
55:         OP1 = OP1+ READ_INT32(FEPBANK2);           ▷ accumulate
56:         OP2.real = OP2.real+ READ_INT32(FEPBANK2 + 32);   ▷ accumulate
57:         OP2.imag = OP2.imag+ READ_INT32(FEPBANK2 + 36);   ▷ accumulate
58:         OP3 = OP3+ READ_INT32(FEPBANK2 + 64);      ▷ accumulate
59:         S1 = S1 − 1;                               ▷ mark source buffer as free
60:     end for
61:     PUSH({idx, OP1, OP2, OP3});                    ▷ push in software FIFO
62:     idx = idx + 1;                                 ▷ increment unique index
63: end procedure;
```

CPU of the baseband processor is neglected because it represents only four 64-bits adds per segment. As for energy detection and Welch periodogram detection, we can see that the required processing power is extremely small on realistic targets and even on the much slower SACRA prototype the HOC application fits without problem. Table 5.6 also presents the estimated power consumption of FEP for the different configurations of the application, in the 22 nm version of the baseband processor at 1 GHz and for the two different operating voltages. As can be seen the power consumption of the HOC application is almost negligible.

BW	Clock cycles (two CWM and one CWS)	F (MHz)	Load (%)	Power (mW) High Vdd	Power (mW) Low Vdd
5 MHz	3108	100	11.66	-	-
		500	2.33	-	-
		1000	1.17	0.52	0.34
10 MHz	3108	100	23.31	-	-
		500	4.66	-	-
		1000	2.33	1.05	0.68
20 MHz	3108	100	46.62	-	-
		500	9.32	-	-
		1000	4.66	2.10	1.36

Table 5.6: FEP loads for different HOC use cases

5.1.4 Proposed Software API

The host PC controlling the sensing drives the HOC application and retrieves the computed energies through a simplified C-language API, which will also be available from Matlab®. Two different modes are supported: the one-shot mode in which the application is launched, returns an energy level and quits, and the continuous mode in which the application runs continuously and periodically stores energy levels in a software FIFO. The one-shot mode API is depicted in listing 5.1 and the continuous mode API in listing 5.2.

```
/* HOC returns -1 on failure (HOC already running or invalid parameters) and 0
 * on success. When successful, HOC returns after energy computation (blocking
 * call) and the score set is stored in the 4-cells int64_t array h:
 * h[0] = OP1;
 * h[1] = OP2.real;
 * h[2] = OP2.imag;
 * h[3] = OP3;
 * */
int HOC(
    /* int BW, bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    /* int Fc, centre frequency (Hz), in [300e6 ... 6e9] */
    int M,  /* sensing period (number of 2048-samples segments), in [1 ... 255] */
    int64_t *h /* must point to an already allocated 4-cells int64_t array */
    );
```

Listing 5.1: HOC API in one-shot mode

```
/* HOC_start returns -1 on failure (HOC already running or invalid parameters)
 * and 0 on success. HOC_start returns immediately (non blocking call) and, when
 * successful, starts storing score sets in the software FIFO, altogether with a
```

```
 * unique, self-incremented index. The index of the first score set is 0. The
 * software FIFO and the unique index are re-initialized upon HOC_start call.
 * Score sets that were still in the FIFO are lost. */
int HOC_start(
    /* int BW, bandwidth, in {0, 1, 2} for {5, 10, 20} MHz respectively */
    /* int Fc, centre frequency (Hz), in [300e6 ... 6e9] */
    int M   /* sensing period (number of 2048-samples segments), in [1 ... 16383] */
    );

/* HOC_stop returns -1 on failure (HOC not running) and 0 on success. HOC_stop
 * returns immediately (non blocking call) and, when successful, the currently
 * running HOC application is stopped at the end of the current HOC computation.
 * */
int HOC_stop(
    void
    );

/* HOC_pop returns -1 on failure (software FIFO empty), else pops the next
 * {index, OP1, OP2, OP3} set from the software FIFO, decrements the counter of
 * FIFO entries, and returns the current value of the counter. HOC_pop returns
 * immediately (non blocking call). On return the 4-cells int64_t array h
 * contains the scores:
 * h[0] = OP1;
 * h[1] = OP2.real;
 * h[2] = OP2.imag;
 * h[3] = OP3;
 * */
int HOC_pop(
    int *idx,  /* unique index */
    int64_t *h /* must point to an already allocated 4-cells int64_t array */
    );
```

Listing 5.2: HOC API in continous mode

As also explained in D3.3 [4], the non-normalized computed energies have to be indicated through the interface to the host PC. Please note that consecutive computed energies may be necessary in order to decide if classification is reliable or not.

5.2 Signal Classification without Quiet Period based on Cyclostationary Properties

The Cyclic Autocorrelation Function (CAF) of order 2 is the coefficient of the autocorrelation function decomposed in Fourier series expansion. In D2.2 [5] it has been mentioned that CAF can be used for signal classification, but as further seen, this method is more computationally intensive than HOC. CAF of any received signal y dependents on both cyclic frequency α and delay τ and can be written as:

$$R_y(\alpha,\tau) = \lim_{T\to\infty} \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} y^*(t)\, y(t+\tau) \exp(-j2\pi\alpha t)\, dt \tag{5.7}$$

Similarly, one could define CAFs for different orders. The expression described above could be numerically approximated by

$$R_y(\alpha,d) = \frac{1}{N} \sum_{n=0}^{N-1} y^*(n) y(n+d) \exp(-j2\pi\alpha n\Delta t) \tag{5.8}$$

where d represents the delay time normalized by the sampling period Δt. Please also note that it should be possible to use the FFT operation of the EMBB processor to perform CAF. However, the numerical conversion CAF-FFT is not yet straight forward and should be further investigated.

With all previous notations, the classification algorithm becomes as follows:

1. Compute the value of the CAF $R_y(\alpha, \tau)$ for a given delay τ and a cyclic frequency α. The delays τ and cyclic frequencies α could be used directly from a data base (see Table 5.7) or they could be estimated if necessary. Take the real and the imaginary part of this CAF:

$$r_y(\alpha, \tau) = \left[Re\left\{R_y(\alpha, \tau)\right\}, Im\left\{R_y(\alpha, \tau)\right\}\right] = \left[r_{y1}, r_{y2}\right]. \tag{5.9}$$

2. For a fixed delay τ, compute $Q_1(\alpha, \tau)$ and $Q_2(\alpha, \tau)$ defined by:

$$Q_1(\alpha, \tau) = \frac{1}{NL} \sum_{l=-\frac{L-1}{2}}^{\frac{L-1}{2}} W(l) R_y\left(\alpha - \frac{l}{N\Delta t}, \tau\right) R_y\left(\alpha + \frac{l}{N\Delta t}, \tau\right) \tag{5.10}$$

or, if approximated as symmetrical it might be simplified by a factor of 2

$$Q_1(\alpha, \tau) = \frac{2}{NL} \sum_{l=1}^{\frac{L-1}{2}} W(l) R_y\left(\alpha - \frac{l}{N\Delta t}, \tau\right) R_y\left(\alpha + \frac{l}{N\Delta t}, \tau\right) + \frac{1}{NL} W(0) R_y^2(\alpha, \tau) \tag{5.11}$$

where N has the same definition as previous and

$$Q_2(\alpha, \tau) = \frac{1}{NL} \sum_{l=-\frac{L-1}{2}}^{\frac{L-1}{2}} W(l) R_y^*\left(\alpha + \frac{l}{N\Delta t}, \tau\right) R_y\left(\alpha + \frac{l}{N\Delta t}, \tau\right) \tag{5.12}$$

or

$$Q_2(\alpha, \tau) = \frac{2}{NL} \sum_{l=1}^{\frac{L-1}{2}} W(l) R_y^*\left(\alpha + \frac{l}{N\Delta t}, \tau\right) R_y\left(\alpha + \frac{l}{N\Delta t}, \tau\right) + \frac{1}{NL} W(0) \left|R_y(\alpha, \tau)\right|^2 \tag{5.13}$$

where W is a Kaiser window and L its length. Please note that the Kaiser window is normally defined as

$$W(n) = \frac{I_0\left[\beta_{Kaiser} \sqrt{1 - \left(\frac{2n}{L-1} - 1\right)^2}\right]}{I_0\left[\beta_{Kaiser}\right]} \tag{5.14}$$

with $0 \leq n \leq L-1$, where I_0 is the zero order Modified Bessel function of the first kind:

$$I_0(x) = \sum_{m=0}^{\infty} \frac{1}{m!\Gamma(m+1)} \left(\frac{x}{2}\right)^{2m} \tag{5.15}$$

where

(a) Γ(y) is the gamma function;

(b) β_{Kaiser} is an arbitrary real number that determines the shape of the window. In the frequency domain, it determines the trade-off between main-lobe width and side lobe level, which is a central decision in window design. Increasing β_{Kaiser} widens the main lobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation). For $\beta_{Kaiser} = 0$ one can obtain the same performance as the rectangular window, for $\beta_{Kaiser} = 5,4414$ the Hamming window, and for $\beta_{Kaiser} = 8,885$ the Blackman window;

(c) L is the length of the sequence.

However, due to the change of variable $l = n - \frac{L-1}{2}$, in our case the Kaiser window is now defined as

$$W(l) = \frac{I_0\left[\beta_{Kaiser}\sqrt{1-\left(\frac{2l}{L-1}\right)^2}\right]}{I_0\left[\beta_{Kaiser}\right]} \tag{5.16}$$

with $-\frac{L-1}{2} \leq l \leq \frac{L-1}{2}$.

3. Compute the covariance matrix $\sum_y(\alpha)$:

$$\Sigma_y(\alpha) = \begin{bmatrix} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{bmatrix} = \begin{bmatrix} Re\left(\frac{Q_1(\alpha,\tau)+Q_2(\alpha,\tau)}{2}\right) & Im\left(\frac{Q_1(\alpha,\tau)-Q_2(\alpha,\tau)}{2}\right) \\ Im\left(\frac{Q_1(\alpha,\tau)+Q_2(\alpha,\tau)}{2}\right) & Re\left(\frac{Q_2(\alpha,\tau)-Q_1(\alpha,\tau)}{2}\right) \end{bmatrix} \tag{5.17}$$

with its inverse function

$$inv\left(\Sigma_y(\alpha)\right) = \frac{1}{\Delta}\begin{bmatrix} S_{2,2} & -S_{1,2} \\ -S_{2,1} & S_{1,1} \end{bmatrix} \tag{5.18}$$

where

$$\Delta = \det\left(\Sigma_y(\alpha)\right) = S_{1,1}\cdot S_{2,2} - S_{2,1}\cdot S_{1,2} \tag{5.19}$$

4. Compute the test statistic:

$$T_{GLRT}(y,\alpha) = N \cdot r_y(\alpha,\tau)\cdot inv\left(\sum_y(\alpha)\right)\cdot r_y^T(\alpha,\tau) \tag{5.20}$$

where inv is the inversion operation of the covariance matrix. This can be further resumed to

$$T_{GLRT}(y,\alpha) = \frac{N}{\Delta}\cdot\left(S_{2,2}\cdot r_{y1}^2 + S_{1,1}\cdot r_{y2}^2 - (S_{1,2}+S_{2,1})\cdot r_{y1}\cdot r_{y2}\right). \tag{5.21}$$

The test statistic is compared with a threshold γ. This threshold is computed according to P_{FA} = 1 - $\Gamma(1,\gamma/2)$, where Γ is the incomplete gamma function. When P_{FA} = 0.1 threshold is 4.60518 and when P_{FA} = 0.01 threshold is 9.212. Please note that when implementing detection or classification, all normalization factors should be integrated into the threshold, instead of being applied at each step of the computation of the test statistic. This will reduce computation. Classification result is 1 if $T_{GLRT}(y,\alpha) > \gamma$ and 0 if $T_{GLRT}(y,\alpha) < \gamma$. Consecutive classification results may be necessary in order to decide if classification is reliable or not.

For an OFDM signal, the peaks in the Cyclic Autocorrelation function are dependent on T_U and total symbol duration which is T_U + T_G. For a sum of multiple OFDM signals with different parameters (different

Useful symbol duration	$T_U = 224\ \mu$s
Cyclic prefix length	$T_G = (1/4)T_U$
(Maximum) classification time	$T_C = 250$ ms
Number of OFDM symbols	$T_C/(T_U + T_G)$
P_{FA} / γ (threshold)	0.1 / 4.60518 or 0.01 / 9.2120
Delays	$\tau = \pm T_U$
Cyclic positions	$\alpha = \pm 1/(T_U + T_G)$
Sampling period	$\Delta t = 1/(4 \times 3.84MHz)$ and $\Delta t = 1/(2 \times 3.84MHz)$ for 10 MHz and 5 MHz LTE systems
Number of samples	$N = T_C/\Delta t$
Length of Kaiser window	$L = 101$
β_{Kaiser}	1

Table 5.7: Input parameters for the signal classification without quiet period based on cyclostationary properties

T_U and different T_U + T_G), multiple distinct peaks should appear on the cyclic autocorrelation function. A complete description of the parameters is given in Table 5.7, for a $2k$ DVB-T signal.

Please note that the L=101 Kaiser window from Table 5.7 can be also interpolated for $L = 2 \cdot p + 1$ with 2p+1<101. For example L=7 Kaiser window is [0.7898, 0.9034, 0.9754, 1.0000, 0.9754, 0.9034, 0.7898].

The goal of this study was to see if cyclostationary properties could be used for signal separation and identification on the EMBB processor. Matlab® simulations show that FFT can be used to perform CAF. However, it is not clear if this algorithm can use the FFT operation from the platform and this probably has to be further investigated. The implementation thus currently covers only the first step of the software design flow as described in Chapter 3.

6 CONCLUSION

We described the complete implementation of two sensing and one classification applications on the SACRA baseband processor, EMBB. A classification application, without quiet period, and using cyclostationary properties has also been partly implemented and described. Several lessons that can be learned from these experiments.

First, the 3 fully implemented algorithms had to be reworked a bit in order to comply with the target processor but this was rather straightforward. This is a good indication that the EMBB processor is a reasonable one for this kind of applications and that the requested software design effort is very limited. An extra operation has been designed and added to ease the implementation of the HOC-based classification but this was not mandatory and not using this extra operation would have only slightly degraded the performance. All in all, the three algorithms would represent only a negligible part of the available processing power and power budget of an EMBB instance manufactured in an advanced micro-electronic process (22 nano-meter).

Another partial lesson comes from the fourth algorithm: implementing the signal classification without quiet period using cyclostationary properties, as it is described today, would probably lead to a very high computation load. Reworking the algorithm seems significantly more difficult than with the 3 others because it is not just a matter of splitting large vector computations in smaller sub-vectors. What is probably required is a deep re-formulating based on Fourier transforms. This kind of situations is probably unavoidable. Some algorithms can be designed in many different ways, all about equivalent when working at a high abstraction level (Matlab®), but completely different at implementation time. Considering the specificities of the target execution platform during the very first stages of the design flow is an option but, of course, is not always practical. And unfortunately real time hardware baseband processors have very different characteristics than the high level software tools used by algorithms designers.

Finally, without doubt, the most challenging task when porting applications on EMBB is the parallelization phase. This is not very surprizing but has been confirmed by experiments. The parallelization between the 3 different applications has not been done yet. As at least one of the two sensing applications must run concurrently with the classification, this will have to be done for the final demonstration. The raw loads and memory usages show that it should not really be a problem.

7 ACRONYMS

3GPP	3rd Generation Partnership Project
nm	nano-meter
A/D	Analogue/Digital
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
CWA	Component Wise Addition
CWL	Component Wise filter by a Lookup table
CWM	Component Wise square of Modulus
CWP	Component Wise Product
CWS	Component Wise Square
CWL	Component-Wise Lookup
CAF	Cyclic Autocorrelation Function
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DL	DownLink
DSP	Digital Signal Processor or Digital Signal Processing
DVB-T	Digital Video Broadcasting-Terrestrial
DVFS	Dynamic Voltage and Frequency Scaling
ED	Energy Detector
eNB	Evolved Node B
FCC	Federal Communications Commission
FEP	Front-End Processor
FIFO	First In First Out buffer
FIR	Finite Impulse Response
FFT	Fast Fourier Transforms
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
FDD	Frequency Division Duplex
FT	Fourier Transform
HDL	Hardware Description Language
HOC	Higher Order Cumulants
I/Q	In-phase/Quadrature-phase
IC	Integrated Circuit

Continued on next page –

	– continued from previous page
IMT	International Mobile Telecommunications
ISS	Instruction Set Simulator
ISR	Interrupt Service Routines
ITRS	International Technology Roadmap for Semiconductors
LTE	Long Term Evolution
MMU	Memory Management Unit
MOV	Component wise copy
MSC	Message Sequence Chart
Ms/s	Mega-Sample per Second
OFDM	Orthogonal Frequency-Division Multiplexing
PC	Personal Computer
PP	Pre-Processor
PD	Probability of Detection
PFA	Probability of False Alarm
PMSE	Programme Making and Special Events
PSK	Phase Shift Keying
PU	Primary User
QAM	Quadrature Amplitude Modulation
RF	Radio Frequency
RTOS	Real Time Operating System
RX	Reciever
SDK	Software Design Kit
SU	Secondary User
SMA	Sum, Min-max, Argmin-argmax
SNR	Signal to Noise Ratio
TDD	Time Division Duplex
TVWS	TeleVision White Space
TX	Transmitter
UE	User Equipment
WPD	Welch Periodogram Detector

BIBLIOGRAPHY

[1] Aeroflex Gaisler: `http://www.gaisler.com/`.

[2] MutekH, a free and portable operating system for embedded platforms: `http://www.mutekh.org/`.

[3] SoCLib, an open platform for virtual prototyping of multi-processors system on chip (MP-SoC): `http://www.soclib.fr/`.

[4] Konstantinos Chatzikokolakis, Panagiotis Spapis, George Katsikas (UoA), Bassem Zayen (Eurecom), Abdoulaye Bagayoko, Dorin Panaitopol (NTUK), Farouk Aissanou, Alain Petrowski, and Djamal Zeghlache (IT). SACRA deliverable D3.3: Control loops drive models & resource management assessment, 2012.

[5] Bassem Zayen (EURECOM), Wael Guibene (EURECOM), Dorin Panaitopol (NTUK), and Atso Hekkala (VTT). SACRA deliverable D2.2: Specification of signal classification techniques for cognitive radios, 2012.

[6] A. Hekkala, I. Harjula, D. Panaitopol, T. Rautio, and R. Pacalet. Cooperative spectrum sensing study using Welch periodogram. In *Proceedings of the 2011 11th International Conference on Telecommunications (ConTEL)*, June 2011.

[7] H. Holma and A. Toskala. *WCDMA for UMTS: HSPA Evolution and LTE*, page 577. John Wiley & Sons, fifth edition, 2010.

[8] Renaud Pacalet (IT). SACRA deliverable D5.2: Report on SACRA embedded software library, RF BB co-design, RF BB interface, functional and performance validations, first appendix: ExpressMIMO user guide, 2012.

[9] ITRS. International Technology Roadmap For Semiconductors 2011 Edition Executive Summary, 2011.

[10] S. Maleki, A. Pandharipande, and G. Leus. Two-stage spectrum sensing for cognitive radios. In *Proceedings of ICASSP 2010*, 2010.

[11] Second memorandum opinion and order. FCC Report 10-174, "Unlicensed Operation in the TV Broadcast Bands and Additional Spectrum for Unlicensed Devices Below 900 MHz and in the 3 GHz Band", 23^{rd} September 2010.

[12] Dorin Panaitopol (NTUK), Bassem Zayen (EURECOM), Wael Guibene (EURECOM), and Atso Hekkala (VTT). SACRA deliverable D2.5: Development and evaluation of energy efficient multiband spectrum sensing algorithms, 2012.

[13] Dorin Panaitopol (NTUK), Atso Hekkala (VTT), Pertti Jarvensivu (VTT), Abdel Waheb Marzouki (IT), Wael Guibene (EURECOM), and Umer Salim (INFINEON). SACRA deliverable D2.1: Preliminary report on sensing and access techniques, 2010.

[14] D. Panaitopol, A. Bagayoko, P. Delahaye, and L. Rakotoharison. Fast and Reliable Sensing Using a Background Process for Noise Estimation. In *Proceedings of CrownCom 2011*, 2011.

[15] H. Sesia, I. Toufik, and M. Baker. *LTE – The UMTS Long Term Evolution From Theory to Practice*. John Wiley & Sons, first edition, 2009.