AUDITOR – GA 687367

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

D3.1 Version 1.0

GNSS receiver design and implementation

Contractual Date of Delivery:	M16 (Apr, 2017)
Actual Date of Delivery:	**28.04.2017**
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:	*WP3 – GNSS receiver module*
Security:	**CO**
Nature:	**R**
Version:	1.0
Total number of pages:	59

Abstract:

This document contains the GNSS receiver design and implementation that is based on the on the previous “D2.1 Architecture definition” and “D2.2 Subsystem specification”. The GNSS receiver is composed mainly of two hardware elements a custom multiband/multisystem RF Front-End and a commercial ARM/FPGA processing platform. The configurable RF front-end and the integration of multiple custom software/firmware components in the ARM/FPGA platform provide a flexible and high performance open GNSS receiver implementation. The RF front end supports Galileo/GPS bands (E1/L1, L2 o E5a/L5).

Document Control

Version	Details of Change	Author	Approved	Date
0.1	First version of the document	JD		03/04/2017
0.2	Updated contents and TOC reorganization	JD		12/04/2017
0.3	Updated with design issues	JD		21/04/2017
0.4	Merged ACORDE/CTTC contributions added conclusions	JD		25/04/2017
0.5	Some minor corrections and reordering	CF		28/04/2017
1.0	Consolidated Version		JD	28/04/2017

Executive Summary

This document summarizes the GNSS receiver design already proposed in D2.1 Architecture Definition and extended in D2.2 Subsystem specification.

The GNSS receiver is composed of two hardware elements:

- A custom RF front-end designed and implemented within AUDITOR
- A signal processing platform based on the Zynq-7000 All Programmable SoC that includes a FPGA/ARM configuration with custom accelerators and the gnss-sdr-org modules.

The RF front-end provides two simultaneous receiver chains for Galileo/GPS bands. The first receiver chain is fixed to the E1/L1 band. The second receiver chain can be configured either in the E5a/L5 band or the L2. The RF front-end provides the sampled I/Q signals for these bands to the processing platform.

The processing platforms embed custom accelerators, implemented within WP3, that provide a set of high performance and efficient pre-processing functions. Those accelerators are embedded in the FPGA and contain multiple correlators and FIFO modules that allows the real-time processing of multiple GNSS signals. Moreover the ARM contains the receiver monitoring and control modules and the high level GNSS related functions that are based on the open source project GNSS-SDR.org. This open source project is maintained by CTTC and enables Linux distributions to implement a GNSS software receiver.

In this document the design and implementation of both components is detailed as well as the main internal and external monitoring and control interfaces. The GNSS receiver implements an innovative ionospheric model that is supported by data streams from cloud computing platforms both developed within WP4/WP5.

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
EMI	ElectroMagnetic Interference
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
IF	Intermediate Frequency
IOP	Input / Output Peripherals
IP	Intellectual Property

Term	Description
IRQ	Interrupt ReQuest
LPDDR	Low Power Double Data Rate
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 a previous deliverable [1] the overall architecture for the AUDITOR system has been presented. This architecture was further refine in [2]. The architecture introduced [1] and [2] is summarized in Figure 1.1.

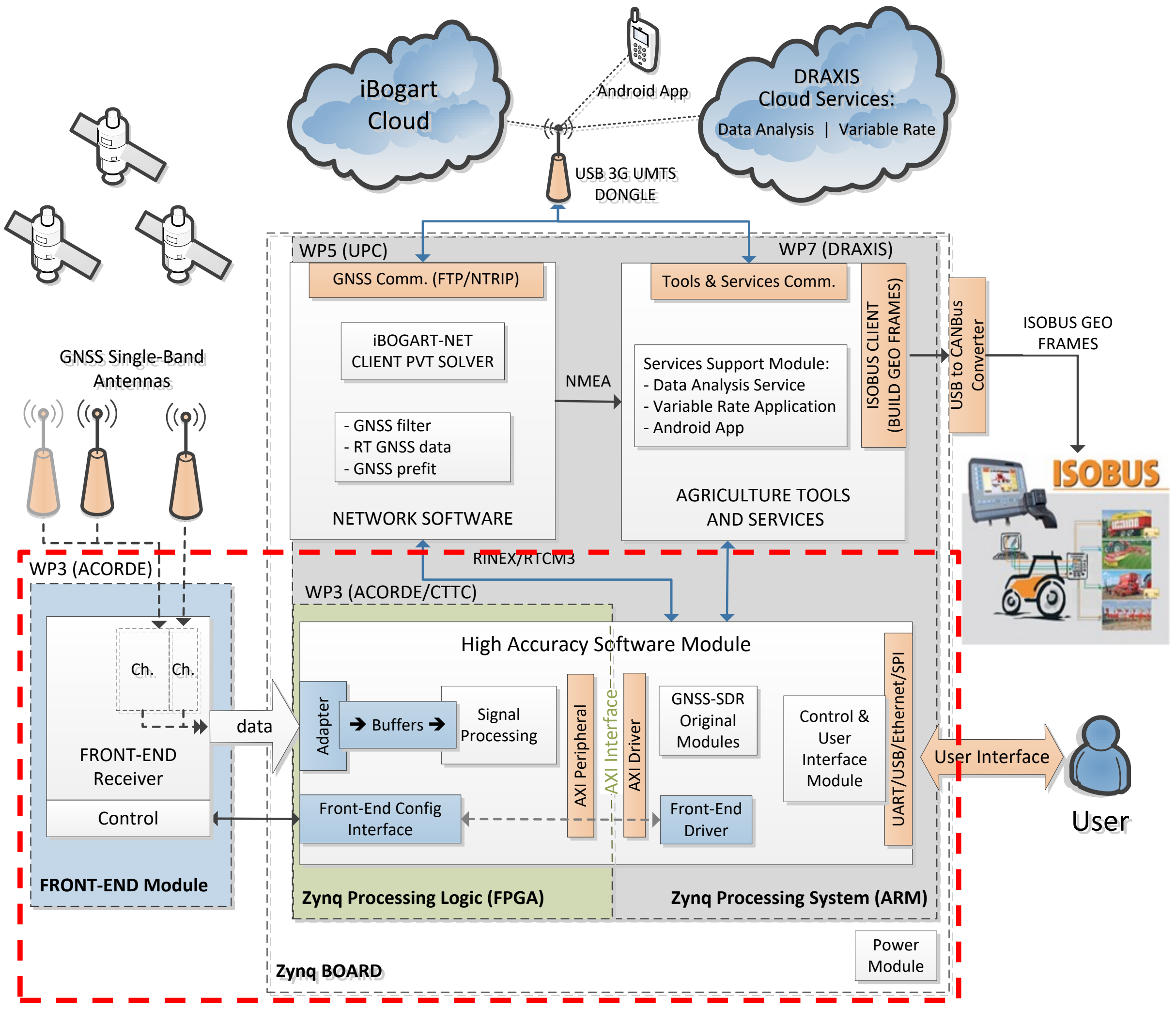


Figure 1.1: System Architecture

This document is focused in the work performed by ACORDE FRONT-END Module (see Figure 1.1 left), CTTC High Accuracy Software Module (see Figure 1.1 bottom-right). The work carried out by UPC (Network Software) in collaboration with TUM that involves the iBOGART Cloud related to the innovative ionospheric model and the generation/processing of its data streams is detailed in the WP5/WP4 deliverables [3] and [4], also submitted in M16 .

The GNSS receiver is composed of two hardware elements depicted in Figure 1.1:

- **RF front-end module**
- **Digital processing platform**

This document in the following sections details the design and implementation of both elements including their custom hardware and main software/firmware modules.

2. GNSS RF Front-End Module (ACORDE)

The RF FE down converts the **GALILEO and GPS bands L1/E1, L2 or L5/E5a to a low intermediate frequency and provides the digitalized I/Q stream**. It is composed of two independent RF channels. The first channel receives the E1/L1 band and down converts it to IF in a single step. The configurable channel, either for the L2 or E5a/L5 bands, includes a similar schema but adding a twostep downconverter. The fixed and one of the configurable bands can operate simultaneously. The configurable band can only operate L2 or E5a/L5 as these bands share parts of the configurable receiver RF chain.

One critical element of the RF FE is the **clock generation and distribution**. All the clock signals need to be phase-synchronized; therefore they are generated from a single low noise reference oscillator. This reference signal is pre-scaled and distributed to the mixers/downconverters and ADC in order to extend this synchronization to the digital generation of the I/Q samples.

The RF FE embeds also its monitoring and control logic that can be interfaced via a standard serial port and allows enable/disable RF elements and configuring the down conversion and acquisition parameters of both RF chains.

Two versions of the RF front-end (FE) were designed, v1.0 and v2.0. The specifications used as reference for the design were defined in collaboration with other partners in [2] Section 2. The first version (v1.0) is focused on the validation of the RF design. This preliminary functional version allows an early assessment within the digital platform. The second version (v2.0) rationale is to improve the RF performance mitigating the effects of electromagnetic interference (EMI) and the redesign of the clock distribution to optimize the synchronization characteristic. This second version was also resized taking into account the form factor of the proposed high performance digital platform module in [2], the Zynq ZC706 Evaluation Kit shown in Figure 2.1, which provides extension with external modules via FMC connectors.

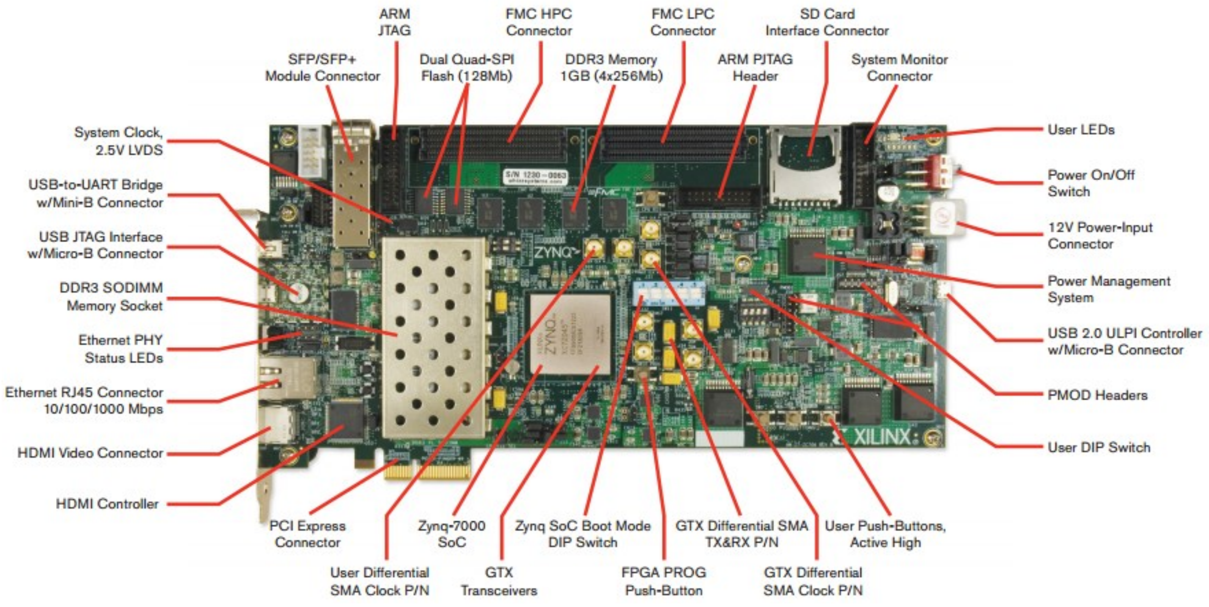

ARM JTAG
FMC HPC Connector
FMC LPC Connector
SD Card Interface Connector
SFP/SFP+ Module Connector
Dual Quad-SPI Flash (128Mb)
DDR3 Memory 1GB (4x256Mb)
ARM PJTAG Header
System Monitor Connector
System Clock, 2.5V LVDS
USB-to-UART Bridge w/Mini-B Connector
USB JTAG Interface w/Micro-B Connector
DDR3 SODIMM Memory Socket
Ethernet PHY Status LEDs
Ethernet RJ45 Connector 10/100/1000 Mbps
HDMI Video Connector
HDMI Controller
User LEDs
Power On/Off Switch
12V Power-Input Connector
Power Management System
USB 2.0 ULPI Controller w/Micro-B Connector
PMOD Headers
User DIP Switch
PCI Express Connector
Zynq-7000 SoC
Zynq SoC Boot Mode DIP Switch
GTX Differential SMA TX&RX P/N
User Push-Buttons, Active High
User Differential SMA Clock P/N
GTX Transceivers
FPGA PROG Push-Button
GTX Differential SMA Clock P/N

Figure 2.1: ZC706 Evaluation Kit top layer layout

The main parameters of the **FE specification** were defined in [2] and have been included here in Table 2.1 for convenience. An **additional column "Implementation Comments"** to extend the initial specification with the implementation results of both v1.0 and v2 has been added.

Table 2.1: Front End Specification

Channel			***Implementation Comments***
Channel 1 (fixed)	L1/E1	***yes***	*Yes, in FE v1.0 and v2.0*
Channel 2 (configurable)	L2C	***yes***	*Yes, in FE v1.0 and v2.0*
	L5/E5a	***yes***	
Frequency bandwidth			***Implementation Comments***
Sampling frequency (in MHz)[1]	L1/E1	***~4 MHz + 2xIF***	*Configurable IF filter 26MHz/5=5.2MHz*
	L2C	***~4 MHz + 2xIF***	*Sampled at:* - *26MHz/5=5.2MHz* - *26MHz/4=6.5MHz*
	L5/E5a	***~25 MHz + 2xIF***	*Sampled at 26/1=26MHz*
	Option a: L1/E1 and L2C	***~4 MHz + 2xIF***	*Sampled at 26MHz/5=5.2MHz*
	Option b: L1/E1 and L5/E5a	***~25 MHz + 2xIF***	*Sampled at 26/1=26MHz*
Bits per sample			***Implementation Comments***
Each channel will generate 8 bits I + 8 bits Q in 2's complement.			*Yes, in FE v1.0 and v2.0*
The adapter module in the Zynq will be in charge of reading these inputs and converting them to baseband.			*Tested in v1.0, implemented now by CTTC.*
Intermediate Frequency			***Implementation Comments***
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***	*L1 IF 418kHz*
	L2C	***Up to 1 MHz***	*L2 IF 609kHz*
	L5/E5a	***Up to 1 MHz***	*L5 IF 404kHz*
AGC			***Implementation Comments***
• 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).			*Configurable amplifier stages via serial port 0dB-69dB*
Reference oscillator			***Implementation Comments***
Accuracy	**<= 0.5ppm**		*<0.2ppm*
External oscillator option	**YES (SMA female connector)**		*Added in RF FE v2.0*
Antenna input			***Implementation Comments***
Connector type	**SMA female**		*Yes, in FE v1.0 and v2.0*
Impedance	**50 ohms**		*Yes, in FE v1.0 and v2.0*
On-board DC Bias-T (for active GNSS antenna)	**YES (5v DC output is required to power the GNSS antenna LNA)**		*DC power for external active antennas reduced to 3.3V*
Signals Summary			***Implementation Comments***
CH1 DATA In-phase component (real part)		8	*Yes, in FE v1.0 and v2.0*
CH1 DATA Quadrature component		8	*Yes, in FE v1.0 and v2.0*

[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.

(imaginary part)		
CH2 DATA In-phase component (real part)	8	*Yes, in FE v1.0 and v2.0*
CH2 DATA Quadrature component (imaginary part)	8	*Yes, in FE v1.0 and v2.0*
Sample CLOCK	1	*Yes, in FE v1.0 and v2.0*
VCC	1	*Yes, in FE v1.0 and v2.0*
GND	1	*Yes, in FE v1.0 and v2.0*

As it can be seen in the previous Table 2.2, all the specification have been achieved or improved and more configurable options are included in terms of IF selection or sampling bandwidth.

In terms of the **electrical specification** for both designed FE their key parameters are summarized in Table 2.2.

Table 2.2: Electrical specifications

Parameter	Conditions	Min.	Typ.	Max.	Units
POWER SUPPLY					
V_{CC}	Supply voltage operation		5 (v1.0) 12 (v2.0)		V
I_{CC}	Current consumption		500		mA
DIGITAL INTERFACE					
Digital Logic-High	All pins		2.3		V
Digital Logic-Low	All pins		1		V
OUTPUT CLOCK					
$CLK\_OUT_{FREQUENCY}$	System clock frequency	4.25	6.5	26	MHz
RF INTERFACE					
Z_{IN}	Input impedance		50		Ω
NF	Front-End Noise Figure			3	dB
V_{ANT}	Active antenna supply voltage		3.3		V
I_{ANT}	Antenna supply current			20	mA

The main difference between both FE v1.0 and FE v2.0 in terms of electrical characteristic is the increase of the supply voltage to 12V. This voltage supply level is the standard value for Zynq boards (MicroZed, ZedBoard…) and is also commonly one of the main power supplies in ground vehicles.

2.1 RF Front-End v1.0

The design and implementation of the first prototype was based on ACORDE previous experience with earlier E1/E6 front-end designs, as the one presented in [5] and shown here in Figure 2.2.

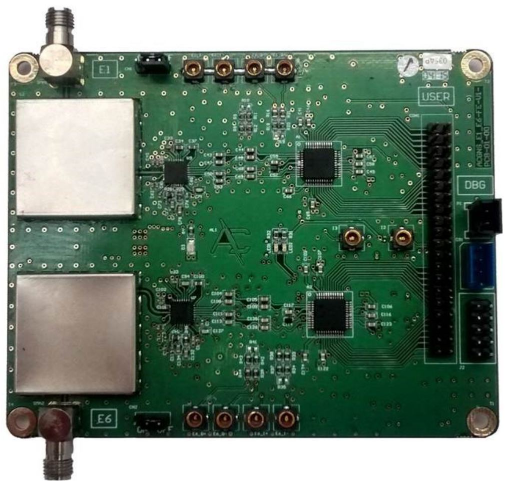


Figure 2.2: ACORDE's previous E1/E6 Front-End v2 (size 10 x 8 cm2)

The more challenging elements in AUDITOR design are the integration of multiple bands and the correct synchronization of the digital sampling clock and RF reference oscillator:

- <u>Reference clock:</u>
  - Distribution required either square or sin signals depending on the component RF/digital nature.
  - Voltage must be carefully distributed to feed modules with different voltage levels and/or DC terms.
  - Squared digital signals introduced several harmonics that could potentially interfere with other digital/RF modules.
- <u>Configurable Downconverter:</u>
  - Full input dynamic power need to be provided in order to work in optimal conversion conditions.
  - Must minimize the mix with unwanted internal or external frequencies.
  - Inductions and capacitance effects need to be minimized on all inputs/outputs to reduce the noise and possible spurious frequencies.
  - Shall provide enough configuration range to cover both L2 and E5a/L5 bands.
- <u>Fixed Downconverter:</u>
  - The optimal input power need to be provided also taking into account its internal configuration options that allows setting different gain factors.
  - Exact configuration of its internal parameters is needed but datasheet information is not full documented. This requires to perform additional standalone measurements to assess its performance with different internal configurations.
  - AGC module needs to be built externally to the downconverter to take into account all the external LNAs and provide the optimal dynamic range for both downconverters and finally the output ADC.

2.1.1 Design

The initial design and specification identified in [2], was refined to the design shown in Figure 2.3.

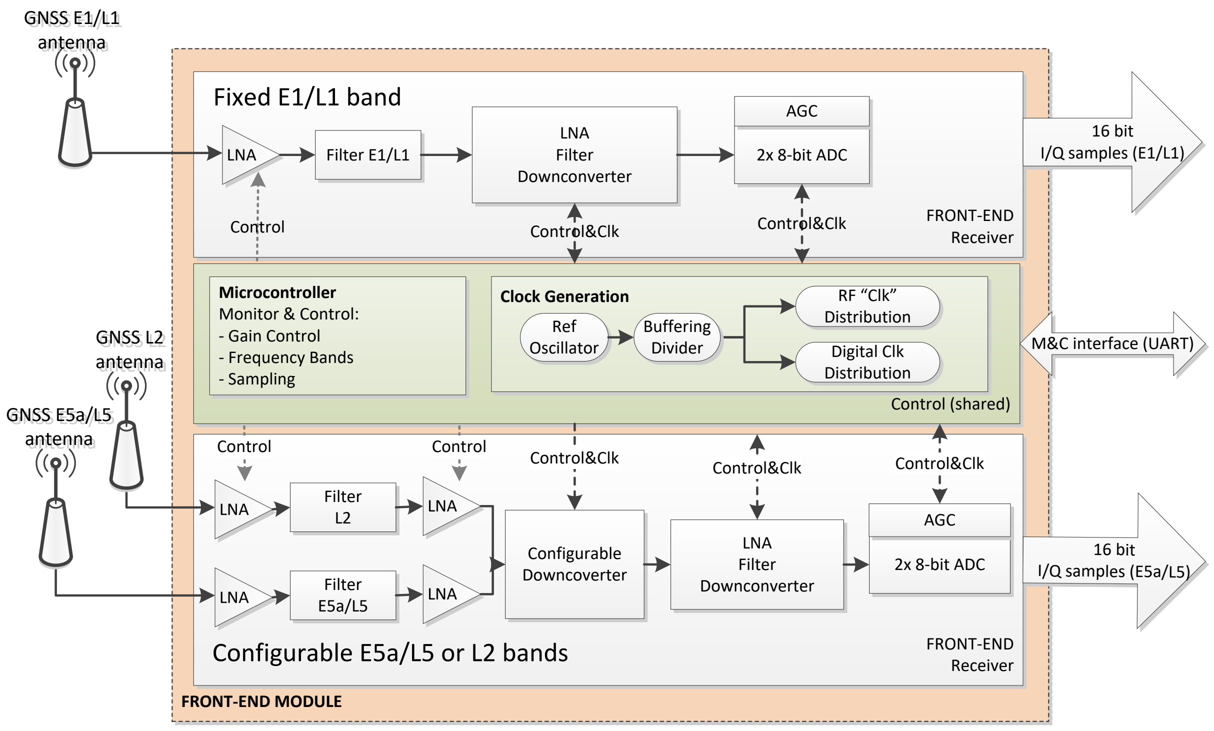


Figure 2.3: RF FE v1.0 design

This design follows closely the specification of [2] but introduces two different RF chains for the L2 and E5a/L5 bands that **shared the same digital control & monitor logic**. An important part of the design is the **clock generation** module that from a single reference crystal creates all the reference oscillators and digital clocks signals to distribute to the downconverters and ADCs.

The **two antennas** connectors simplify the initial tests with independently single-band antennas. The next FE version proposes one multiband antenna instead that are more widely commercialized and provide up to three bands reception in a single module.

The fixed and configurable band uses a similar output dowconverter (MAX2769) and ADC stage in order to simplify the overall design and provide similar performance for all bands. In this way, the configurable band requires and additional downconversion stage to adapt, either the input L2 or the E5a/L5 band to the final dowconverter band that is centered in all cases at L1.

2.1.1.1 Interfaces

Four external interfaces are included in the design:

- Power connector: power supply.
- Serial connector: UART0 of the embed control microcontroller.
- JTAG connector: for programming and debugging the control logic.
- User connector: 20+20pin custom connector for **data out and external control**.

The serial connector provides a basic debug interface while the power connector is the main energy source of the board provided from a Zynq compatible board. The Table 2.3 summarizes the pin-out for both connectors.

Table 2.3: Serial and power connector

Serial debug connector

Pin	Description
1	GND
2	UART RX (3.3V)
3	UART TX (3.3V)
4	aux

Power connector

Pin	Description
1	+5V
2	GND
3	GND
4	+12V (do not connect)

The user connector is the main data interfaces to the digital processing platform detailed in section 3. Table 2.4 lists with different colors all the pins in the 20+20 pins USER connector that is mainly composed of I/Q data bits for both channels, the reference clock and the monitoring & control serial port.

Table 2.4: 20+20 pins USER connector

Pin	Description	Direction	Pin	Description	Direction
1	I7 (E1)	Out	2	I6 (E1)	Out
3	I5 (E1)	Out	4	I4 (E1)	Out
5	I3 (E1)	Out	6	I2 (E1)	Out
7	I1 (E1)	Out	8	I0 (E1)	Out
9	Q0 (E1)	Out	10	Q1 (E1)	Out
11	Q2 (E1)	Out	12	Q3 (E1)	Out
13	Q4 (E1)	Out	14	Q5 (E1)	Out
15	Q6 (E1)	Out	16	Q7 (E1)	Out
17	GND	N/A	18	UART RX	In
19	Do not connect	N/A	20	UART TX	Out
21	CLOCK	Out	22	NC	N/A
23	NC	N/A	24	GND	N/A
25	I7 (E6)	Out	26	I6 (E6)	Out
27	I5 (E6)	Out	28	I4 (E6)	Out
29	I3 (E6)	Out	30	I2 (E6)	Out
31	I1 (E6)	Out	32	I0 (E6)	Out
33	Q0 (E6)	Out	34	Q1 (E6)	Out
35	Q2 (E6)	Out	36	Q3 (E6)	Out
37	Q4 (E6)	Out	38	Q5 (E6)	Out
39	Q6 (E6)	Out	40	Q7 (E6)	Out

Where the I/Q samples are expressed in **2's complement**. All of these pins are 3.3v digital signals except the clock signal that is an analog sine signal.

2.1.2 Implementation

The layout for the designed FE was implemented in a four layer schema that includes two signal planes. The designed layout is shown in Figure 2.4.

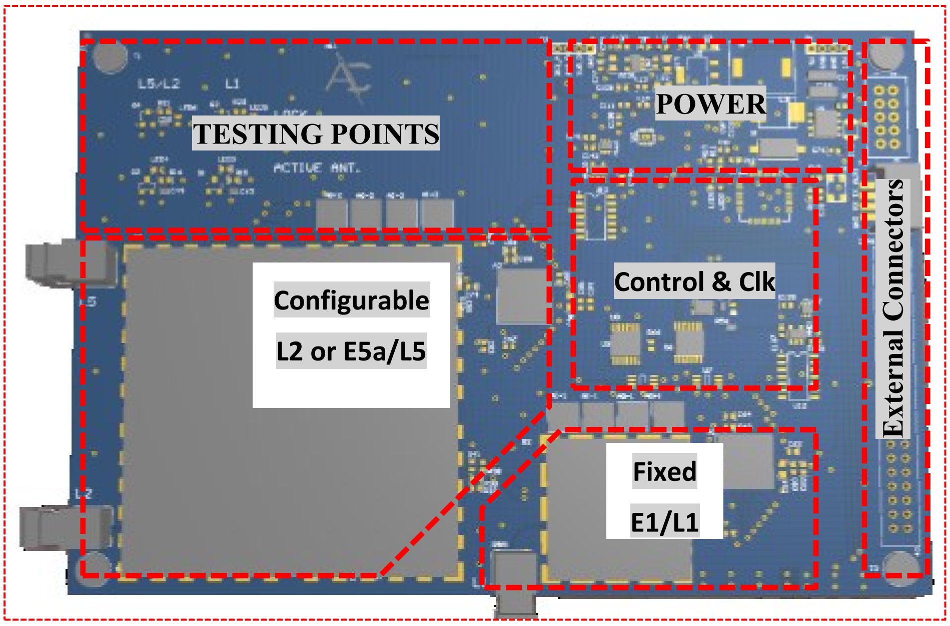


Figure 2.4: FE v1.0 top layer layout, (153mm x 104mm)

The configurable L2 or E5a/L5 RF chain can be identified on the left with its main RF parts covered by a large RF shield. Two different antenna connectors were used in order to add more flexibility, simplify the input stage and better isolated both channels as introduced in section 2.1.1.

On the lower-right area the fixed E1/L1 chain, that is simpler than the configurable chain, is shown also including its shielding and antenna connector.

The center-right of the PCB is dedicated to the clock generation and the embedded microcontroller that monitors and controls all the RF elements and provides the serial external interfaces via de User Connector.

Several testing points and led indicators are located on the upper-left to ease the testing and validation procedures.

On the upper-right several regulators define the core components that implement the power networks to provide the 5V supply voltage.

The designed layout was manufactured and the main components mounted as show in Figure 2.5.

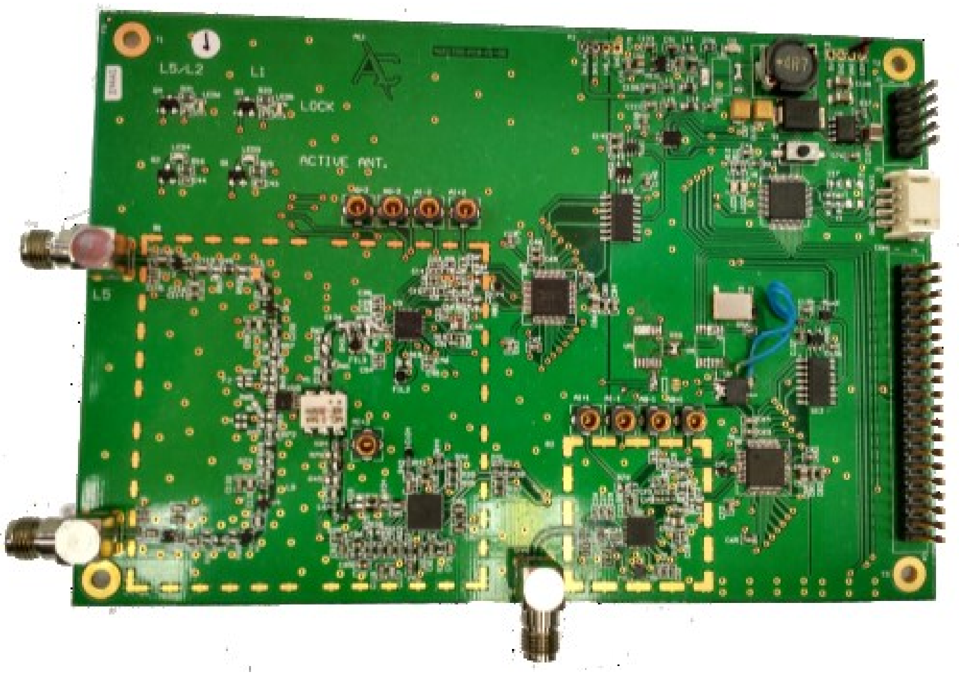


Figure 2.5: FE v1.0 PCB top layer

The main external connectors of this PCB located on the right side (excluding the antenna connectors) are as show in Figure 2.6:

- Power connector labelled as P1.
- Serial connector labelled as P3.
- JTAG connector labelled as J1.
- User connector 20x20 pint out.

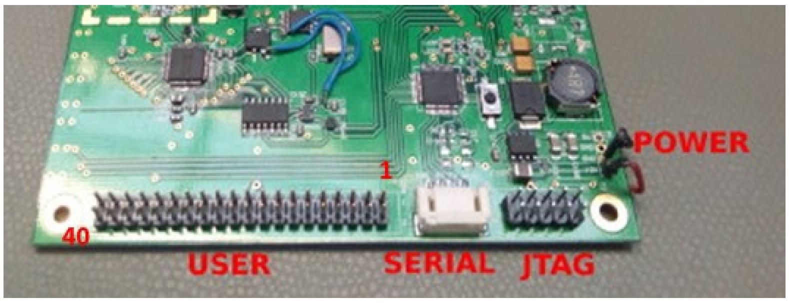


Figure 2.6: FE v1.0 external connectors

This PCB was tested in laboratory using a wide range of RF equipment, the main test performs and its results are summarized in 2.1.2.1.

Several issues were identified during the laboratory tests that required the redesign of the clock generation/distribution section and the improvement of resilience to EMI. The clock issues could be solved in the current PCB by carefully bypassing and replacing some digital components, while the EMI was reduced by additional external shields and improving the grounding of the main RF components. Theses fixes and the modification of the form factor of the PCB were the main design parameters to drive the design and implementation of the FE v2.0 detailed in section 2.2.

2.1.2.1 Tests

In this section snapshots of some measurements performed in the RF laboratory are summarized. For each measurement its **rationale, the main inputs/outputs and its results** are listed with photos of the different equipment involved.

Testing equipment setup:

- Measurement rationale: check quality of the desired band that will be sampled at **L2.**
- Input: carrier frequency at **L2.**
- Output: IF signal of **L2** carrier after downconversion.
- Summary: A clean signal can be seen in Figure 2.7 with not relevant spurious near the IF frequency.

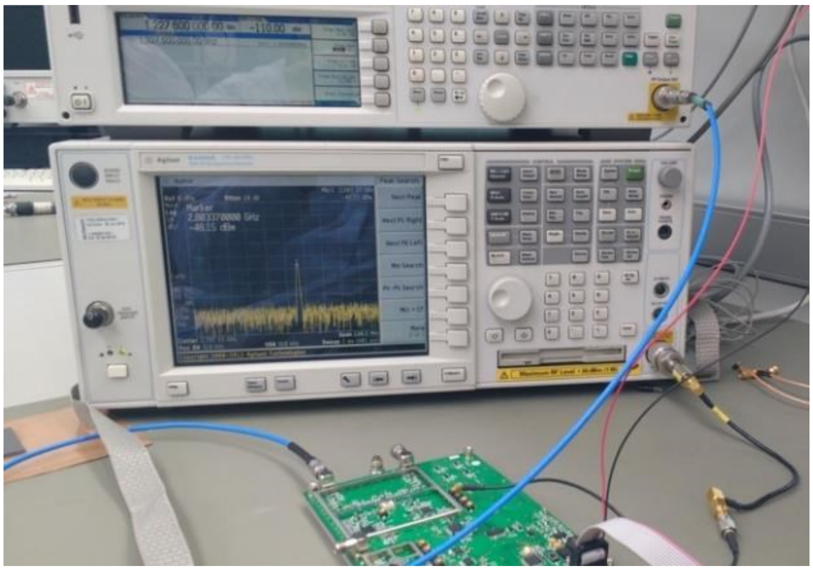


Figure 2.7: FE v1.0 measurement, testing equipment.

GPS L1 band single tone:

- Measurement rationale: check quality of the desired band that will be sampled at **L1.**
- Input: carrier frequency at **L1.**
- Output: IF signal of **L1** carrier after downconversion.
- Summary: A clean amplified signal and bandwidth curve can be seen in Figure 2.8 without any other spurious near the IF. The aim is to obtain maximum signal power without spurious and a bandwidth clean from other signals to be sampled at **L1**.

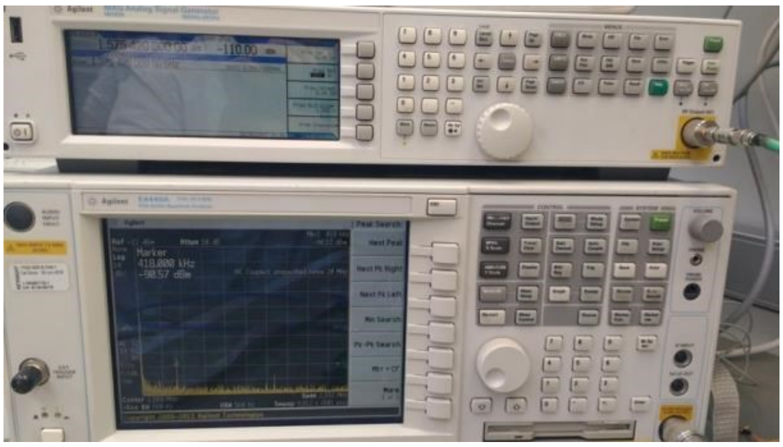


Figure 2.8: FE v1.0 measurement, GPS L1 band single tone, IF at 418 kHz (BW 2.5 MHZ).

GPS L2 band single tone:

- Measurement rationale: check quality of the desired BW that will be sampled at **L2.**
- Input: carrier frequency at **L2.**
- Output: IF signal of **L2** carrier after downconversion.
- Summary: A clean amplified signal and bandwidth curve can be appreciated in Figure 2.9 without any other spurious near the IF. The main objective of these measurements is to maximize signal power without increasing the spurious contributions at **L2.**

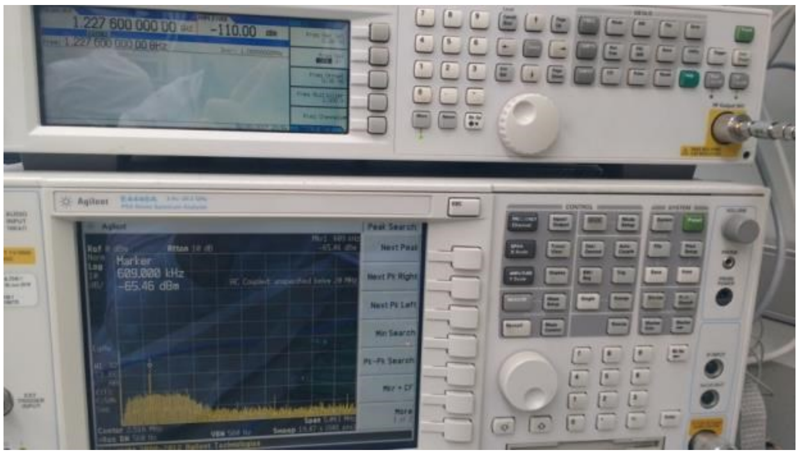


Figure 2.9: FE v1.0 measurement, GPS L2 band single tone, IF at 609 kHz (BW 2.5 MHZ).

GPS L5 band single tone:

- Measurement rationale: check quality of the desired band that will be sampled at L5.
- Input: carrier frequency at **L5.**
- Output: IF signal of **L5** carrier after downconversion.
- Summary: A clean amplified signal and bandwidth curve can be appreciated in Figure 2.10 without any other spurious near the IF. The main objective of these measurements is to maximize signal power without increasing the spurious contributions at **L5.**

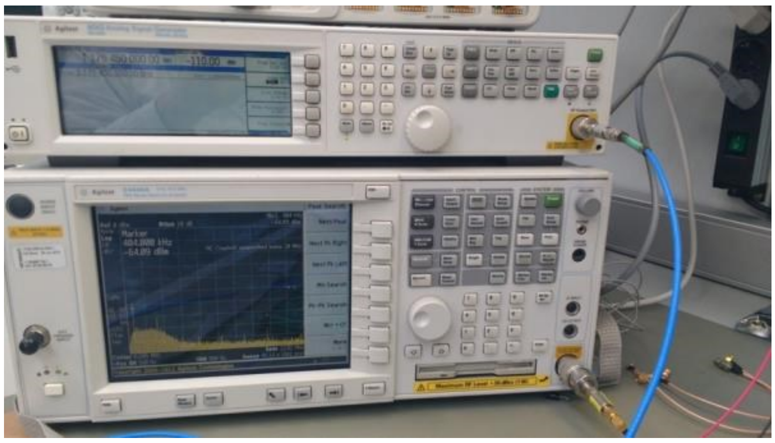


Figure 2.10: FE v1.0 measurement, GPS L5 band single tone, IF at 404 kHz. (BW 12.5 MHz).

<u>OL phase noise:</u>

- Measurement rationale: the quality of the reference oscillator is a critical parameter to the downconversion of Galileo/GPS bands in order to not to degraded the input signal.
- Input: Reference oscillator.
- Output: phase noise of this local oscillator signal.
- Summary: measure is below phase noise limits identified in [6], values shown in Figure 2.11 are:
    - -70.28 dBc/Hz @ 100Hz
    - -81.13 dBc/Hz @ 1 kHz
    - -80.78 dBc/Hz @ 10 kHz
    - -105.53 dBc/Hz @ 100 kHz
    - -118.59 dBc/Hz @ 1MHz

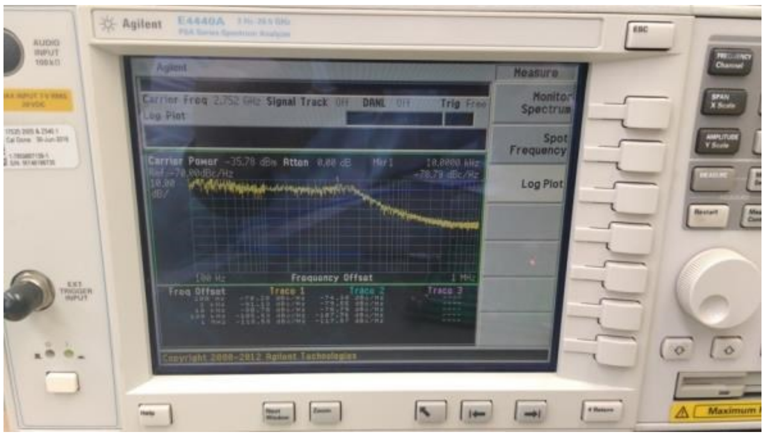


Figure 2.11: OL phase noise measurement.

2.2 RF Front-end v2.0

Taking into account the experience acquire with the RF FE v1.0 and the feedback from AUDITOR partners a new version of the FE was designed. This redesign focused on:

- The clock generation/distribution.
- Reduction of EMI between the different RF chains and digital components.
- Layout refactor and an additional FMC-LPC connector to interconnect to the ZC706 Evaluation kit.

2.2.1 Design

The new design of the RF FE does not introduce important modifications in the architecture design. The main architecture changes were the use of a single multiband antenna and reorganization of the upper section of the two configurable bands as shown in Figure 2.12.

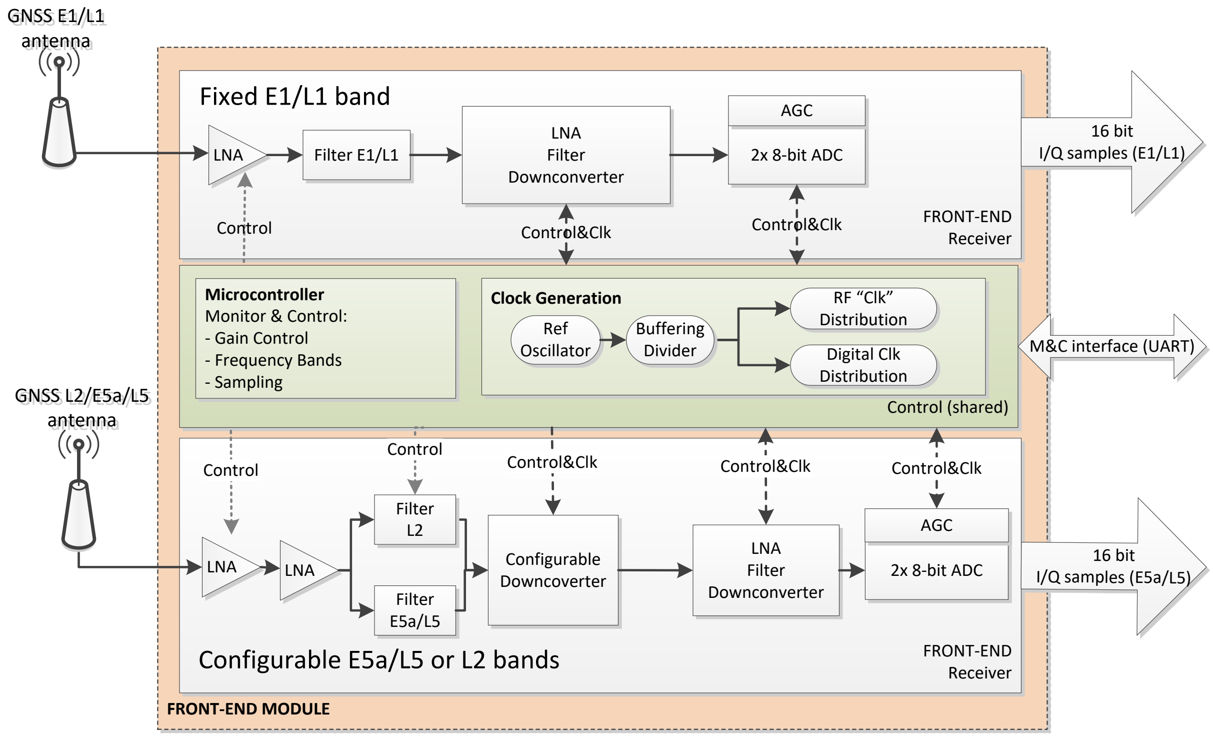


Figure 2.12: RF FE v2.0 design

Additionally, the following improvements were applied to the overall design:

- Increased filtering in all digital lines to mitigate ringing and high frequency coupling,
- Two independent clock circuits are included based on two different distribution principles to evaluate the clock quality and isolation of both options with the other components.
- Integrated a single-ended IC (MAX444) to allow in-PCB single-ended measurements without additional external elements.
- Merged in the configurable band both RF chains to simplify input stages and optimize the input power to the first downconverter.
- An additional UART and several digital I/O have been added to the existing interfaces to add more configuration and testing flexibility.
- Distributed power supplies have been integrated that offer better linearity which will directly impact in the improvement of the quality of the amplified signals for all LNAs.

In this new FE **a common two level amplifier stage now** is located close to the antenna. The filtering for the different bands is separated just before the first configurable down converter that feeds the second fixed downconverter. Using this shared L2/E5a/L5 approach optimizes the layout space while reducing power consumption and simplifies the RF design.

The RF chain gains have been optimized using two LNA to use the full dynamic range of the two downconverters and avoid saturation that would lead to an increase of the output spurious signal power.

2.2.1.1 **Interfaces**

The four external interfaces included in the v1.0 design have not been deeply modified in order to provide backward compatibility. The main changes can be summarized as:

- Power connector: changed to multiple power connectors ±12V, ±5V, 3.3V
- Serial connector: No changes in the existing one but added an additional UART1. In v1.0 UART0 was the only one available both in the serial and user connector.
- JTAG connector: No changes
- User connector (20+20pin custom connector): No changes in the user connector, added an additional FMC LPC.

The main changes involved the user connector that shares the same distribution as the v1.0. However, an additional FMC-LPC connector with similar pinout has been added in order to provide direct compatibility with the ZC706 evaluation kit FMC connector.

2.2.2 Implementation

In the FE v2.0 several modifications were applied to the layout in order to minimize EMI, clock and power distribution issues.

- The **vias** density has been increased and its layout has been optimized to improve the ground layer and the guidance of high frequency signals.
- **Power supplies** (±12V, ±5V, 3.3V) have been redesigned and its layout has been distributed in several groups close to their powered components.
- **RF layout** has been straightened as much as possible to minimize EMI corner issues, therefore RF components needed also to be redistributed to follow those new linear paths.
- **Test points** are now included later in the lab within the RF line if needed.
- **Additional filters** have been added to the layout to mitigate inducted signals for adjacent lines.
- All IC are better **decoupled** with inline protection resistors.
- **Power supply lines** now included fusible resistor in order to track possible short circuit issues and also avoid ringing and the propagation of unwanted signals thought VCC networks.

The redesigned layout is shown in Figure 2.13, as it can be easily appreciated that its composition has suffer important changes since the v1.0 show in Figure 2.4.

On the left side both RF fixed/configurable channels can be identified with their respective antenna connectors. The lower-right part of the PCB is now dedicated mainly to the digital and clock modules. The new form factor is compliance with the ZC706 evaluation kit and an additional FMC-LPC connector in the bottom layer to allow direct connection.

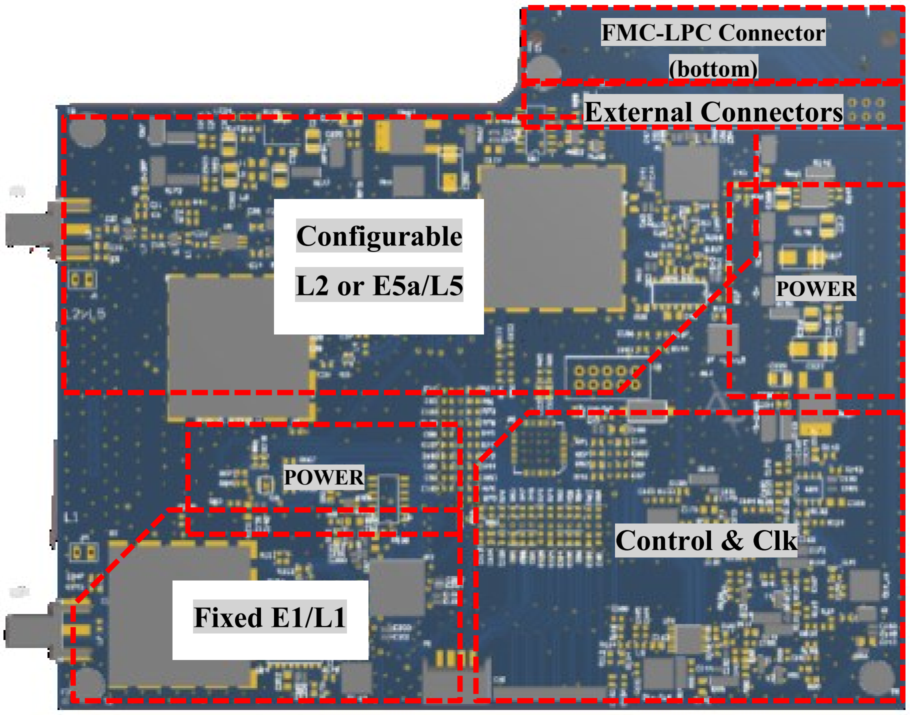


Figure 2.13: FE v2.0 top layer layout, (153mm x 108.9mm+17.9mm)

The interconnection concept for the final configuration of the FE v2.0 and the ZC706 evaluation kit board is pictured in Figure 2.13. The direct connection simplifies the tests avoiding the use of high performance FMC cables which still is possible as the FE v2.0 can be stacked below the ZC706 board.

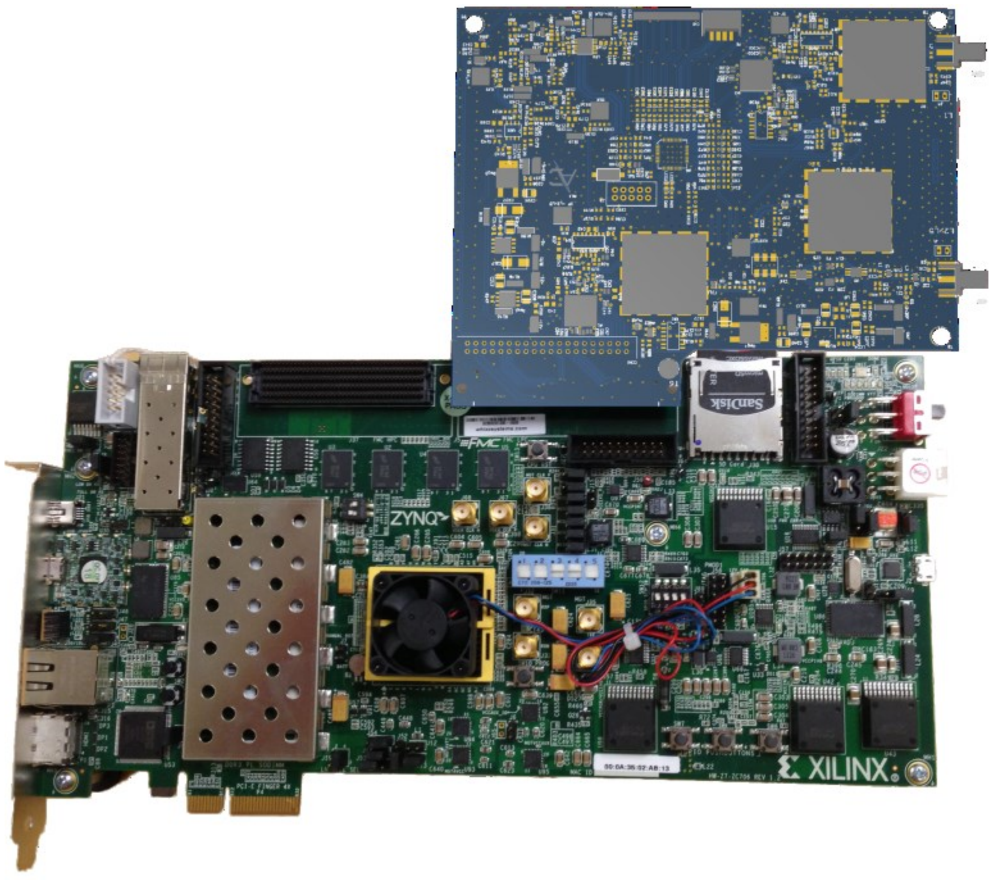


Figure 2.14: ZC706 connected via FMC-LPC to FE v2.0 layout

2.2.3 Tests

The FE v2.0 PCB is currently being manufactured. Tests performed in v1.0 will be repeated in this new version to confirm that the design and layout improvements result in noticeable performance benefits. FE v1.0 is already a functional E1/L1, L2 or E5a/L5 functional RF front-end.

2.3 Firmware

The FE is fully configurable through serial port P3 (+3.3v) that is connected to the embedded microcontroller. A custom firmware has been developed to monitor and control the main FE parameters from the digital processing platform.

A typical boot phase can be described as:

- Once the system powered (5V v1.0 or 12V v2.0), the module starts loading the default configuration where the ADCs are not enable,
- GPS L2 band is selected and a sampling clock of 6.5MHz is used,
- L1 downconverter IC is enabled too,
- All integrated circuits (IC) need to be enable/disable to set the desired configuration using the serial port.

If the user needs to change the configuration to down convert GPS L5 to IF the typical steps should be:

1. Plug the L5 antenna.
2. Set multiplexer to L5 instead of L2
3. Enable VCO integrated circuit.
4. Config VCO IC with the L5 default configuration.
5. Enable L5 max IC downconverter.
6. Enable L5 ADC.
7. Choose the sampling clock if a different frequency is needed.
8. Select different IF low pass filter if it is needed to reduce noise.
9. Change last downconverter IF stage gain if it is needed to achieve the optimal signal power.

On the other hand, if the desired band is L1, the first stage mixing is not necessary, so the steps needed to down convert this band starts at step 5 of the check list.

2.3.1 Message formats

The UART parameters to communicate with the AUDITOR front-end should be configured at:

- **9600 baud,**
- **8bits,**
- **Parity None,**
- **One stop**.

The messages sent to the FE that comply with the protocol are responded with and ACK message in the form of an ASCII ***"OK"*** or an NACK message "***NACK***". Other serial messages that are not well-formed are considered incorrected and therefore ignored. All serial port messages must end with one carrier-return followed by one end-of-line character (\r\n).

Recommended initial steps are:

1. Once the serial port is open the user can **check the correct communications** requesting "*ping message*" to the front-end and the front-end should respond an ACK message "*OK*".

2. The second recommended step is to **check the current front-end configuration** that can be obtained sending the *"???"* message to the front-end. The response should be a human readable message where the current state of all ICs is explained, followed by an *"OK"* message.
3. After these two steps the user need to **send the messages needed to achieve the configuration** to the main downconverter, as explained in section 2.3 for reference.

Following the example in section 2.3, Table 2.5 details the serial messages needed to perform the configuration.

Table 2.5: Serial messages example

Step	Command	Serial message

1	Set multiplexer to L5 instead of L2	`< cML2<CR><LF>`
		`> OK`

2	Enable VCO integrated circuit.	`< cVe<CR><LF>`
		`> OK`

3	Config VCO IC with the L5 default configuration.	`< cV5<CR><LF>`
		`> OK`

4	Enable L5 max IC downconverter.	`< cL2o1<CR><LF>`
		`> OK`
		`< cL2t1<CR><LF>`
		`> OK`

5	Enable L5 ADC.	`< cA2e<CR><LF>`
		`> OK`

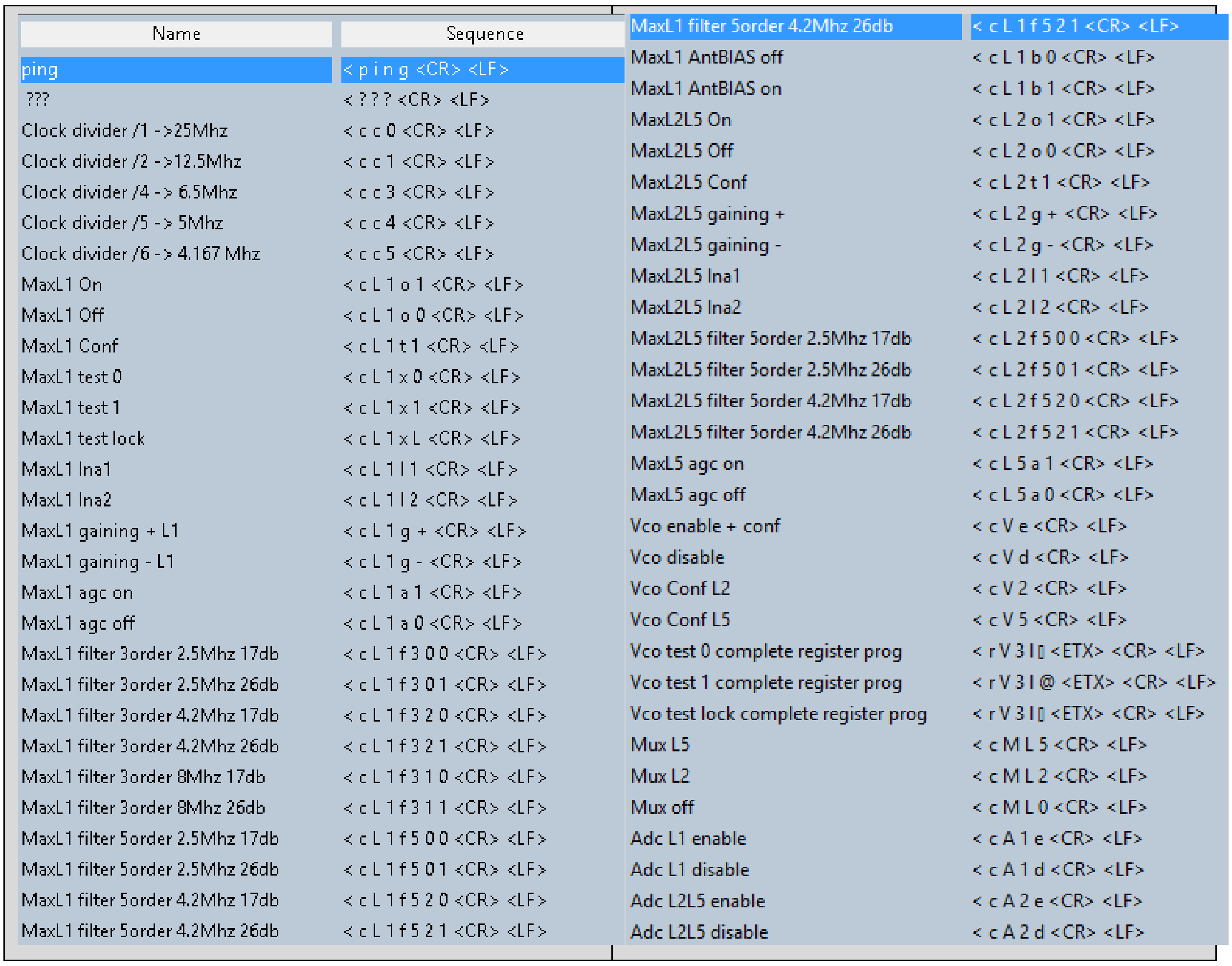
6	Choose the sampling clock if a different frequency is needed.	`< cc3<CR><LF>`
		`> OK`

7	Select different IF low pass filter (18Mhz) if it is needed.	`< cL2f531<CR><LF>`
		`> OK`

Sending the previous list of commands and applying an IF 1176.45 MHz sine signal to L5 inputs, the signal in Figure 2.10 will be shown in the spectrum analyzer.

The following Table 2.6 lists the full list of UART serial messages that can be used to configure the FE.

Table 2.6: list of serial messages.



Name	Sequence
ping	< p i n g <CR> <LF>
???	< ? ? ? <CR> <LF>
Clock divider /1 ->25Mhz	< c c 0 <CR> <LF>
Clock divider /2 ->12.5Mhz	< c c 1 <CR> <LF>
Clock divider /4 -> 6.5Mhz	< c c 3 <CR> <LF>
Clock divider /5 -> 5Mhz	< c c 4 <CR> <LF>
Clock divider /6 -> 4.167 Mhz	< c c 5 <CR> <LF>
MaxL1 On	< c L 1 o 1 <CR> <LF>
MaxL1 Off	< c L 1 o 0 <CR> <LF>
MaxL1 Conf	< c L 1 t 1 <CR> <LF>
MaxL1 test 0	< c L 1 x 0 <CR> <LF>
MaxL1 test 1	< c L 1 x 1 <CR> <LF>
MaxL1 test lock	< c L 1 x L <CR> <LF>
MaxL1 Ina1	< c L 1 I 1 <CR> <LF>
MaxL1 Ina2	< c L 1 I 2 <CR> <LF>
MaxL1 gaining + L1	< c L 1 g + <CR> <LF>
MaxL1 gaining - L1	< c L 1 g - <CR> <LF>
MaxL1 agc on	< c L 1 a 1 <CR> <LF>
MaxL1 agc off	< c L 1 a 0 <CR> <LF>
MaxL1 filter 3order 2.5Mhz 17db	< c L 1 f 3 0 0 <CR> <LF>
MaxL1 filter 3order 2.5Mhz 26db	< c L 1 f 3 0 1 <CR> <LF>
MaxL1 filter 3order 4.2Mhz 17db	< c L 1 f 3 2 0 <CR> <LF>
MaxL1 filter 3order 4.2Mhz 26db	< c L 1 f 3 2 1 <CR> <LF>
MaxL1 filter 3order 8Mhz 17db	< c L 1 f 3 1 0 <CR> <LF>
MaxL1 filter 3order 8Mhz 26db	< c L 1 f 3 1 1 <CR> <LF>
MaxL1 filter 5order 2.5Mhz 17db	< c L 1 f 5 0 0 <CR> <LF>
MaxL1 filter 5order 2.5Mhz 26db	< c L 1 f 5 0 1 <CR> <LF>
MaxL1 filter 5order 4.2Mhz 17db	< c L 1 f 5 2 0 <CR> <LF>
MaxL1 filter 5order 4.2Mhz 26db	< c L 1 f 5 2 1 <CR> <LF>
MaxL1 filter 5order 4.2Mhz 26db	< c L 1 f 5 2 1 <CR> <LF>
MaxL1 AntBIAS off	< c L 1 b 0 <CR> <LF>
MaxL1 AntBIAS on	< c L 1 b 1 <CR> <LF>
MaxL2L5 On	< c L 2 o 1 <CR> <LF>
MaxL2L5 Off	< c L 2 o 0 <CR> <LF>
MaxL2L5 Conf	< c L 2 t 1 <CR> <LF>
MaxL2L5 gaining +	< c L 2 g + <CR> <LF>
MaxL2L5 gaining -	< c L 2 g - <CR> <LF>
MaxL2L5 Ina1	< c L 2 I 1 <CR> <LF>
MaxL2L5 Ina2	< c L 2 I 2 <CR> <LF>
MaxL2L5 filter 5order 2.5Mhz 17db	< c L 2 f 5 0 0 <CR> <LF>
MaxL2L5 filter 5order 2.5Mhz 26db	< c L 2 f 5 0 1 <CR> <LF>
MaxL2L5 filter 5order 4.2Mhz 17db	< c L 2 f 5 2 0 <CR> <LF>
MaxL2L5 filter 5order 4.2Mhz 26db	< c L 2 f 5 2 1 <CR> <LF>
MaxL5 agc on	< c L 5 a 1 <CR> <LF>
MaxL5 agc off	< c L 5 a 0 <CR> <LF>
Vco enable + conf	< c V e <CR> <LF>
Vco disable	< c V d <CR> <LF>
Vco Conf L2	< c V 2 <CR> <LF>
Vco Conf L5	< c V 5 <CR> <LF>
Vco test 0 complete register prog	< r V 3 I ▯ <ETX> <CR> <LF>
Vco test 1 complete register prog	< r V 3 I @ <ETX> <CR> <LF>
Vco test lock complete register prog	< r V 3 I ▯ <ETX> <CR> <LF>
Mux L5	< c M L 5 <CR> <LF>
Mux L2	< c M L 2 <CR> <LF>
Mux off	< c M L 0 <CR> <LF>
Adc L1 enable	< c A 1 e <CR> <LF>
Adc L1 disable	< c A 1 d <CR> <LF>
Adc L2L5 enable	< c A 2 e <CR> <LF>
Adc L2L5 disable	< c A 2 d <CR> <LF>

2.4 Summary of RF testing

In [6] the unit tests for the RF hardware were specified (subsection 4.1). In Table 2.7 the results from the compliance with the proposed test, are summarized.

Table 2.7: GNSS RF chain tests

Test ID	Objective	Results
RF testing		
GNSS RF input chain	Verify functionality of the GNSS RF chain	*Output tones input to downconverter within correct power and frequency levels.*
Reference clock	Verify clock stability and amplitude	*Measured phase noise ref TCXO (26MHz):* *-103 dBc/Hz <-103@1KHz* *-119 dBc/Hz < -116@10KHz* *-121 dBc/Hz < -122 @100KHz* *-123 dBc/Hz < -135 @1MHz*

		Measured phase noise ref oscillator (2750 MHz): *-116 dBc/Hz < -80 dBc/Hz @10Khz* *-122 dBc/Hz < -107 dBc/Hz @ 100KHz* *-135 dBc/Hz < -129 dBc/Hz @ 1MHz*
Intermediate Frequency	Verify IF output	*IF frequencies correctly configured deviation below 10KHz*
Wireless interface	Verify functionality of the wireless interface (Wi-Fi or 3G, TBC)	*Not applicable to the RF FE, will be implemented in the digital processing platform using a standard wireless adapter.*
RF front-end electrical performance		
Data bandwidth	Verify data bandwidth for each band	Generated samples at *5.2MHz for L1* *Generated samples at 5.2/6.5MHz for L2* *Generated samples at 26MHz for L5* *(decimation of samples performed in digital processing platform)*
Out-of-band filtering	Verify filtering of out-of-band using testing tones	Filtering of out-of-band >60dB @ CF ±50 MHz
Quantization bits	Verify ADC bit resolution by histogram examination	Generated histograms via the offline post-processed of the ADC inputs. Optimized AGC to use full ADCs dynamic ranges.
Noise figure	Noise figure estimation	2-3dB (@ dowconverter input) < 4dB
Frequency bands	Verify available frequency bands simultaneously or selectable	L1, L2 bands tested I/Q samples generated at baseband
Electromagnetic shielding	Verify GNSS and wireless interface RF shielding	EMI reduced by multiple shielding covers and signal layout. Several improvements performed in v2.0.
PCB Testing procedures		
Printed circuit board	Verify functionality of the printed circuit board	PCB electrical tests performed, no noticeable manufacturing issues detected.

3. Digital Processing Platform (CTTC)

The Digital Processing Platform is the device in charge of executing the software-defined GNSS receiver and other related controlling programs. In this WP, the design described in Deliverable 2.2 has been consolidated and implemented. In summary, it consists of a Xilinx's Zynq-based platform, a System-on-Chip (SoC) that contains an ARM processor and a FPGA processor. The ARM processor (known as Processing System or PS) runs GNSS-SDR, the open source software receiver developed within AUDITOR, along with some control scripts, and the FPGA processor (known as Processing Logic or PL) executes some specific functions of the software receiver in order to increase the number of satellites that can be acquired and tracked in real time.

According to some experiments (which results were reported in [7]), the dual-core ARM processor that is shipped in a Zynq is not powerful enough to sustain real-time processing of GNSS signals even in the most basic configuration (GPS L1 C/A, 2 Msps). Other ARM-based platforms (such as Raspberry Pi 3, which ships a quad-core processor) are able to sustain about 6-8 satellites in real-time, which are enough for getting PVT fixes but not for applications targeting high accuracy. Thus, the acceleration provided by the PL is of key importance in order to deliver quality GNSS observables in real time when using an embedded system, and benefiting from their low power consumption, small size, rugged operating ranges, and low per-unit cost.

3.1 Zynq Processing System (PS)

3.1.1 Development cycle

This Section describes the development cycle for building and executing GNSS-SDR, its corresponding Quality Assessment code and the control system scripts in an embedded computer. In this example, we are working with a ZedBoard (a development board that ships a Xilinx Zynq-7000 all-programmable SoC, which houses two ARM and one FPGA processor in a single chip), but this procedure is applicable to other embedded platforms without much modification.

Once all the required dependencies are already installed, GNSS-SDR can be built from source in ARM processors without requiring any extra configuration step. However, this building process can easily take more than 10 hours if it is executed on the Zynq device. Thus, in order to speed up the development cycle from a change in the source code to the execution in an embedded platform, we need to resort to cross-compilation.

Cross-compilation consists of a building framework capable of creating executable code for a platform other than the one on which the compiler is running. In our example, we would like to build GNSS-SDR with the powerful, fast processor of a general-purpose desktop computer, and to generate binaries that can be directly executed by the Zynq device. By using cross-compilation, we can shorten the building time from more than 10 hours to less than 10 minutes. This improves **Testability** (defined in Deliverable 1.3, see [6]), as one of its requirements is that a testing cycle has to be *fast*.

The cross-compilation environment proposed here is based on OpenEmbedded (see http://www.openembedded.org), a building framework for embedded Linux. OpenEmbedded offers a best-in-class cross-compile environment, allowing developers to create a complete, custom GNU/Linux distribution for embedded systems. Using OpenEmbedded, we created a software developer kit (SDK) that installs a ready-to-use cross-compilation environment in the user's computer. The SDK has been made publicly and freely available at http://gnss-sdr.org/docs/tutorials/cross-compiling/, along with detailed instructions to allow users to build their own customized SDK.

The SDK provides the toolchain installer, a script that installs a cross-compiler, a cross-linker and a cross-debugger, forming a completely self-contained toolchain which allows users to cross-develop on the host machine for the target hardware. Cross-compilation if of paramount importance to accelerate the development and testing of GNSS-SDR in embedded systems.

The general procedure can be summarized as follows (more details in the website):

1) **Get the Software Development Kit**. There are two options here:
   a. Download it from http://gnss-sdr.org/docs/tutorials/cross-compiling/
   b. Customize and build your own SDK (instructions provided in the website)

2) **Install the SDK**. This consists of a one-line command which executes a shell script:

```
$ sudo sh oecore-x86_64-armv7ahf-neon-toolchain-nodistro.0.sh
```

3) **Setting up the cross-compiling environment**. Running the environment script will set up most of the variables required to compile GNSS-SDR This has to be executed each time you want to run the SDK (and since the environment variables are only set for the current shell, you need to source it for every console you will run the SDK from):

```
$ . /usr/local/oecore-x86_64/environment-setup-armv7ahf-neon-oe-linux-gnueabi
```

4) **Cross-compiling GNSS-SDR** and installing it on the target filesystem

```
$ git clone https://github.com/gnss-sdr/gnss-sdr.git
$ cd gnss-sdr
$ git checkout next
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains/oe-sdk_cross.cmake \
-DCMAKE_INSTALL_PREFIX=/usr ..
$ make
$ sudo make install DESTDIR=/usr/local/oecore-x86_64/sysroots/armv7ahf-neon-oe-linux-
gnueabi/
```

   Please note that we set the install prefix to `/usr`. That will be the installation location of the project on the embedded device. We use this because all links and references within the file system will be based on this prefix, but it is obviously not where we want to install these files on our own host system. Instead, we use the make program's `DESTDIR` directive. On the device itself, however, the file system would have this installed onto `/usr`, which means all our links and references are correct as far as the device is concerned.

5) **Copying an image file to the SD card** that will be then inserted into the ZedBoard. The website http://gnss-sdr.org/docs/tutorials/cross-compiling/ describes several methods to do this.

This procedure generates a fully functional, customized Linux distribution that will run on the Processing System, allowing the execution of GNSS-SDR module and all the required control and interface scripts.

3.1.2 GNSS-SDR module

GNSS-SDR is an open source project that implements a global navigation satellite system software defined receiver in C++. With GNSS-SDR, users can build a GNSS software receiver by creating a graph where the nodes are signal processing blocks and the lines represent the data flow between them. The software provides an interface to different suitable RF front-ends and implements the entire receiver's chain up to the navigation solution. Its design allows any kind of customization, including interchangeability of signal sources, signal processing algorithms, interoperability with other systems, output formats, and offers interfaces to all the intermediate signals, parameters and variables.

The goal is to provide efficient and truly reusable code, easy to read and maintain, with fewer bugs, and producing highly optimized executables in a variety of hardware platforms and operating systems. In that sense, the challenge consists of defining a gentle balance between level of abstraction and performance, addressing:

- Concurrency (take advantage of multicore processors).
- Efficiency (take advantage of the specific processor architectures).
- Performance (and how to measure it!).
- Portability (should live in a complex, dynamic ecosystem of operating systems and processor architectures).
- Ability to run in real-time or in post-processing.
- Extendibility (easy addition and test of new algorithms and implementations).

The proposed software receiver runs in a wide range of processor architectures (including AUDITOR's) and provides interfaces to a variety of either commercially available or custom-made RF front-ends (such as AUDITOR's), adapting the processing algorithms to different sampling frequencies, intermediate frequencies and sample resolutions. It also can process raw data samples stored in a file. The software performs signal acquisition and tracking of the available satellite signals, decodes the navigation message and computes the observables needed by positioning algorithms, which ultimately compute the navigation solution. It is designed to facilitate the inclusion of new signal processing techniques, offering an easy way to measure their impact in the overall receiver performance. Testing of all the processes is conducted both by the systematic functional validation of every single software block and by experimental validation of the complete receiver using both real and synthetic signals. The processing output can be stored in Receiver Independent Exchange Format (RINEX), used by most geodetic processing software for GNSS, or transmitted as RTCM 3.2 messages through a TCP/IP server in real-time.

GNSS-SDR module's design was described in Deliverable 2.2 (see [2]), Section 3.4.2. Basically, it consists of a control plane and a signal processing plane, described below.

3.1.2.1 **Control plane**

The Control Plane (which design was described in Deliverable D2.2 [2], Section 3.4.2.1) is in charge of creating a flow graph of interconnected nodes. The nodes represent signal processing blocks, and the link between nodes represent unidirectional flows of data. Then, an underlying scheduling system takes data samples from signal sources (that is, a file or actual digitized GNSS signals coming from the output of a RF front-end) to signal sinks (that is, blocks in charge of displaying or storing the final results of the signal processing).

This process scheduling is a key feature to achieve real-time, and to scale well when the software is executed in a more powerful processor. Task parallelization focuses on distributing execution processes (threads) across different parallel computing nodes (processors), each executing a different thread (or process) on the same or different data. Spreading processing tasks along different threads must be carefully designed in order to avoid bottlenecks (either in the processing or in memory access) that can block the whole processing chain and prevent it from attaining real-time operation. This section provides an overview of the task scheduling strategy implemented in GNSS-SDR.

GNSS-SDR uses a "**thread-per-block**" scheduler, which means that each instantiated processing block runs in its own thread. This architecture scales very well to multicore processor architectures. The implementation is provided by GNU Radio (see https://gnuradio.org), whose flow graph computations can be jointly modelled as a **Kahn process** [8], [9]. A Kahn process describes a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or tokens and send them along a communication channel where they are consumed by the waiting destination process. Communication channels are the only method processes may use to exchange information. Kahn requires the execution of a process to be suspended when it attempts to get data from an empty input channel. A process may not, for example, test an input for the presence or absence of data. At any given point, a process can be either enabled or blocked waiting for data on only one of its input channels: it cannot wait for data from more than one channel. Systems that obey Kahn's mathematical model are determinate: the history of tokens produced on the communication channels does not depend on the execution order [8]. With a proper scheduling policy, it is possible to implement software defined radio process networks holding two key properties:

- **Non-termination**: understood as an infinite running flow graph process without deadlocks situations, and
- **Strictly bounded**: the number of data elements buffered on the communication channels remains bounded for all possible execution orders.

An analysis of such process networks scheduling was provided in [10]. By adopting GNU Radio's signal processing framework, GNSS-SDR bases its software architecture in a well-established design and extensively proven implementation.

Software defined receivers can be represented as flow graph of nodes. Each node represents a signal processing block, whereas links between nodes represents a flow of data. The concept of a flow graph can be viewed as an acyclic directional graph with one or more **source blocks** (to insert samples into the flow graph), one or more **sink blocks** (to terminate or export samples from the flow graph), and any **signal processing blocks** in between. The diagram of a processing block (that is, of a given node in the flow graph), as implemented by the GNU Radio framework, is shown in Figure 3.1. Each block has a completely independent scheduler running in its own execution thread and a messaging system for communication with other upstream and downstream blocks. The actual signal processing is performed in the `work()` method.

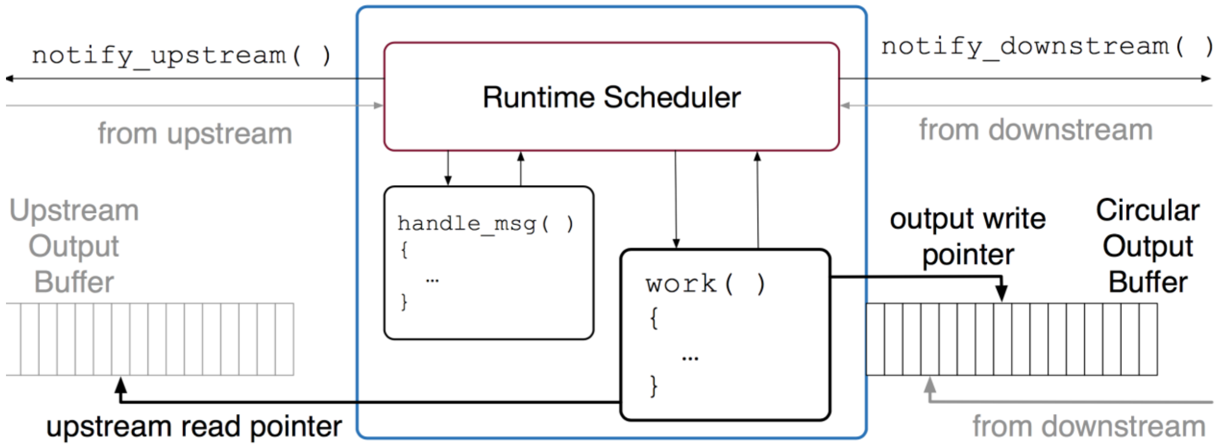


Figure 3.1: Diagram of a signal processing block, as implemented by GNU Radio. Figure from [7].

Each block can have an arbitrary number of input and output *ports* for data and for asynchronous message passing with other blocks in the flow graph. In all software applications based on the GNU Radio framework, the underlying process scheduler passes items (i.e., units of data) from sources to sinks. For each block, the number of items it can [8] process in a single iteration is dependent on how much space it has in its output buffer(s) and how many items are available on the input buffer(s). The larger that number is, the better in terms of efficiency (since the majority of the processing time is taken up with processing samples), but also the larger the latency that will be introduced by that block. On the contrary, the smaller the number of items per iteration, the larger the overhead that will be introduced by the scheduler.

Thus, there are some constraints and requirements in terms of number of available items in the input buffers and in available space in the output buffer in order to make all the processing chain efficient. In GNU Radio, each block has a runtime scheduler that dynamically performs all those computations, using algorithms that attempt to optimize throughput, implementing a process network scheduling that fulfills the requirements described in [10]. Each processing block executes in its own thread. A detailed description of the GNU Radio internal scheduler implementation (memory management, requirement computations, and other related algorithms and parameters) can be found in [11], and of course in GNU Radio source code (available at https://github.com/gnuradio/gnuradio).

Under this scheme, software-defined signal processing blocks read the available samples in their input memory buffer(s), process them as fast as they can, and place the result in the corresponding output memory buffer(s), each of them being executed in its own, independent thread. This strategy **results in a software receiver that always attempts to process signal at the maximum processing capacity**, since each block in the flow graph runs as fast as the processor, data flow and buffer space allows, regardless of its input data rate. Achieving real-time is *only* a matter of executing the receiver's full processing chain in a processing system powerful enough to sustain the required processing load, but it does not prevent from executing exactly the same process at a slower pace, for example, by reading samples from a file in a less powerful platform.

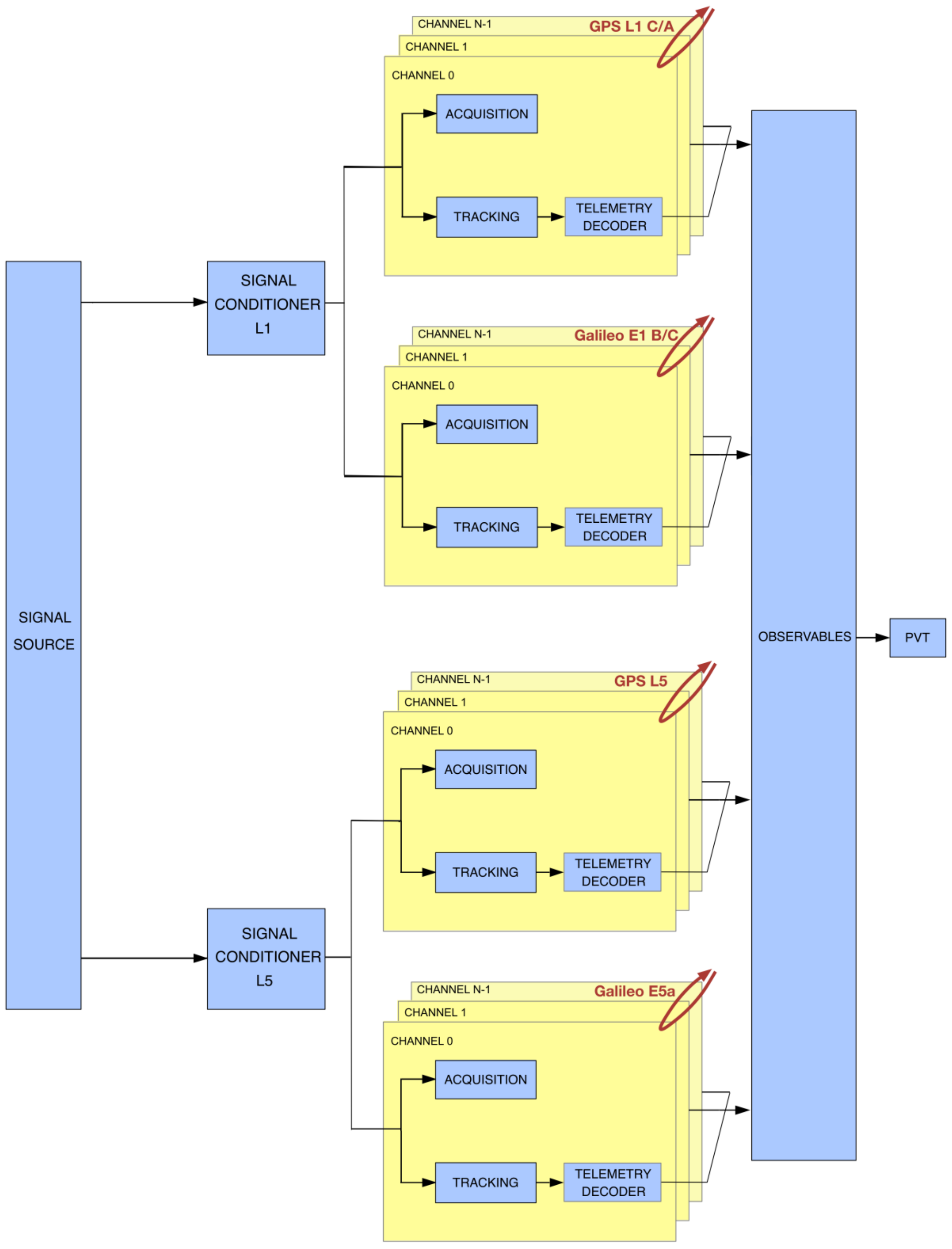


Figure 3.2: Diagram of a multi-band, multi-system GNSS-SDR flow graph

Figure 3.2 shows a possible flow graph diagram used in GNSS-SDR. There is a signal source block (a dual-band radio-frequency front-end) writing samples in a memory buffer at a given sampling rate;

some signal conditioning (possible data type adaptation, filtering, frequency downshifting to baseband, and resampling); a set of parallel channels, each one reading form the same upstream buffer and targeted to a different satellite; a block in charge of the formation of observables collecting the output of each satellite channel after the despreading (and thus in a much slower rate); and a signal sink, responsible for computing the position-velocity-time solution from the obtained observables and providing outputs in standard formats (such as KML, GeoJSON, RINEX, RTCM and NMEA).

The flow graph in Figure 3.2 defines a multi-band, multi-system GNSS receiver. In all cases, each of the processing blocks will be executing its own thread, defining a multi-threaded GNSS receiver that efficiently exploits task parallelization.

3.1.2.2 Signal processing plane

The Signal Processing Plane (which design was described in Deliverable D2.2, see [2] Section 3.4.2.2) is in charge of defining and implementing all the processing blocks that will form the receiver's flow graph defined in the Control Plane.

In the system developed within AUDITOR, the Processing System of the Zynq device will run exactly the same code than in the open source version of GNSS-SDR but the implementation of specific functions (described in Section 3.2), which are **off-loaded to the FPGA in order to accelerate** their execution. The ARM – FPGA interface for computation off-loading is described in Section 3.1.4.

The documentation about the available blocks and their configuration parameters are published online at http://gnss-sdr.org/docs/sp-blocks/

3.1.3 System control

3.1.3.1 Receiver configuration

GNSS-SDR's configuration mechanism design was described in Deliverable 2.2 (see [2]), Section 3.4.2.1.1, and in this WP the design has been consolidated, implemented and tested, demonstrating its flexibility to accommodate the continuously growing number of receiver's configuration parameters and options.

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 processing blocks with the names of the parameters in the configuration.

For instance, parameters related to *SignalSource* should look like this:

```
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:

```
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:

```
; THIS IS A COMMENT
SignalConditioner.implementation=Pass_Through ; THIS IS ANOTHER COMMENT
```

In this way, a full GNSS receiver can be uniquely defined in one text file in INI format:

```
$ gnss-sdr --config_file=/path/to/my_receiver.conf
```

GNSS-SDR allows the user to define a custom GNSS receiver, including its architecture (number of bands, channels per band and targeted signal) and the specific algorithms and parameters for each of the processing blocks through a single configuration file (a simple text file in INI format). Thus, **each configuration file defines a different GNSS receiver**. Some examples of such files are available at `gnss-sdr/conf`.

3.1.3.2 **Startup scripts and status monitoring**

To be defined in WP6.

3.1.4 PS-PL low speed communications

The control of the GNSS-SDR acquisition and tracking hardware accelerators and the front-end configuration requires a bi-directional low speed communication pipe between the software plane and the PL. Next table introduces a summary of the data exchange to and from the software plane and the hardware plane:

Table 3.1: Data exchange to and from the software plane and the hardware plane

Data source	Data sink	Message type	Maximum frequency
GNSS-SDR	Acquisition accelerator	Initialization data: PRN code and acquisition parameters	1 Hz.
Acquisition accelerator	GNSS-SDR	Acquisition result: Code delay, Doppler frequency, and test statistics	1 Hz.
GNSS-SDR	Tracking accelerator	Initialization data: PRN code and pull-in parameters	1 Hz.
GNSS-SDR	Tracking accelerator	Tracking loop parameters: Code NCO and Carrier NCO commands	1 kHz.
Tracking accelerator	GNSS-SDR	Correlators output	1 kHz.
GNSS-SDR	Front-end hardware module	Initialization parameters: Carrier frequency, number of channels, sampling frequency, gain parameters	One-time configuration at the receiver startup.
Front-end hardware module	GNSS-SDR	Front-end status feedback	0.01 Hz.

A suitable communication mechanism, which provides enough bandwidth and low latency with low complexity, is the so-called memory mapped registers. Each hardware accelerator publishes a set of AXI registers at specific memory addresses as described in Section 3.2. In order to minimize the latency and the required computational resources, each accelerator provides an interrupt signal that triggers the data gathering from GNSS-SDR.

3.1.4.1 **User Input-Output (UIO) driver framework**

Linux provides a standard called UIO (User I/O) framework for developing user-space-based device drivers. The UIO framework defines a small kernel-space component that performs two key tasks:

- Indicate device memory regions to user space.
- Register for device interrupts and provide interrupt indication to user space.

The kernel-space UIO component then exposes the device via a set of sysfs entries like `/dev/uioXX`. The user-space component searches for these entries, reads the device address ranges and maps them to user space memory.

The user-space component can perform all device-management tasks including I/O from the device. For interrupts however, it needs to perform a blocking `read()` on the device entry, which results in the kernel component putting the user-space application to sleep and waking it up once an interrupt is received.

In order to activate the Linux kernel UIO driver, the device tree description for the **hardware accelerators** must contain the generic-uio string. Next device tree DTS entry shows an example of a description for a tracking accelerator:

```
multicorrelator_resampler_S00_AXI_0: multicorrelator_resampler_S00_AXI@43c00000 {
                             compatible = "generic-uio";
                             interrupt-parent = <&intc>;
                             interrupts = <0 31 4>;
                             reg = <0x43c00000 0x10000>;
                             xlnx,s-start-count = <0x20>;
          };
```

3.1.5 PS-PL high speed communications

The baseband signal samples flow from the AUDITOR custom front-end ADCs to the hardware accelerators is routed directly to the Zynq FPGA fabric, thus, there is no need of any communication mechanism in the PS plane. However, for debugging purposes and to support USB-based front-ends, such as the Ettus Research Universal Software Radio Peripheral (USRP) family, it is required a high-speed communication mechanism from PS to PL.

The selected mechanism is based on the Xilinx AXI Direct Memory Access (AXI DMA) IP core (see http://www.wiki.xilinx.com/DMA+Drivers+-+Soft+IPs) with a user space DMA driver.

The AXI DMA IP provides high-bandwidth direct memory access between memory and AXI4-Stream-type target peripherals. Its optional scatter gather capabilities also offload data movement tasks from the CPU in processor-based systems. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface.

Features Supported

- AXI4 and AXI4-Stream compliant
- Optional Scatter/Gather (SG) DMA support. When Scatter/gather mode is not selected the IP operates in Simple DMA mode.

- Primary AXI4 Memory Map and AXI4-Stream data width support of 32, 64, 128, 256, 512, and 1024 bits
- Optional Data Re-Alignment Engine
- Optional AXI Control and Status Streams
- Multi-channel mode
- Support for up to 64-bit Addressing

The user space driver is based on the open source project EZDMA (see https://github.com/jeremytrimble/ezdma) Figure 3.3 shows how EZDMA works. The driver publish a set of sysfs files to trigger DMA transfers to/from the user space with simple read() and write() operations. This is also known as zero-copy operations.

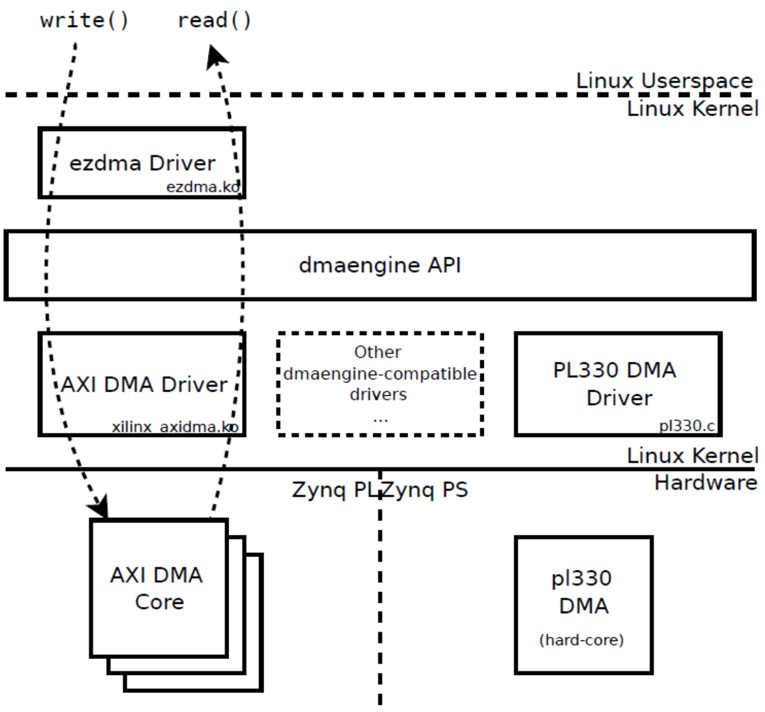


Figure 3.3: EZDMA driver architecture (from [12]).

In order to activate the EZDMA kernel driver, the device tree description must contain the DMA device descriptions and the EZDMA entry. Next device tree DTS subsection shows an example of an AXI DMA + EZDMA description:

```
axi_dma_0: dma@40400000 {
        #dma-cells = <1>;
        compatible = "xlnx,axi-dma-1.00.a";
        interrupt-parent = <&intc>;
        interrupts = <0 29 4 0 30 4>;
        reg = <0x40400000 0x10000>;
        xlnx,include-sg ;
        dma-channel@40400000 {
                compatible = "xlnx,axi-dma-mm2s-channel";
                dma-channels = <0x1>;
                interrupts = <0 29 4>;
                xlnx,datawidth = <0x20>;
```

```
                                xlnx,device-id = <0x0>;
                        };
                        dma-channel@40400030 {
                                compatible = "xlnx,axi-dma-s2mm-channel";
                                dma-channels = <0x1>;
                                interrupts = <0 30 4>;
                                xlnx,datawidth = <0x20>;
                                xlnx,device-id = <0x0>;
                        };
                };
ezdma0: dmatest@0 {
        compatible = "ezdma";
        dmas = <&axi_dma_0 0 &axi_dma_0 1>;
        dma-names = "loop_tx", "loop_rx";
        ezdma,dirs = <2 1>;   // direction of DMA: 1 = RX (dev->cpu), 2 = TX (cpu->dev)
};
```

3.2 Zynq Processing Logic (PL)

As shown in [7], current ARM-based platforms are not fast enough for a multiple-constellation, multiple-band GNSS receiver configuration working in real time, as required by AUDITOR. Hence, the most computational demanding operations are off-loaded to the FPGA device.

This allows a clear interface between free and open source software (that is, GNSS-SDR) and proprietary software. The open source version is a fully functional GNSS receiver, that users can execute in their own desktop or laptop computer. However, in order to attain real-time in an embedded device (specially for multi-band, multi-constellation configurations), users must resort to the FPGA off-loading provided by the **intellectual property (IP) cores developed within AUDITOR**. In this way, the whole receiver benefits from the extensive testing and collaborative development of GNSS-SDR, while professional users have the possibility to embed the software in small form factor devices.

3.2.1 Hardware accelerators

3.2.1.1 **Overview**

The AUDITOR GNSS receiver is based on a Zynq System-on-Chip (SoC). The Zynq SoC contains a two-core ARM processor and an FPGA together in the same chip. The Zynq is designed to maximize the advantages of using an ARM software-based system combined with FPGA-based HW accelerators. All together the system can obtain high performance in terms of signal processing speed while minimizing the energy consumption.

The ARM processors run a Linux operating system and could in principle execute all the tasks of the AUDITOR GNSS receiver. However, the ARM processors cannot deliver real-time performance when running the GNSS receiver due to their limited performance.

In order to produce a GNSS receiver that can function properly in real time, hardware accelerators were implemented for various functionalities of the physical layer.

The hardware accelerators are implemented in VHDL and encapsulated as IP modules. These IP modules can be reused and instantiated as needed in the main Zynq design. The IP modules replace various functionalities that were earlier implemented in software, which have stringent demands in terms of real-time performance.

3.2.1.2 **Implementation**

The AUDITOR GNSS receiver performs two types of tasks that require very intensive computations: the **acquisition process and the tracking multi-correlation process** of the physical layer. These tasks work at the received sampling rate, whereas the remaining tasks work at the received code rate, which is much lower than the sampling frequency. Therefore, two types of hardware accelerators were implemented:

- Acquisition
- GPS and Galileo Multicorrelators

Figure 3.4 shows a block diagram of the main HW components involved in the data flow.

The **analog front-end** receives data using two frequency bands simultaneously. One of those two frequency bands is fixed to the GPS L1/CA, Galileo E1, and GPS SBAS band. The other frequency band can be switched either to GPS L2C or Galileo E5A and GPS L5

The **buffers** compensate for temporary delays caused by the hardware accelerator modules. The hardware accelerator modules can process samples faster than the received data rate on average, but they might introduce temporary delays due to processing or communication with the ARM processors. We call buffer 1 the buffer that stores the signal of the GPS L1/CA, Galileo E1 and GPS SBAS band. We call buffer 2 the buffer that stores the signal of the GPS L2C or the signal of the Galileo E5A and GPS L5 band.

The **Acquisition** HW accelerators runs a parallel code phase search Acquisition. It frees the ARM processors from this task. The ARM processors configure the Acquisition HW module and read the Acquisition results using the low speed communication bus (see section 3.1.4) and the hardware interrupts.

The **GPS Multicorrelator** hardware accelerators and the **Galileo Multicorrelator** hardware accelerators perform the Doppler wipeoff and the multi-correlation between the received signal and a local copy of the GNSS code. They free the ARM processors from this task. The ARM processors configure the Multicorrelator modules and read the multi-correlator results using the low speed communication bus (see section 3.1.4) and the hardware interrupts.

The Acquisition HW accelerators and the Multicorrelator HW accelerators keep track of all the samples that are received from the analog front-end in order to keep the synchronization between them. The acquisition returns a sample pointer that is then used by the Multicorrelator HW accelerators to process the received signal at the right synchronization point.

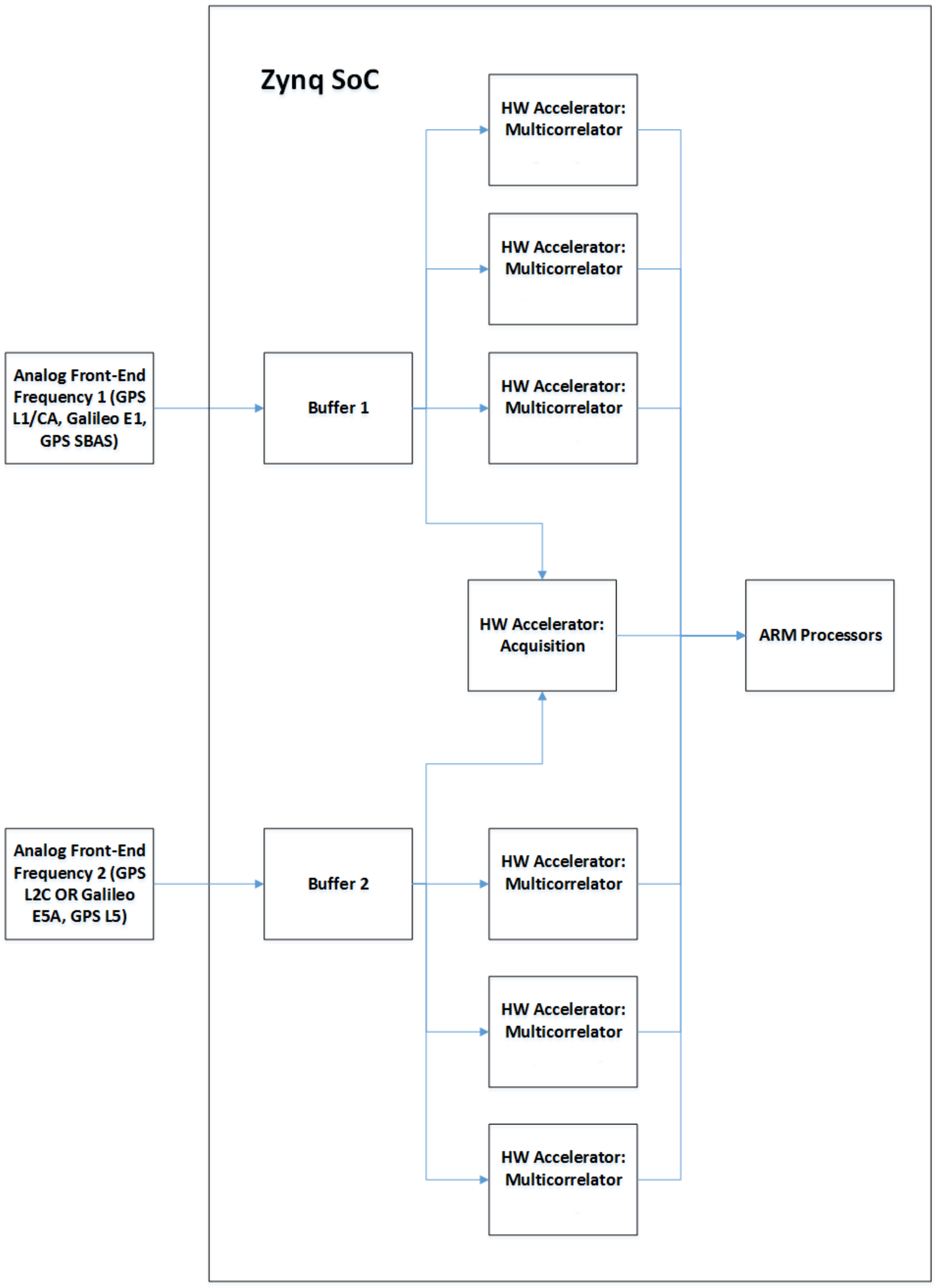

Zynq SoC
Analog Front-End Frequency 1 (GPS L1/CA, Galileo E1, GPS SBAS)
Buffer 1
HW Accelerator: Multicorrelator
HW Accelerator: Multicorrelator
HW Accelerator: Multicorrelator
HW Accelerator: Acquisition
ARM Processors
Analog Front-End Frequency 2 (GPS L2C OR Galileo E5A, GPS L5)
Buffer 2
HW Accelerator: Multicorrelator
HW Accelerator: Multicorrelator
HW Accelerator: Multicorrelator

Figure 3.4: Block Diagram of the data flow inside the HW part of the receiver

The **Acquisition HW accelerator** can be used to run the acquisition with both GPS and Galileo signals. It has two input high speed ports, such that it can be connected to both Buffer 1 and Buffer 2.

The **GPS Multicorrelator** hardware accelerators can only be used for GPS signals. They have one input high speed port.

The **Galileo Multicorrelator** hardware accelerators can in principle be used for both GPS and Galileo signals. However, they occupy more FPGA resources than the GPS Multicorrelator. Therefore, to save space in the FPGA, the GPS signals are tracked using the GPS Multicorrelator and not the Galileo Multicorrelator.

When testing without the analog front-end, the DMA is used to transfer samples from the ARM processors main memory to the HW accelerators. This is shown in Figure 3.5.

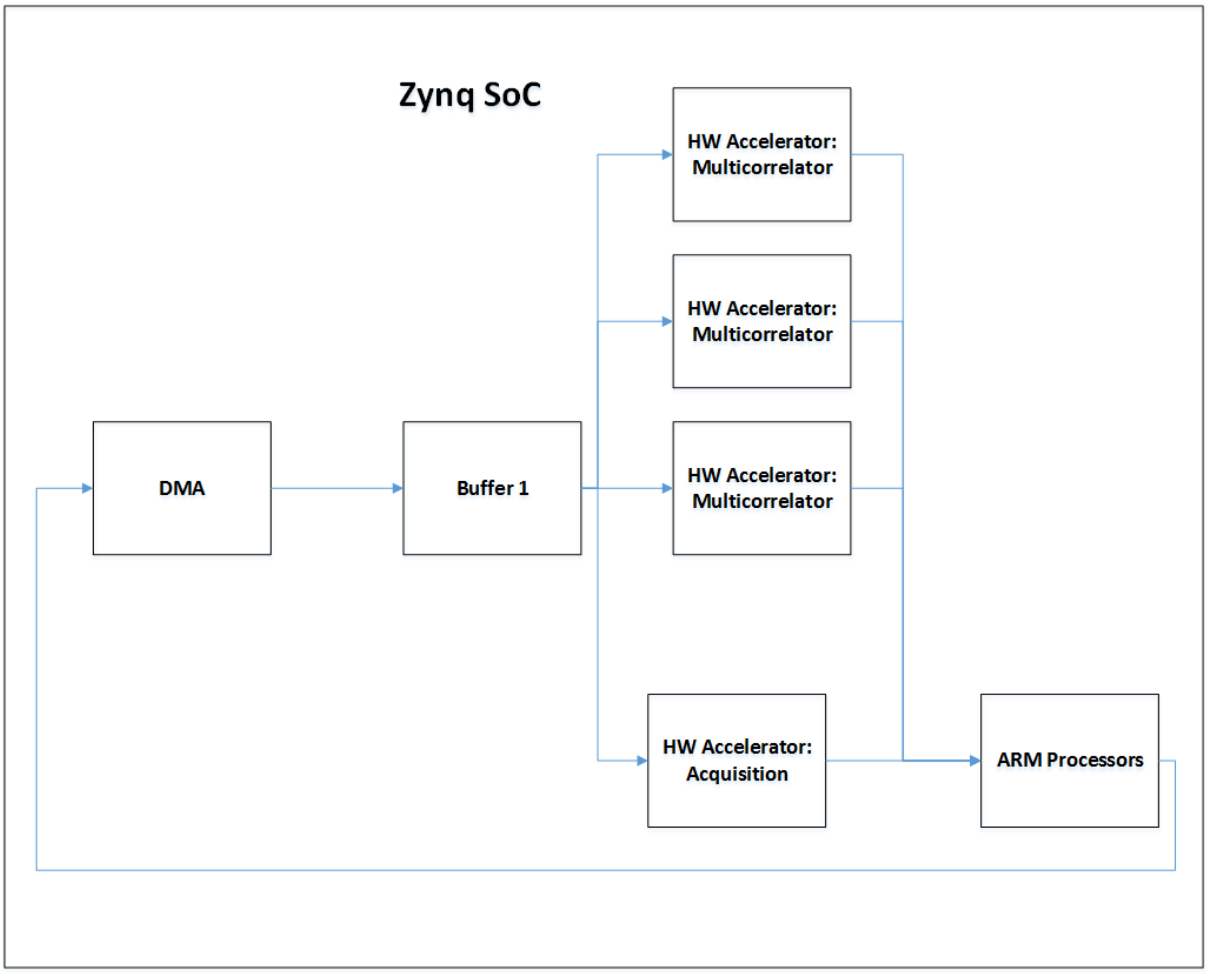

Zynq SoC
HW Accelerator: Multicorrelator
HW Accelerator: Multicorrelator
DMA
Buffer 1
HW Accelerator: Multicorrelator
HW Accelerator: Acquisition
ARM Processors

Figure 3.5: Block Diagram of the data flow inside the HW part of the receiver when testing with the DMA

Figure 3.6 shows an example of several hardware accelerator modules instantiated in a Zynq design. The main parts of the design are the ARM processors, the DMA, the Buffer, the Multicorrelator modules and the Acquisition module. This is an example of a design used for testing. The ARM processors use the DMA to send samples stored in a file to the Acquisition and Multicorrelator modules using the Buffer. The path DMA -> Buffer -> Acquisition/Multicorrelator hardware accelerator modules is connected using the high speed PS-PL communications explained in section 3.1.5. The DMA itself and the Acquisition/Multicorrelator hardware accelerators are connected to the ARM using the PS-PL low speed communications explained in section 3.1.4. and the hardware interrupts. The Acquisition and Multicorrelator accelerator modules are memory mapped to the ARM processors using this low speed PS-PL communications bus. The ARM processors control the hardware accelerators using these memory mapped registers.

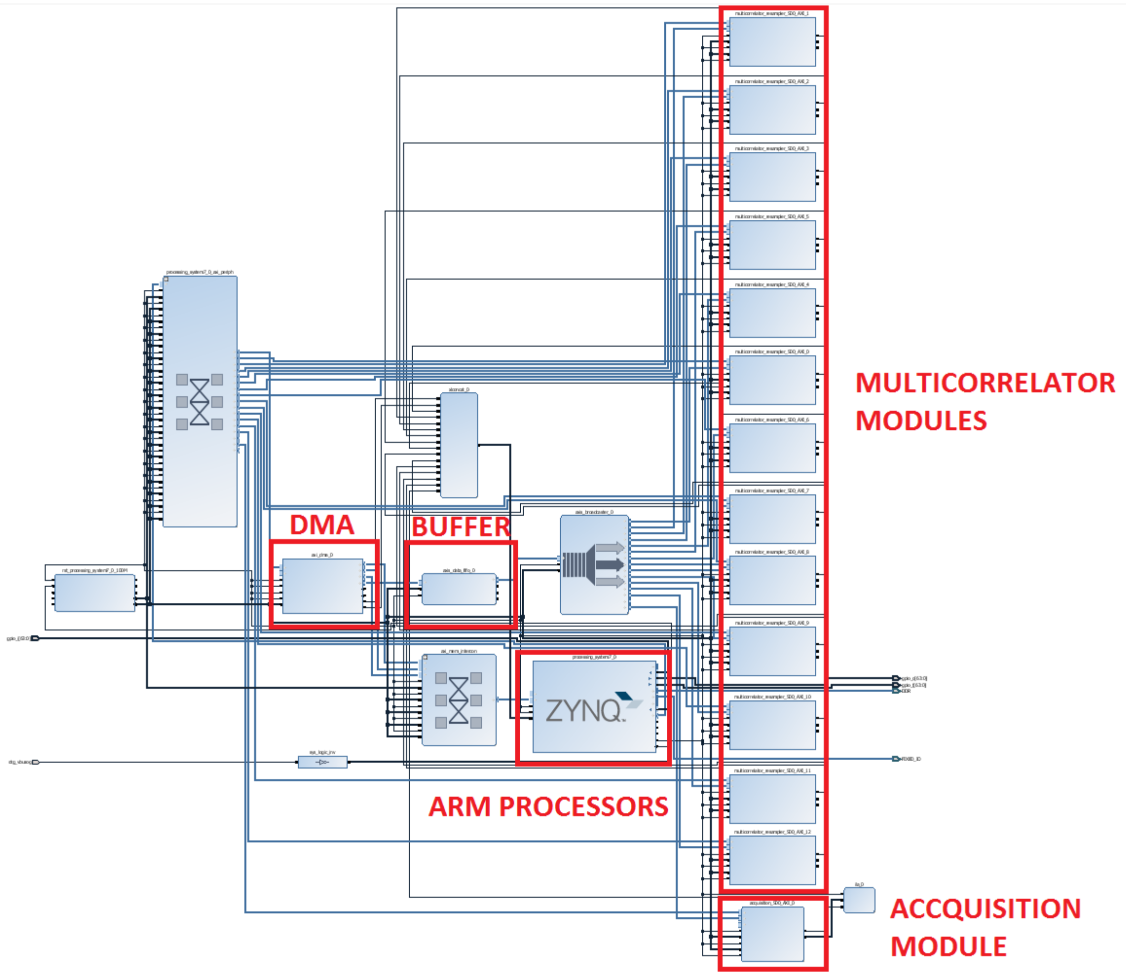


Figure 3.6: Hardware Accelerator Modules

3.2.1.3 **Buffers**

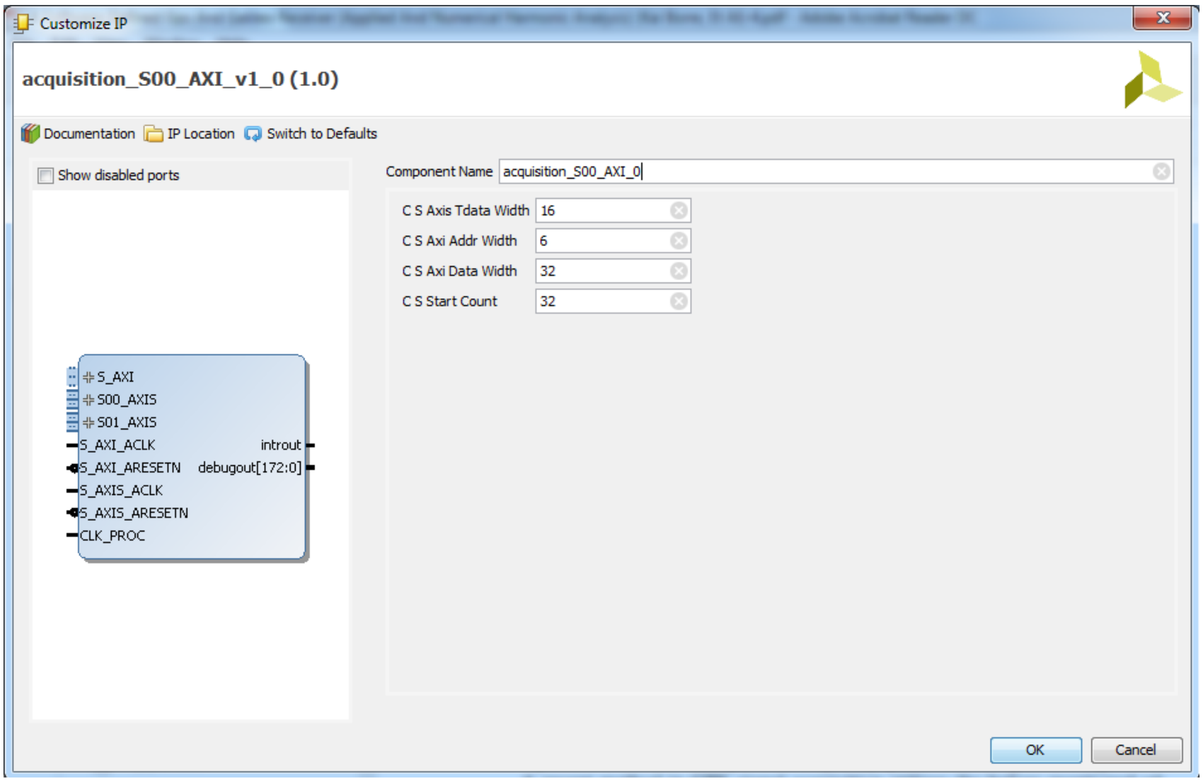
In WP3, the buffer design described in [2] section 3.2.2.1, has been consolidated and implemented. As mentioned in section 3.2.1.2 of this document, the buffers are used to compensate for temporary delays introduced by the hardware accelerators during acquisition or tracking. The hardware accelerators can process samples faster than the received sample rate on average. However temporary delays might be introduced due to signal processing calculations, communication between the hardware accelerators and the ARM processors, etc. The buffers are dimensioned in order to occupy the minimum space in the FPGA while ensuring that no samples are lost during operation.

3.2.1.4 **Signal Acquisition**

In WP3, the Signal Acquisition design described in [2] section 3.2.2.2, has been consolidated and implemented. The Acquisition module runs a parallel code phase search acquisition algorithm. The result of the acquisition is an estimation of the synchronization point in the received signal and an estimation of the Doppler frequency. The Doppler sweep is done in the software, this is: the software runs the hardware Acquisition accelerator for each Doppler frequency candidate.

The Acquisition module has two input high speed communication ports (AXI-Stream ports) used to connect the Acquisition hardware with the corresponding buffer.

Figure 3.7 shows the aspect of the Acquisition IP block. The `debugout` port is an optional port for debugging which will not be used in the final version.



acquisition_S00_AXI_v1_0 (1.0)

Component Name: acquisition_S00_AXI_0

C S Axis Tdata Width	16
C S Axi Addr Width	6
C S Axi Data Width	32
C S Start Count	32

Figure 3.7: Acquisition Hardware Accelerator

The input/output ports shown in Figure 3.7 are detailed in Table 3.2.

Table 3.2: Acquisition Hardware Accelerator ports

Port	Input/Output	Explanation
S_AXI	Input	AXI port (low speed communications). This port is used to connect the Acquisition hardware accelerator to the ARM processors.
S00_AXIS	Input	AXI-Stream port 1 (high speed communications port 1). This port is to be used to connect the Acquisition hardware accelerator to Buffer 1, see section 3.2.1.1).
S01_AXIS	Input	AXI-Stream port 2 (high speed communications port 2). This port is to be used to connect the Acquisition hardware

		accelerator to Buffer 2, see section 3.2.1.1).
S_AXIS_ACLK	Input	Digital clock of the AXI-stream buses
S_AXI_ARESETN	Input	Reset of the AXI bus (also used as main reset for this module)
CLK_PROC	Input	Clock for the processing logic: by default the module has separate clocks for the AXI bus (not shown by default), the AXI-Stream bus (shown above) and the processing logic. The reason why there are three separate clocks is to allow for the possibility of using a higher frequency clock for part of the processing logic. By default the module is internally designed such that all three clocks are the same. Raising the clock of the processing logic would require a modification in the VHDL code.
Introut	Output	Output interrupt line. The interrupt is asserted when the acquisition process is finished.

3.2.1.5 **Signal tracking**

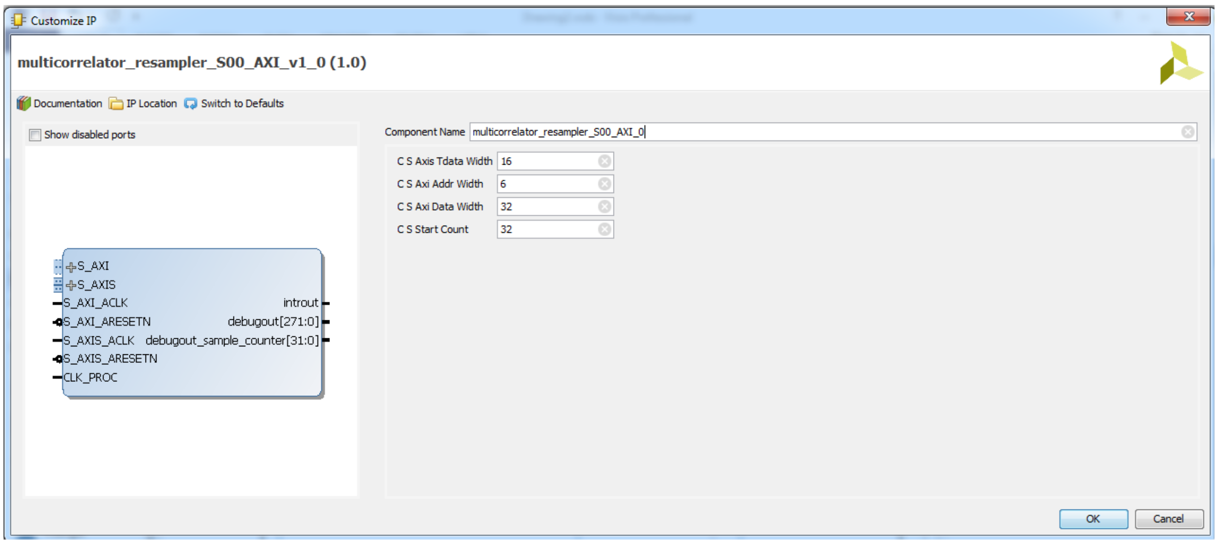
In WP3, the Signal Tracking design described in [2] section 3.2.2.3 has been consolidated and implemented. There are two Multicorrelator HW accelerator modules the GPS Multicorrelator module and the Galileo Multicorrelator module.

As mentioned in section 3.2.1.2, the Galileo Multicorrelator hardware accelerators can in principle be used for both GPS and Galileo signals. However, they occupy more FPGA resources than the GPS Multicorrelator. Therefore, to save space in the FPGA, the GPS signals are tracked using the GPS Multicorrelator and not the Galileo Multicorrelator.

3.2.1.6 **GPS Multicorrelator**

The GPS Multicorrelator module runs a Doppler wipe-off and a multicorrelation between the received signal and a local copy of the GPS code.

Figure 3.8 shows the aspect of the GPS Multicorrelator IP block. The debugout ports is an optional port for debugging which will not be used in the final version.



multicorrelator_resampler_S00_AXI_v1_0 (1.0)

Component Name: multicorrelator_resampler_S00_AXI_0

C S Axis Tdata Width: 16
C S Axi Addr Width: 6
C S Axi Data Width: 32
C S Start Count: 32

Figure 3.8: GPS Multicorrelator HW accelerator

The input/output ports shown in Figure 3.8 are detailed in Table 3.3.

Table 3.3: GPS Multicorrelator Accelerator ports

Port	Input/Output	Explanation
S_AXI	Input	AXI port (low speed communications). This port is used to connect the Acquisition hardware accelerator to the ARM processors.
S_AXIS	Input	AXI-Stream port (high speed communications). This port is to be used to connect the Acquisition hardware accelerator to Buffer 1 or Buffer 2, see section 3.2.1.1).
S_AXI_ACLK	Input	Digital clock of the AXI bus
S_AXI_ARESETN	Input	Reset of the AXI bus (also used as main reset for this module)
S_AXIS_ACLK	Input	Digital clock of the AXI-stream bus
S_AXIS_ARESETN	Input	Reset of the AXI Stream bus
CLK_PROC	Output	Clock for the processing logic: by default the module has separate clocks for the AXI bus (not shown by default), the AXI-Stream bus (shown above) and the processing logic. The reason why there are three separate clocks is to allow for the possibility of using a higher frequency clock for part of the processing logic. By default the module is internally designed such that all three clocks are the same. Raising the clock of the processing logic would require a modification in the VHDL code.
Introut	Output	Output interrupt line. The interrupt is asserted when the acquisition process is finished.

3.2.1.7 **Galileo Multicorrelator**

The Galileo Multicorrelator module runs a Doppler wipe-off and a multicorrelation between the received signal and a local copy of the Galileo code.

The Galileo Multicorrelator HW accelerators are the same as the I/O ports of the GPS Multicorrelator HW accelerator (see Figure 3.3).

3.3 Summary of indicators status for the software-defined receiver

In Deliverable 1.3 (see [6]) we identified a set of indicators for the software-defined receiver and provided a report on its status at the start of AUDITOR. Below, we reproduce the same table, indicating the progress achieved since the beginning of the project.

Table 3.4: GNSS-SDR current status.

Indicator	Requirements	Current status	Status M0	Status M16
Definition of project's geographical coordinate frame and geodetic datum.	Position must be expressed in an adequate coordinate reference system and geodetic datum.	The World Geodetic System (WGS84) is a standard for use in cartography, geodesy, and navigation (including GPS and Galileo). Position fixes will be expressed in geographic coordinates (latitude and longitude) and height above the gravity geoid (*i.e.*, orthometric height).	✓	✓
Definition of accuracy metrics and procedures.	Definition of metrics for 2D and 3D positioning. Detailed definition of measurement procedures.	AUDITOR accuracy metrics: • for 2D positioning: Distance Root Mean Square (DRMS) and the Circular Error Probability (CEP), in meters, and, • for 3D positioning: the Mean Radial Spherical Error (MRSE) and the Spherical Error Probable (SEP), in meters. Measurement procedures defined in Deliverable 1.3.	✓	✓
Meet AUDITOR's static positioning accuracy requirements.	Not defined.	Testing software implemented. Experiments to be addressed in WP6.	❌	Ongoing
Meet AUDITOR's dynamic positioning accuracy requirements.	Not defined.	Testing software implemented. Experiments to be addressed in WP6.	❌	Ongoing
24/7 service	24/7 service	Measurement procedures defined in Section 6 in Deliverable 1.3. Experiments to be addressed in WP6.	❌	Ongoing
Typical duration of usage session	Targeting > 24 h.	Measurement procedures defined in Section 6 in Deliverable 1.3. Experiments to be addressed in WP6.	❌	Ongoing

Indicator	Requirements	Current status	Status M0	Status M16
Acquisition sensibility	Not defined.	Measurement procedures defined in Section 6. Experiments to be addressed in WP6.	✗	Ongoing
Tracking sensibility	Not defined.	Measurement procedures defined in Section 6. Experiments to be addressed in WP6.	✗	Ongoing
Time to First Fix	Definition of detailed TTFF measurement procedures in different receiver's status.	Cold start TTFF measurement procedures defined in Section 6 in Deliverable 1.3.	✗	✓
		Warm start TTFF measurement procedures defined in Section 6 in Deliverable 1.3.	✗	✓
		Hot start TTFF measurement procedures defined in Section 6 in Deliverable 1.3.	✗	✓
Reacquisition	TBD	Measurement procedures defined in Section 6. Experiments to be addressed in WP6.	✗	Ongoing
Number of parallel channels that the software receiver can sustain in real time, given the targeted GNSS signal of each channel and the computational resources available for signal processing.	More than 8 channels per GNSS signal in a multiband configuration.	More than 8 channels per GNSS signal in a multiband configuration.	✗	✓
Power consumption for a given host computer and computational load in terms of number of signals and channels to be processed.	TBD	Not measured	✗	✗
Availability of profiling tools for identifying processing bottlenecks and measuring efficiency.	Allow for statistical profiling tools. Allow for CPU profiling tools. Allow for instrumenting profiler tools. Statistical execution time measurement tool available for the supported processing platforms.	GNSS-SDR allows for the usage of several software profiling tools, see http://gnss-sdr.org/documentation/how-profile-code for some examples. The suggested approach consists of using a set of freely available profiling tools that use different techniques, in the hope of taking advantage of their complementary nature and obtain a better insight about how the code is performing.	✗	✓
Possibility to either use synthetically generated or real-life GNSS signals.	-	Implemented	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
Possibility to process signals either in real time or in post-processing time (only limited by the computational capacity of the host machine).	-	AUDITOR's software receiver architecture makes use of a thread-per-block strategy and GNU Radio's task scheduler, which manages a flow graph of nodes. Each node represents a signal processing block, whereas links between nodes represents a flow of data. Under this scheme, software-defined signal processing blocks read the available samples in their input memory buffer(s), process them as fast as they can, and place the result in the corresponding output memory buffer(s). This strategy results in a software receiver that always process signal at the maximum processing capacity, regardless of its input data rate. Achieving real-time is *only* a matter of executing the full receiver's processing chain in a processing system powerful enough to sustain the required processing load, but it does not prevent from executing exactly the same processing at a slower pace, for example by reading samples from a file, in a less powerful platform.	✓	✓
Possibility to use different radio frequency front-ends.	-	AUDITOR's software receiver configuration system allows the selection of UHD and OsmoSDR compatible RF front-ends. AUDITOR's front-end still TBD but both receiver's software architecture and configuration systems allows for easy addition.	✓	✓
Possibility to define custom receiver architectures.	-	Implemented	✓	✓
Possibility to easily define / interchange implementations and parameters for each processing block.	-	AUDITOR's software receiver configuration system allows for an unlimited number of block implementations and number of parameters for each particular implementation.	✓	✓
Possibility to deploy different receiver architectures and components.	-	Implemented	✓	✓
Possibility to allow for unit/component/integration/system testing.	-	Implemented	✓	✓
Possibility to be executed in different processing platforms (mainframes, personal computers, embedded systems, etc).	-	AUDITOR's software receiver building system, based on CMake, currently supports i386, x86_64/amd64, armhf and arm64 processor architectures.	✓	✓
Flexible configuration	The software defined receiver must be fully	AUDITOR's GNSS software defined receiver follows a single-file configuration strategy.	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
mechanism.	configured in a single file. Configuration system must be arbitrarily extendable and easy to use. Allow possibility of overriding parameters via commandline flags. Required tools/dependencies released under open source license.	Properties are passed around within the program using a configuration interface class. Classes that need to read configuration parameters will receive instances of such interface class from where they will fetch the values. 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 processing block factory for that role. Since the configuration is just a set of property names and values without any meaning or syntax, the system is very versatile and arbitrarily 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 the INI format, see https://en.wikipedia.org/wiki/INI_file) For commandline flags management, AUDITOR's GNSS software defined receiver relies on **gflags** (formerly Google Commandline Flags), see https://github.com/gflags/gflags. Gflags is the commandline flags library used within Google, and differs from other libraries in that flag definitions can be scattered around the source code, and not just listed in one place such as main(). In practice, this means that a single source-code file will define and use flags that are meaningful to that file. Any application that links in that file will get the flags, and the gflags library will automatically handle that flag appropriately. There is significant gain in flexibility, and ease of code reuse, due to this technique. Gflags is released under the BSD 3-clauses license.		
"Operation modes"	Receiver operable automatically.	GNSS-SDR can be completely configured and executed automatically by a shell script. The software receiver has also been virtualized in form of Docker container.	❌	✓
GNSS Signals	Signal acquisition, tracking, decoding of the navigation message and generation of code and phase observables.	GPS L1 C/A	✓	✓
		GPS L2C(M)	❌	✓
		GPS L5	❌	Ongoing
		Galileo E1b/c	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
		Galileo E5a	✗	✓
		Galileo E5b	✗	Ongoing
	Signal acquisition, tracking, and decoding of navigation message	EGNOS	✗	Ongoing
RF frontend drivers	The GNSS software receiver must receive data at an adequate bandwidth and sampling frequency.	AUDITOR's RF front-end	✗	✓
		USRP family	✓	✓
		OsmoSDR-compatible	✓	✓
Input data types for raw samples	AUDITOR's GNSS software defined receiver must allow for most usual input raw sample data types (i.e. bit length, integer and floating point encodings) delivered by available GNSS radio-frequency front-ends' analog to digital converters and associated software drivers.	AUDITOR's raw sample data type	✗	✓
		Reading samples represented by 2 bits (sign and magnitude).	✗	✓
		Reading real (IF) samples represented by 1 byte (8-bit signed integer)	✓	✓
		Reading real (IF) samples represented by 1 *short* (16-bit signed integer)	✓	✓
		Reading real (IF) samples represented by 1 *float* (32-bit floating point)	✓	✓
		Reading I&Q (IF or baseband) interleaved samples represented by 1 byte (8-bit signed integer)	✓	✓
		Reading I&Q (IF or baseband) interleaved samples represented by 1 *short* (16-bit signed integer)	✓	✓
		Reading complex (baseband) samples represented by 1 byte (8-bit signed integer) each component.	✓	✓
		Reading complex (baseband) samples represented by 1 *short* (16-bit signed integer) each component.	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
		Reading complex (baseband) samples represented by 1 *float* (32-bit floating point) each component.	✓	✓
Output formats	The "output products" of the GNSS software defined receiver (position, observables, navigation data) must be delivered in (preferably open) standards in order to maximize interoperability with other software packages.	RINEX v2.11 and 3.02 file generation (obs and nav). RINEX (Receiver Independent Exchange Format) is an interchange format for raw satellite navigation system data, covering observables and the information contained in the navigation message broadcast by GNSS satellites. This allows the user to post-process the received data to produce a more accurate result (usually with other data unknown to the original receiver, such as better models of the atmospheric conditions at time of measurement).	✓	✓
		RTCM SC-104 provides standards that define the data structure for differential GNSS correction information for a variety of differential correction applications. Developed by the Radio Technical Commission for Maritime Services (RTCM), they have become an industry standard for communication of correction information. GNSS-SDR implements RTCM version 3.2, defined in [RTCM 10403.2].	✗	✓
		NMEA 0183 is a combined electrical and data specification for communication between marine electronics such as echo sounder, sonars, anemometer, gyrocompass, autopilot, GPS receivers and many other types of instruments. It has been defined by, and is controlled by, the U.S. National Marine Electronics Association. At the application layer, the standard also defines the contents of each sentence (message) type, so that all listeners can parse messages accurately. Those messages can be sent through the serial port (that could be for instance a Bluetooth link) and be used/displayed by a number of software applications such as gpsd, JOSM, OpenCPN, and many others (and maybe running on other devices).	✓	✓
		GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON) supported by numerous mapping and GIS software packages, including OpenLayers, Leaflet, MapServer, GeoServer, GeoDjango,GDAL, and CartoDB. It is also possible to use GeoJSON with PostGIS and Mapnik, both of which handle the format via the GDAL OGR conversion library. The Google Maps Javascript API v3 directly supports the integration of GeoJSON data layers, and GitHub also supports GeoJSON rendering. Format specification freely available at http://geojson.org/geojson-spec.html	✗	✓
		KML (Keyhole Markup Language) is an XML grammar used to encode and transport representations of geographic data for display in an earth browser. KML is an open standard	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
		officially named the OpenGIS KML Encoding Standard (OGC KML), and it is maintained by the Open Geospatial Consortium, Inc. (OGC). KML files can be displayed in geobrowsers such as Google Earth, Marble, osgEarth, or used with the NASA World Wind SDK for Java. Open standard freely available at http://www.opengeospatial.org/standards/kml		
UNIX-friendly	The binary file that executes the software receiver must be system-wide installable. Configuration files must be in plain text format and human-readable. Executable must accept commandline flags.	Implemented	✓	✓
Source code under a version control system.	Available under open source license. Free public access to the source code repository.	AUDITOR uses **Git** as a distributed version control system and **GitHub** as the Git server hosting service. See https://github.com/gnss-sdr/gnss-sdr Git is available under the GNU General Public License v2.	✓	✓
Automated documentation system.	Usable in C++, Python and VHDL source code. Available under open source license, and in all supported environments. Easily maintainable.	AUDITOR uses **Doxygen** (http://www.stack.nl/~dimitri/doxygen/) the *de facto* standard tool for generating documentation from annotated C++ sources, but it also supports other programming languages such as C, Python, and VHDL. The documentation is written within code, and is thus relatively easy to keep up to date. Doxygen can cross reference documentation and code, so that the reader of a document can easily refer to the actual code. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically. Doxygen is highly portable, and can generate documentation in HTML and PDF formats. Available under GNU General Public License v2.	✓	✓
Automated build environments.		Provided by Launchpad (https://launchpad.net/)	✗	✓
Availability of "debugging modes" and tools.		Provided by CMake.	✓	✓
Static code analysis		Provided by Coverity Scan (https://scan.coverity.com/)	✓	✓
Dynamic code analysis		Provided by Valgrind (http://valgrind.org/)	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
Definition of a source tree structure		Described in the README file at the root of the source code tree.	✓	✓
Availability of a coding style guide.		Coding guidelines and conventions can be found at: http://gnss-sdr.org/coding-style/	✓	✓
Supported processor architectures		AUDITOR's processor architecture	✗	✓
		i386 processor architecture supported by GNSS-SDR v0.0.6	✓	✓
		x86_64 / amd64 processor architecture supported by GNSS-SDR v0.0.6	✓	✓
		armhf processor architecture supported by GNSS-SDR v0.0.6	✓	✓
		arm64 processor architecture supported by GNSS-SDR v0.0.6	✓	✓
Usage of a well-established building tool	Available for all the supported processor architectures, and under an open source license.	AUDITOR uses **CMake** (https://cmake.org) as a building tool for its GNSS software defined receiver. Available under BSD 3-Clause license.	✓	✓
Availability of software package	AUDITOR's software defined receiver should be easily built and installed, ideally requiring one single line in the user terminal.	GNSS-SDR v0.0.6 is currently undergoing the acceptance process to be included as a software package (".deb") in Debian 9 (and possibly followed by Ubuntu and others).	✗	✓
Freely available, industry-proven software compilers.	Usable in the processor architectures and Operating Systems described above. Available under open source license.	AUDITOR's software defined receiver can use **GCC** (available under the GNU General Public License v3) and **LLVM** (available under University of Illinois/NCSA Open Source License)	✓	✓
Available for most popular (Unix-based) operating system distributions.	AUDITOR's software defined receiver must be compilable and executable (including the availability of all required dependencies in a given environment).	Debian 8 and above (32 and 64 bits)	✓	✓
		Ubuntu 14.04 LTS and above (32 and 64 bits)	✓	✓
		Linaro 15.03 and above	✓	✓
		Apple's Mac OS X 10.9 and above	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
Source code released under an open license.		GNSS-SDR is released under the GNU General Public License v3, as specified in the COPYING file at the root of the source code tree (as it is standard practice in the discipline). It can be checked online at https://github.com/gnss-sdr/gnss-sdr	✓	✓
Unique identifier for source code snapshots.	Every single change in the source tree (either on the reference development branch or in any other) must be uniquely identified and retrievable, keeping an annotated history of source code evolution.	Git assigns a 40 character-long identifier to every revision (i.e, specific snapshot of the status of every single file present in the repository), which is the output of the SHA-1 algorithm applied to a set of information required to recreate the full source tree. Hence, every single bit change in the source code is registered by Git, with the added benefit of *integrity* over the source code identification.	✓	✓
Availability of a Digital Object Identifier for GNSS-SDR releases	Automated and persistent DOI assignment to GNSS-SDR stable releases.	GNSS-SDR v0.0.9 DOI: 10.5281/zenodo.291371 Every new release of GNSS-SDR will obtain a new DOI. Automated DOI assignment already set-up, service freely provided by ZENODO and GitHub.	✓	✓
Quasi-linear acceleration with the number of cores/ processors		Demonstrated in [7]	✗	✓
Arbitrarily scalable receiver's software architecture.		Implemented.	✗	✓
Arbitrarily scalable configuration system.		See indicator "Flexible configuration mechanism."	✓	✓
Logging system	Possibility to set up severity levels and verbose modes for messages. Possibility to set up conditional / occasional logging. Available under open source license and for all supported platforms	AUDITOR uses **Google Glog**, Google's C++ logging system (see https://github.com/google/glog). It provides simple yet powerful APIs to various log events in the program. Messages can be logged by severity level, and the users can control logging behaviour from the command line, log based on conditionals, abort the program with stack trace when expected conditions are not met, and introduce their own verbose logging levels. Available under BSD 3-clause license.	✓	✓
Unit test software framework	Test should be easy to write for programmers, letting test writers to focus on the test *content*. Test should be easy to read for programmers. Test should be order independent. Test should be deterministic. Test should be versionable. Test should be automatic. Tests should be	AUDITOR uses **Google Test**, Google's C++ test framework (https://github.com/google/googletest), which meets all the required features. Available under BSD 3-clause license.	✓	✓

Indicator	Requirements	Current status	Status M0	Status M16
	independent and *repeatable*. Tests should be well *organized* and reflect the structure of the tested code. Tests should be *portable* and *reusable*. When tests fail, they should provide as much *information* about the problem as possible. Tests should be *fast* (less than 5-10 minutes) to execute. Testing framework available under open source license, and in all the supported environments.			
Public source code repository	• Freely accessible. • Robust hosting service. • Management tools. • Bug tracking system. • Allowing automated building and other hooks on new code changes.	Available at https://github.com/gnss-sdr/gnss-sdr.git	✓	✓
Communication channels	Public mailing list	Public mailing list	✓	✓
Documentation for users	Howtos, tutorials. Documentation is a never-ending process that continually gets feedback from users.	Available at http://gnss-sdr.org/docs/	❌	✓
Documentation for developers	Identification of a source-code based automated documentation tool. Documentation is a never-ending process that continually gets feedback from users.	AUDITOR uses **Doxygen** as the automated source code documentation tool. Doxygen is free software, released under the terms of the GNU General Public License. In-code documentation, presented in an ordered manner and generated automatically, helps to keep an updated version of the source code manual.	❌	✓

4. Conclusion

This deliverable details the design and implementation internals of AUDITOR's GNSS receiver. The hardware and software elements of the two main components of the receiver, based on D2.2 [2], have been defined. The elements devoted to the implementation and communication of the iBOGART model are detailed in a parallel deliverable also due for submissions in M16, D4.2 [4] and D5.2 [3].

The development of the RF Front-End (FE) required the design, implementation and validation of a custom RF board with two custom RF chains and additional controlling firmware. The proposed design, as defined in the D2.2 specification, receives simultaneously two different bands one fixed E1/L1 and one configurable L2 or E5a/L5.

Several issues related to EMI, clock generation/distribution and optimization of the chains gains have been identified and solved during the validation phase of the first version, v1.0. These issues and the adaptation of the FE to a more convenient form-factor compatible with the high performance ZC706 eval board has motivated the design of a new version, v2.0. This new version is currently being manufactured and its design and layouts modifications should provide a more robust FE to EMI, with clock quality improvements and better overall performance in terms of power and spurious signal rejection.

The development of the Digital Processing Platform required the design, implementation and validation of multiple complex software elements. This platform is based on a commercial available System-on-Chip (Zynq) that provides multiple ARM cores and a FPGA processor.

The GNSS-SDR module is one of the key elements that are part of the software receiver. This open-source project, maintained by CTTC partners, provides the architecture and core blocks to implement a software GNSS receiver that tackles the challenges of concurrency, efficiency, performance, portability, real-time/post-processing and extendibility. In order to accomplish this, the design decouples receiver's internal control and signal processing, while providing a flexible configuration system that allows for the required flexibility to implement different processing approaches.

A key feature of the AUDITOR software receiver is the implementation of several IP modules in VHDL language to allow the real-time efficient processing of multiple Galileo and GPS signals in an embedded platform. These high optimized modules provide a buffering, acquisition and tracking of multiple Galileo/GPS signals.

The summary of the indicators status introduced in D2.2 has been updated showing clearly the advance in the implementation of the required functionalities.

The AUDITOR GNSS receiver has recently started the integration activities where the full RF FE and software receiver are being validated. The objective is to complete the implementation of the full system showed in Figure 1.1, which includes the iBOGART model and network software based on embed and cloud services to provide high level agriculture tools and services.

5. References

[1] AUDITOR-D2.1 Architecture definition.

[2] AUDITOR-D2.2 Subsystem Specification.

[3] AUDITOR-D5.2 GNSS Network software implementation and test.

[4] AUDITOR-D4.2 Development and validation of the sequential NABS model.

[5] (AUDITOR) Part B.

[6] AUDITOR-D1.3 Test definition.

[7] 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.

[8] G. Kahn, «The semantics of a simple language for parallel programming,» *Information processing,* p. 471–475, 1974.

[9] G. Kahn, «Coroutines and networks of parallel processes,» *Information Processing,* p. 993–998, 1977.

[10] T. Parks, Bounded Scheduling of Process Networks, Berkeley, CA: Ph.D. thesis, University of California, 1995.

[11] N. M. a. T. O. T. W. Rondeau, «SIMD programming in GNU Radio: Maintainable and user-friendly algorithm optimization with VOLK,» de *Wireless Innovation Forum Conference of Wireless Communication Technologies and Software Defined Radio*, Washington, DC, 2013.

[12] J. Trimble, «Linux on Zynq, ECE 699 Hardware/SOftware Codesign,» 2015.

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

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

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

[16] 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.

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

[18] AUDITOR-D1.2 Requirement definition.

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