AUDITOR – GA 687367

Advanced Multi-Constellation EGNSS Augmentation and Monitoring Network and its Application in Precision Agriculture

D2.2 Version 1.0

Subsystem specification

Contractual Date of Delivery:	M6 (Jun, 2016)
Actual Date of Delivery:	
Editor:	*Jacobo Dominguez (ACORDE)*
Author(s):	*Esther López, Jacobo Domínguez, David Abia, José Manuel Sánchez (ACORDE); Carles Fernandez-Prades, Marc Majoral, Javier Arribas (CTTC); Alberto García Rigo, Manuel Hernández-Pajares (UPC);*
Work package:	*WP2 - Receiver architecture definition*
Security:	**CO**
Nature:	**R**
Version:	1.0
Total number of pages:	62

Abstract:

This document contains the internal details of the different subsystems that are part of the AUDITOR system architecture which foundations were introduced in D2.1 Architecture definition. A custom RF Front-End and a commercial ARM/FPGA processing platform provide the base hardware to implement a flexible GNSS receiver. Multiple low level modules and higher level algorithms are presented in this document which integrated into the hardware platforms provide a flexible high performance open GNSS receiver supporting multi-channel and Galileo/GPS bands (E1/L1, L2 o E5a/L5). Moreover, the embedded subsystem is supported by remote cloud services to obtain ionospheric data corrections and extends its user services.

Document Control

Version	Details of Change	Author	Approved	Date
1.0	First complete version of the document	EL	EL	30/06/2016

Executive Summary

This document extends D2.1 Architecture Definition adding detailed information about the different modules and interfaces that are embedded into the proposed AUDITOR GNSS receiver architecture.

The two core hardware elements included in the architecture are the RF front-end module and the signal processing board based on the Zynq-7000 All Programmable SoC. Additional cloud computing platforms are integrated into the system either to obtain ionospheric corrections or to provide valuable web users services.

The front-end acquires the GNSS signals from two simultaneous channels and provides the samples to the signal processing board via a high performance parallel bus.

The processing platform implements an efficient real-time pre-processing of the GNSS samples in its FPGA architecture while higher level algorithms and applications are developed in its ARM core to offer high precision positioning features.

Several internal and external interfaces are defined into this document to provide the needed interconnection between the previous modules and subsystems. These include the necessary software interfaces and messages formats using custom or standard communication protocols.

Authors

Partner	Name	e-mail
ACORDE	Esther López	esther.lopez@acorde.com
	Jacobo Domínguez	jacobo.dominguez@acorde.com
	David Abia	david.abia@acorde.com
	José Manuel Sánchez	josemanuel.sanchez@acorde.com
CTTC	Carles Fernández-Prades	carles.fernandez@cttc.es
	Marc Majoral	marc.majoral@cttc.es
	Javier Arribas	javier.arribas@cttc.es
UPC	Alberto García Rigo	alberto.garcia.rigo@upc.edu
	Manuel Hernández-Pajares	manuel.hernandez@upc.edu

Table of Contents

List of tables

List of Figures

List of Acronyms and Abbreviations

Term	Description
ACP	Accelerator Coherency Port
ADC	Analog-to-Digital Conversion
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
APU	Application Processor Unit
ASSP	Application Specific Standard Product
AXI	Advanced eXtensible Interface
CAN	Controller Area Network
CLB	Configurable Logic Block
CPU	Central Processing Unit
DAP	Debug Access Port
DDR	Double Data Rate
DevC	Device Configuration interface
DMA	Direct Memory Access
DMIPS	Dhrystone Million Instructions Per Second
DSP	Digital Signal Processor
ECC	Error Correction Checking
EHCI	Enhanced Host Controller Interface
EMIO	Extendable Multiplexed Input / Output
FE	Front End
FPGA	Field Programmable Gate Array
GIC	General interrupt controller
GMII	Gigabit Media-Independent Interface
GNSS	Global Navigation Satellite System
IOP	Input / Output Peripherals
IP	Intellectual Property
IRQ	Interrupt ReQuest
LPDDR	Low Power Double Data Rate

Term	Description
LUT	Look-Up Table
MAC	Media Access Control
MIO	Multiuse Input / Output
MMU	Memory Management Unit
OCM	On-Chip Memory
OTG	On-The-Go
PCAP	Processor Configuration Access Port
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
RGMII	Reduced Gigabit Media-Independent Interface
ROM	Read Only Memory
SGMII	Serial Gigabit Media-Independent Interface
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SWDT	System Watch Dog Timer
TTC	Triple Timers / Counters
UART	Universal Asynchronous Receiver-Transmitter
ULPI	UTMI+ Low Pin Interface
UTMI	USB 2.0 Transceiver Macrocell Interface
VFPU	Vector Floating Point Unit
WDT	Watch Dog Timers

1. Introduction

In previous deliverable D2.1 [1] the overall architecture for the AUDITOR system has been presented. This architecture is summarized in Figure 1.1, that was already introduced in [1] and it's included here for convenience:

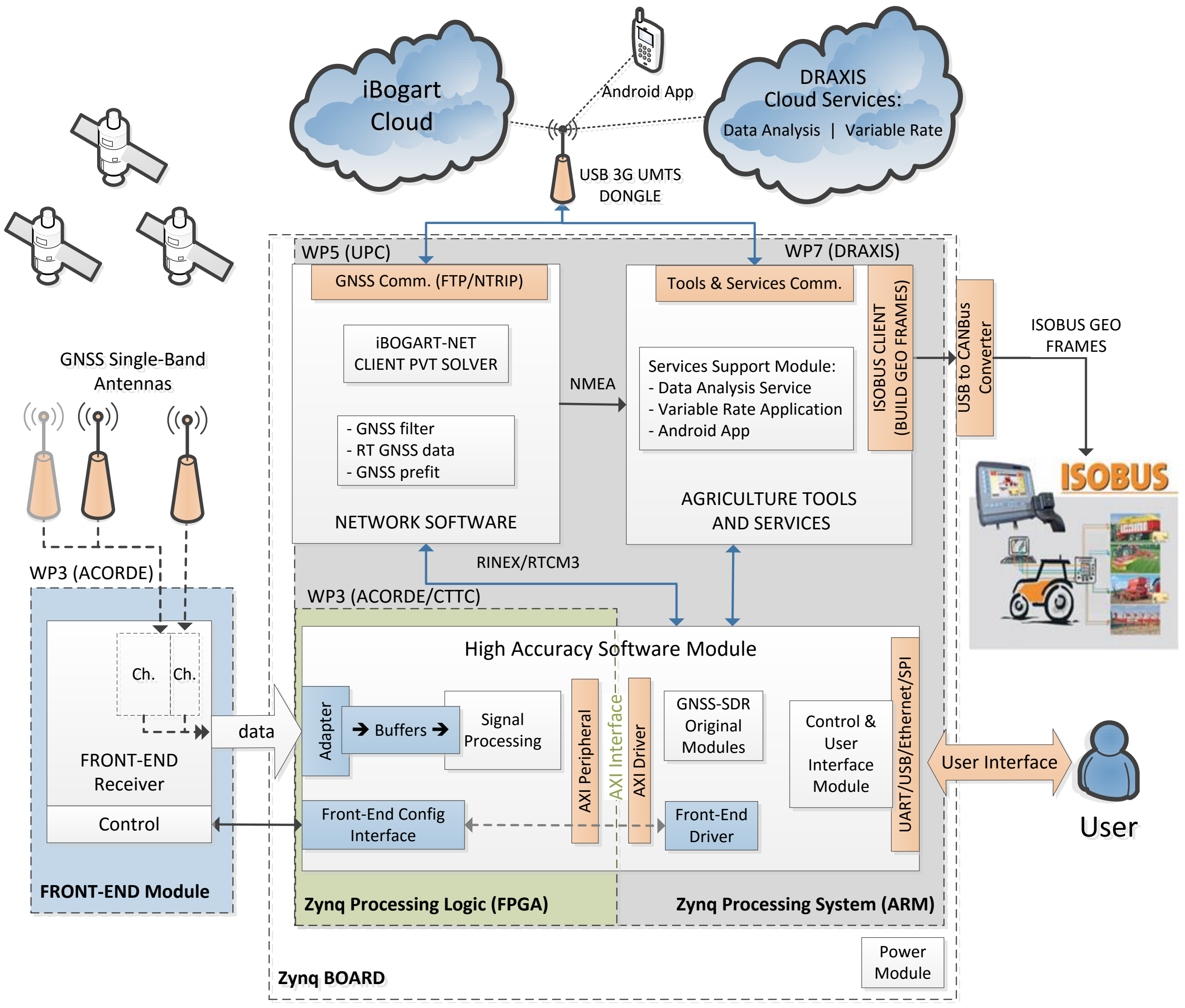


Figure 1.1: System Architecture

The AUDITOR system is composed of a custom **RF front-end module**, a **processing platform** (Zynq Board) and several **cloud systems** to either consume ionospheric corrections or provide user services. In this deliverable the subsystem specification for each module will be detailed as well as, the multiple internal/external interfaces and message formats that supports the actual interconnection of these modules. This document establishes the initial subsystem specification but is subjected to changes as the project implementation advances.

2. GNSS Front-End Module

The GNSS Front-End module introduced in [1] implements two GNSS channels to support simultaneous data acquisition from two different bands. The general interfaces for this module are shown in Figure 2.1.

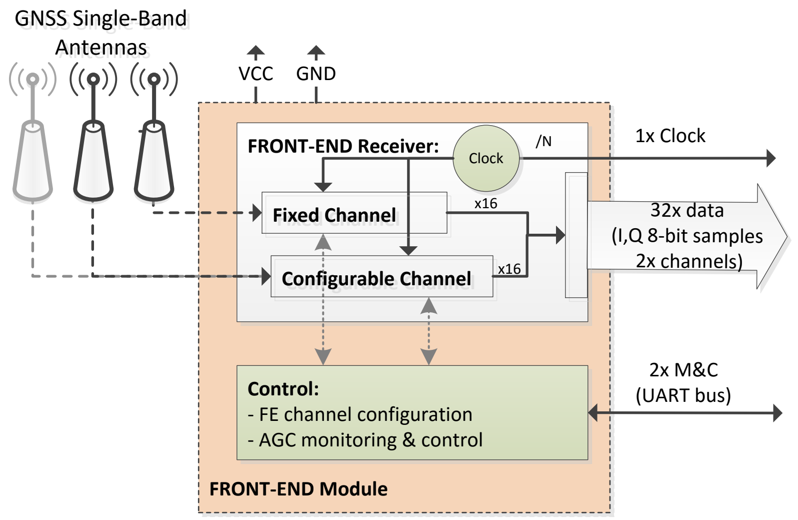


Figure 2.1: Front-End interfaces

The main input interfaces for the FE module are the external GNSS antennas. Different single-band antennas have been selected to reduce costs, add flexibility and simplify the FE layout.

The main output interfaces are the I,Q data samples and its associated reference clock. Each signal (I and Q) is sampled with 8-bit resolution generating a 16-bits parallel stream. The two GNSS channels acquired produce a total of 32-bits (1 sample) in the output parallel data bus. This data stream is synchronized with the provided clock signal.

The FE is based internally in two single band chains two support simultaneously the two proposed GNSS bands in a real time manner. Each band implements different RF Front-End modules while sharing the same clock and M&C logic. A bidirectional interface implemented via a UART bus provides the external M&C capabilities. Further details for each chain are described in the following sections.

A summary of the front-end initial specification is shown in Table 2.1.

Table 2.1: Front End Specification

Channel		
Channel 1 (fixed)	L1/E1	***yes***
Channel 2 (configurable)	L2C	***yes***
	L5/E5a	***yes***
Frequency bandwidth		
Sampling frequency (in MHz)[1]	L1/E1	***~4 MHz + 2xIF***
	L2C	***~4 MHz + 2xIF***
	L5/E5a	***~25 MHz + 2xIF***

[1] Exact value depends on local oscillator configuration and internal crystal parameters, a configurable sampling frequency of 6.25/12.5/25 Msps will be available for all channels to assess different data rates capabilities.

	Option a: L1/E1 and L2C	***~4 MHz + 2xIF***
	Option b: L1/E1 and L5/E5a	***~25 MHz + 2xIF***
Bits per sample		
Each channel will generate 8 bits I + 8 bits Q in 2's complement.		
The adapter module in the Zynq will be in charge of reading these inputs and converting them to baseband.		
Intermediate Frequency		
Depending on the band and architecture, Zero-IF may not be achieved. Different IF frequencies (from several kHz to a few MHz) would be used instead. Sampling frequency would higher than defined to cope with the IF		
IF	L1/E1	***Up to 1 MHz***
	L2C	***Up to 1 MHz***
	L5/E5a	***Up to 1 MHz***
AGC		
• The gain provided by the AGC needs to be known in the receiver side (can be at low rate, using SPI or I2C). • Possibility to fix that gain by software (required by some test procedures).		
Reference oscillator		
Accuracy	**<= 0.5ppm**	
External oscillator option	**YES (SMA female connector)**	
Antenna input		
Connector type	**SMA female**	
Impedance	**50 ohms**	
On-board DC Bias-T (for active GNSS antenna)	**YES (5v DC output is required to power the GNSS antenna LNA)**	
Signals Summary		
CH1 DATA In-phase component (real part)		8
CH1 DATA Quadrature component (imaginary part)		8
CH2 DATA In-phase component (real part)		8
CH2 DATA Quadrature component (imaginary part)		8
Sample CLOCK		1
VCC		1
GND		1

2.1 Fixed Band

The fixed band schema is show in Figure 2.2. It is based on an embedded front RF downconverter and a high performance ADC backend. Both elements are synchronized via a common clock source which is also provided externally as the reference data clock.

The AGC module continuously evaluates the GNSS signal level generates an 8-bit digital word that provide a link quality indicator (LQI) to the FE microcontroller.

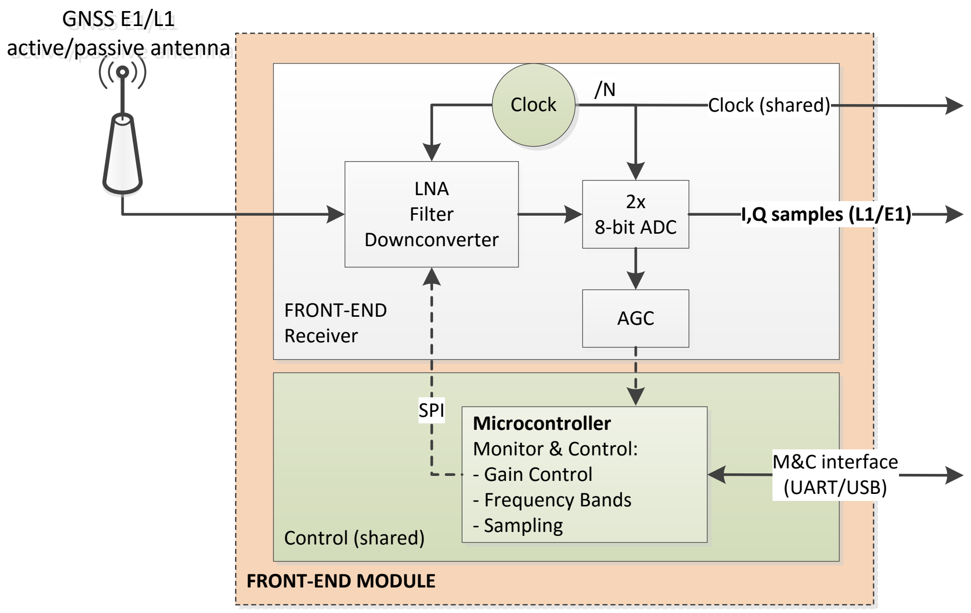


Figure 2.2: Single E1/L1 bands

A fixed 8-bit dual ADC provides a continuous I,Q 16-bit data stream to be consumed directly by the Zynq Board.

2.2 Configurable Band

The configurable band schema follows a similar approach than the single band using a configurable RF downconverter and a custom LNA/Filter chain to select the proper GNSS band to be received. Either one single-band antenna or a dual-band antenna could be connected to the RF input.

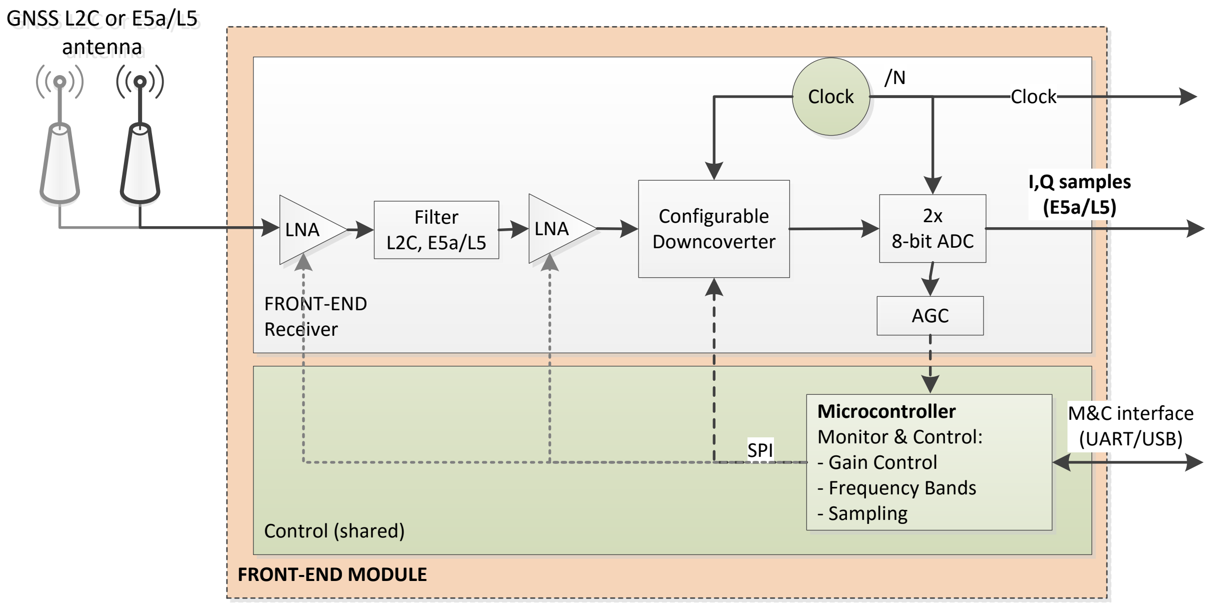


Figure 2.3: Single L2C or E5a/L5 band

The clock and M&C logic is shared with the single band while this chain adds another 16-bit I,Q samples to the output data bus.

2.3 M&C commands

In order to configure, fine tune and test the FE module the commands show in Table 2.2 will be implemented.

Table 2.2: FE M&C commands

Message	Description
Configure ch2	Select L2C or E5a/L5
Enable/Disable AGC	Enables or disables the AGC process
Set fixed gain value per ch.	Manually set a fixed gain value per channel.
Read current gain value per ch.	Gets the current gain applied by AGC or manually fixed.
Set/Get sampling frequency	Establish the reference sampling frequency for all channels between a list of valid values
Get firmware version	Get the current internal firmware version
Get AGC counter	Only for debugging
Set/Get downconverter config	For testing and debugging purposes

3. Signal Processing

GNSS baseband signal processing (understood as the process of computing GNSS observables from raw samples obtained at the output of a radio frequency front-end) will be defined by software and executed by the processing platform described below.

Most of the operations required by GNSS processing can be executed in real-time in ARM processors. Indeed, the software receiver GNSS-SDR can already be built and executed in such processors. Recent experiments shown that the software can even attain real-time execution in high-end, ARM-based devices [2], but only for very basic configurations (on the order of eight channels targeting GPS L1 C/A signals with a sampling rate of 2 Msps).

In order to attain the electric characteristics targeted in AUDITOR, and thus its computational requirements, today's ARM-based platforms are still not fast enough.

The strategy adopted in AUDITOR consists of offloading some of the processing work from the central processing unit to a field-programmable gate array (FPGA) device. This Section describes a system-on-chip architecture that integrates all-purpose ARM processors and FPGAs in a single chip

3.1 Zynq Platform

The Zynq SoC comprises two main parts:

- **Processing System (PS):** the processing system contains a dual-core ARM Cortex-A9 processor, a number of processor peripherals (DDR controller, SPI, DMA controller, internal RAM memory, interrupt controller, etc.) and a number of interfaces to the PL.
- **Processing Logic (PL):** the processing logic contains an Artix or Kintex FPGA with all of its logic cells and processing modules (DSP48, FIFOs, RAMs, etc.) and a number of interfaces to the PS.

The table below shows the elements that are contained in each part more info can be found in [3].

Table 3.1: Zynq SoC components

Part	Contents
PS	Dual Core ARM Cortex-A9 processor with NEON floating point units, Memory Management Units and L1 and L2 cache memories.
	Internal hard-coded non-accessible BOOT ROM to run basic system configuration and to load the first stage boot loader (FSBL) from an external resource.
	256 kB RAM OCM (On Chip Memory)
	Generic Interrupt Controller with private interrupts (for each core), software interrupts, peripheral interrupts and interrupts coming from the PL.
	Timers and watchdogs for the ARM cores and the PL
	DMA controller
	Flash/asynchronous SRAM memory controller
	Quad-SPI flash controller

	SD/SDIO controller
	General Purpose I/O peripheral
	USB Host, Device and OTG controller
	Gigabit Ethernet controller
	SPI Controller
	CAN controller
	UART Controller
	I2C controller
	Interface to the PL
PL	Artix or Kintex FPGA
	Interface to the PS

The PS and the PL are in separate areas of the SoC and have their own pins. By default the peripherals listed in Table 3.1 can only be accessed from the pins that lay in the PS area because the peripherals are physically situated in the PS area. However, some peripheral interfaces can be internally routed to the PL and accessed from FPGA pins that lie in the PL.

The PS and PL will contain different processing elements of the AUDITOR project. Figure 3.1 shows the Digital Processing Platform (PS and PL) inner modules including the interconnection with the radio-frequency front-end.

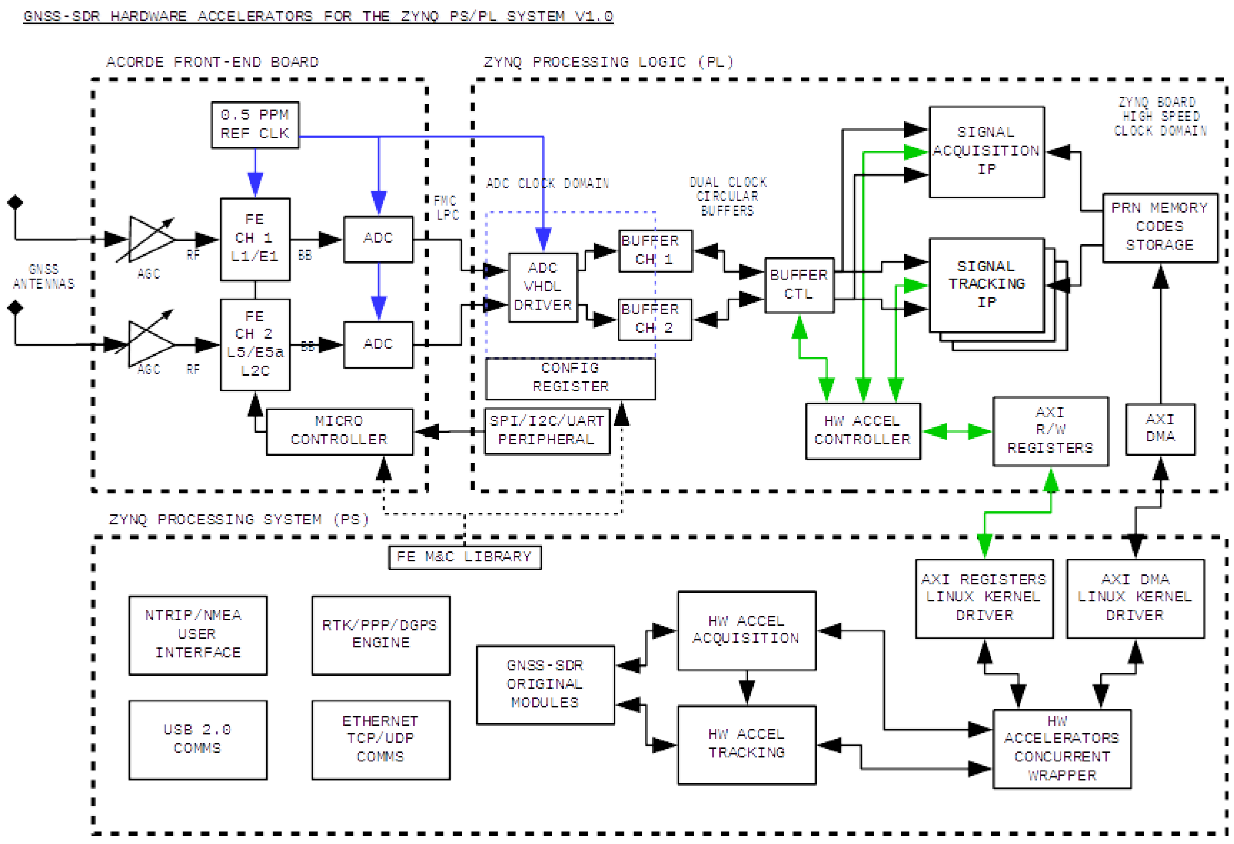


Figure 3.1: Overview of AUDITOR's digital processing platform

3.2 Zynq Processing Logic (PL)

In the PL section of the Zynq platform custom units of logic, also known as intellectual property (IP) cores, will be developed to build low level processing blocks to transform the GNSS raw data stream for the higher level algorithms. In this manner the processing is divided in two phases. Initially a "front-end" processing of the raw GNSS samples produced by the RF FE is defined to generate a fixed continuous data stream. Secondly a "back" processing or hardware accelerators, devoted to the extraction of the GNSS data embedded in the logic channels, is implemented via multiple buffers and parallel correlators.

3.2.1 Front-end logic

Taking into account the architecture overview shown in Figure 1.1, Front-end logic is allocated into the High Accuracy Software module that runs in the Zynq platform to implement high efficient pre-processing. The main features of the Front-end logic are:

- Physically **interconnect** with the FE module.
- **Adapt** the maximum 8-bits/sample to lower values.
- Provide a flexible **buffering** to feed the higher processing algorithms.

The relation interconnection between these elements is depicted in Figure 3.2 which components are detailed next.

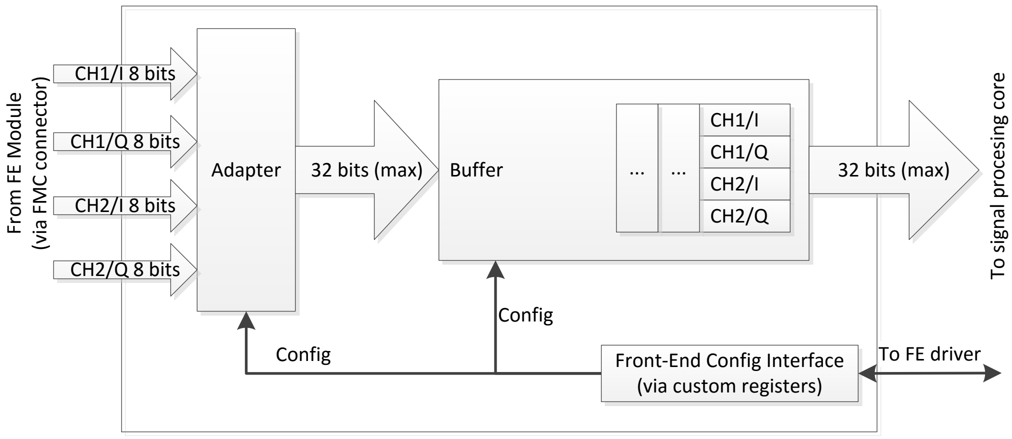


Figure 3.2: FE logic schema

Physical Connections

The front-end logic interfaces with the FE output data signals and M&C UART bus. Considering the ZedBoard as the targeting host processor the first approach to implement the interconnection would be the use of the Pmod connectors [4]. These connectors provide easier access to signals during prototype development. Four Pmods are available embedding 8 data lines each as show in Figure 3.3.

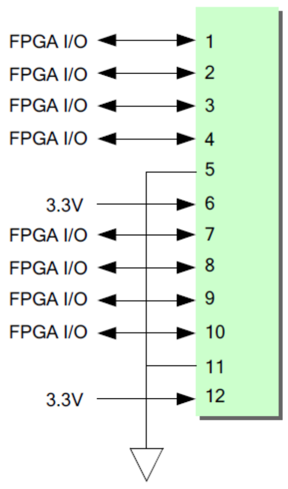


Figure 3.3: ZedBoard Pmod pinout.

Parallel bits sample transfer will require 32 data signals and 1 associated clock. This requires the use of all the four Pmods to receive the parallel GNSS multichannel data stream. In order to simplify the wiring and relax the requirements to fully occupy the Pmod connectors (which are not available in high performance Zynq platforms such as the ZC706 only one PMOD exposed) and alternative solution based on the FMC breakout board is followed.

The FMC breakout board, see Figure 3.4, exposes the entire FMC I/O and clock lines to a commodity 2.54 mm pins. Front-end PCB could be connected to the FMC breakout board using a standard 40 pins 2.54 mm female connector.

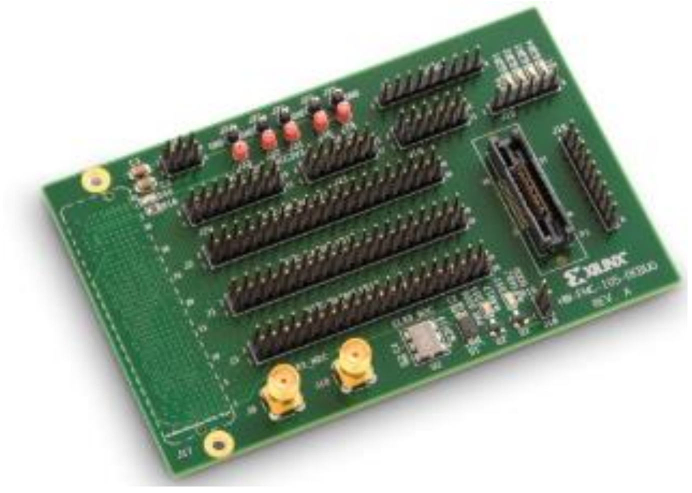


Figure 3.4: FMC XM105 Debug Card.

This solution reduces the initial effort for the development of the AUDITOR system adding multiple I/O lines using a standard ZedBoard, and allows the future evolution into and more embedded prototype as the MicroZed or ZC706 board which exposes only the FMC connectors. Thanks to the Zynq architecture flexibility the use of different Pmod or FMC wiring schemas could be easily converted by rerouting the Zynq internal constraints.

Adapter

The input GNSS I/Q samples from both channels are provided by a fixed 8-bit ADC module. In order to add additional flexibility to reduce this rate and relax the processing requirements from the Zynq platform lower bits/sample can be configured. This module will be implemented as a VHDL core that will provide standard bits per sampled used in GNSS receiver: 2-bit, 4-bit or 8-bit per sample.

Buffer

The decimated samples provided by the Adapter module need to be streamed in a continuous way to be consumed by pre-processing algorithms. In order to do this a Buffer is provided implemented as a high performance VHDL core that can be read/write at different speeds. The buffer packing strategy could be configured taking into account the Adapter bits per sample and the fixed buffer word size of 32-bits, the following strategies will be evaluated and partially implemented:

- Full 8-bits: 32 bits
- 4-bits: 16 data bits, 16 dummy bits
- 4-bits: 16 data bits sample 1, 16 data bits sample 2 (output read frequency/2)
- 2-bits: 8 data bits, 24 dummy bits
- 2-bits: 4 data bits sample 1, sample 2, sample 3, sample 4 (output read frequency/4)

FE config interface

The FE config size will provide via custom registers the configuration of the Adapter and Buffer parameters:

- Adapter bits-sample
- Buffer packing strategy

The configuration of these register will be available from the Zynq Processing system running a Linux OS via the Front-end driver (see subsection 3.2.2)

3.2.2 Hardware accelerators

The hardware accelerators implement signal processing functions that require a lot of computational power. These functions are implemented in the Zynq Processing Logic (PL) to offload these tasks from the ARM cores (this is: the Zynq processing system or PS). The hardware accelerators process data at the base-band sampling rate. There are two types of hardware accelerator modules: the acquisition modules and the correlation modules (tracking).

The PL not only implements the hardware accelerators but also the reception of the base-band samples coming from the A/D converters and the buffering of the samples, possibly including digital down-sampling filters. Figure 3.5 shows a block diagram of the PL, including the hardware accelerators and the buffering of the received samples.

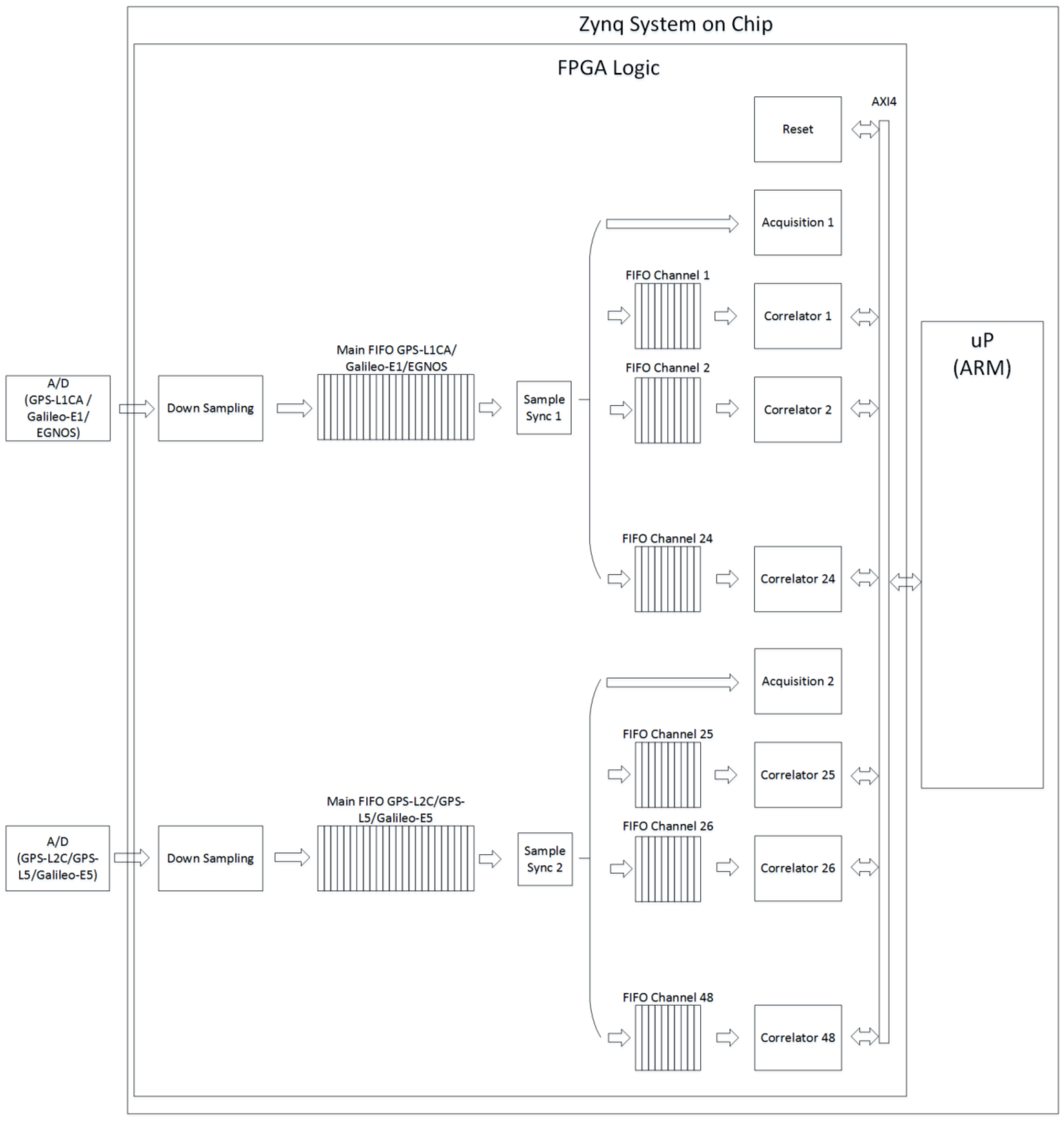


Figure 3.5: Block diagram of the PL

The PS instantiates the following modules:

- Reset: this module performs a total RESET of the FPGA modules and logic upon request of the uP (the PS).
- Down-sampling: digital down-sampling filters to reduce the sampling frequency of the base-band signal. The down-sampling filters can be implemented at the input or at the output of the main FIFO.
- Main FIFO GPS-L1CA/Galileo-E1/EGNOS: queue that stores the complex I/Q samples delivered by the A/D converter that is tuned to the RF frequency band of the following signals: GPS-L1CA, Galileo-E1 and EGNOS. These signals use the same RF carrier frequency and can be received simultaneously. The FIFO receives I/Q samples at the A/D sampling rate.

- Main FIFO GPS-L2C/GPS-L5/Galileo-E5: queue that stores the complex I/Q samples delivered by the A/D converter that is tuned to any of the following bands: GPS-L2C, GPS-L5 or Galileo-L5. Note that the GPS-L5 and the Galileo-E5a RF carrier frequencies are the same and can be received simultaneously. The other RF carrier frequencies are different and cannot be received simultaneously. The FIFO receives I/Q samples at the A/D sampling rate.
- Acquisition modules: module that runs the acquisition algorithm. Acquisition module 1 is connected to the GPS-L1CA, Galileo-E1 and EGNOS signals. Acquisition module 2 is connected to the GPS-L2C, GPS-L5 and Galileo-E5 signals.
- Sample Sync 1: this module synchronizes the output of the Main FIFO GPS-L1CA/Galileo-E1/EGNOS, the Acquisition 1 and the channels FIFOs 1 to 24. This module ensures that every FIFO output sample is captured by the Acquisition Module and all the FIFO channels that are connected to it.
- Sample Sync 2: this module synchronizes the output of the Main FIFO GPS-L2C/GPS-L5/Galileo-E5, the Acquisition 2 and the channels FIFOs 25 to 48. This module ensures that every FIFO output sample is captured by the Acquisition Module and all the FIFO channels that are connected to it.
- Channel FIFOs: the channel FIFOs store the channel input samples temporarily. The purpose of the channel FIFOs is to avoid the main FIFOs from being blocked when one of the channels blocks the input samples temporarily. The Acquisition modules do not require a FIFO channel because they drop the samples that they don't use, therefore the acquisition modules never block the data flow. Note: depending on the overall performance that is achieved by the main FIFO, the channel FIFOs may not be needed.
- Correlator modules: the correlator modules implement the Doppler wipe-off and the correlation between the received PRN sequence and the local PRN sequence. We expect to be able to use a maximum of 24 correlators connected to the GPS-L1CA/Galileo-E1/EGNOS analog front-end, and 24 correlators connected to the GPS-L2C/GPS-L5/Galileo-E5 analog front-end. However the number of correlators might be limited by the capacity of the PL.

The interface between the hardware accelerators and the PS is implemented using the AXI protocol (see section 0).

<u>Coordination between the hardware accelerators and the PL</u>:

When the receiver is turned on the receiver needs to locate the GNSS satellites that are available at that moment. To do so and to track the existing satellites, the PS runs the following algorithm:

Table 3.2: Use of the hardware accelerators to locate and track the GNSS satellites.

Step	**Action**
1	Load the local codes and configuration data corresponding to the candidate satellites to be located first to the PL acquisition modules
2	Run the acquisition modules.
3	When one of the acquisition modules finishes check the results to see if a satellite has been detected or not.
4	If a satellite has been detected enable one of the tracking modules and set it to track that satellite.

5	Load the local code of another candidate satellite to the acquisition module that is ready and run the acquisition module again.
6	In the meantime the PS receives data from the tracking modules that are locked to a GNSS satellite. Simultaneously to the location of new GNSS satellites by the acquisition modules in the PL, the PS processes the data obtained from the tracking modules.
7	If any tracking module loses contact with the GNSS satellite that it is tracking then disable the tracking module. It becomes then available to track new satellites.
8	Go back to step 3

The buffer control and the hardware accelerators are explained in the following subsections.

3.2.2.1 **Buffer control**

The PS receives the base-band samples from the A/D board at a fixed sampling rate. The receiver buffering performs three tasks:

- Store the received samples before they are processed by the acquisition and/or the correlation modules.
- Make the FPGA clock independent of the A/D sampling clock.
- Distribute the received samples among all the hardware accelerator modules (the acquisition module and the correlation modules).

<u>Buffering control</u>:

The primary sample buffering is performed by the main FIFOs (see Figure 3.5).

The hardware accelerators (acquisition and correlation modules) can process the received samples faster than the sampling clock. However the hardware accelerators may block the input buffer from time to time when they are running internal calculations. When this happens the main FIFO stores the received samples temporarily. When the hardware accelerators are ready again they get more samples from the FIFO and they catch up with the received signal.

The sample synchronization modules (sample sync 1 and sample sync 2) make sure that the samples delivered by the main FIFOs are captured by all the hardware accelerators. This means that a received sample is not removed from the FIFOs unless all the hardware accelerators that are connected to that FIFO are ready to read the sample.

Because of the sample synchronization modules, if one hardware accelerator blocks the sample flow temporarily, none of the other hardware accelerators can read samples from the main FIFO. In order to avoid this, there is the possibility of adding smaller FIFOs between the main FIFO and every hardware accelerator module. These are the channel FIFOs shown in Figure 3.5. The way these FIFOs work is the following: if one or some of the hardware accelerators block the sample flow temporarily (because they are not ready to read samples from the FIFO), then the corresponding channel FIFO reads the samples from the Main FIFO and stores the samples temporarily for that hardware accelerator only. In this way the hardware accelerators that are ready to read samples are not blocked by the modules that are blocking the flow of data. When the blocking modules are ready they read the samples from their channel FIFOs instead of the main FIFO, they process the samples and they catch up with the other modules.

The acquisition modules drop the samples that they don't need to process; therefore they don't need the channel FIFOs. The tracking modules cannot drop any samples, if they are not ready to process samples they block the data flow, therefore the channel FIFOs are possibly needed for them.

Independence of the sampling clocks:

This is a common hardware issue. The PS and the PL work with their internal sampling clocks. The A/Ds deliver the samples to the FPGA using the A/D sampling clocks. The FPGA captures the A/D samples using the sampling clocks of the A/D but it needs to process the samples using the internal PS and PL clocks.

This clock conversion is done in the PS, in the buffering itself. The buffering is implemented using Xilinx FIFOs, which are hardware designed to use different clocks at the input and at the output. The Xilinx FIFO modules read samples using the external A/D sampling clock but they output samples using the sampling clock of the PL. This process is called clock domain crossing (CDC).

Sample distribution among the hardware accelerators

As explained above in the buffering paragraphs, the sample distribution among the hardware accelerators is implemented in the sync modules. These modules ensure that all the samples are captured by all the hardware accelerator modules and no module misses any sample.

3.2.2.2 **Signal Acquisition**

The Acquisition modules run the acquisition algorithm.

The Acquisition module processes samples on a block by block basis. The samples that are not processed are dropped. As the non-processed samples are dropped, the Acquisition module does not need to have an input FIFO like the other channels: from the moment the Acquisition module starts processing a block of samples, all the incoming samples are dropped until a new block of samples is processed. There is no requirement on how fast the acquisition has to be able to process the samples, apart from the fact that it should process them as fast as possible in a way that the total acquisition time is acceptable for the user. Therefore the Acquisition module captures a new block of samples for processing as soon as the processing of the current block of samples is finished, but there is no need to buffer the received samples.

In spite of the fact that the samples that are not processed are dropped, the Acquisition module needs to keep track of the number of samples that have entered acquisition since the FPGA was booted. When the acquisition module detects a satellite, it reports both the sample number of the first sample of the block of samples that has been processed with the acquisition algorithm and the synchronization point or offset from the first sample, apart from other estimated parameters. This information is needed by the tracking algorithms such that they can locate the synchronization point in their buffers.

In order for the sample counter of the acquisition module and the sample counters of the correlator modules to be synchronized, the sample counters of all the modules count up all the received samples. All the modules (acquisition and correlators) receive samples all the time from the moment the FPGA is programmed, regardless whether the modules are enabled or disabled, regardless whether the modules are processing samples or not and regardless whether the modules are have some kind of interaction with the PS or not. The following table summarizes the behaviour of the acquisition module:

Table 3.3: Processing of the received samples and behaviour of the input sample counter in the processing module

Status of the Acquisition module	**Processing of the input samples**	**Input sample counter**
Disabled	Receive the input samples and drop them. Do not block the input data flow.	Increment the input sample counter for each received complex sample (I/Q).
Configuration (by the PS)	Receive the input samples and drop them. Do not block the input data flow.	
Capturing samples for processing	Capture the input samples and store them in an input buffer	
Processing a captured block of samples and reporting data to the PS	Receive the input samples and drop them. Do not block the input data flow.	

As the input sample counter cannot have an unlimited number of bits, when it reaches its maximum value it wraps back to zero and restarts. This is expected and it is a normal behaviour. As the input sample counters from all the modules (acquisition and correlators) have the same number of bits, the channels are still able to use the input sample counter and synchronization point reported by the acquisition module. The only trick is that the correlator algorithms have to check for a possible counter wrapping back to zero.

The acquisition module produces output data only when they have valid output data available

Table 3.4: Production of output data in the acquisition modules

Status of the correlator module	**Production of output data**
Disabled	The acquisition module does not produce any output data
Configuration (by the PS)	The acquisition module does not produce any output data

Status of the correlator module	Production of output data
Capturing samples, processing and reporting to the PS	The acquisition module produces output data only when it has valid data

3.2.2.3 Signal Tracking

The correlator modules run the correlator algorithm. Each tracking module corresponds to one channel. The correlator modules work with the following GNSS signals (GPS L1, GPS L5, Galileo L1 and Galileo L5).

The following block diagram is a description of the correlator modules.

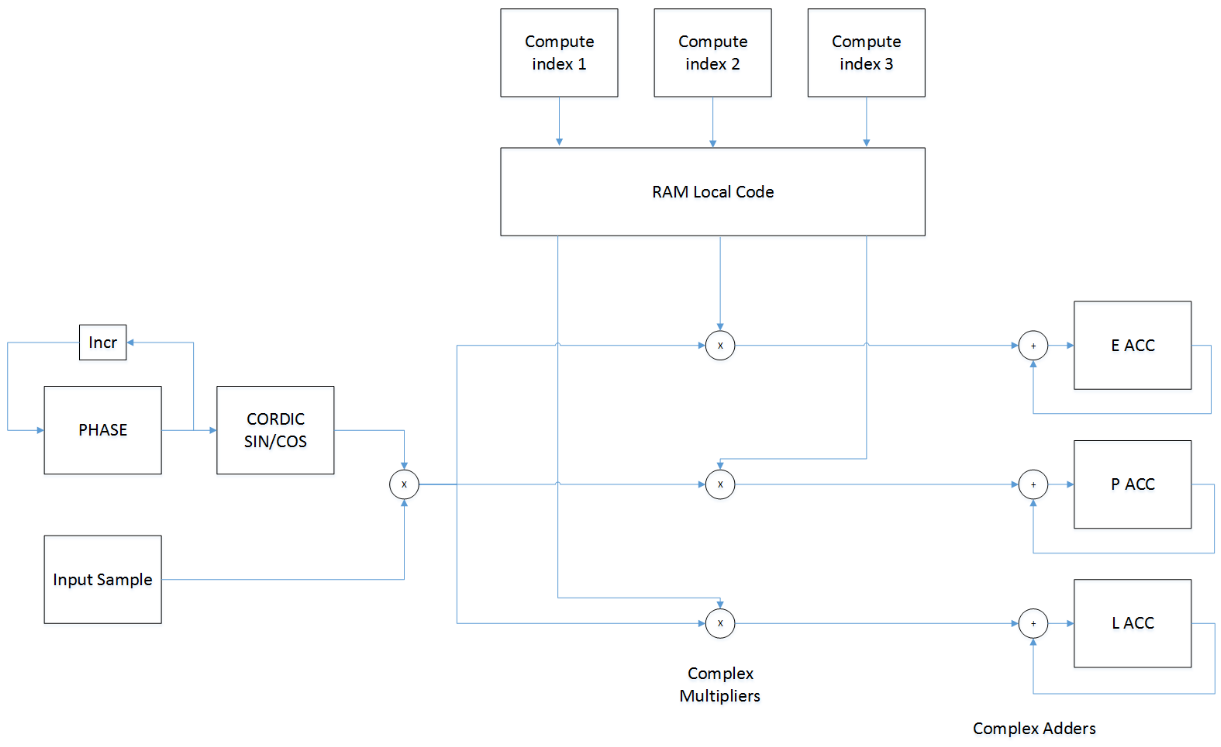


Figure 3.6: Correlation Module block diagram

The correlator modules run the Doppler wipe-off on the received signals (using the CORDIC SIN/COS algorithm) and correlate the signals with various shifted versions of the local GNSS codes in order to keep the synchronization between the receiver and the GNSS transmitters in the satellites. Index 1, index 2 and index 3 in Figure 3.6 point to various shifted positions in the memory that contains the local code. The result of the correlator module is stored in the Early, prompt and late accumulators (E ACC, P ACC and L ACC). In Figure 3.6 there are three correlation lines (early, prompt and late). Depending on the type of GNSS signal, up to 5 correlator lines may be needed to keep the receiver synchronized with the satellite.

As opposed to the acquisition algorithm, the correlator algorithms use all the received samples. It is expected that sometimes the correlator module has to stop the input flow of data because of some internal calculations or some interaction with the PS. Because of this the correlator modules use the channel FIFOs in the input.

As in the Acquisition module, the correlator modules need to keep track of the number of samples that have entered acquisition since the FPGA was booted.

In order for the sample counter of the acquisition module and the sample counters of the correlator modules to be synchronized, the sample counters of all the modules count up all the received samples. All the modules (acquisition and correlators) receive samples all the time from the moment the FPGA is booted, regardless whether the modules are enabled or disabled, regardless whether the modules are processing samples or not and regardless whether the modules are have some kind of interaction with the PS or not. The following table summarizes the behaviour of the correlator modules:

Table 3.5: Processing of the input samples in the channel modules

Status of the correlator module	**Processing of the input samples**	**Input sample counter**
Disabled	Receive the input samples and drop them. Do not block the input data flow.	Increment the input sample counter for each received complex sample (I/Q).
Configuration (by the PS)	Receive the input samples and drop them. Do not block the input data flow.	
Reporting data to the PS	Block the flow of input samples to avoid losing data	
Capturing samples and processing	Capture the input samples and process them. If the channel module has a temporary latency because of its normal working behaviour then block the input data flow (prevent new samples from coming in) during this latency.	

As in the case of the acquisition, the input sample counter cannot have an unlimited number of bits. When it reaches its maximum value it wraps back to zero and restarts. This is expected and it is a normal behaviour. As the input sample counters from all the modules (acquisition and correlators) have the same number of bits, the channels are still able to use the input sample counter and synchronization point reported by the acquisition module. The only trick is that the correlator algorithms have to check for a possible counter wrapping back to zero.

The correlator modules produce output data only when they have valid output data available (see Table 3.6).

Table 3.6: Status of the correlator

Status of the correlator module	**Production of output data**
Disabled	The correlator module does not produce any output data

Configuration (by the PS)	The correlator module does not produce any output data
Initialization (by the PS)	The correlator module does not produce any output data
Capturing samples and processing	The correlator module produces output data only when it has valid data

3.3 FPGA-ARM interface (AXI)

The AXI interface (Advanced eXtensible Interface) is a bus interface specification defined in AMBA (Advanced Microcontroller Bus Architecture), an open standard on-chip interconnect specification. The AXI bus is targeted at high performance, high clock frequency system designs. It includes features that make it suitable for high speed interconnection of microprocessors and peripherals.

The AXI bus is used by the Xilinx devices (FPGAs and SoCs) to exchange information between hardware accelerators and between hardware accelerators and microprocessors (the PL and the PL). For this reason there are a number of AXI interfaces between the PL and the PS.

The interfaces between the PS and the PL are the following:

Table 3.7: Interfaces between the PS and the PL

Interface	Characteristics
2x M_AXI_GP	Two 32-bit General Purpose AXI Master Interfaces (PS Master -> PL slave)
2x S_AXI_GP	Two 32-bit General Purpose AXI Slave Interfaces (PS Slave <- PL Master)
4x S_AXI_HP	Four 32/64-bit High Performance AXI Slave Interfaces (PS Slave <- PL Master)
1x S_AXI_ACP	One 32-bit cache-coherent AXI Slave Interface (PS Slave <- PL Master)

The characteristics of them are explained in the table below:

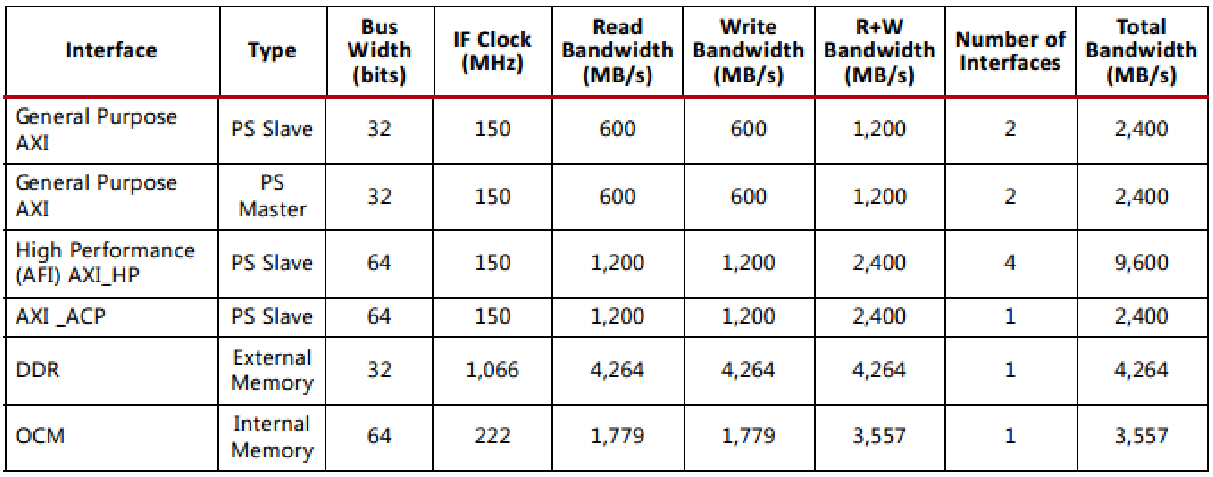
Table 3.8: Characteristics of the various AXI interfaces

AXI Interface	Characteristics
AXI_ACP (cache coherent)	Using this interface a PL master can access the following memories: DDR and the 256 kB OCM (On Chip Memory in the PS) The particularity of this type of interface is that it goes through the Snoop Control Unit, which means that whatever it is written or read to/from this memory is coherent with the contents of the L1/L2 Cache of the ARM processors and with what the ARM processors see when they access the same piece of memory. In exchange for this coherency, this interface may present some more

	latencies than the other AXI interfaces.
AXI_HP (High Performance)	Using this interface a PL master can access the DDR and the 256 kB ROM of the PS. This interface is optimized for high performance, meaning that it is optimized for throughput. The difference between this type of interface and the other interfaces is that this one is queued and it is not cached. It contains FIFOs to ensure that the PL can exchange data with the PS at a constant throughput regardless of temporary latencies in the AXI interconnect. The FIFOs absorb these latencies.
AXI_GP (General Purpose)	General Purpose AXI interface. This type is the only type of interface that contains AXI Masters on the PS side. They are used by the PS to access devices in the PL that are memory-mapped to the AXI addresses.

An aspect to consider is the theoretical bandwidth of the various interfaces is reported in the Xilinx manual (see table below).

Table 3.9: Theoretical bandwidth of PS-PL and PS memory interfaces (from Xilinx)



Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI _ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400
DDR	External Memory	32	1,066	4,264	4,264	4,264	1	4,264
OCM	Internal Memory	64	222	1,779	1,779	3,557	1	3,557

The figure below shows the various AXI interfaces, the OCM (On Chip Memory) and the DDR (Double Data Rate RAM memory) interface, and can be used graphically to see what resources can be accessed from which AXI interface, including the AXI interfaces that are internal to the PS.

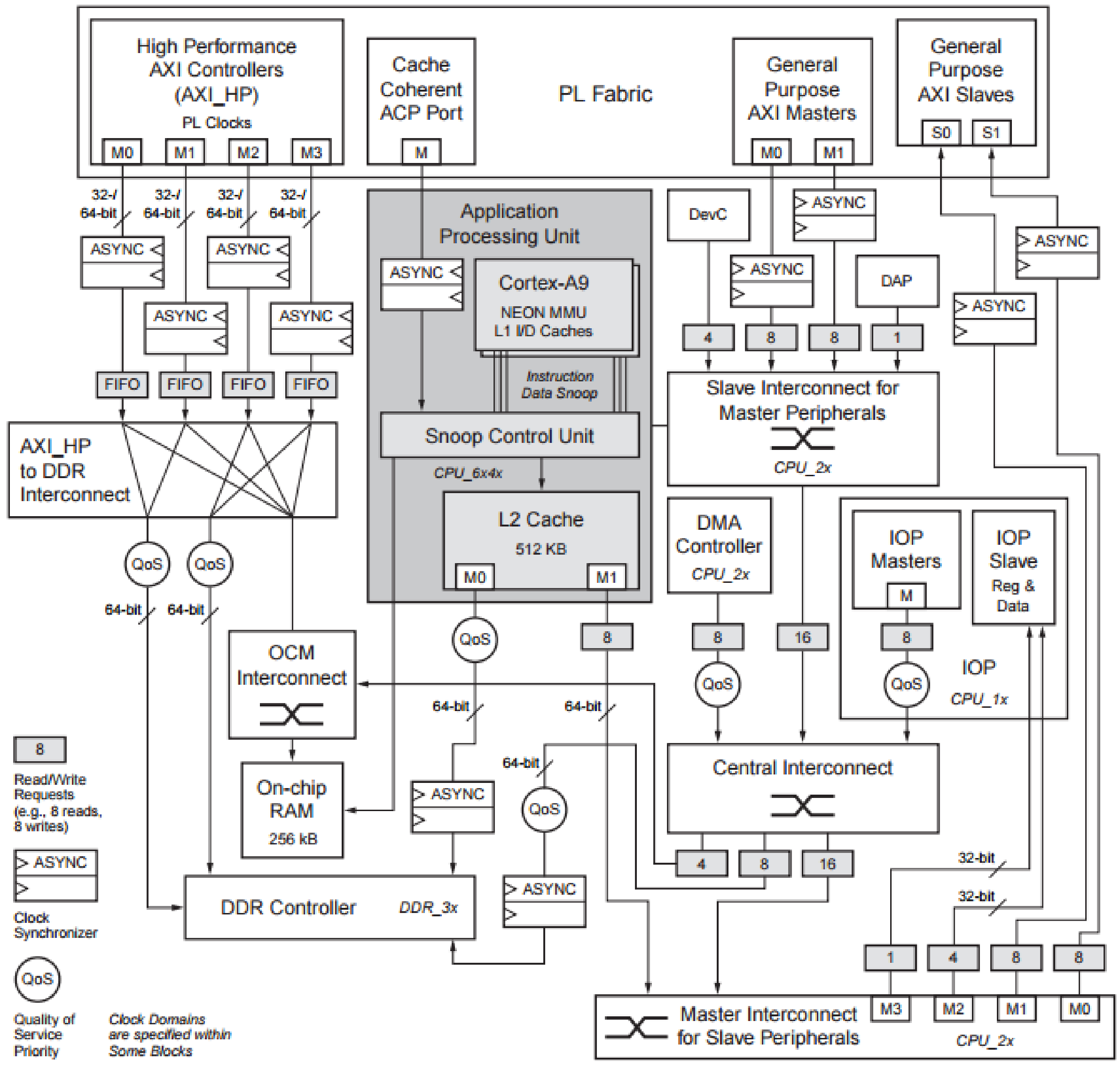


Figure 3.7: APU (Application Processor Unit) system view diagram

3.4 Zynq Processing System (PS)

The processing systems include multiple processes that run in a Linux OS to implement the high level algorithms and user services of the GNSS receiver. The use of a Linux OS speeds the development due to the high availability of different third-party libraries and utilities while the more critical raw real-time pre-processing is controlled by the PL.

3.4.1 Front-end driver

The Front-end driver is a software component to provide a common configuration interface for the FE logic and the FE module from the Zynq Processing system running a Linux OS.

The main parameters that can be configured using this module are shown in Table 2.2 and extended with the configuration parameters of the Adapter and Buffer cores included in the FE logic detailed in 3.2.1:

Table 3.10: Front-end logic commands

Command	Description
Configure adapter decimation	Select 8-bit, 4-bit or 2-bit.
Configure buffer pack strategy	32 bits with dummy bits or full 32 data bits with multiple samples.

3.4.2 GNSS-SDR module

GNSS-SDR is the software application in charge of computing GNSS data (that is, GNSS observables, including pseudorange, pseudorange rate, phase range and signal strength, and the corresponding navigation messages) from the raw signal samples delivered by the radio frequency front-end.

A GNSS software receiver is a complex system, which description needs to be addressed at different abstraction layers. Hereafter we describe the software architecture implemented in GNSS-SDR, which is based on GNU Radio (see http://gnuradio.org); a well-established framework that provides the signal processing runtime and processing blocks to implement software radio applications. Frameworks are a special case of software libraries - they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries: the overall program's flow of control is not dictated by the caller, but by the framework; and it can be extended by the user usually by selective overriding or specialized by user code providing specific functionality. Software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time. GNSS-SDR proposes a software architecture that builds upon the GNU Radio framework in order to implement a GNSS receiver.

The general overview is as follows:

- **The Control Plane** is in charge of creating a *flow graph* in which a sample stream goes through a network of connected signal processing blocks up to the position fix. The nature of a GNSS receiver imposes some requirements in the architecture design: since the composition of the received GNSS signals will change over time (initially, some satellites will be visible, and after a while, some satellites will not be visible anymore and new ones will show up), some channels will lose track of their signals and some new channels will have to be instantiated to process the new signals. This means that the receiver must be able to activate and deactivate the channels dynamically, and it also needs to detect these changes during runtime.
- **The Signal Processing Plane**, consisting of a collection of blocks that actually implement digital signal processing algorithms. Efficiency is specially critical before and during correlations (the most complex operation in terms of processing load, but from which sample rate decreases three orders of magnitude), and even a modern multi-purpose processor must be properly programmed in order to attain real-time.

3.4.2.1 **The Control Plane**

The Control Plane is in charge of creating a flow graph according to the configuration and then managing the modules. Configuration allows users to define in an easy way their own custom receiver by specifying the flow graph (type of signal source, number of channels, algorithms to be used for each channel and each module, strategies for satellite selection, type of output format, etc.). Since it is difficult to foresee what future module implementations will be needed in terms of configuration, we used a very simple approach that can be extended without a major impact in the code. This can be achieved by simply mapping the names of the variables in the modules with the names of the parameters in the configuration.

3.4.2.1.1 The configuration mechanism

Properties are passed around within the program using the `ConfigurationInterface` class. There are two implementations of this interface: `FileConfiguration` and `InMemoryConfiguration`.

- `FileConfiguration` reads the properties (pairs of property name and value) from a file and stores them internally.
- `InMemoryConfiguration` does not read from a file; it remains empty after instantiation and property values and names are set using the `set_property` method.

`FileConfiguration` is intended to be used in the actual GNSS-SDR application, whereas `InMemoryConfiguration` is intended to be used in tests to avoid file-dependency in the file system.

Classes that need to read configuration parameters will receive instances of `ConfigurationInterface` from where they will fetch the values. For instance, parameters related to `SignalSource` should look like this:

Table 3.11: Configuration file syntax: setting parameters of the `SignalSource` module

```
SignalSource.parameter1=value1
SignalSource.parameter2=value2
```

The name of these parameters can be anything but one reserved word: `implementation`. This parameter indicates in its value the name of the class that has to be instantiated by the factory for that role. For instance, if we want to use the implementation `Pass_Through` for module `SignalConditioner`, the corresponding line in the configuration file would be

Table 3.12: Configuration file syntax: setting the implementation of the `SignalConditioner` module

```
SignalConditioner.implementation=Pass_Through
```

Since the configuration is just a set of property names and values without any meaning or syntax, the system is very versatile and easily extendable. Adding new properties to the system only implies modifications in the classes that will make use of these properties. In addition, the configuration files are not checked against any strict syntax so it is always in a correct status (as long as it contains pairs of property names and values in INI format. An INI file is an 8-bit text file in which every property has a name and a value, in the form `name = value`. Properties are case-insensitive, and cannot contain spacing characters. Semicolons (`;`) indicate the start of a comment; everything between the semicolon and the end of the line is ignored.

3.4.2.1.2 The GNSS Block Factory

Hence, the application defines a simple accessor class to fetch the configuration pairs of values and passes them to a factory class called `GNSSBlockFactory`. This factory decides, according to the configuration, which class needs to be instantiated and which parameters should be passed to the constructor. Hence, the factory encapsulates the complexity of blocks' instantiation. With that approach, adding a new block that requires new parameters will be as simple as adding the block class and modifying the factory to be able to instantiate it. This loose coupling between the blocks' implementations and the syntax of the configuration enables extending the application capacities in a high degree. It also allows to produce fully customized receivers, for instance a testbed for acquisition algorithms, and to place observers at any point of the receiver chain.

3.4.2.1.3 The GNSS Flow Graph

The `GNSSFlowgraph` class is responsible for preparing the graph of blocks according to the configuration, running it, modifying it during run-time and stopping it.

Blocks are identified by its role. This class knows which roles it has to instantiate and how to connect them to configure the generic graph that is shown in Figure 3.8. It relies on the configuration to get the correct instances of the roles it needs and then it applies the connections between GNU Radio blocks to make the graph ready to be started.

The complexity related to managing the blocks and the data stream is handled by GNU Radio's `gr::top_block` class. Hence, `GNSSFlowgraph` wraps the `gr::top_block` instance so we can take advantage of the GNSS block factory, the configuration system and the processing blocks. This class is also responsible for applying changes to the configuration of the flow graph during run-time, dynamically reconfiguring channels: it selects the strategy for selecting satellites. This can range from a sequential search over all the satellites' ID to smarter approaches that determine what satellites are most likely in-view, based on rough estimations of the receiver position, in order to avoid searching satellites in the other side of the Earth. This class internally codifies actions to be taken on the graph. These actions are identified by simple integers. `GNSSFlowgraph` offers a method that receives an integer that codifies an action, and this method triggers the action represented by the integer.

Actions can range from changing internal variables of blocks to modifying completely the constructed graph by adding/removing blocks. The number and complexity of actions is only constrained by the number of integers available to make the codification.

This approach encapsulates the complexity of preparing a complete graph with all necessary blocks instantiated and connected. It also makes good use of the configuration system and of the GNSS block factory, which keeps the code clean and easy to understand. It also enables updating the set of actions to be performed to the graph quite easily.

3.4.2.1.4 The Control Thread

The `ControlThread` class is responsible for instantiating the `GNSSFlowgraph` and passing the required configuration. Once the flow graph is defined and its blocks connected, it starts to process the incoming data stream. A `ControlThread` object is then in charge of reading the control queue and processing all the messages sent by the processing blocks via the thread-safe message queue.

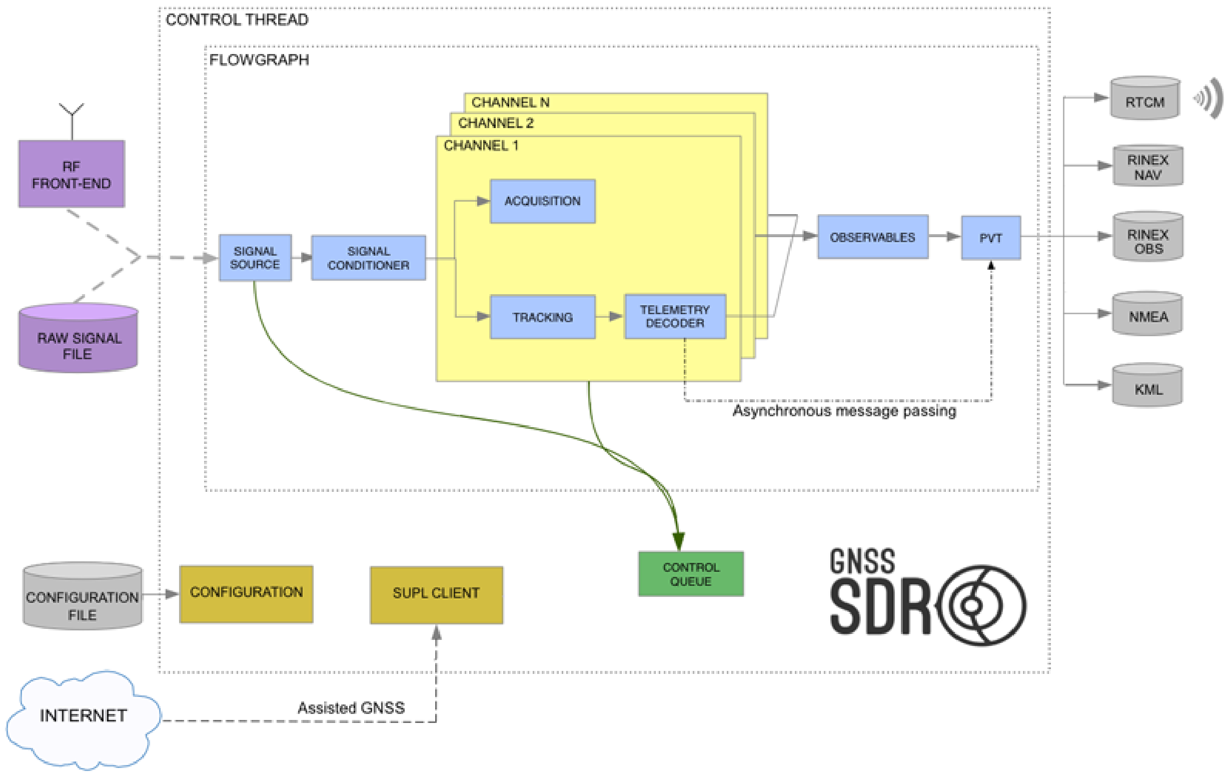


Figure 3.8: General block diagram of a GNSS receiver.

3.4.2.2 The Signal Processing Plane

GNU Radio's class hierarchy imposes a thread-per-block architecture that allows automatic scheduling in multicore processors, hiding all the complexity behind a simple and robust API. It uses shared memory to manage efficiently the flow of data between blocks, and offers a large set of well-programmed blocks that provide implementations for very common signal processing tasks. In contrast, GNU Radio does not provide any standard way to provide control over the blocks.

The user can build a receiver by creating a graph where the nodes are signal processing blocks and the lines represent the data flow between them. Conceptually, blocks process infinite streams of data flowing from their input ports to their output ports. The blocks' attributes include the number of input and output ports they have as well as the type of data that flows through each one of them. Once they are connected and form a flow graph, the application can run and data will be put into the stream. As long as there are data available, the working threads will run the code of the different blocks.

A key aspect of an object-oriented software design is the class hierarchy, depicted in Figure 3.9. The notation is as follows: we used a very simplified version of the Unified Modelling Language (UML), a standardized general-purpose modelling language in the field of object-oriented software engineering. In this document, classes are described as rectangles with two sections: the top section for the name of the class, and the bottom section for the methods of the class. A dashed arrow from `ClassA` to `ClassB` represents the dependency relationship. This relationship simply means that class A somehow depends upon class B. In C++, this almost always results in a `#include`. Inheritance models "is a" and "is like" relationships, enabling you to reuse existing data and code easily. When `ClassA` inherits from `ClassB`, we say that `ClassA` is the subclass of `ClassB`, and `ClassB` is the superclass (or parent class) of `ClassA`. The UML modelling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass.

As shown in Figure 3.9, `gr::basic_block` is the abstract base class for all signal processing blocks, a bare abstraction of an entity that has a name and a set of inputs and outputs. It is never instantiated directly; rather, this is the abstract parent class of both `gr::hier_block2`, which is a recursive container that adds or removes processing or hierarchical blocks to the internal graph, and `gr::block`, which is the abstract base class for all the processing blocks. A signal processing flow is constructed by creating a tree of hierarchical blocks, which at any level may also contain terminal nodes that actually implement signal processing functions.

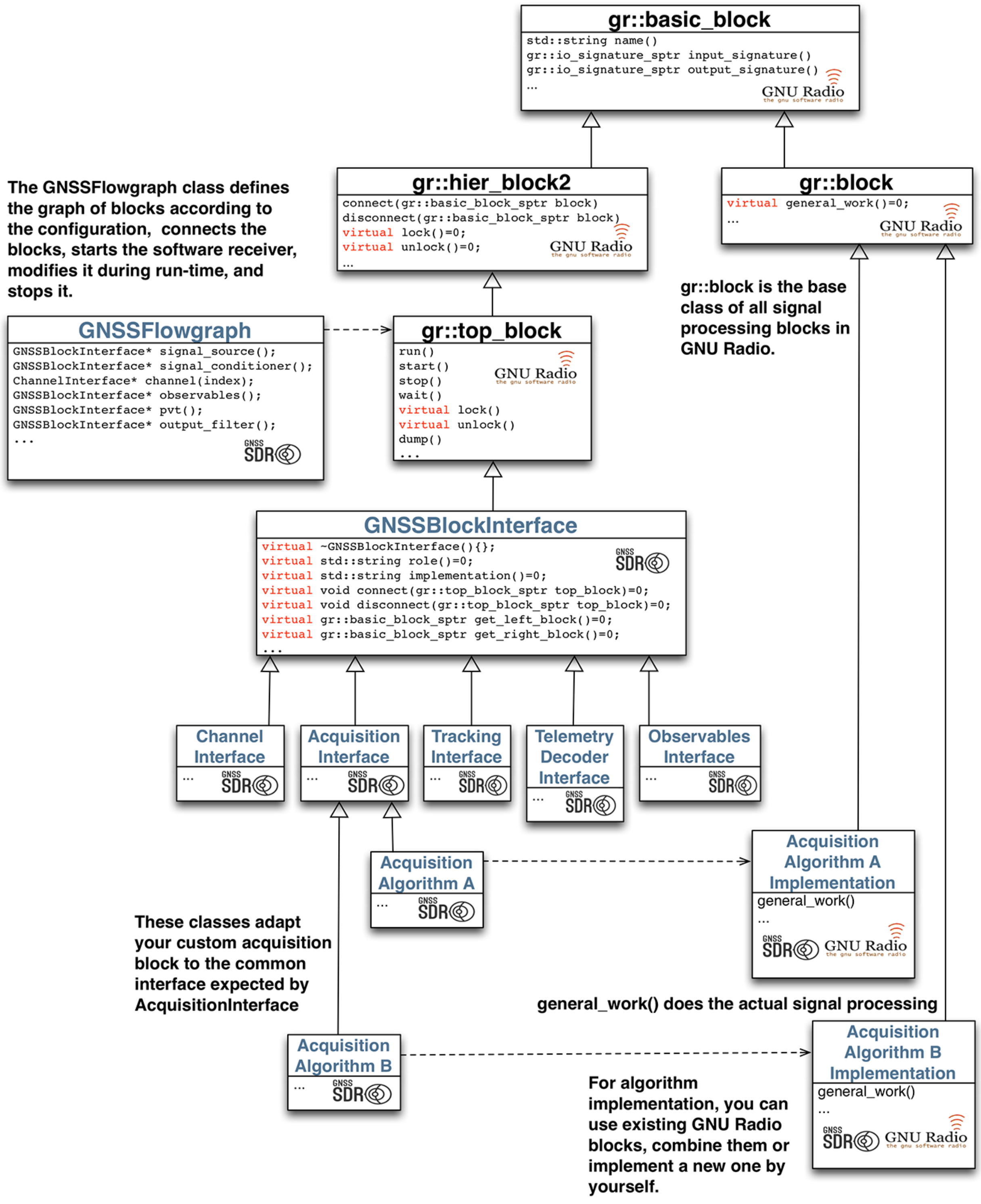

gr::basic_block
std::string name()
gr::io_signature_sptr input_signature()
gr::io_signature_sptr output_signature()
...
GNU Radio
gr::hier_block2
connect(gr::basic_block_sptr block)
disconnect(gr::basic_block_sptr block)
virtual lock()=0;
virtual unlock()=0;
...
gr::block
virtual general_work()=0;
...
The GNSSFlowgraph class defines the graph of blocks according to the configuration, connects the blocks, starts the software receiver, modifies it during run-time, and stops it.
gr::block is the base class of all signal processing blocks in GNU Radio.
GNSSFlowgraph
GNSSBlockInterface* signal_source();
GNSSBlockInterface* signal_conditioner();
ChannelInterface* channel(index);
GNSSBlockInterface* observables();
GNSSBlockInterface* pvt();
GNSSBlockInterface* output_filter();
...
gr::top_block
run()
start()
stop()
wait()
virtual lock()
virtual unlock()
dump()
...
GNSSBlockInterface
virtual ~GNSSBlockInterface(){};
virtual std::string role()=0;
virtual std::string implementation()=0;
virtual void connect(gr::top_block_sptr top_block)=0;
virtual void disconnect(gr::top_block_sptr top_block)=0;
virtual gr::basic_block_sptr get_left_block()=0;
virtual gr::basic_block_sptr get_right_block()=0;
...
Channel Interface
Acquisition Interface
Tracking Interface
Telemetry Decoder Interface
Observables Interface
Acquisition Algorithm A
Acquisition Algorithm A Implementation
general_work()
These classes adapt your custom acquisition block to the common interface expected by AcquisitionInterface
general_work() does the actual signal processing
Acquisition Algorithm B
Acquisition Algorithm B Implementation
general_work()
For algorithm implementation, you can use existing GNU Radio blocks, combine them or implement a new one by yourself.

Figure 3.9: Receiver's class hierarchy.

Class `gr::top_block` is the top-level hierarchical block representing a flow graph. It defines GNU Radio runtime functions used during the execution of the program: `run()`, `start()`, `stop()`, `wait()`, etc. As shown in Figure 3.10, a subclass called `GNSSBlockInterface` is the common interface for all the GNSS-SDR modules. It defines pure virtual methods, which are required to be implemented by a derived class. Classes containing pure virtual methods are termed "abstract;" they cannot be instantiated directly, and a subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class.

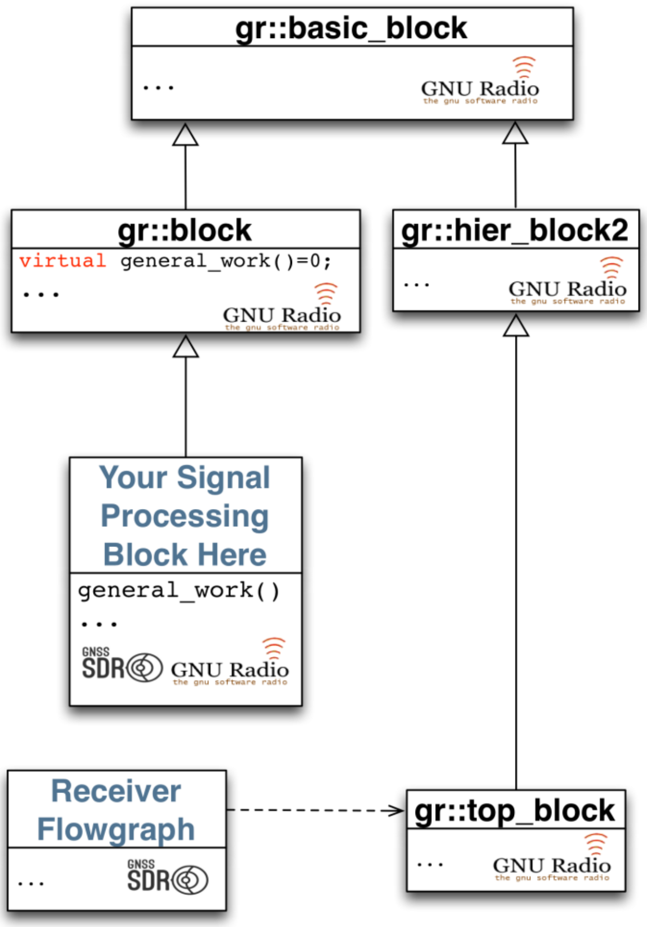


Figure 3.10: General interface for signal processing blocks.

Subclassing `GNSSBlockInterface`, we defined interfaces for the receiver blocks defined in Figure 3.8. This hierarchy, shown in Figure 3.9, provides the definition of different algorithms and different implementations, which will be instantiated according to the configuration. This strategy allows multiple implementations sharing a common interface, achieving the objective of decoupling interfaces from implementations: it defines a family of algorithms, encapsulates each one, and makes them interchangeable. Hence, we let the algorithm vary independently from the program that uses it.

3.4.2.3 **Example: dual-frequency receiver architecture**

A single Signal Source can be equipped with more than one radio-frequency chain. This is the case of AUDITOR's front-end, which will be able to deliver signals in two frequency bands. Those cases implies not only the configuration of the Signal Source, but also there is a need to set up different Signal Conditioners for each band, and configure the Channel implementations for the different signals present on each band.

An example of such configuration is provided in Table 3.13. The generated is shown in Figure 3.11.

Table 3.13: Configuration example for a dual-band receiver.

```
SignalSource.RF_channels=2
SignalSource.implementation=UHD_Signal_Source
...
SignalSource.subdevice=A:0 B:0
...
SignalSource.freq0=1575420000
SignalSource.freq1=1227600000
...

SignalConditioner0.implementation=...
DataTypeAdapter0.implementation=...
InputFilter0.implementation=...
Resampler0.implementation=...

SignalConditioner1.implementation=...
DataTypeAdapter1.implementation=...
InputFilter1.implementation=...
Resampler1.implementation=...


...
Channels_1C.count=8
Channels_2S.count=8

; # Channel connection
Channel0.RF_channel_ID=1
Channel1.RF_channel_ID=1
Channel2.RF_channel_ID=1
Channel3.RF_channel_ID=1
Channel4.RF_channel_ID=1
Channel5.RF_channel_ID=1
Channel6.RF_channel_ID=1
Channel7.RF_channel_ID=1
Channel8.RF_channel_ID=0
Channel9.RF_channel_ID=0
Channel10.RF_channel_ID=0
Channel11.RF_channel_ID=0
Channel12.RF_channel_ID=0
```

```
Channel13.RF_channel_ID=0
Channel14.RF_channel_ID=0
Channel15.RF_channel_ID=0

; Channel signal
Channel0.signal=1C
Channel1.signal=1C
Channel2.signal=1C
Channel3.signal=1C
Channel4.signal=1C
Channel5.signal=1C
Channel6.signal=1C
Channel7.signal=1C
Channel8.signal=2S
Channel9.signal=2S
Channel10.signal=2S
Channel11.signal=2S
Channel12.signal=2S
Channel13.signal=2S
Channel14.signal=2S
Channel15.signal=2S

...

Acquisition_1C.implementation=...
   ; or Acquisition_1C0, ..., Acquisition_1C7
Acquisition_2S.implementation=...
   ; or Acquisition_2S8, ..., Acquisition_2S15

Tracking_1C.implementation=...
   ; or Tracking_1C0, ..., Tracking_1C7
Tracking_2S.implementation=...
   ; or Tracking_2S8, ..., Tracking_2S15

TelemetryDecoder_1C.implementation=...
   ; or TelemetryDecoder_1C0, ..., TelemetryDecoder_1C7
TelemetryDecoder_2S.implementation=...
```

```
   ; or TelemetryDecoder_2S8, ..., TelemetryDecoder_2S15


...
```

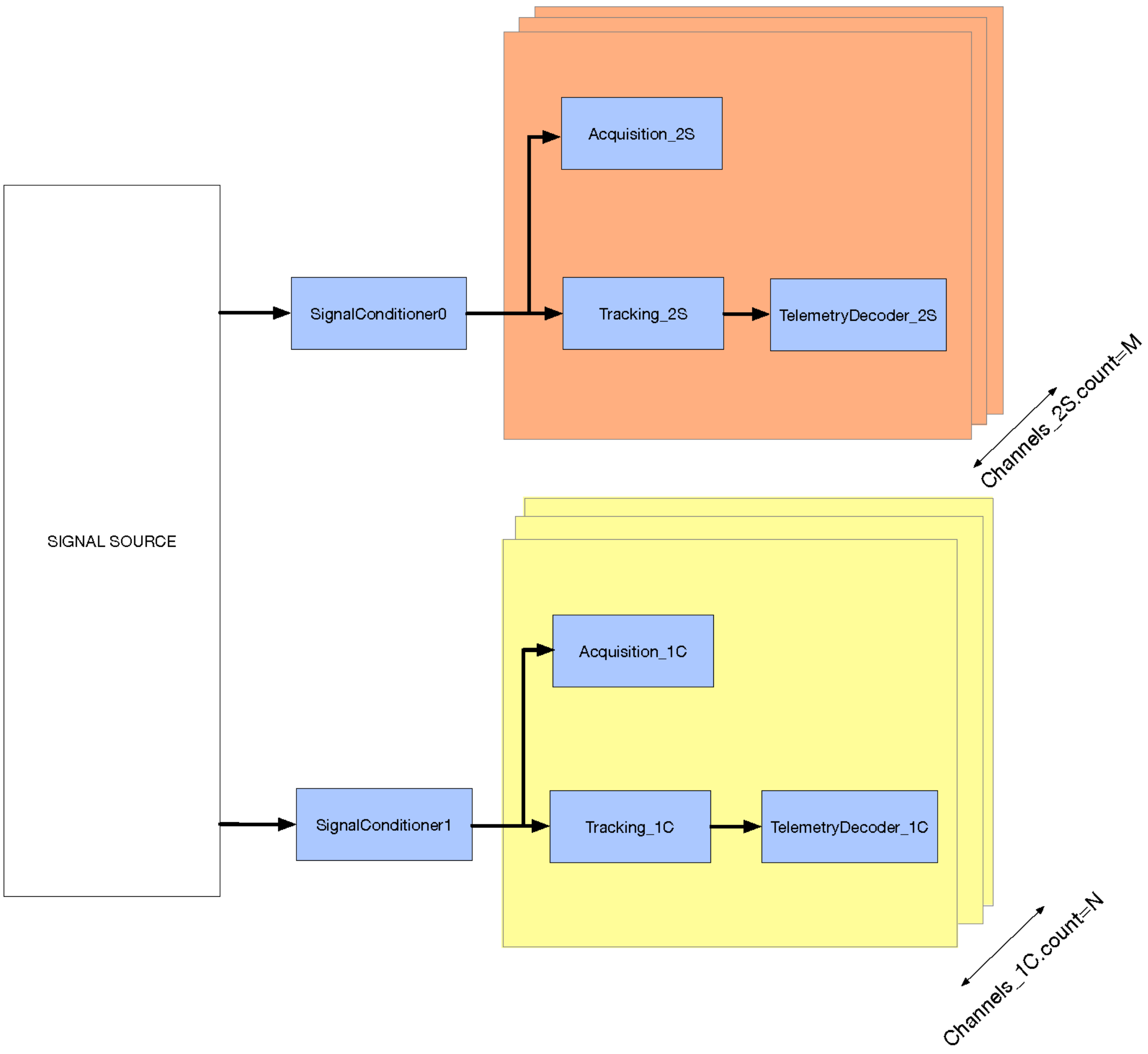


Figure 3.11: Simplified block diagram of a dual-band receiver of GPS L1 C/A and GPS L2C (M) signals.

3.4.3 iBOGART user net and Client PVT Solver

A modified RTKLIB implementation (AUDITOR RTKLIB) will be used as client PVT solver, to be installed in the Zynq Processing system running the Linux OS (as part of the High Accuracy Software Module). This is of key relevance since we know in advance that there is compatibility between RTKLIB and GNSS-SDR. In addition, this allows RTCM handling and the possibility to provide the outputs in multiple formats, including NMEA.

RTKLIB is an open source positioning package and thus, any user can have access to its source code (mainly implemented in C-language). In order to adapt it to AUDITOR, it will be necessary to modify several RTKLIB routines, mainly to allow the applicability of the WARTK corrections in the PVT solver.

AUDITOR RTKLIB PVT solver module will be fed mainly by two inputs, the user raw GNSS measurements and the WARTK correction messages, corresponding mainly to precise predicted orbits and ionospheric delay model and will provide position (and velocity) as outputs. More in detail, these aspects are covered in the following sections.

3.4.3.1 **iBOGART-user-net and PVT Inputs**

As input, it will be necessary to retrieve the WARTK messages from the iBOGART Cloud (through an FTP connection and/or considering RTCM) certain existing products available from IGS servers (like clock and orbit corrections) as well as WARTK messages distributed through the internet.

For additional details on RTCM, please refer to the corresponding section.

3.4.3.2 **iBOGART-user-net and PVT baseline core: AUDITOR RTKLIB**

RTKLIB can be the baseline solution for the core of the iBOGART-user-net and PVT at the user side. This is because this open source software allows precise positioning based on double differences or PPP techniques. For such purpose, it can retrieve data from other reference stations, as well as corrections, via internet, and it can apply DGNSS (see page 161 in the RTKLIB manual; available at http://www.rtklib.com/prog/manual_2.4.2.pdf) and PPP (see page 171 in the manual) algorithms. In addition, it can handle RTCM Messages (see page 124 in the manual). Nonetheless, since RTCM does not allow the proper distribution of WARTK messages, an update of RTKLIB will be necessary in the context of AUDITOR.

Considering a modified RTKLIB (referred to as AUDITOR RTKLIB) is of key importance to easily reach the market. It should be taken into account that this is an Open Source solution that could be easily embedded in the Zynq Processing System within the GNSS-SDR receiver.

The main scripts of RTKLIB are programmed in C and located in <RTKLIB root>/src. Their corresponding total number of lines, for the present version, is 36636, distributed as follows:

Table 3.14: RTKLIB scripts and number of lines

File Name	Number of Lines
convkml.c	197
convrnx.c	1058
datum.c	132
download.c	837
ephemeris.c	750
geoid.c	7490
ionex.c	477
lambda.c	188
options.c	510

pntpos.c	585
postpos.c	1232
ppp_ar.c	471
ppp.c	1054
preceph.c	589
qzslex.c	618
rcvraw.c	412
rinex.c	2482
rtcm2.c	428
rtcm3.c	2101
rtcm3e.c	2161
rtcm.c	377
rtkcmn.c	3688
rtkpos.c	1792
rtksvr.c	882
sbas.c	916
solution.c	1533
stec.c	341
stream.c	2190
streamsvr.c	588
tle.c	557
Total	**36636**

In AUDITOR, it will be necessary to modify several of such C-programs to allow for the application of WARTK messages. The RTCM conversion tool, part of RTKLIB package, will also play an important role to decode the messages when transmitted in RTCM format.

In particular, ionospheric corrections to be applied for standard and PPP precise positioning can be selected from a set of options, including the broadcast Klobuchar model, IONEX GIMs and SBAS ionospheric models (like NeQuick for EGNOS). As an example, this can be seen in the function ionocorr, which computes the ionospheric corrections within the pntpos.c code (to enable standard positioning):

Table 3.15: Ionospheric corrections in within ionocorr function

```
    /* broadcast model */
      if (ionoopt==IONOOPT_BRDC) {
          *ion=ionmodel(time,nav->ion_gps,pos,azel);
          *var=SQR(*ion*ERR_BRDCI);
          return 1;
      }
      /* sbas ionosphere model */
      if (ionoopt==IONOOPT_SBAS) {
          return sbsioncorr(time,nav,pos,azel,ion,var);
      }
      /* ionex tec model */
      if (ionoopt==IONOOPT_TEC) {
          return iontec(time,nav,pos,azel,1,ion,var);
      }
      /* qzss broadcast model */
      if (ionoopt==IONOOPT_QZS&&norm(nav->ion_qzs,8)>0.0) {
          *ion=ionmodel(time,nav->ion_qzs,pos,azel);
          *var=SQR(*ion*ERR_BRDCI);
          return 1;
      }
      /* lex ionosphere model */
      if (ionoopt==IONOOPT_LEX) {
          return lexioncorr(time,nav,pos,azel,ion,var);
      }
```

Several options are also present in the case of ppp.c code (script to enable PPP positioning). In the case of rtkpos.c (script to enable precise positioning based on double differences), the ionospheric corrections are not applied. Then, it will be necessary to apply the corrections derived from WARTK messages in a similar way as in the case of standard or PPP positioning.

The different functions called to derive the ionospheric corrections and related parameters are included in other C programs like rtkcmn.c (broadcast model case within the RTKLIB common functions C script), ionex.c (tec model case; see also next paragraph) and sbas.c (SBAS model case). In the AUDITOR case, the corrections to be applied at the user location will need to be derived from DSM WARTK messages. For such a purpose, a new program will be implemented, with filename wartk.c, to implement the corresponding corrections based on the double differenced STEC values for each satellite. Another possibility would be to send the corrections in IONEX-like format and then adapt the existing ionex.c script when necessary

Note that the C scripts on IONEX (ionex.c) and STEC (stec.c) contain several functions to allow reading IONEX format files (like GIMs), computing interpolated ionospheric values from a IONEX at the user location, derive the associated ionospheric delay, add STEC data to a grid, among other functions.

Regarding ambiguity resolution techniques, there are two already implemented in RTKLIB (TCAR, i.e. three carrier ambiguity resolution, and WL-NL, i.e. wide lane – narrow lane ambiguity resolution). Their implementation can be found in <RTKLIB root>/src/rtkpos_gsi.c (script to allow precise positioning for a certain experiment named GSI). In the frame of AUDITOR, there will be the need to implement a WARTK ambiguity resolution/constrain technique, mainly based on the findings of [5].

After any new routine is implemented (or any update on existing routines), it will be necessary to recompile the corresponding source codes, for instance by means of a makefile file (taking into account the existing one used to compile the Linux version of RTKLIB package and associated tools; see <RTKLIB root>/app/makefile and other makefile scripts within <RTKLIB root>/app).

It is important to remark that even though this approach is the most suitable one to allow reaching the market, it has some associated risks that might be critical to reach the precise positioning goal within AUDITOR. In this regard, it shall be remarked that many routines may probably be not tuned in the same way as WARTK CPF PVT solver at the user side. For instance, how the Kalman filter is configured (process noise matrix for each modelled term may differ substantially) or how cycle-slips are detected.

In this context, the iBOGART-user-net and PVT solver will be mainly based on GPS L1 and L2 (or L2C) measurements to maximize the reliability of the adopted solution, to minimize risks associated to further complexity of considering a multi-GNSS.

3.4.3.3 **iBOGART-user-net and PVT Outputs**

As it has been pointed out,the provision of the navigation solution (position and velocity) in NMEA format is also feasible by means of RTKLIB (see details provided in previous section). Nevertheless NTRIP could be supported in the future since GNSS-SDR allows the generation of GNSS data in a standard format ready to be streamed over the network in real time. For instance, the software receiver can act as a Ntrip Source feeding a Ntrip Server, just as a “professional” receiver does (thus allowing the deployment of a GNSS reference station).

3.4.4 System control

The GNSS receiver is composed of several processes that runs simultaneously and must be both synchronized and supervised from a common control plane. The control plane is implemented in the CPU and it is composed of the following software components:

- Operating system
- Receiver configuration
- Startup scripts / status monitoring

The main goal is to configure and launch the GNSS-SDR receiver process as fast as possible and keep the system running within the desired parameters. If some anomalous situation is detected (i.e. iBOGART network disconnect or hardware accelerator problem / hardware glitch) the control plane will trigger a corrective action (i.e. restart the communications or restart the receiver signal processing). Next sections are devoted to give details of each of the system control layers.

3.4.4.1 Operating system

In contrast to Personal Computers (PC), embedded devices usually require custom versions of firmware to setup and configure all the hardware components. In classical PC systems, these tasks are implemented by the mainboard manufacturer in the Basic Input Output System (BIOS) firmware, and the new hardware additions, like expansion cards, are based in standards already implemented in BIOS.

However, the selected GNSS-SDR embedded platform is based on a hybrid ARM/FPGA processor that requires a custom implementation of the whole boot process. Typically when booting an embedded processing system, there are a series of low-level device-specific operations that are executed to bring the system to a basic operating state. Tasks to be performed may include processor register initialization, memory initialization, peripheral detection and verification, and cache activation. At this stage there are very few resources available, and hence the bootstrap or first-stage loader and its associated software must be as compact as possible.

A second stage loader is generally required to load an operating system. This is beyond the capabilities of the bootstrap loader, so as its final task it loads and passes control to a larger and more capable second stage program. For the purposes of running Linux on an embedded system, U-boot provides all the features needed to act as the second stage.

U-boot is the de-facto loader for embedded Linux because it has been adopted by almost every distribution (i.e. it is natively supported by Debian and Ubuntu). It runs on a wide range of processors and has been ported to innumerable boards. Online information is readily available and there are many support forums to aid developers in porting to new architectures. U-boot includes a command line interface and many configuration options, including the ability to pass parameters to the Linux kernel to alter how the boot will proceed. It can load the kernel, the root file system (RFS) and the **device tree** and perform validation on the installation before passing control to the kernel for execution.

The **Device Tree** is a file which contains a data structure describing the hardware system. The information is used by Linux during the kernel boot process to map device parameters such as device type, memory location and interrupt signals. Although we have already initialized and booted the ARM processor cores into a standalone running state, Linux has no knowledge of this and by default assumes that it must perform all initialization on its own. To do this it needs to set up virtual memory, print to the console, and locate all of the installed hardware in the system and load software drivers.

These operations are carried out by writing to registers, but the Linux kernel needs a method to discover the parameters associated with the current hardware system. In a fixed system this could be handled with static header files or kernel configuration at build time, but for many dynamic devices it is much more desirable to obtain the configuration information at run time. This is further complicated by the endless customization available on an FPGA, which would quickly result in an unmanageable number of possible kernel header and configuration combinations.

On a PC, the device information is supplied by the BIOS, but the ARM processors don't have anything comparable. So the chosen solution is a device tree, also referred to as Open Firmware (abbreviated OF) or Flattened Device Tree (FDT). This is a text file which contains all the hardware information about the system, such as device addresses, interrupt vectors and bus addresses, compiled into byte code format. U-boot copies the compiled data into a known address in the RAM before jumping to the kernel's entry point.

The last but not least task is the **Linux kernel customization** to support the custom peripherals required to access to the PL hardware accelerators (i.e. including the Xilinx AXI DMA driver). It usually requires rebuilding a custom the kernel from its sources. Both the main processor and the integrated peripherals manufacturers provide a public source code repository with the kernel customizations for its developer boards.

Summarising, for the ZedBoard hardware developer platform we have identified the following stable configurations and firmware versions:

Table 3.16: ZedBoard configurations and firmware versions

	Config #1 (Analog Devices)	**Config #2 (AD PL + Xilinx kernel)**	**Mixed #1 (AD PL + Xilinx Vivado auto Device Tree)**
PL FIRMWARE	https://github.com/analogdevicesinc/hdl.git GIT TAG: origin/hdl_2014_r2 Migrated to Vivado 2014.4 and compiled with 2014.4 (no problems found)	https://github.com/analogdevicesinc/hdl.git GIT TAG: origin/hdl_2014_r2 Migrated to Vivado 2014.4 and compiled with 2014.4 (no problems found)	https://github.com/analogdevicesinc/hdl.git GIT TAG: origin/hdl_2014_r2 Migrated to Vivado 2014.4 and compiled with 2014.4 (no problems found)
PS COMPILER	CROSS_COMPILE=arm-xilinx-linux-gnueabi- source /opt/Xilinx/Vivado/2015.2/settings64.sh	CROSS_COMPILE=arm-xilinx-linux-gnueabi- source /opt/Xilinx/Vivado/2015.2/settings64.sh	CROSS_COMPILE=arm-xilinx-linux-gnueabi- source /opt/Xilinx/Vivado/2015.2/settings64.sh
DEVICETREE	Included in AD Kernel: /arch/arm/boot/dts/zynq-zed.dtsi /arch/arm/boot/dts/zynq.dtsi make ARCH=arm zynq-zed-adv7511.dtb cd arch/arm/boot/dts mv zynq-zed-adv7511.dtb devicetree.dtb	Included in AD Kernel: /arch/arm/boot/dts/zynq-zed.dtsi /arch/arm/boot/dts/zynq.dtsi make ARCH=arm zynq-zed-adv7511.dtb cd arch/arm/boot/dts mv zynq-zed-adv7511.dtb devicetree.dtb	Vivado 2014.4 SDK generated devicetree from new example project: http://www.wiki.xilinx.com/Build+Device+Tree+Blob git://github.com/Xilinx/device-tree-xlnx.git

U-BOOT	https://github.com/Xilinx/u-boot-xlnx.git (master) (modified include/configs/zynq_zed.h to launch root fs from sdcard)	https://github.com/Xilinx/u-boot-xlnx.git (master) (modified include/configs/zynq_zed.h to launch root fs from sdcard)	https://github.com/Xilinx/u-boot-xlnx.git (master) (modified include/configs/zynq_zed.h to launch root fs from sdcard)
KERNEL	https://github.com/analogdevicesinc/linux.git GIT BRANCH: remotes/origin/xcomm_zynq_new_pcore_regmap	https://github.com/Xilinx/linux-xlnx.git git checkout tags/xilinx-v2015.1	AD Kernel
ROOT SYSTEM	UBUNTU 14.10 (utopic) ARMHF on EXT4 SDCARD	UBUNTU 14.10 (utopic) ARMHF on EXT4 SDCARD	UBUNTU 14.10 (utopic) ARMHF on EXT4 SDCARD
Results	ALL GOOD. Features: • ETH • USB 2.0 OTG • HDMI Display • AXI CDMA	ALL GOOD. Features: • ETH • USB 2.0 OTG • HDMI Display • AXI CDMA	ALL GOOD. Features: • ETH • USB 2.0 OTG • HDMI Display • AXI CDMA

3.4.4.2 **Receiver configuration**

All the receiver operating parameters are stored in the GNSS-SDR configuration file (see Section XXX_the_control_plane for more details on the available configuration options). Depending on the user requirements and on the embedded hardware platform resources, such as the available satellite channels and the desired GNSS observables output rate, different configuration files could be selected by the startup script.

3.4.4.3 **Startup scripts and status monitoring**

Once the Linux kernel is fully loaded, the receiver startup scripts are in charge of:

- Initializing the OS communications: USB/CANBUS, NMEA, Ethernet/3G network
- Configure and initialize the front-end
- Launch GNSS-SDR
- Launch RTKLIB/PVT solver

In Debian-based systems, the so-called SystemV, *init* is the program which spawns all other processes. It runs as a daemon (a process that runs silently in background) and typically has Process ID (PID) 1. It is the parent of all processes. Its primary role is to create processes from a script stored in the file /etc/inittab file. The Runlevels in SystemV describe certain states (i.e. Runlevel 0 is halt state, 1 is Single user mode, and 6 is a rebooting state)

All System V init scripts are stored in /etc/rc.d/init.d/ or /etc/init.d directory. These scripts are used to control the system startup and shutdown.

The mandatory init script commands are:

- Start
- Stop
- Restart

The GNSS receiver platform will implement custom SystemV init scripts to automate the receiver startup process. The user interface can trigger a partial or a fully receiver restart just using a command line instruction from a SSH shell terminal session.

In addition to the start and stop process, SystemV also provides service monitoring functionality. If the service is irresponsive or has suffered a crash, it is possible to configure an automatic restart. All the GNSS receiver restart events are recorded in a log file for further analysis.

4. iBOGART Central Processing Facility (iBOGART Cloud)

The Central Processing Facility will be located in the iBOGART Cloud, which will be implemented at UPC premises (see D2.1). The CPF will offer a passive service, computing and distributing WARTK corrections through the Internet to any potential AUDITOR users (mainly targeting South Europe) mainly based on L1 and L2 (or L2C) frequencies. For this purpose, a unidirectional link from the CPF is needed to distribute the corrections to any interested AUDITOR user.

The design of the Central Processing Facility is explained below in two main subsections: (1) the CPF real time implementation, including the SW modules and WARTK messages definition, and (2) the CPF messages dissemination, including how RTCM and NTRIP could be useful and also Bandwidth considerations.

4.1 CPF real time implementation

The software sets the required real-time environment to execute the WARTK CPF in true real-time conditions. For such a purpose, the software retrieves real-time GPS observables from different NTRIP data streams directly obtained using BKG's BNC open source software (https://igs.bkg.bund.de/ntrip/download). Then, OB1 messages are generated in real-time continuously with the input data streams flow thanks to Linux pipe processes, it updates precise predicted orbit sets every 6h from IGS, generates real-time plots and provides additional features.

The whole package consists of several main C-shell scripts (hereinafter CSH), some secondary C and FORTRAN programs and many AWK scripts, and requires some extra open software to be installed. Specifically it needs the following OpenSource applications: BNC, wget, gawk, gnuplot, GraphicsMagick command-line utilities, and Randomize Lines utility, among others. It has been designed to be 100% portable within different Linux machines, and it has been named WARTK-RT. In this way, we foresee that the integration in the Zynq Processing System will be feasible with few adjustments.

4.1.1 Software structure

WARTK-RT software components are allocated in different folders depending on their purpose. The main folders are the following ones:

- <wartk-rt root>/bin/: This directory contains all WARTK-RT related programs, binary code, source code and scripts of any nature.
- <wartk-rt root>/dat/: It contains static files, which are needed to run the software, such as antenna phase center definition file, a priori permanent station coordinates, GPS constellation status, leap seconds information.
- <wartk-rt root≥/input parameters/: All configurable files are inside this directory.

- <wartk-rt root>/datasets/: This folder contains several databases automatically or time-to-time created by WARTK-RT software, containing files downloaded from the Internet. It contains the SINEX, SP3, DCBs between C1 and P1 and troposphere information, among others.
- <wartk-rt storage folder>/run/: Real time growing files are inside this folder. It mainly allocates real-time inputs, real-time outputs, and real-time log files. It contains three main folders, one named starting_up_bkg-ds to initialize, maintain and provide in the right format the GNSS observations gathered by BNC (NTRIP datastreams), the bkg_2_prefits to compute, in the same seamless stream, the prefit residuals after (and prefits_2_krigingVTEC, to derive real time ionospheric VTEC maps, not necessary in principle in this project, and allocated in the corresponding folder prefits_and_krigingVTEC_2_WARTK_Messages).
- <wartk-rt root>/templates/: This directory contains gnuplot templates, which are necessary to build the real time plots, and namelists on multiple parameters the CSH scripts require.

WARTK-RT software is structured in different CSH scripts which run different processes and subscripts in foreground and background mode, coordinately, in order to be able to run the WARTK-CPF FORTRAN core.

Once WARTK-RT is launched, real-time data flows in different directions, from different scripts and programs to others harmonically, and finally feeds WARTK-CPF core. An output message is continuously generated, and real-time plots are built. Figure 4.1 shows a diagram of the stable situation achieved when WARTK-RT software is running.

4.1.1.1 WARTK-RT CPF main components description

WARTK CPF can run either in real time or for a predefined interval of past days. As a first step, one must decide which station will be treated as reference station, which set of permanent stations should fix their phase range ambiguities (these must be within a certain range from the reference station), which permanent stations must be used in the computations (helping to the ionospheric model mainly) but will not fix their ambiguities, and finally which stations should be treated as rovers in certain critical aspects, as an autonomous CPF monitoring system to detect potential real-time failures.

The software starts generating a grid of points defining the ionospheric voxel centers, and storing it into a file (c.pri). Then it continues gathering data for the previously selected set of stations. It downloads their observables in RINEX format (in the case of real time streams, they are gathered by means of BKG's BNC), and also extracts their precise coordinates from an automatically downloaded SINEX file. The next step consists on downloading the corresponding ultrarapid predicted IGS orbit datasets. Then, it generates the WARTK CPF input messages[2]. With all these data, WARTK CPF can start the computing process, which should last from several hours in cold start before the starting of the service (to guarantee the convergence of the main system parameters) to many weeks in a seamless mode if needed.

[2] Known and named as messages1.input file or stream.

When the observations corresponding to a given time tag (for instance each 30 seconds) are all of them read in the datastream chain, the WARTK CPF processing generates new output messages[3].

WARTK CPF core

The main WARTK CPF core is implemented considering the following two FORTRAN programs (see also Figure 4.1):

- messages1input_2_prefits_v*.f
- prefits_2_messages1output_v*.f

These programs are the ones dealing with the computational load, and the ones that will also be used (and slightly modified) for AUDITOR real-time implementation. In brief, they are described below.

On the one hand, messages1input_2_prefits script implements the required steps to compute GNSS prefit residuals. For such purpose, it is necessary to acquire and preprocess input data (such as GNSS observations, precise ephemeris and clocks), computing combinations of pseudorange/phase observables (such as the ionospheric-free, the geometry-free and the wide-lane combinations), and detect cycle slips, among other aspects. In order to derive the prefit residuals it is necessary to model to the required extent possible multiple terms affecting GNSS signals, including tides, wind up, orbits and clocks (interpolated to the time of reception), relativity corrections, Delay Code Biases, troposphere wet and dry delays, antenna phase center offsets, etc. It is also necessary to compute transmission and reception times and interpolate satellite clocks to the time of reception, among other features. These steps are carried out in real time through piped processes and the data are converted to the main output messages "PR2", containing the prefit residuals for all the CPF receiver considered measurements, and additional internal messages (like the one called TI, or TI-CLASSIC) to easy the following steps in the processing chain.

On the other hand, prefits_2_messages1output script implements the required steps to run TOMION ionospheric model and compute the WARTK messages. In this regard, it is necessary to run a Kalman filter (taking into account a process noise for each term to be modelled) to solve the navigation equations and derive the post-fit residuals as well as the partial TEC values for each ionospheric voxel in a two-shell tomographic modelling. The script is also responsible to apply double differences and derive double differenced postfit residuals, fixing or constraining WARTK-required ambiguities (Bc, Bi, Bwc), checking wrong ambiguity fixes and looking for small undetected cycle-slips within the messages1input_2_prefits script. It also computes the Dilution Of Precision (DOP) and can estimate ionospheric effective height (important for AUDITOR WP4). Within this script, we keep track of the reference receiver (s and rover-like receivers -if any; for testing purposes-). Furthermore, the STEC and dSTEC adjustment per satellite (together with the corresponding covariance matrix) are computed to be transmitted to the users as part of the WARTK DSM messages.

It is also remarkable that this last script allows parallel running considering a number of CPUs.

[3] A file named messages1.output.

Last but not least, in both cases there are multiple tests to verify the correct execution of the scripts. In addition, the scripts allow carrying out tests considering independent stations for validation purposes.

As a final remark, in AUDITOR it is not expected to initially modify these scripts very significantly. The effort would be more focused at the user side, as described in Section 3.4.3.

4.1.1.2 **WARKT-RT Inputs**

The WARKT-RT software is fed by three types of different inputs depending on their required refresh rate: static inputs, semi-static inputs, and streamed and dynamic inputs.

Static Inputs

These inputs are defined during WARTK-RT software initialization and then remain constant during all the execution time. To launch the WARTK CPF in true real-time mode we need to define a set of options including the service area delimited in longitude and latitude (to select permanent GNSS stations in the area and potential reference stations), the types of receivers and their Antenna Phase Centers (APCs), among other information.

- <wartk-rt root>/input_parameters/rover_like.sta: This file must contain a list of stations. If available, these stations will be treated as rover-like users, specifically to test the main ionospheric corrections, and overall WARTK performance. thus WARTK-CPF will not use their observations to compute the corrections. Using independent rover-like stations is very useful to quickly check WARTK-CPF performance.
- <wartk-rt root>/input_parameters/wartk-rt_var-setup: Setup file which contains all the configuration variables used by all WARTK-RT packages scripts.

Once WARTK-RT software has been initialized, it creates all required CPF core static files in the run directory. These are the ones showed in Figure 4.1, some of them are: ant_info.abs.unix, c.pri, reference_receiver.sta, fixing_ambiguities.sta, rover_like.sta and to_estimate_coordinates.sta.

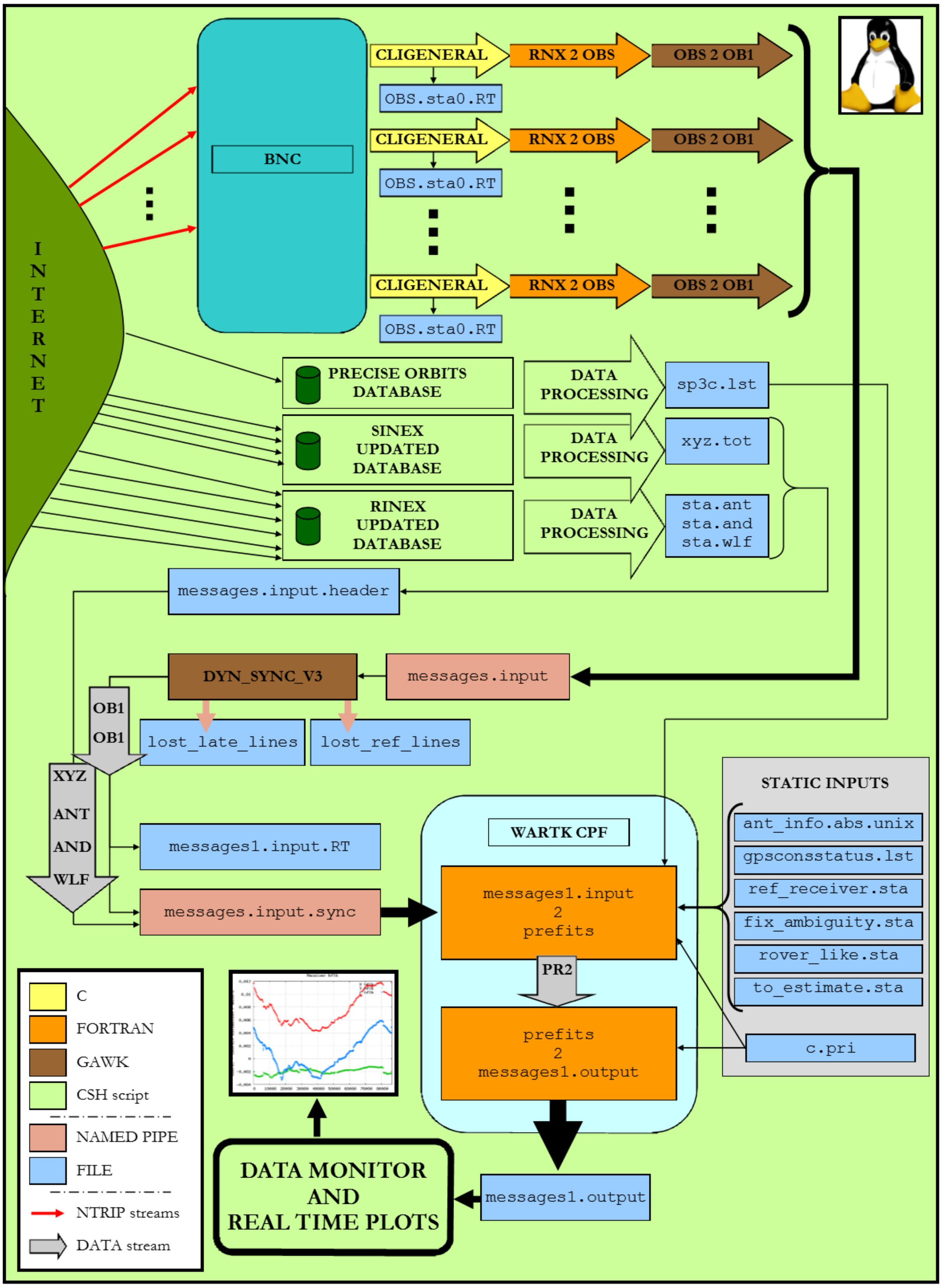


Figure 4.1: WARTK-RT software data flow diagram

4.1.1.3 **Semi-static Inputs**

As WARTK-RT software is working in real-time conditions, the majority of the data inputs will be dynamically updated. However there are a set of inputs which require low update rates, these are the so-called semi-static inputs. The software user does not have to worry about these inputs because they are downloaded and updated automatically by the WARTK-RT software.

Within this group, we can distinguish between three types of sources: the GPS constellation status file, the SINEX files, and the RINEX files.

- **GPS constellation status file** is updated every time WARKT-RT software is relaunched. It contains the PRN-to-SV dictionary and it is downloaded from the University of New Brunswick servers4 but it is in process of being updated to be gathered from the file containing the antenna phase center information, in ANTEX format. This file is adapted to WARTK core programs and stored in the run directory with the name: gpsconsstatus.lst
- **SINEX files** contain information about GPS permanent station precise coordinates, in case it has not been computed recently by the same WARTK-RT software. These files are weekly generated by different providers. WARTK-RT software builds a local database with all downloaded SINEX files. The present version uses a pair of providers: CDDIS5 and BKG's GNSS Data Center6, and it downloads one file (codXXXX7.snx.Z) from CDDIS, and three files (eurXXXXmr.snx.Z, nkgXXXX7.snx.Z and olgXXXX7.snx.Z) from BKG servers. Then, station precise coordinates are extracted and stored in sta_pos.tot file. With this information, the next step is to build the WARTK messages for the necessary stations.
- **RINEX files** are known as a standard way to enclose GPS observable information. In addition, the RINEX headers have useful information that WARTK-CPF core requires. And at least one updated RINEX file is needed for each used station. WARTK-RT software also builds a RINEX database with all these files. It seeks necessary RINEX files mainly from the BKG NTRIP casters in case of real time mode of operation (and CDDIS and GARNER GNSS servers in case of post-process mode).

WARTK-RT tries to download the newest versions of RINEX files, extracting their headers and generating three WARTK-CPF input messages: ANT, AND and WLF. They include antenna descriptions, antenna phase centers, and receiver wavelength factors for each GPS permanent station.

All these inputs are called semi-static because their update status is not very critical. It is not crucial to use, for example, a five weeks old SINEX file because the precise coordinates of the stations within the file are taken as a priori guess (their value are refined, with an a priori variance of few centimetres, in the same CPF computation chain). All these files are updated once WARTK-RT

[4] http://gge.unb.ca/Resources/GPSConstellationStatus.txt

[5] Crustal Dynamics Data Information System, http://cddisa.gsfc.nasa.gov

[6] Bundesamt für Kartographie und Geodäsie, http://igs.bkg.bund.de

software is relaunched, and they stay typically static the rest of the time, but they might be "injected" on-the-fly at any time if needed.

4.1.1.4 **Streamed and Dynamic Inputs**

These are the true real-time inputs. If these inputs are not properly updated, the WARTK-CPF core might stop; so these are the most important inputs and should be checked continuously. This includes: (1) the new precise orbit sets must be transferred to the CPF every six hours, and (2) the real-time low latency observables that can be gathered from BKG's NTRIP (at 1 Hz rate though downdated[7] afterwards according to the user needs and machine CPU8 capabilities).

- **Ultrarapid Predicted Precise Orbits** files are continuously generated by the IGS community, up to 24 hours in advance, and stored in IGU files (named iguXXXXD YY.sp3.Z, where XXXX stands for the GPS week, D for the day of the week (from 0 to 6) and YY for the issued hour). They are downloaded mainly from CDDIS servers, but if not available they can be automatically downloaded from alternative sites, such as GARNER. WARTK-RT software will automatically download the latest file version four times every day. For instance at: 3:10h, 9:10h, 15:10h and 21:10h (UTC time). Then, the file format is slightly modified and a sp3c.lst file is created, so that the WARTK core knows that a new set of orbits is ready to be read.
- **Real-Time Observables.** In this context, observables (L1, L2, P1, P2, C/A) of the selected GPS dual frequency receivers are gathered in real-time conditions with 1 Hz sampling rate from the BKG's BNC. In normal conditions, WARTK-CPF can be connected up to more than 100 GNSS receivers. Information coming from these streams must be downdated up to one epoch each 30s to assure WARTK-CPF performance is achieved. Nonetheless, higher time rates could be assessed. Furthermore, observations coming from different streams must be ordered and synchronized in sequential epochs. For this purpose an AWK script was designed. It is also important to say that these real-time streams are continuously checked by WARTK-RT software, a real-time global map is generated every 6 minutes showing which streams (locations) are working and which are down (see Figure 4.1).

4.1.1.5 **WARTK-RT Outputs**

WARTK-CPF core generates uninterruptedly one file named messages1.output which concentrates all CPF output messages. It contains many different messages, including the ionospheric model, the tropospheric estimation, the satellite clocks, and the rover-like navigation status, among others.

This file is examined automatically every 15 minutes, and three sets of plots are depicted. A first plot shows the total ambiguity fixing performance, a second set of plots shows the performance of the double difference correction of STECs for each satellite, DDSTEC21, and a third set of plots draws X, Y and Z coordinates accuracy of each of the rover-like stations used for real-time screening of the CPF performance.

[7] Instead of sending observables every second, they can be sent every 30 second in order to avoid overloading the machine.

[8] Central Processing Unit, main processor of a machine.

Furthermore, the real-time GPS observables streams are checked every six minutes, a stream-log file is generated, stopped streams are detected, and the whole behaviour is plotted on a map (see example in Figure 4.1).

At the end of the day, all relevant data, logs, input and output messages, plots and maps are automatically stored in the run folder, sorted by day.

4.1.1.6 **WARTK correction message requirements**

This section briefly describes the current WARTK user messages, which may evolve during the project to meet specific requirements on precision agriculture. For instance, it is currently being assessed the suitability of only transmitting messages on prefits mode in order to save bandwidth.

4.1.1.6.1 "OB1" message: measurements of the nearest reference station

The rover user should receive the observations OB1 message at a typical rate of 1 Hz, referring to the reference station to form the double differences. The OB1 message is obtained easily after a simple processing of the GNSS station observables.

Reference station observations (such as the pseudoranges P1 or C/A and P2, as well as the carrier phases L1 and L2, in the case of GPS) can be obtained directly from a real-time datastream server through the Internet, in RINEX[9] or RTCM3 format. Then, these observations are processed and converted to the non-standardized "OB1" message format (an internal format for WARTK processing). It is remarkable that this format is very useful because it contains information on the real-time occurrence of cycle slips [10].

Table 4.1 shows all fields that an OB1 message must contain in the current implementation.

Table 4.1: "OB1" message field description

Field	Parameter	Meaning	Format	Examples
1	type=OB1	The following parameters describe the measurements of one reference receiver (i.e. the nearest), including cycle-slip detection.	3 Characters. A3, 1X	OB1 (the only acceptable string)
2	PRN	Satellite number	Integer, I2, 1X	1, 4, 15, 21
3	year	Year of the measurement	Integer. I2.2, 1X	99, 00, 05, 06
4	month	Month of the measurement	Integer. I2, 1X	1, 4, 10, 12
5	day	Day of the measurement	Integer. I2, 1X	4, 19, 28, 31
6	hour	Hour of the measurement	Integer. I2, 1X	5, 11, 19, 23

[9] Receiver INdependent EXchange, GNSS standard for raw measures.

[10] Cycle slip: A discontinuity of an integer number of cycles in the measured (integrated) carrier phase resulting from a sudden loss-of-lock in the carrier tracking loop of a GNSS receiver.

7	minute	Minute of the measurement	Integer. I2, 1X	2, 23, 56, 59
8	second	Second of the measurement	Float. F11.7, 1X	0.0000000, 8.0000000, 43.0000000
9	refrec	Reference receiver name	String. A4, 1X	lliv, ebre, plan
10	ksat	Kind of satellite.	String. A1, 1X	G (= GPS), R (=GLONASS). Etc
11	epochflag	Power failure between previous and current epoch or event flag.	Integer (from 0 to 6). I1	0, 1, 5, 6
12	recclock	Receiver clock estimate (if available)	Float. F12.9	0.000000000
13	nobs	Number of available observables.	Integer. I6	0, 1, 4, 6
...				
13 + i	kobs(i)	Kind of the i-th observable.	2 Characters. 4X, A2	L1, P2, C1, D1
13+i+1	kobs(i+1)	Kind of the i-th+1 observable.	2 Characters. 4X, A2	L1, P2, C1, D1
...				
14+2*nobs	narch1	Number of continuous carrier-phase arch	Integer, i5	1,2,...,19,...
15+2*nobs	Cycleslip	Flag indicating whether or not there is one cycle-slip.	Logical, l1	.F., .T.

Table 4.2: "OB1" message example

```
OB1  11  08  1  30  11  46  30.0000000  ebre  G  0  0.000000000  4 L1 L2 P2 C1
124026977.63649 96644386.00647 23601544.0234 23601547.6914 17 F
```

4.1.1.6.2 "S3C" message: ultrarapid (predicted) IGS orbits

Accurate predicted orbits and clocks information can be downloaded from IGS servers. IGS issues IGU[11] files four times every day, with a delay of 3 hours. Every file has information on the satellite positions and clocks of the previous 24 hours and of the next 24 hours, sampled every 15 minutes. These files are published in IGS' SP3 format.

S3C message is obtained after a simple processing of the IGU files. This processing adds an orbit flag that informs whether an IGU satellite orbit set can be accepted by the user or not. This is because sometimes IGU predicted information is badly computed. In these cases, the user must use the common GPS broadcasted orbit parameters for the affected satellites (still useful in double-difference precise positioning).

[11] Code that identifies IGS Ultrarapid orbit files

Table 4.3 shows all fields that an S3C message must contain in the current implementation.

Table 4.3: "S3C" message field description

Field	Parameter	Meaning	Format	Examples
1	type=S3C	The following parameters describe the ultrarapid precise IGS orbits	3 Characters. A3	S3C (the only acceptable string).
2	PRN	Satellite number	Integer. 1X, I2	1, 4, 15, 21
3	year	Year of the measurement	Integer. 1X, I2.2	99, 00, 05, 06
4	month	Month of the measurement	Integer. 1X, I2	1, 4, 10, 12
5	day	Day of the measurement	Integer. 1X, I2	4, 19, 28, 31
6	hour	Hour of the measurement	Integer. 1X, I2	5, 11, 19, 23
7	minute	Minute of the measurement	Integer. 1X, I2	2, 23, 56, 59
8	second	Second of the measurement	Float. 1X, F11.8	0.00000000
9	x	Satellite's x-axis coordinate	Float. 1X, F13.6	-24730.078897, -21724.190244
10	y	Satellite's y-axis coordinate	Float. 1X, F13.6	6340.852653, -8418.584775
11	z	Satellite's z-axis coordinate	Float. 1X, F13.6	-7248.582843, -12629.254112
12	dt	Satellite's clock error	Float. 1X, F13.6	389.055848157, 32.218068
13	orbflag	Orbital information flag	Integer. 1X, I2 (possible values: 0/1)	0, 1

Table 4.4: "S3C" message example

```
S3C 1 08 1 29 6 0 0.00000000 16418.552828
-19056.394552  8632.355341  181.568916  0
```

4.1.1.6.3 "DSM" message: Total Electron Content adjustment performed independently for each satellite

WARTK CPF generates a wide variety of messages. One of them contains information about ionospheric status. This ionospheric information is compacted in a slant ionospheric model per satellite, and stored in DSMs messages. This is typically slightly better in Wide Area scenarios than broadcasting 3D ionospheric grid values, with an implicit mitigation of the mapping function errors, for distances up to several hundreds of kilometres, among other advantages.

These messages are indispensable for the rover user. With them the user WARTK receiver can build its own STEC values through interpolation, and get a prompt precise guess of the ambiguities of both carrier phases. DSM messages can be broadcasted each minute (usually enough) or at higher rates if needed.

Table 4.5 shows all fields that a DSM message must contain in the current implementation.

Table 4.5: "DSM" message field description

Field	Parameter	Meaning	Format	Examples
1	type=DSM	The following parameters describe the final adjustment of the total electron content, independently performed per each satellite	3 Characters. A3	DSM (the only acceptable string)
2	iprn1	Satellite number	Integer. 1X, I2	1, 4, 15, 21
3	iyy0	Year of the measurement	Integer. 1X, I2.2	99, 00, 05, 06
4	imo0	Month of the measurement	Integer. 1X, I2	1, 4, 10, 12
5	ida0	Day of the measurement	Integer. 1X, I2	4, 19, 28, 31
6	iho0	Hour of the measurement	Integer. 1X, I2	5, 11, 19, 23
7	imi0	Minute of the measurement	Integer. 1X, I2	2, 23, 56, 59
8	sec0	Second of the measurement	Float. F11.7	17.0000000, 49.0000000
9	ndstec	Number of receivers involved in the adjustment	Integer. 1X, I3	3, 5, 9, 10
10	rms postfit	STEC Postfit RMS	Float. 1X, F9.4	0.00189, 0.00708, 0.00438
11	bias postfit	STEC Postfit Bias	Float. 1X, F9.4	0.00189, 0.00708, 0.00438
12	chi2	STEC Postfit Chi squared.	Float. 1X, E14.6	0.326456D+00, 0.152913D+01
13	nunk dstec	Number of unknowns involved in the adjustment	Integer I3	2, 4
…	…	…	…	…
12 + 2*i	x dstec(i)	I-th adjustment parameter	Float. 1X, F10.5	-0.02916
13 + 2*i	St dev dstec(i)	Standard deviation of i-th ad-justment parameter	Float. 1X, F12.7	0.666E-01, 0.451E-01.

12+2*(i+1)	x dstec(i+1)	I-th + 1 adjustment parameter	Float. 1X, F10.5	0.11884, 0.00560
13+2*(i+1)	st dev dstec(i+1)	Standard deviation of i-th + 1 adjustment parameter	Float. 1X, F12.7	0.838E-01, 0.517E-01.
…	…	…	…	…
14+2*nunk dstec	refrec	Reference receiver character id.	4 characters. X, a	brus
15+2*nunk dstec	irefrec	Reference receiver integer id.	Integer, 1x,i3	1,5,23,...
16+2*nunk dstec	ra refrec	Right Ascension of the IPP of the ref. receiver observation for the given satellite.	Float. F12.6	119.637337, 130.848918
17+2*nunk dstec	dec refrec	Latitude of the IPP of the reference receiver observation for the	Float. F12.6	29.751108, 43.952362, 32.882576
18+2*nunk dstec	stec refrec	Slant Total Electron Content of the reference receiver.	Float. 1X, F10.4	2.5540, 0.8460, 2.6961.
19+2*nunk dstec	sigma stec refrec	Standard deviation (formal error) corresponding to the Slant Total Electron Content Reference.	Float. 1X, F10.5	0.09772, 1.62279, 0.76316.
20+2*nunk dstec	vtec1 refrec	Vertical Total Electron Content reference receiver	Float. 1X, F10.5	2.5540, 0.8460, 2.6961.
21+2*nunk dstec	iprnref	Reference satellite PRN	Integer, 1X,I2	12,23,7,...
22+2*nunk dstec	sele0	Elevation over spherical horizon	Float. F8.3	11.196, 38.154, 81.151.
23+2*nunk dstec	adjusting tec instead of dtec	Whether adjusting zero-differenced TEC instead of the single difference referred to the reference receiver observation (per each satellite, and performed independently).	Logical, 1X, L1	.T.
24+2*nunk dstec	adjusting dvtec instead of	Whether adjusting zero-differenced TEC instead of the single difference referred	Logical, 1X, L1	.F.

	dstec	to the reference receiver observation (per each satellite, and performed independently).		
25+2*nunk dstec	iyear0	4-digit year	Integer. 1X,I4	2016
26+2*nunk dstec	Idoy0	3-digit day of year	Integer. 1X,I3	286
27+2*nunk dstec	tsecday0	5-digit second of day	Float. 1X,F8.2	43200.00
28+2*nunk dstec	tdjmobs0	GPS Time in Modified Julian days	Float. 1X,F12.6	54020.055556

Table 4.6: "DSM" message example

```
DSM 8 08 1 29 23 57 45.0000000 16 0.0298 -0.0004 0.158224E+02  6
          0.1454827E+01  0.3548E-02  -0.2400633E-01
          0.1402E-02 -0.6604203E-01 0.9057E-03 0.2740417E-02
          0.5688E-03 -0.8131125E-02 0.2512E-03 0.9116094E-02
          0.5684E-03 karl 1 148.523000 50.541000 1.4398 0.0063
          0.7318  26  22.530  T  F  2008  29  86265.00  54494.998437
```

4.1.1.7 **WARTK CPF messages dissemination: FTP/RTCM**

Regarding WARTK messages dissemination, it must be remarked that presently there is not a well-defined RTCM standard capable to enclose all information that WARTK CPF needs to provide to the rovers. Therefore, the possibility to send the messages in textual ASCII format via datastream or FTP is considered necessary and taken as the baseline. In this way, we make sure they are not contaminated by any means due to any encapsulation limit on accuracy. Nonetheless, RTCM 3.2 standard is expected to soon cover such needs thanks to the so-called State Space Representation (SSR) new set of messages to move towards RTK-PPP (see deliverable D1.2) allowing the distribution of precise orbits, precise clock bias (first stage of SSR messages already completed and documented in the current RTCM 3.2 specifications), a VTEC ionospheric model in spherical harmonics (second stage, message 1264; not documented in the specifications) and, in the near future, STEC messages (third stage; not supported). In this way, it will be assessed the use of RTCM to the extent possible.

It is worth mentioning that RTCM's VTEC SSR messages can already be decoded by using BKG's BNC. The following information is directly extracted from BNC's help manual and shall be considered confidential.

Table 4.7: Example for block 'VTEC' carrying ionospheric corrections

```
> VTEC 2015 06 17 11 43 35.0 6 1 CLK93
 1  6  6   450000.0
  17.6800    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
   4.5200    8.8700    0.0000    0.0000    0.0000    0.0000    0.0000
 -4.6850  -0.3050   1.1700   0.0000   0.0000   0.0000   0.0000
 -2.2250  -1.3900  -1.0250  -0.1300   0.0000   0.0000   0.0000
  0.8750  -0.3800   0.2700  -0.1300   0.0400   0.0000   0.0000
  1.2150   0.9050  -1.0100   0.3700  -0.1450  -0.2450   0.0000
 -0.8200   0.4850   0.2300  -0.1750   0.3400  -0.0900  -0.0400
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000  -0.0700   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   0.5800  -1.4150   0.0000   0.0000   0.0000   0.0000
  0.0000  -0.6200  -0.1500   0.2600   0.0000   0.0000   0.0000
  0.0000   0.0700  -0.0900  -0.0550   0.1700   0.0000   0.0000
  0.0000   0.5000   0.3050  -0.5700  -0.5250  -0.2750   0.0000
  0.0000   0.0850  -0.4700   0.0600   0.0700   0.1600   0.0400
```

The second record in this block provides four parameters:
  Layer number
  Maximum degree of spherical harmonics
  Maximum order of spherical harmonics
  Height of ionospheric layer [m]

Subsequent records in this block provide the following information:
  Spherical harmonic coefficients C and S, sorted by degree and order (0 to maximum)

In this context, note that the possibility to use VTEC SSR message may imply a limited accuracy for the dissemination of WARTK messages, due to the spherical harmonic implementation and since the number of decimals would be limited by the format.

Regarding RTKLIB's RTCM support, there are routines that allow RTCM encoding (rtcm3e.c) and decoding (rtcm3.c). Then, in case transmitting WARTK messages in RTCM-like format, it would be needed to modify such routines so that the messages are correctly supported by RTKLIB.

5. Conclusion

In this document internal specification for the different hardware/software subsystems that are part of the AUDITOR GNSS receiver have been detailed. Several physical and logical interfaces have been also described between the different elements described. This deliverable provided an initial overview of the architecture and multiple subsections describing the flow of the GNSS data from the antennas to the final post processed positioning data.

The custom RF front-end (FE) details have been presented its main electrical components, configuration parameters, physical interfaces and data acquisition parameters.

The processing platform has been detailed divided into its processing logic (PL or FPGA) and processing system (PS or ARM). The PL receiver the front-end data samples and provides low level real time processes to feed data to the PS subsystem through the AXI bus.

The processing logic embed a set of low level processes which implement adapters, buffers, correlators, hardware accelerators and signal tracking. These configurable elements transform the raw GNSS data samples into a continuous pre-processed GNSS data stream.

The processing system runs a Linux distribution which supports the core of the AUDITOR GNSS algorithms. One of the key libraries that will be included in the PS is the open-source GNSS-SDR which provided a complex GNSS data computer with several configurable abstraction layers. The core elements of this library and its relation to the processing of the AUDITOR GNSS data have been included as well as its integration with a PVT solver based on the RTKlib.

Finally the iBOGART cloud platform has been referenced by describing its main messages types and formats that need to be used via a remote TCP/IP connection to obtain the ionospheric data corrections.

6. References

[1] AUDITOR-D2.1 Architecture definition.

[2] C. Fernández-Prades, J. Arribas y P. Closas, «Accelerating GNSS Software Receivers,» de *Proc. of the 29th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+)*, Portland, OR, 2016.

[3] Xilinx, «Zynq-7000 All Programmable SoC Overview - Product Specification,» San José, CA, 2016.

[4] Digilent, «Digilent Pmod Interface Specification,» Pullman, WA, 2011.

[5] J. J. J. S. O. C. Hernández-Pajares M., «Application of ionospheric tomography to real-time GPS carrier-phase ambiguities resolution, at scales of 400-1000 km and with high geomagnetic activity,» *Geophysical Research Letters,* vol. 27(13), pp. 2009-2012, 2000.

[6] ARM, «AMBA AXI and ACE Protocol Specification,» 2011.

[7] AUDITOR-D1.1 State of the Art.

[8] AUDITOR-D1.2 Requirement definition.

[9] AUDITOR-D1.3 Test definition.

[10] GNSS-SDR project: http://gnss-sdr.org/.