UNIVERSIDAD
COMPLUTENSE
MADRID

FPGA IMPLEMENTATION OF AN AD-HOC RISC-V SYSTEM-ON-CHIP FOR INDUSTRIAL IOT

IMPLEMENTACIÓN FPGA DE UN SoC BASADO EN RISC-V PARA IoT INDUSTRIAL

DANIEL LEÓN GONZÁLEZ

MSc. THESIS – MASTER'S DEGREE INTERNET OF THINGS
FACULTAD DE INFORMÁTICA – UNIVERSIDAD COMPLUTENSE DE MADRID
JULY 2020 – ORDINARY CALL

ADVISED BY:
DANIEL ÁNGEL CHAVER MARTÍNEZ
LUIS PIÑUEL MORENO

Grade/Calificación: 10

ABSTRACT

Node devices for IoT[1] need to be energy efficient and cost effective, but they do not require a high computing power in a large number of scenarios. This changes substantially in an Industrial IoT environment, where massive sensor utilization and a fast pace of events require more processing power. A custom developed node, using an efficient processor and a high performance and feature-full operating system, may balance these requirements and offer an optimal solution. This project addresses the hardware implementation, using an Artix-7 FPGA[2], of a prototype IIoT[3] node based on the RISC-V processor architecture. The project presents the implemented custom SoC[4] and the development of the necessary Zephyr OS drivers to support a proof-of-concept application, which is deployed in a star network around a custom border router. End-to-end messages can be sent and received between the node and the ThingSpeak cloud platform. This document includes an analysis of the existing RISC-V processor implementations, a description of the required elements and a detailed guide of environment configuration and steps to build the complete project.

Keywords

RISC-V, System on Chip, SoC, Zephyr, IoT, Internet of Things, FPGA, Artix 7, IIoT

RESUMEN

Los dispositivos de nodo para IoT necesitan, generalmente, ser eficientes energéticamente y tener un coste contenido, pero no precisan de una gran potencia de cómputo en un gran número de escenarios. Esto cambia sustancialmente en un entorno de IoT Industrial, donde los requerimientos sensoriales y de tiempo de respuesta precisan de una potencia de cálculo mayor. Un nodo desarrollado a medida, sobre un procesador eficiente y un sistema operativo de altas capacidades, puede balancear estos requerimientos ofreciendo una solución óptima. Este trabajo aborda la implementación hardware, sobre FPGA Artix-7, de un prototipo de nodo IIoT basado en la arquitectura de procesador RISC-V. El proyecto presenta la creación de un System-on-Chip a medida y el desarrollo de los drivers necesarios sobre el sistema operativo Zephyr para soportar una aplicación de prueba de concepto, que se despliega en una red de estrella con un rúter de borde. Mensajes de extremo a extremo pueden ser enviados y recibidos entre el nodo y la plataforma ThingSpeak en la nube. El documento incluye un análisis de las implementaciones existentes de procesadores RISC-V, una descripción de los elementos necesarios y una guía detallada de configuración de entorno y pasos para construir el proyecto completo.

Palabras clave

RISC-V, Sistema embebido, SoC, Zephyr, IoT, Internet de las cosas, FPGA, Artix-7, IIoT

[1] **IoT**: Internet of Things
[2] **FPGA**: Field-Programmable Gate Array
[3] **IIoT**: Industrial Internet of Things
[4] **SoC**: System-on-Chip

INDEX

1. Introduction

As we are about to leave behind the first six months of the decade, IoT stands as one of the main technology trends in the market, with overwhelming accomplished CAGR[5] figures of 43% and a similarly strong expected future growth. Estimated 2020 revenue for IoT is $390.000 Million and forecasted revenue by 2026 reaches $1 Billion[6] [1]. To put these figures into perspective, this is the data for three well-known industries, two in the technology field and one in the durable consumer goods industry:

- Laptops [2]: 2017 global laptop sales totaled $101.700 Million. Forecasted CAGR through 2025 is just 0.4%
- Gaming [3]: The gaming industry estimated revenue for 2020 is 164.600 Million, with a forecasted CAGR 2018-2022 of 9%.
- Automotive [4]: 2019 revenue for the top 10 automotive industry makers was $448.600 Million, with less than 0.1% growth over 2018.

Industrial IoT (IIoT) comprises all IoT deployments in manufacturing, energy, healthcare and transportation and its forecasted revenue for 2021 is $200.000 Million [5], becoming one of the key technology markets by itself.

The Industrial Internet Consortium [6], enumerates the following interest areas for IIoT:

- Intelligent warehousing
- Remote & predictive maintenance
- Tracking of transported goods
- Connected logistics
- Smart metering, smart grid
- Smart agriculture and livestock control
- Industrial security systems
- Energy consumption optimization
- Industrial heating and cooling, air conditioning
- Manufacturing monitoring
- Gas concentration and temperature monitoring in industrial environments

While energy saving is a key factor in all IoT deployments, an important requirement for most of these applications is the need for high processing power and low latency. This is especially critical in manufacturing control and security systems, where actions must be taken immediately should any situation requires it.

Creating an ad-hoc SoC, i.e.: a custom hardware embedded system, for a specific solution offers a perfect balance between three variables: processing power, cost and energy requirements. Most one-fits-all stock SoCs and complete systems come with many superfluous features that increase any number of these three variables.

[5] **CAGR**: Compound Annual Growth Rate
[6] Using Billion as in Europe: 10^{12}

Moreover, a candidate IoT node should not only be fast and energy efficient, but also have access to a feature-rich and well supported operating system enabling fast and secure development of the desired user applications.

This project[7] goal is to explore available options and build a high performance, low power IIoT SoC integrating sensors, actuators and end-to-end networking for a defined task of reading and transmitting acceleration data and switches status, and receiving information to show in a display. The project outcomes are:

- A Proof-of-Concept prototype of the SoC and its deployment, including a border router to connect to cloud services, where data is received in real time and node-triggered actions, as well as cloud-triggered actions, are performed. The overall scope of the pilot is summarized in Figure 1. Note that the border router support up to six nodes by simply enabling their addresses in its radio driver.
- A step-by-step guide for setting the development environment and replicating the project from scratch. This guide also includes the testing configuration and steps for all the elements.

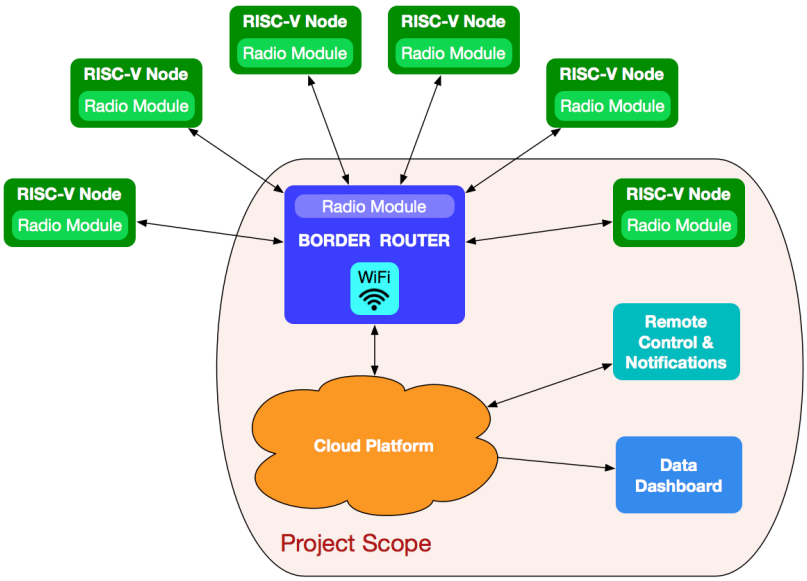


Figure 1: Project scope. Proof-of-Concept prototype environment (own elaboration)

Given the aforementioned requirements for IIoT, the selected processor architecture is RISC-V [7], an open processor architecture offering modular functionality, and the operating system is Zephyr [8], an emerging, IoT-oriented, RTOS supported by the Linux Foundation [9].
The SoC is built on a development board with a Xilinx Artix-7 [10] FPGA. An external radio transceiver is used for network communications.

[7] This project connects with the ongoing project "Desarrollo de RVfpga", a UCM Art. 83 [81] contract signed on March 2020. The project team is formed by members of the ArTeCS [66] department and Imagination Technologies [80]. The author of this thesis is part of the project as an "external person".

The SoC block diagram of the involved elements, plus the border router and the cloud services, are presented in Figure 2. This diagram will be used at the beginning of each chapter to highlight the element covered in that chapter.

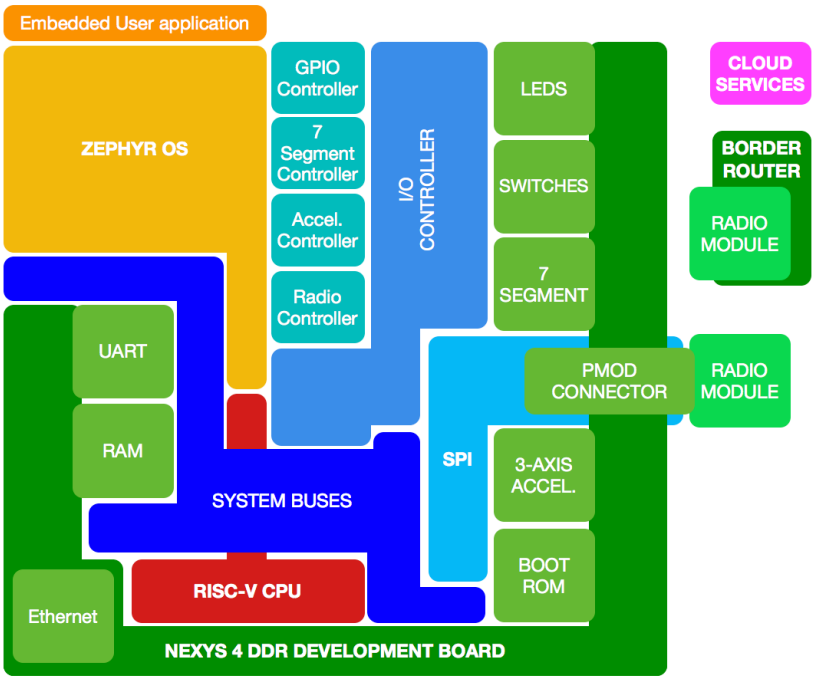


Figure 2: Block diagram of the project's SoC components plus the border router and cloud services (own elaboration)

The main elements of the project are:

- RISC-V cores and SoCs market research
- Hardware implementation, in Verilog, of the SoC extensions to support the project required hardware elements
- Integration of Zephyr OS with the SoC
- Development of low-level drivers for hardware elements
- Development of a Zephyr IIoT application
- Development of the border router using a microcontroller
- Creation of cloud services for the proof of concept

This document introduces the hardware, hardware cores and software elements used in the prototype SoC and the border router, presents a detailed step-by-step guide describing all configuration and development aspects and includes the code for the node and, also, for the border router.

2. Acknowledgements

I would like to extend my deepest gratitude to both my thesis advisors, Daniel Chaver, for his unparalleled support and practical suggestions, and Luis Piñuel, for his invaluable insights and advice. Thank you both very much, too, for inviting me to participate in this amazing project.

I also wish to thank Ignacio Gómez for helping me demonstrate the final proof of concept during the thesis defence.

I am also grateful to all the UCM FDI[8] IoT Master's teachers and classmates for creating a productive, collaborative and friendly environment, for their teachings and experiences and for the laughs we shared over the past year.

Thanks to Zvonimir Bandic, President of Chips Alliance and Senior Director at Western Digital, Olof Kindgren, maintainer of SweRVolf, at FOSSi Foundation and Robert Owen, Director Worlwide University Programme at Imagination Technologies, for their kind feedback on this MSc. thesis and their words of encouragement.

Many thanks to Olga Peñalba, Director of the Polytechnic School and Dean of the Computer Engineering degree at the Francisco de Vitoria University of Madrid (UFV) and to Carolina Rubio, Computer Engineering degree coordinator at the UFV, for adapting my working schedule to be compatible with the studies of the IoT Master's degree.

Finally, I cannot begin to express my thanks to my wife and partner-in-crime, Mónica, and my three-year-old son, Gonzalo, for their loving care and understanding during the Master's degree studies and, especially, during the period of this thesis development and writing where, notwithstanding the difficulties caused by the global Covid-19 pandemic and the three-month confinement law, they granted me the time and space to work the way I needed.

[8] **FDI**: Facultad De Informática

3. RISC-V

This chapter covers the description of the RISC-V architecture and a comparison between different RISC-V cores and SoCs. RISC-V processor sits at the core of the solution, offering an open and configurable central processing unit. Figure 3 highlights the RISC-V CPU within the node.

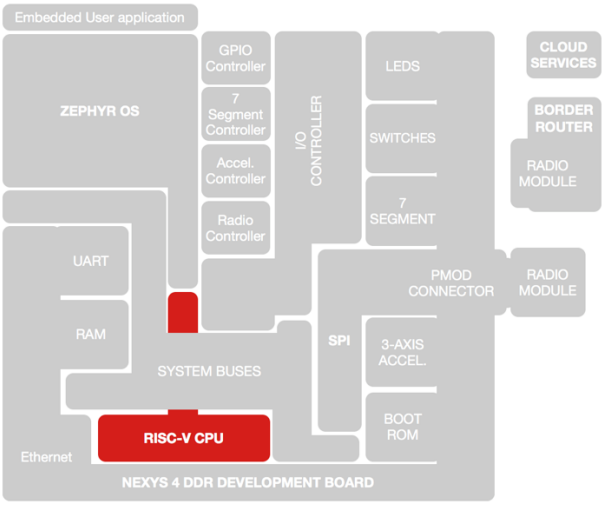


Figure 3: RISC-V CPU within the project scope (own elaboration)

3.1 RISC-V Instruction Set Architecture

RISC-V is an ISA[9] that was created in 2011 in Par Lab, Bekerley, aiming to become a "Universal ISA" for processors used in different applications, from constrained, low resources IoT devices up to supercomputers. It established five principles to achieve this goal:

- It must be compatible with a wide range of software packages and programming languages.
- Its implementation must be feasible in all technology options, from FPGAs to ASICs[10] and even new technologies yet to come.
- It must be efficient in all different micro-architectures scenarios, including those implementing microcode or hardwired control, in-order or out-of-order pipelines, different kinds of parallelism, etc.
- RISC-V must be able to be tailored to specific tasks to achieve the required maximum performance without any drawbacks imposed by the ISA itself.
- Its base instruction set must be stable and long lasting, offering a common and solid framework for developers.

It is an open standard, in fact, the specification is public domain, and it is managed by the "RISC-V Foundation" since 2015, a non-profit organization promoting the

[9] **ISA**: Instruction Set Architecture
[10] **ASIC:** Application-Specific Integrated Circuit

development of hardware and software for RISC-V architectures. In 2018, the RISC-V Foundation started an ongoing collaboration with the Linux Foundation and, very recently, in March 2020, RISC-V International Association was incorporated in Switzerland, dissipating any concern the community may have had about future openness of the standard.
Currently, the RISC-V Foundation and RISC-V international are supported by up to 200 key players from the technology, education and services industries, including Microchip, NXP, Samsung, Qualcomm, Micron, Google, Alibaba, Hitachi, Nvidia, Huawei, Western Digital, Imagination Technologies and the Complutense University as a community member.

RISC-V is one of the few ISAs, and probably the only relevant, created in the past 10 years or even in this century. Besides being an open standard, it presents yet another major difference when compared to traditional ISAs such as x86 or ARM: the RISC-V ISA is *modular* instead of *incremental*.
An incremental architecture, the preferred approach for all key players, implements all past instructions in each new processor, even those that are tagged as "obsolete". This is done to ensure binary compatibility with all previous programs. As an example, x86, that began with 80 instructions, has now over 1300, or 3600 if considering all different opcodes in machine code. This results in processors implementing x86 needing a very high number of gates and a higher instruction length, since most of the shorter opcodes, or low word instructions, are already used.
On the other hand, RISC-V presents its architecture as *modular*. Its base core comprises four different implementations, as shown in Table 1, and it is able to execute all programs. These base or core ISAs are fixed and will not change in the future, allowing for a common ground in software and compiler development and low gate-count implementations, which is a key factor in energy constrained devices used in IoT. On top of this core module, new instructions packages can be added in hardware allowing for specific tasks such as floating point, multipliers and dividers or vector operations, among other options. These specialized hardware instructions are also included in the standard and they are known to the compilers, so enabling the right options in the compiler will allow for a targeted binary code generation.
Each of these extension modules is identified by a letter that must be added to the core ISA to represent the hardware capabilities of the implementation.

Name	Functionality
RV32I	Integer Instruction set. 32bits and 32 registers
RV32E	Integer Instruction set for embedded devices. 32 bits and 16 registers
RV64I	Integer Instruction set. 64bits and 32 registers
RV128I	Integer Instruction set. 128bits and 32 registers

Table 1: RISC-V Base ISAs (own elaboration, [11])

The standard extension modules are shown in Table 2. The letter G, as in "general", is used to denote the inclusion of all MAFD extensions, as a shorter way to do it since it is expected that most implementations will include these. Note that an organization or an individual may develop its proprietary extensions using opcodes that are guaranteed not to be used in standard modules. This allows third-party implementations in a faster time-to-market. Two examples are the Crypto extensions of PQField and the V5 extensions of Andes.

	Letter	Functionality
G	M	Integer multiplication and division
	A	Atomic Instructions
	F	Single precision floating point
	D	Double precision floating point
	Q	Quad precision
	L	Decimal floating point
	C	Compressed instructions (16 bit instructions)
	B	Bit manipulation
	J	Dynamically translated languages
	T	Transactional memory
	P	Packed SIMD instructions
	V	Vector operations
	N	User level interrupts
	H	Hypervisor

Table 2: RISC-V Standard ISA extensions (own elaboration, [11])

Therefore, a 64-bin RISC-V implementation, including all four general extensions, plus bit manipulation and user level interrupts is referred to as RV64GBN.

All these modules are covered in the *unprivileged* or *user* specification. The RISC-V foundation also covers a set of requirements and instructions for *privileged* operations, required for running general-purpose operating systems and attaching specific hardware. This document is currently at its 1.1 release and specifies three different privilege levels and a reserved level as shown in Table 3. The coverage of the privileged mode exceeds the scope of this thesis, as the Zephyr operating system runs in the basic level (User/Application), not requiring privileged modes.

Level	Encoding	Name	Abreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

Table 3: RISC-V privilege levels (own elaboration, [11])

RISC-V is a very good positioned option for IoT deployments, based on the openness of the standard with no license fees, the ability to tailor the ISA to the specific application requirements, the overall high performance per Megahertz of clock frequency (MHz), critical for battery-powered IoT applications, and the support of major software tools and languages as well as from the key players in the market.

3.2 RISC-V available cores

All known available cores are listed at the RISC-V website [7]. Using this information as a base, and extending it with additional Western Digital Cores, Table 4 presents a summary of the commercial licensed cores, including hardware definition language and privileged/unprivileged coverage of the standard.

Core Name	Priv. Spec.	User Specification	BITS	HDL	Company
A25	1.11	RV32GCP + SV32 + Andes V5 ext.	32	Verilog	Andes
A25MP	1.11	RV32GCP + SV32 + Andes V5 ext. + Multi-core	32	Verilog	Andes
BI-350	1.10	RV32IMAFC + multi-core	32	SystemVerilog	CloudBear
Bk3	1.10	RV32EMC / RV32IMFC	32	Verilog	Codasip
Bk5	1.10	RV32IMFC / RV64IMFC	32	Verilog	Codasip
BM-310	1.10	RV32IMC	32	SystemVerilog	CloudBear
D25F	1.11	RV32GCP + Andes V5 ext.	32	Verilog	Andes
E2	1.11	RV32I(E)MAFC	32	Verilog	SiFive
E3	1.11	RV32I(E)MAFDC	32	Verilog	SiFive
E7	1.11	RV32I(E)MAFDC	32	Verilog	SiFive
N22	1.11	RV32IMAC/EMAC + Andes V5/V5e ext.	32	Verilog	Andes
N25F	1.11	RV32GC + Andes V5 ext.	32	Verilog	Andes
Pluto	1.11	RV32IMC / RV32EMC + Crypto Functions	32	Verilog	PQShield
RV32EC_FMP5	Custom	RV32EC	32	SystemVerilog	IQonIC Works
RV32EC_P2	1.11	RV32EMC / RV32IMC	32	SystemVerilog	IQonIC Works
RV32IC_P5	1.11	RV32MNAC	32	systemVerilog	IQonIC Works
XuanTie E902	1.10	RV32EMC / RV32IMC	32	Verilog	T-Head (Alibaba)
AX25	1.11	RV64GCP + SV39/48 + Andes V5 ext.	64	Verilog	Andes
AX25MP	1.11	RV64GCP + SV39/48 + Andes V5 ext. + Multi-core	64	Verilog	Andes
BI-651	1.10	RV64GC + multi-core	64	SystemVerilog	CloudBear
BI-671	1.10	RV64GC + multi-core	64	SystemVerilog	CloudBear
Bk7	1.10	RV64IMAFDC	64	Verilog	Codasip
NX25F	1.11	RV64GC + Andes V5 ext.	64	Verilog	Andes
S2	1.1	RV64GC	64	Verilog	SiFive
S5	1.11	RV64GC	64	Verilog	SiFive
S7	1.12	RV64GC	64	Verilog	SiFive
SCR7	1.10	RV64GC Multicore	64	SystemVerilog	Syntacore
U5	1.11	RV64GC	64	Verilog	SiFive
U7	1.11	RV64GC	64	Verilog	SiFive
XuanTie C910	1.10	RV64GCV + SV39 + ISA Extension + Memory model Extension + multi-core & multi-cluster(16 core max)	64	Verilog	T-Head (Alibaba)
SCR3	1.10	RV[32/64] EMCA / IMCA	32/64	SystemVerilog	Syntacore
SCR4	1.10	RV[32/64] IMCFDA	32/64	SystemVerilog	Syntacore
SCR5	1.10	RV[32/64] IMCFDA	32/64	SystemVerilog	Syntacore

Table 4: Commercially licensed RISC-V Cores (own elaboration, [7])

For this thesis, the most interesting cores are those with an open source license. Table 5 summarizes the open-source RISC-V cores, including their license type. The requirements of this project do not include privileged modes. Also, a 32 bit architecture is enough, with 16 bit instructions as a plus, and floating point would impose an unnecessary hardware overhead. Our application would benefit from 32 registers instead of 16, so candidate cores are those RV32I with no privileged specification implementation and C (Compressed 16 bits) instructions as a plus.

Core Name	Priv. Spec	User Specification	BITS	HDL	Company	Licence
DarkRISCV	-	RV32I	32	Verilog	Darklife	BSD
Kronos	-	RV32I	32	SystemVerilog	Sonal Pinto	Apache 2.0
Maestro	-	RV32I	32	VHDL	João Chrisóstomo	MIT
OPenV/mriscv	-	RV32I	32	Verilog	OnChipUIS	MIT
ReonV	-	RV32I	32	VHDL	Lucas Castro	GPL V3
RPU	-	RV32I	32	VHDL	Domipheus Labs	Apache 2.0
RSD	-	RV32I	32	SystemVerilog	rsd-devel	Apache 2.0
SERV	-	RV32I	32	Verilog	Olof Kindgren	ISC
Tinyriscv	-	RV32I	32	Verilog	Blue Liang	Apache 2.0
MR1	-	RV32I	32	Spinal HDL	Tom Verbeure	Unlicensed
Instant SOC	-	RV32IM	32	VHDL	FPGA Cores	Apache 2.0
ORCA	-	RV32IM	32	VHDL	VectorBlox	BSD
Taiga	-	RV32IMA	32	SystemVerilog	Simon Fraser Univ.	Apache 2.0
Reve-R	-	RV32IMAC	32	CLD	Gavin Stark	Apache 2.0
RI5CY	-	RV32IMC	32	SystemVerilog	ETH Zurich, Bologne Univ.	Solderpad HW v. 0.51
Riscy Processors	-	RV32IMC	32	BlueSpec SV	MIT CSAIL CSG	MIT
SweRV EH1	-	RV32IMC	32	SystemVerilog	Western Digital Corp.	Apache 2.0
SweRV EH2	-	RV32IMC	32	SystemVerilog	Western Digital Corp.	Apache 2.0
SweRV EL2	-	RV32IMC	32	SystemVerilog	Western Digital Corp.	Apache 2.0
VexRiscv	-	RV32IMC	32	Spinal HDL	SpinalHDL	MIT
PicoRV32	Custom	RV32IMC / RV32EMC	32	Verilog	Clifford Wolf	ISC
Lizard	-	RV64IM	64	PyMTL	Cornell CSL BRG	BSD
SCR1	1.10	RV32EMC / RV32IMC	32	SystemVerilog	Syntacore	SHL 2.0
Minerva	1.10	RV32I	32	nMigen	LambdaConcept	BSD
HummingBird E200	1.10	RV32IMAC	32	Verilog	Bob Hu	Apache 2.0
SSRV	1.10	RV32IMC	32	Verilog	Risclite	Apache 2.0
RiscyOO	1.10	RV64IMAFD	64	BlueSpec SV	MIT CSAIL CSG	MIT
Roa Logic RV12	1.10	RV32I / RV64I	32/64	SystemVerilog	Roa Logic	Non - Commercial
biRISC-V	1.11	RV32IM	32	Verilog	UltraEmbedded	Apache 2.0
RV01	1.11	RV32IM	32	VHDL	Stefano Tonello	LGPL
Ibex	1.11	RV32IMC / RV32EMC	32	SystemVerilog	lowRISC	Apache 2.0
Shatki	1.11	RV64IMAFDC	64	BlueSpec SV	IIT Madras	BSD
rocket	1.11	Multiple	32/64	Chisel	SiFive, UCB Bar	BSD
Freedom	1.11-draft	RV64G	64	Chisel	SiFive	BSD
Ariane	1.11-draft	RV64GC	64	SystemVerilog	ETH Zurich, Bologne Univ.	Solderpad HW v. 0.51
Boom	1.11-draft	RV64GC	64	Chisel	UC Berkeley	BSD

Table 5: Open Source RISC-V Cores (own elaboration, [7])

Out of the list, there are eight cores meeting the aforementioned criteria, but only five of them use the most common HDL[11], Verilog or SystemVerilog, that extends Verilog to

[11] **HDL**: Hardware Description Language

include a HVL[12]. One of them is maintained by a single individual, making its future evolution or bug fixing difficult. Therefore, the candidates are:

RI5CY - SweRV EH1 - SweRV EH2 - SweRV EL2

The comparative performance per MHz of these four cores, among other open-source RISC-V implementations is shown in Figure 4 below.

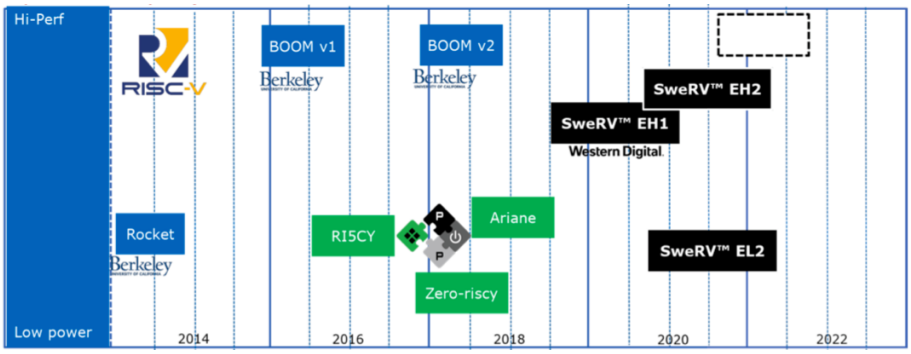


Figure 4: Date of introduction and performance of open source RISC-V Cores [12]

3.2.1 RI5CY

Ri5CY is a RISC-V core developed by the Federal Polytechnic University of Zurich and the Bologne University. It dates back to 2016 and it implements the RV32IMC user specification. Its Github repository [13] is being actively updated. Since February, 2020, it is a part of the OpenHW group, a non-profit global organization promoting open-source cores and related intellectual property, tools and software. The license is a solderpad HW license 0.51, which is a variation of the Apache 2.0 license, but not endorsed by the Apache Foundation.
The core features a 4-stage pipeline and may implement the floating point standard extension as an option, although this is maintained in a different repository. The user manual [14] latest update was in December 2019, but it does not offer much information about the internal structure nor its performance, that appears to be around 3.8 Coremarks according to the data in Figure 4 above.

3.2.2 Western Digital Cores – SweRV

All three SweRV cores have been developed by Western Digital Incorporated (WD), one of the two founding premier partners of RISC-V International. They all have an Apache 2.0 license. WD is a NASDAQ traded company, founded in 1970 and it had a US$20.65 American billion revenue in 2018. It employs over 60.000 professionals. Among different technology products, it is one of the largest hard drive manufacturers worldwide.
The WD's RISC-V web page [15] outlines the three available cores as shown in Table 6.

[12] **HVL**: Hardware Verification Language

Core Name	RISC-V Type	Pipeline Stages	Threads	Size @ TSMC	CoreMarks/Mhz
SweRV Core EH1	RV32IMC	9- dual issue	Single	.11mm @ 28nm	4.9
SweRV Core EH2	RV32IMC	9- dual issue	Dual	.067 @ 16nm	6.3
SweRV Core EL2	RV32IMC	4- single issue	Single	.023 @ 16nm	3.6

Table 6: Overview of WD's SweRV cores [15]

Western Digital is a platinum member of Chips Alliance [16], a Linux Foundation [9] project for developing high-quality hardware designs. Some other relevant members are Google, Intel, SiFive, Alibaba, Samsung and Yale University. Western Digital also intends to use these cores in soon to be released products from the company, guaranteeing a dynamic product evolution.
Out of the three cores, EH1 is preferred for several reasons:

- It presents a high performance/MHz, needed in real-time industrial IoT applications, while keeping a simple unique thread structure, minimizing power requirements.
- Its Apache 2.0 license offers a wide range of usages with no fees
- It has extensive support from Chips Alliance and Western Digital, making it a future-proof solution.

This decision sits well with the available development platform too. The Digilent Nexys 4 DDR is precisely the reference design platform for the testing of the core as shown in the Figure 5. Note that the Nexys 4 DDR is fully compatible with the Nexys A7, so both may be alternatively used.

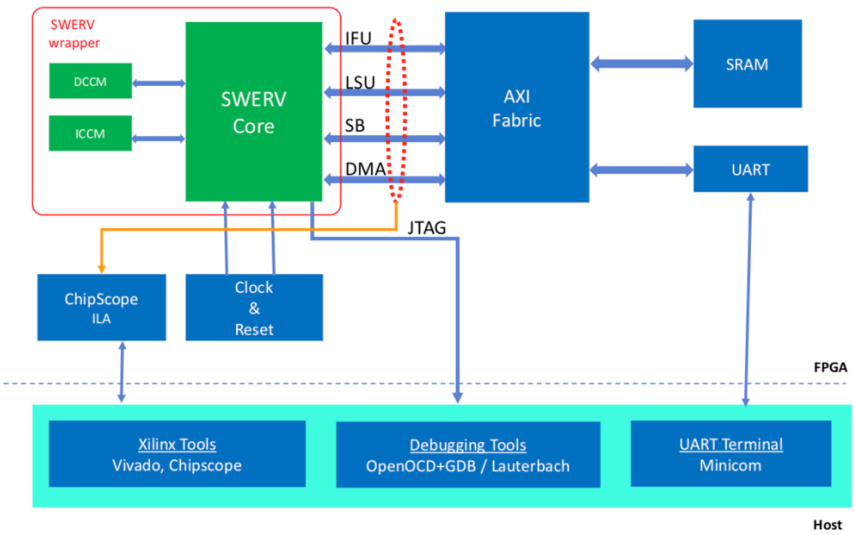


Figure 5: SweRV EH1 Nexys-4 DDR Reference design [17]

3.2.2.1 SweRV EH1

SweRV EH1 is a 9-stage dual issue, in-order, pipeline implementation of the user specification RV32IMC, with extensive documentation and activity at the repository

[18]. The 113-page programmer's reference manual [19], updated on May, 15th 2020, describes in detail all aspects of the core, from its structure to timing information and memory mappings. SweRV EH1 performance per MHz is shockingly high, twice as much as an ARM Cortex A8 and even surpassing an ARM Cortex A15.

Figure 6 shows a comparative performance chart of different cores and processors.

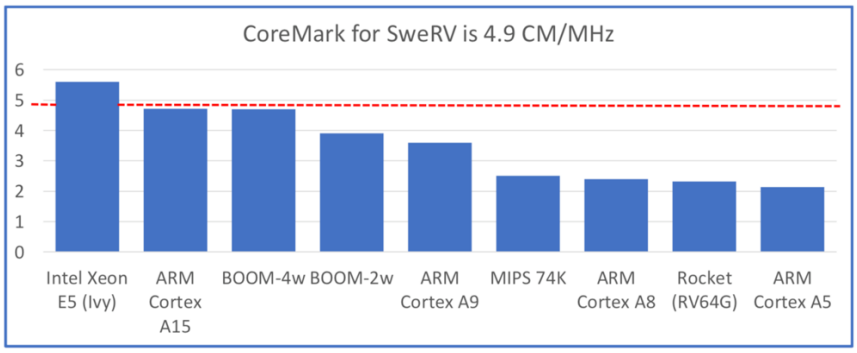


Figure 6: Benchmark comparison per thread and MHz [12]

SweRV EH1 adds some components to the core in what WD calls "*Core Complex*". These components allow debugging via JTAG[13], following the RISC-V debug specification [20], implement a Programmable Interrupt Controller (PIC), manages the bus communication and the access to program and data memories, implementing a modified Harvard architecture. Figure 7 shows the core complex for SweRV EH1.

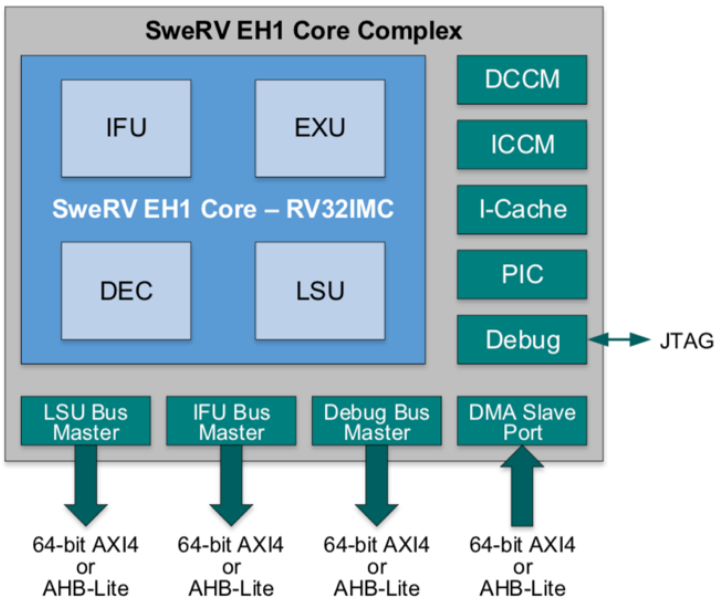


Figure 7: SweRV EH1 Core Complex [18]

The 9-stage pipeline implements a dual-issue pipe for general operations, adding a third pipe for multiply operations (M extension). The divider is left out of the pipe since it

[13] **JTAG**: Join Test Action Group. The acronym usage is extended to name standard debug and programming interfaces.

takes 34 cycles to complete a single operation. Figure 8 shows the SweRV pipeline stages and their paths.

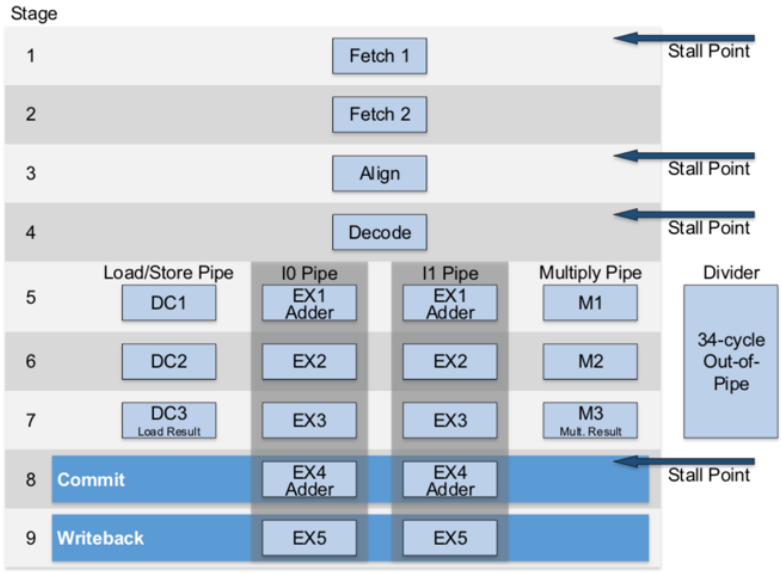


Figure 8: SweRV EH1 9-stage Dual-issue pipeline with Out-of-pipe divider [18]

3.3 RISC-V available SoCs

The RISC-V web page also shows a list of readily available SoCs and their development boards. While using a market-available SoC is a possibility, developing a custom SoC for a specific purpose leads to better performance per Euro, since no unused hardware is implemented and thus lower FPGAs or simpler ASICs can be used. Additionally, no license fees are to be paid when deploying a solution, therefore lowering the CAPEX[14] of the project. Table 7 shows the publicly available open-source RISC-V SoC Platforms.

Platform Name	Core	User Specification	Priv. Spec.	Company	Licence
KRZ	Kronos	RV32I	-	Sonal Pinto	Apache 2.0
Hero	RI5CY	RV32IMC	-	ETH Zurich, Bologne Univ.	Solderpad HW 0.51
OpenPULP	RI5CY	RV32IMC	-	ETH Zurich, Bologne Univ.	Solderpad HW 0.51
PULPino	RI5CY	RV32IMC	-	ETH Zurich, Bologne Univ.	Solderpad HW 0.51
PULPissimo	RI5CY	RV32IMC	-	ETH Zurich, Bologne Univ.	Solderpad HW 0.51
Riscy	RV64I	RV64I	-	Aleksandar Kostovic	MIT
LowRISC	lowRISC	RV64IM	-	lowRISC	BSD
SCR1 SDK	SCR1	RV32EMC / RV32IMC	1.10	Syntacore	SHL 2.0
Icicle	RV32I	RV32EMC / RV32IMC	1.11	Graham Edgecombe	ISC
MIV RV32IMA L1 AHB	rocket	RV32IM	1.11	Microchip	Apache 2.0
MIV RV32IMA L1 AXI	rocket	RV32IMA	1.11	Microchip	Apache 2.0

[14] **CAPEX**: Capital Expense. Money put upfront to acquire assets.

MIV RV32IMAF L1 AHB	rocket	RV32IMAF	1.11	Microchip	Apache 2.0
Rocket Chip	rocket	Multiple	1.11	SiFive / UCB Bar	BSD
Chipyard	Rocket/Boom	RV64GC	1.11-draft	UCB Bar	BSD
ESP	Ariane	RV64GC	1.11-draft	SLD/Columbia Univ.	Apache 2.0
OpenPiton + Ariane	Ariane	RV64GC	1.11-draft	ETH Zurich, Bologne Univ.	Solderpad HW 0.51
PicoSoC	PicoRV32	RV32IMC / RV32EMC	custom	Clifford Wolf	ISC
Raven	PIcoRV32	RV32IMC / RV32EMC	custom	efabless.com	ISC
SweRVolf	SweRV EH1	RV32IMC	Limited 1.1	Chips Alliance	Apache 2.0

Table 7: Open-source RISC-V (own elaboration, [7])

Besides SweRVolf, addressed in a different chapter of the document, the only SoC platforms meeting the initial decision criteria are Hero, OpenPULP, PULPino and PULPissimo. All SoC's are part of the PULP Platform [21], developed by the Federal Polytechnic University of Zurich and the Bologne University. These are based on three different cores: Ibex, RI5CY and Ariane. Ibex is a two-stage pipeline, low performance core, while Ariane is a RISC64 6-stage pipeline implementation. Details on RI5CY are presented above in the RISC-V Available cores chapter.

All SoCs offer adequate documentation and SystemVerilog sources on GitHub repositories. They implement several features as shown in Table 8, however, PULP single core implementations have a lower performance per MHz than SweRV EH1 and going ahead with one of these SoCs would require deleting most of the unused functionality in order to fit the SoC in the smallest FPGA or ASIC possible, without any extra benefit such as an Ethernet implementation.

Platform	Features
Hero	Multi cluster configurations of OpenPULP
OpenPULP	8-core cluster, SPI[15], I2S[16], Camera Interface, I2C[17], UART[18], JTAG, Boot ROM, Interrupt, Timers, Hardware Processing Engines support, GPIO[19]
PULPino	Single core, SPI, I2S, I2C, UART, JTAG, SPI Slave, Boot ROM, Interrupts, Timer, Hardware processing Engines support, GPIO, FLL control. Available as ASIC or FPGA
PULPissimo	Single Core, SPI, I2S, I2C, UART, JTAG, SDCard Input Output, Camera Interface, Clock generator, Timer, DMA[20], Hardware Processing Engines support, GPIO

Table 8: PULP Platform SoCs main features (own elaboration, [21])

[15] **SPI**: Serial Peripheral Interface. A common, 4-wire, protocol for chip communications by Motorola
[16] **I2S**: Integrated InterChip Sound. Standard for digital audio transmission
[17] **I2C**: Inter Integrated Circuit. A common, 2-wire, protocol for chip communications by Philips
[18] **UART**: Universal Asynchronous Receiver Transmitter. Standard Serial port communications
[19] **GPIO**: General Purpose Input Output
[20] **DMA**: Direct Memory Access

Figure 9 shows an overview of all these cores. OpenPULP is not present in the image, but lies alongside Fulmine and Mr. Wolf.

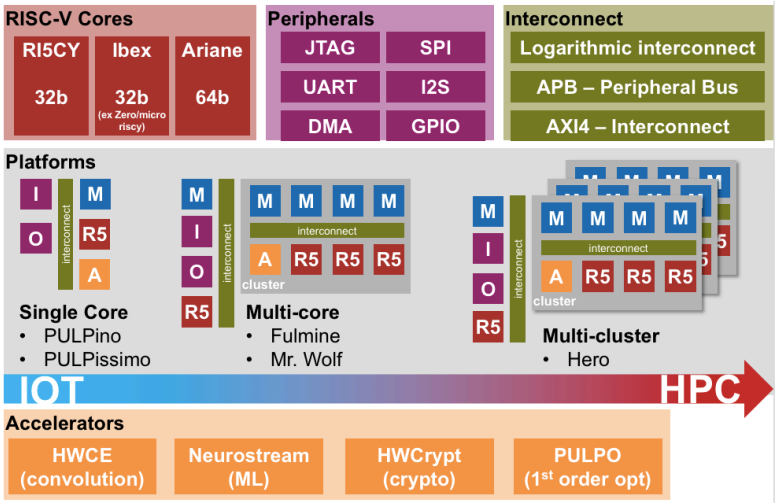


Figure 9: PULP Platform [21]

Out of this state-of-the-art analysis, two different paths are open for developing the project SoC:

- Use a RISC-V core and develop all surrounding buses and control elements.
- Use an already build SoC, as minimalist as possible, and extend it with the requested components for the project.

While the first option is certainly enticing, and will most likely result in a tighter and better integrated SoC, this project schedule demands that this part is jumpstarted by using an already built, functionality-limited SoC. SweRVolf [22], a Chips Alliance SoC that leverages the high-performance SweRV EH1 core, clearly meets the project's demands and it's the selected base SoC on which the remaining components are plugged to complete the ad-hoc SoC. Next chapter covers SweRVolf Core in detail.

4. SwerVolf RISC-V Core

This chapter presents the SweRVolf Core, the base SoC that this project extends to the required ad-hoc SoC. The elements included in the SweRVolf Core are highlighted in Figure 10. Note that many of these elements need adaptations to accommodate and register the new components that are added during the project. Detailed information on these changes and additions is presented in the Project development chapter.

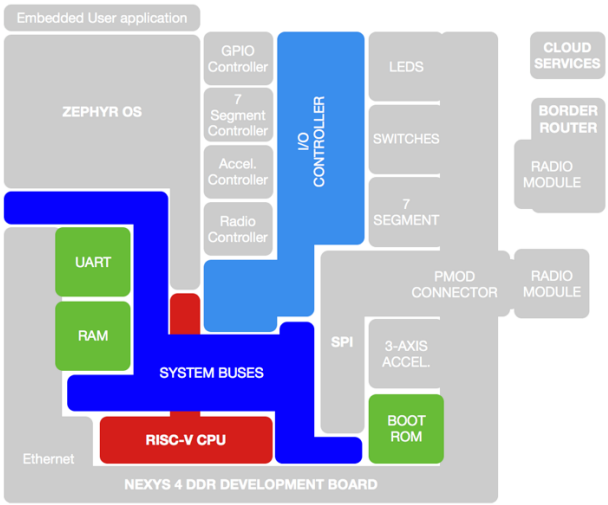


Figure 10: SweRVolf Core situation within the project scope and affected HW elements in green (own elaboration)

SweRVolf [22] RISC-V Core is a Chips Alliance SoC developed by Olof Kindgren, from Qamcom [23] and FOSSi Foundation [24]. It builds around the SweRV EH1 core [18], using the FuseSoC [25] tools from OpenRISC [26], also maintained by Mr. Kindgren, who is also part of Chips Alliance. This project uses SweRVolf version 0.6.

SweRVolf main goal is to jump-start ad-hoc SoC implementations by adding a set of commonly used peripherals, buses and facilities around the SweRV EH1 core. Figure 11 illustrates the overall structure of SweRVolf, that includes a debug module interface and Boot ROM through a dedicated SPI, GPIO and UART through a Wishbone [27] interface.

Peripherals connected to the Wishbone interconnect interface are memory mapped according to the diagram presented in Figure 12. Note that, although the SoC does not include a memory controller, addresses from 0x00000000 to 0x07FFFFFF, representing 128MB of memory, are allocated for RAM.
SweRVolf not only implements this SoC, which covers most of the required peripherals (notably missing is Ethernet) for the project, but it goes even further by providing a Digilent Nexys A7 board integration, a rebranded board to the Digilent Nexys 4 DDR board available at the UCM, so no changes are needed.

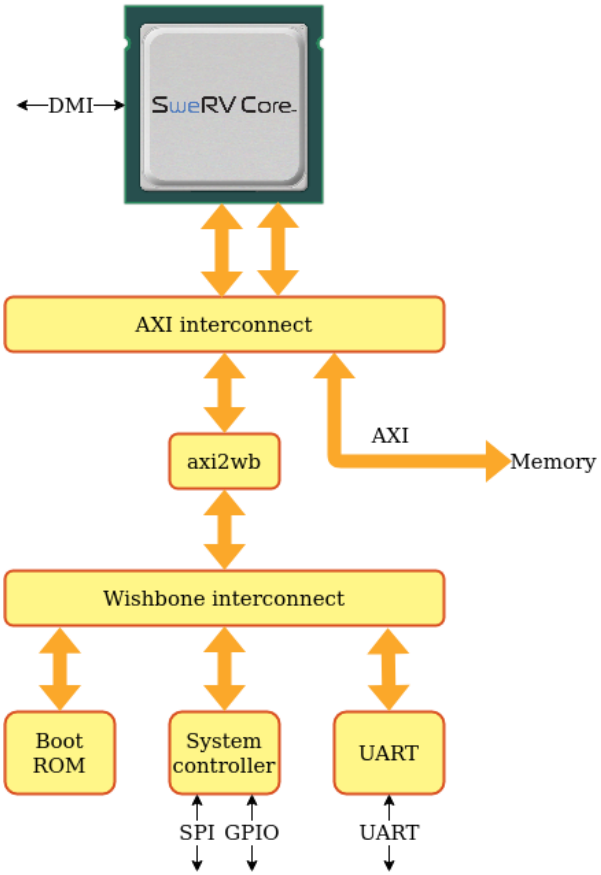


Figure 11: SweRVolf Structure [22]

Core MAP	Memory Addresses
RAM	0x00000000 - 0x07FFFFFF
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x80001FFF
UART	0x80002000 - 0x80002FFF

System Controller peripherals

Peripheral	Addresses
GPIO	0x80001010 - 0x80001017
SPI	0x80001040 - 0x80001060

Figure 12: Relevant memory mappings for SweRVolf (own elaboration)

This Nexys-extended core adds four important components.

- A LiteDRAM [28] memory core for managing the Nexys 128MB of DDR RAM.
- GPIO Ports connected to the switches and leds of the Nexys board
- SPI connected to the SPI Flash memory chip, allowing a RISC-V program to boot from this memory.
- JTAG and UART connected to the USB controller, so all programming, debugging and UART communication can be done through the serial-to-usb controller of the Nexys board.

A diagram of these connections is shown in Figure 13. Using this extended version of SweRVolf provides a very straightforward basic integration with the Digilent Nexys 4 DDR board.

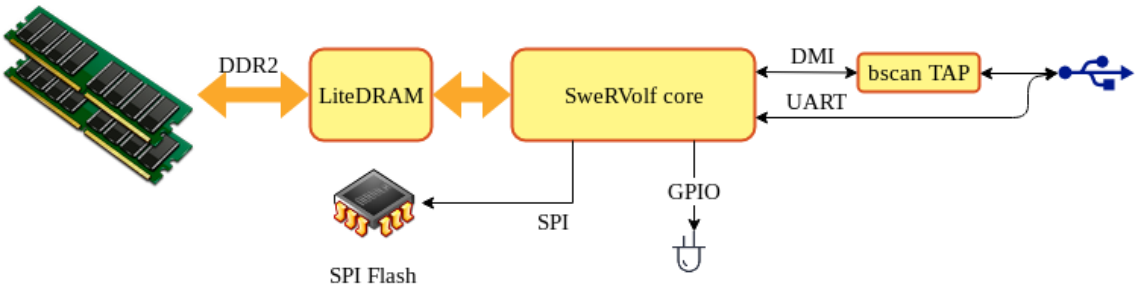


Figure 13: SweRVolf Nexys extension [22]

During the development of this thesis project, the GPIO is disconnected, including a more advanced core through the Wishbone interface at a different memory space. Also an additional dedicated SPI peripheral is implemented, in order to access the accelerometer, and a generic SPI interface is also implemented to interface to the radio module.

The combined final SoC also includes some other peripherals added by the ArTeCS team working in an Art. 83 project between UCM and Imagination; in particular, a timer and a 7-segment controller, of which some of the software developed for this project make use of.

5. Test platform and hardware parts

This chapter describes key hardware elements and additional cores used in the project, from the physical development kit to the cores described in hardware and synthesized to run in the FPGA. It also covers the radio hardware and border router platform, as well as the unimplemented ethernet core and its related physical layer transceiver.

5.1 Nexys 4 DDR development kit

The SoC development board used during this work is a Digilent Nexys 4 DDR. It's highlighted in Figure 14.

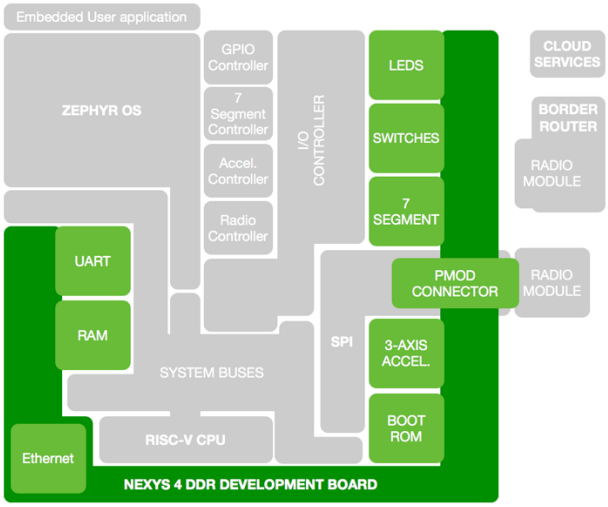
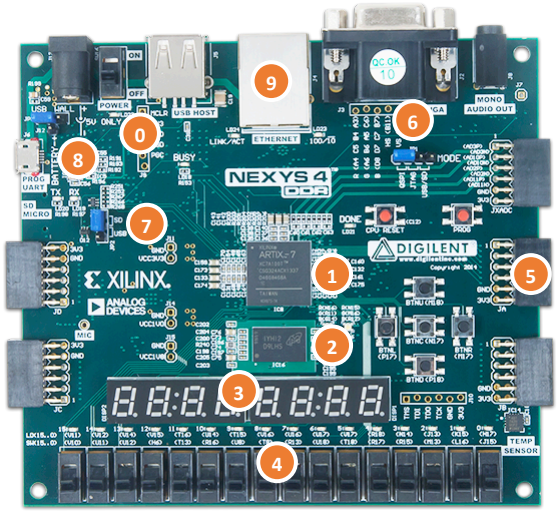


Figure 14: Nexys 4 DDR development board within the project scope (own elaboration)

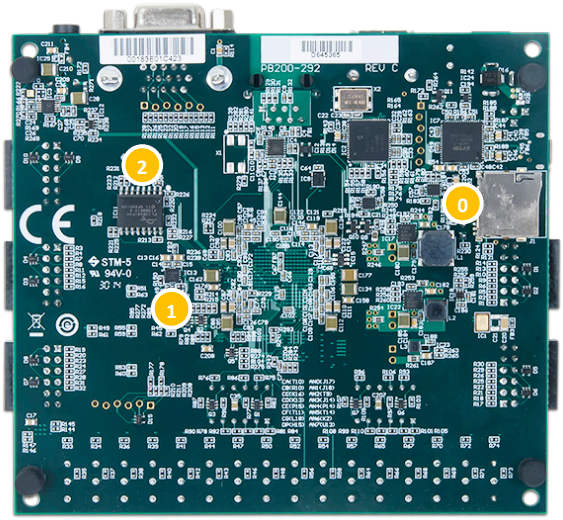
This board was provided by the thesis advisors. The board represents an incremental update (mainly memory technology) over the Nexys 4 and it has recently been rebranded as the Nexys A7. The Nexys A7 has a price of $265 and it is a recommended trainer board for ECE[21] curriculum. Digilent provides an extensive reference manual [29] on this board.

Figure 15 shows the board, pointing out the relevant elements to this project.

[21] **ECE**: Electrical and Computer Engineering



0. Power Switch
1. Artix-7 FPGA
2. 128MB DDR RAM
3. 8 x 7-Segment display
4. 16 x switches and 16 x Leds
5. PMOD A external connector
6. Load FPGA source selection
7. Serial Boot Mode SD/USB
8. Micro USB Connector
9. Ethernet 100 baseT connector



0. MicroSD slot
1. ADXL362 3-axis accelerometer
2. 128Mbit SPI Flash Memory

Figure 15: Digilent Nexys 4 DDR FPGA Trainer board [29] and key elements (own elaboration)

The Nexys 4 DDR board may be powered from a 5V wall wart or from the microUSB connector. A Microchip PIC24 microcontroller manages the loading process into the FPGA, making this board a very user friendly option. It may load the desired configuration to the FPGA from four different sources: a FAT32[22] formatted MicroSD card, a FAT32 formatted USB pendrive, the internal flash memory or a JTAG interface. It is based on a Xilinx Artix-7 FPGA with the following main features:

- 15.850 Logic slices of four 6-input LUTs[23] and 8 flip-flops.
- 4.860 Kbits of total block RAM
- 6 CMTs[24] consisting of a Mixed Mode Clock Manager and a Phase Locked Loop.
- 170 User input-output pins
- 450 MHz internal clock frequency

The board integrates with Xilinx Vivado Design Suite [30], providing constraint files for all hardware connections.

[22] **FAT32**: File Allocation Table 32 bit FileSystem. A proprietary but pervasive filesystem from Microsoft.
[23] **LUT**: Look-Up Table. A logic truth table determining the logic function
[24] **CMT**: Clock Management Tile

Besides the relevant components stated above, the trainer board comprises a plethora of sensors and connectors, including a microphone, an audio jack, a VGA[25] port, USB host port, RGB-Leds, Thermometer, PWM[26], and more.

5.2 ADXL362 accelerometer

The accelerometer is the main sensor used by the PoC[27] prototype node. A tiny ADXL362 accelerometer [31], from Analog Devices is one of the available peripherals in the Nexys 4 DDR board. Figure 16 shows the accelerometer situation within the project scope.

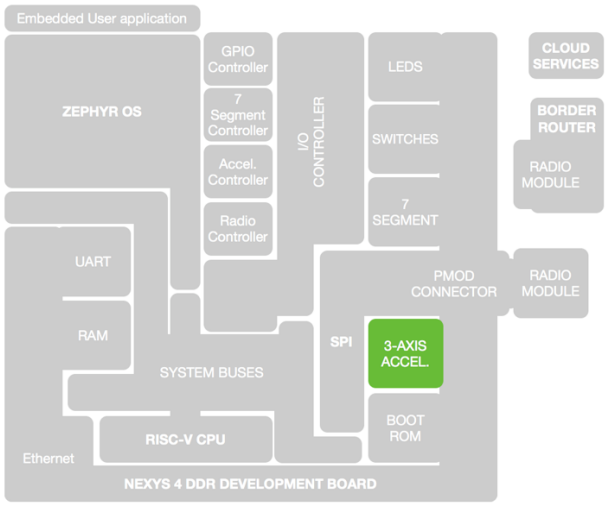


Figure 16: ADXL362 Accelerometer within the project scope (own elaboration)

The ADXL362 is a three-axis G-force configurable (2G, 4G or 8G) digital output MEMS[28] accelerometer accessible through a 4-wire SPI interface with either 12-bit high-accuracy resolution or 8-bit fast resolution. It has a temperature sensor, mainly used for calibration.
It offers three operating modes: standby, wake-up and measurement. This project will use the later for continuous tracking of the tilt of the device. In this mode, measures may be taken from 12,5Hz to 400Hz, with different power requirements. Out of the ranges that the accelerometer can be configured with, this project uses the lowest one, from -2g to +2g. Even though it can be programmed with advanced activity events and interrupts, the project requirement is only a basic polling of the acceleration data.

The device operates in SPI Slave mode, SPI Mode 0, and accepts multi-byte commands, of which the project will use 0x0A, write register command, and 0x0B, read register command.

Figure 17 and Figure 18 show the waveform for a read command and write command respectively.

[25] **VGA**: Video Graphics Array. Standard for analog PC video signals
[26] **PWM**: Pulse Width Modulation
[27] **PoC**: Proof-of-Concept
[28] **MEMS**: Micro Electrical Mechanical System

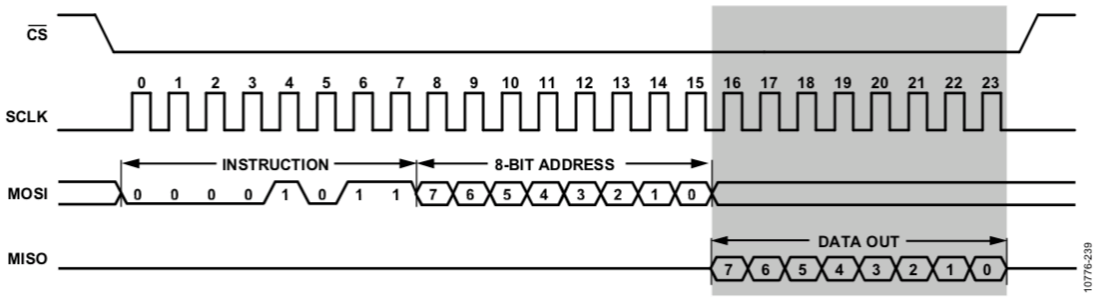


Figure 17: ADXL Read Register Command (0x0B) waveform [31]

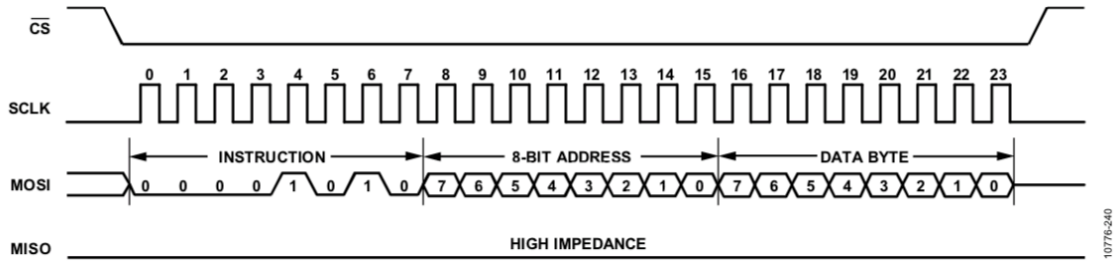


Figure 18: ADXL362 Write Register Command (0X0A) waveform [31]

Communication with the accelerometer is done through an SPI Master device implemented in the FPGA and it is done by entering measure mode and continuously reading the X, Y and Z registers for acceleration data. Table 9 presents the relevant registers used for this project.

Register	Name	Description
0x08	XDATA	8 most significant bits of the X axis accelerometer data
0x09	YDATA	8 most significant bits of the Y axis accelerometer data
0X0A	ZDATA	8 most significant bits of the Z axis accelerometer data
0X1F	SOFT_RESET	Performs a device reset when written with 'R' : 0x52
0x2D	POWER_CTL	Switches between operation modes, accuracy and power requirements

Table 9: Analog Devices ADXL362 Accelerometer relevant registers (own elaboration)

Soft Reset requires a minimum of 0.5 milliseconds for the accelerometer to return to its normal operating condition, so this is taken into consideration when initializing the sensor. The details for the power control (POWER_CTL) register are presented in Figure 19 below.

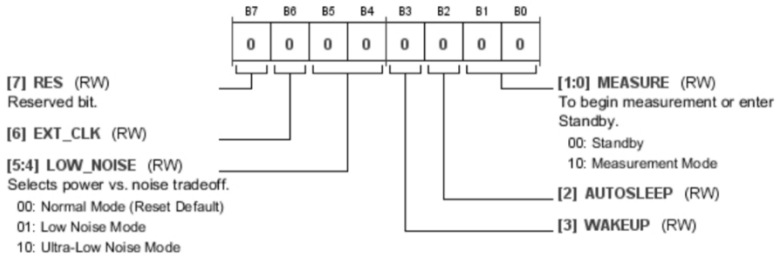


Figure 19: Register fields for ADXL362 Power Control (0x2D) register [31]

Initializing the sensor for polled measurement mode can be done using the Zephyr OS code snippet presented in Code 1. Note that this code uses a custom, low-level SPI software library developed for this project by the MSc. thesis author.

```
setSpiReg(0x1F, 0x52);       // Reset Accelerometer
k_sleep(K_MSEC(1));          // Minimum 0.5ms to allow for Accelerometer reset
setSpiReg(0x2D, 0x02);       // Enable measurement mode
```

Code 1: Zephyr OS code for initializing the ADXL362 accelerometer (own elaboration)

Then, the application may poll for acceleration information using the 8-bit XDATA (0x08), YDATA (0x09) and ZDATA (0x0A) registers. Code 2 shows a Zephyr OS function to read acceleration data into a 3-byte int8_t array. Acceleration data is delivered using two's-complement for positive and negative axis acceleration, so a zero value represents the lack of acceleration or tilt in the axis read from.

```
//adxl362 registers for accelerometer data
#define XAXIS           0x08
#define YAXIS           0x09
#define ZAXIS           0x0A

void getAcl(int8_t *xyz){
        xyz[0] = (int8_t)getSpiReg(XAXIS);
        xyz[1] = (int8_t)getSpiReg(YAXIS);
        xyz[2] = (int8_t)getSpiReg(ZAXIS);
}
```

Code 2: Zephyr OS code for reading 8-bit acceleration data from ADXL362 (own elaboration)

5.3 LAN8720A Ethernet transceiver

An Ethernet PHY[29] is included in the Nexys 4 DDR board. Although this element is not used in the project prototype, it's inclusion is a natural step for a future project. Figure 20 highlights the interface within the project scope.

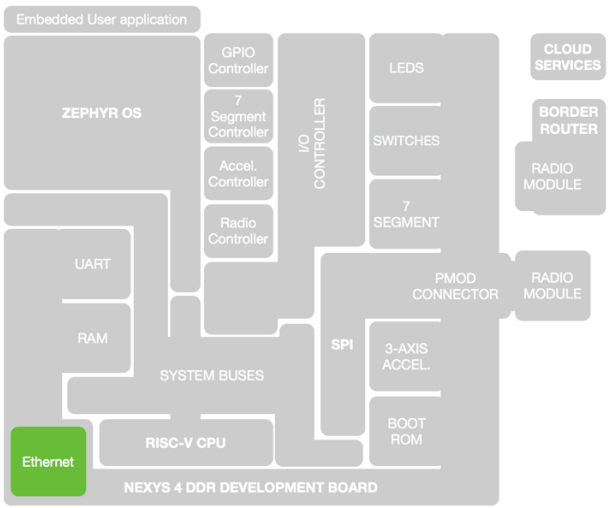


Figure 20: Ethernet transceiver within the project scope (own elaboration)

[29] **PHY**: Physical layer (layer 1) of the OSI network model

The transceiver chip is a LAN8720A [32] from Microchip , and offers 10Mbps and 100Mbs auto negotiated speeds and automatic MDI[30]/MDIX[31] support. This chip provides the physical access to a 10/100 ethernet network over twisted pair and connects to the FPGA through a RMII[32] interface. The block diagram is presented in Figure 21, while an internal configuration of the device, detailing the RMII and control connection port, is shown in Figure 22.

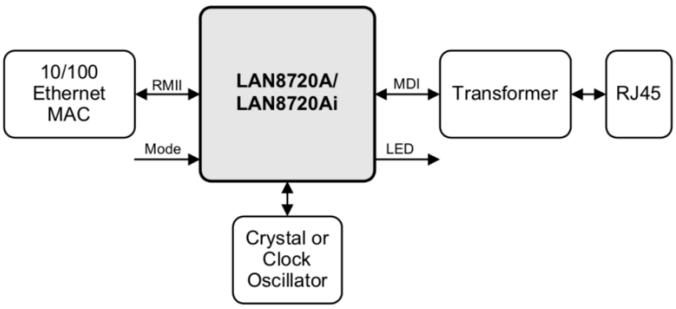


Figure 21: Microchip LAN8720A block diagram [32]

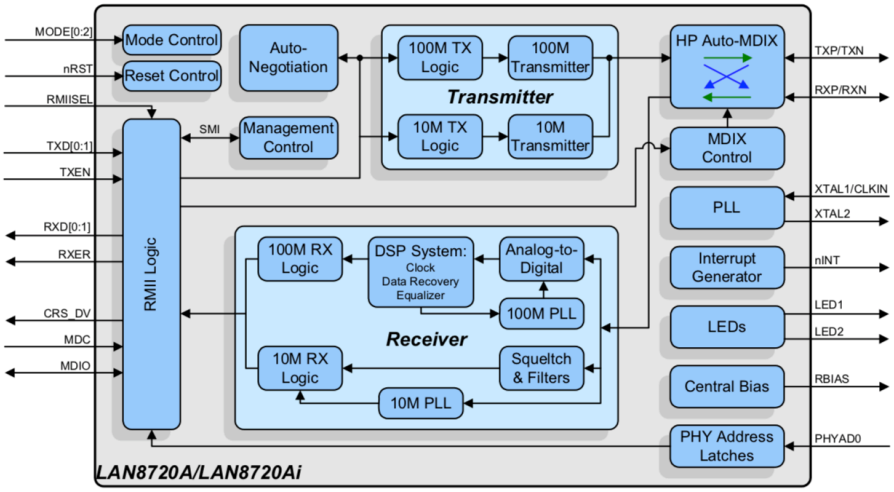


Figure 22: LAN8720A internal diagram and RMII ports [32]

The RMII, supported by the PHY device, communicates at 50MHz with the MAC[33], effectively becoming a multiplexed version of the MII[34], which uses twice the transmit and receive wires at half the speed. It, however, retains the same management interface as MII, grouped into the SMI[35]. These signals are MDC[36], a periodic clock for synchronous serial communication and the MDIO[37] port, used for command reception and status transmission. Figure 23 represents the waveform for sending data from the SMI (operation code 01). Receiving data follows the same diagram with operation code 10. In this latter case, the data is driven by the RMII Logic instead of by the SMI.

[30] **MDI**: Medium Dependent Interface.
[31] **MDIX**: MDI with crossed transmission and reception wiring
[32] **RMII**: Reduced Media-Independent Interface
[33] **MAC**: Medium Access Control Layer. Implemented in layer 2 of the OSI network model
[34] **MII**: Media Independent Interface
[35] **SMI**: Serial Management Interface
[36] **MDC**: Management Device Clock
[37] **MDIO**: Management Device Input Output

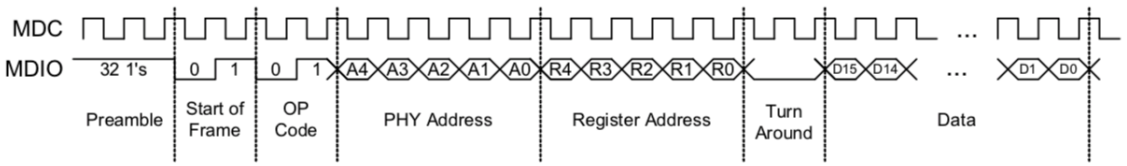


Figure 23: SMI MDC and MDIO signals. Operation 01, Write Cycle [32]

Table 10 defines the remaining signals managed by the RMII logic. The MAC layer is responsible for interfacing with the PHY device following the RMII and SMI described protocol using these signals. This task is covered in the "Ethernet support - MAC Layer core" chapter.

Signal	Description
TXD[1:0]	2 bits for data transmission
TXEN	Transmit strobe signal
RXD[1:0]	2 bits for data reception
RXER	Receive error (optional)
CRS_DV	Carrier sense

Table 10: RMII Data signals (own elaboration)

5.4 NRF24L01+ Radio

An external radio is added to both the Nexys 4 DDR board and the border router to enable two-way wireless communication. The NRF24L01+ radio is selected for its very low cost, versatility, constrained power needs and speed. Figure 24 highlights the radio modules within the project scope.

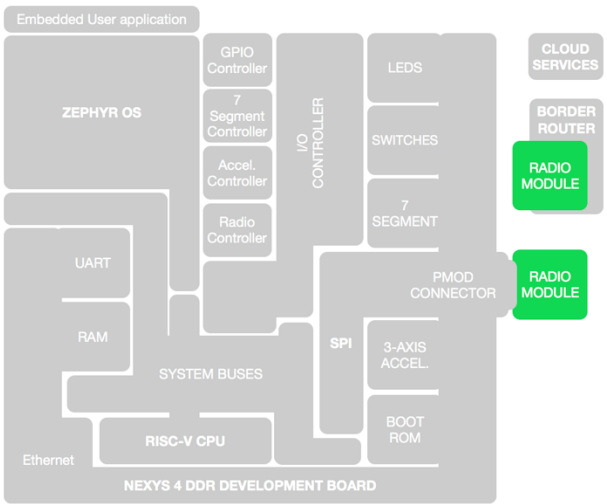


Figure 24: Radio modules within the project scope (own elaboration)

The NRF24L01+ single chip radio transceiver, from Nordic Semiconductor [33], is an ISM[38] band transceiver that includes a baseband to manage the link layer operations, which name is *Enhanced ShockBurst* and it is proprietary to Nordic. It uses GFSK[39] modulation and works in a range of speed from 250Kbps to 2Mbps.

[38] **ISM**: Industrial, Scientific and Medical frequency allocation band. Worlwide is 2.4GHz – 2.4835GHz.
[39] **GSK**: Gaussian Frequency-Shift keying

Each radio device may be configured as a transmitter (PTX) or receiver (PRX) station, but Enhanced ShockBurst allows for baseband-controlled acknowledgement of packages, and it offers the chance to piggyback user data to these ACKs so, for the intended application, there is no need to manually exchange roles for an end-to-end communication. However, it allows for star topology networks in a 1-to-6 shape, where a PRX station can establish two-way communication with different PTXs stations sharing the same frequency. This opens-up the possibility of creating IP-based mesh networks. The Open-Source project RF24 Ethernet [34] does exactly this, by creating an IoT oriented network of NRF24L01+ devices connected to a central gateway or border router.

Enhanced ShockBurst transmitted packages have a dynamic or fixed payload from 0 to 32 bytes and they include a header with a preamble byte, the receiver address that may be from 3 to 5 bytes, a control field carrying the payload length, the packet identification number and a no-acknowledgement flag to indicate whether an ACK[40] is expected for the packet or not. The package last part is a one or two byte long CRC[41] field. Figure 25 describes the packet structure and details on the control field.

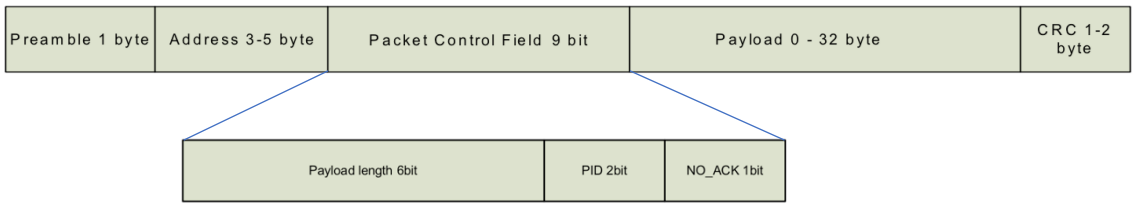


Figure 25: Enhanced ShockBurst packet structure (composition [35])

Electrically, the transceiver operates in a voltage range from 1.9V to 3.6V and it has 5V tolerant inputs. The operation electric power drawn vary from 11.3mA when transmitting at full power to 13.5mA in 2Mbps reception mode. Standby requires 26uA and there is a power down mode that only draws 900nA.

This work uses modules based on the datasheet's [35] reference design, with a very small footprint of barely 23x20 mm. Figure 26 depicts a schematic of the used modules, pointing out the pin signals. These modules are readily accessible from multiple Internet shops at prices from 0.50€. There is an additional design, including an antenna and a redesign for improved range that retails a bit south of 1.5€.

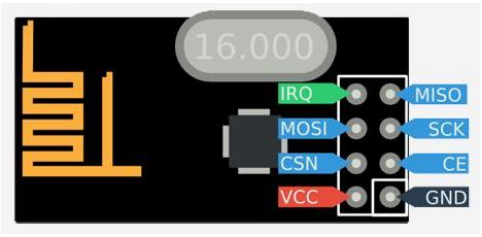


Figure 26: NRF24L01+ Module reference design with pinout [36]

[40] **ACK**: Acknowledgement communication packet indicating the message has been received
[41] **CRC**: Cyclic Redundancy Check. An error-detecting code

The module control interface is accessed through SPI, clocked at a maximum speed of 10MHz, though it also uses two extra signals: CE, used for activating RX or TX mode and signaling a packet transmit in TX mode, and IRQ, an interrupt request signal from the module based on three different signals: Packet arrival, Packet is sent and the maximum number of retransmits without ACK has been reached. Details on the SPI protocol are included in the next chapter "SimpleSPI SPI Core".

Most of the Radio management, apart from the CE controlled actions, is done by sending commands and data through SPI. A command may optionally be followed by relevant data bytes. The list of commands (all of them 1-byte long) used in this work is presented in Table 11.

Command	Format	Databytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read command and status registers. AAA AAAAA = 5 bit Register Map Address
W_REGISTER	000A AAAA	1 to 5 LSByte first	Write command and status registers. AAAAA = 5 bit Register Map Address
R_RX_PAYLOAD	0110 0001	1 to 32 LSByte first	Read RX-payload: 1 – 32 bytes.
W_TX_PAYLOAD	1010 0000	1 to 32 LSByte first	Write TX-payload: 1 – 32 bytes.
FLUSH_TX	1110 0001	0	Flush TX FIFO, used in TX mode
FLUSH_RX	1110 0010	0	Flush RX FIFO, used in RX mode
W_ACK_PAYLOAD	1010 1PPP	1 to 32 LSByte first	Write Payload to be transmitted together with ACK packet on PIPE PPP.
NOP	1111 1111	0	No operation. Returns the contents of the status register

Table 11: NRF24L01+ used commands. (Excerpt [35])

The RF24L01+ Radio has 29 user registers plus 9x32 FIFO[42] registers that are accessible externally. All registers have a width of 8 bits. The FIFO registers are intended for:

- **Transmission**: offering 32 bytes by three levels, for different addresses.
- **Reception:** 32 bytes in three different levels, from different PTX stations.
- **ACK payload:** Up to 96 bytes in a FIFO of ACK payload from PRX to PTX.

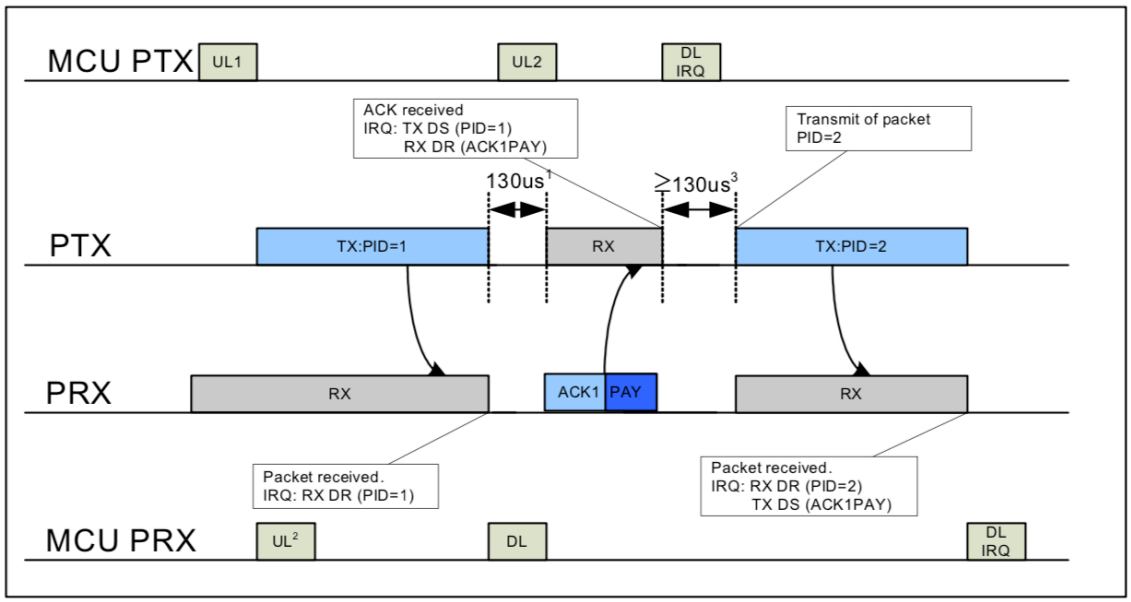
The user registers used for this work are described in Table 12 below.

Name	Address	Description
CONFIG	0x00	Configuration register for interrupt control, CRC configuration, power modes and PTX/PRX setup
EN_AA	0x01	Auto Acknowledgement configuration for data pipes
SETUP_AW	0x02	Address width setup
STATUS	0x07	Interrupt flags and TX and RX queue information
RX_ADDR_P0	0x0A	Address for radio reception on PIPE 0
TX_ADDR	0x10	Address to include into sent packets
RX_PW_P0	0x11	Default payload size for data PIPE 0
DYNPD	0x1C	Enable dynamic payload length
FEATURE	0x1D	Enables ack payload and other options

Table 12: Used NRF24L01+ user registers (own elaboration)

[42] **FIFO**: First In First Out. List queue operation format

Figure 27 depicts the TX/RX cycles on both ends, including the ACK with payload. The upper part represents the microcontroller to which a PTX module is connected whereas the lower half is the reception side. Note that the automatic turnaround of roles for ACK and attached payload reception is done without the MCU[43] intervention.



1 Radio Turn Around Delay
2 Uploading Payload for Ack Packet
3 Delay defined by MCU on PTX side, ≥ 130us

Figure 27: NRF24L01+ Enhanced ShockBurst TX/RX cycles with ACK Payload [35]

5.5 SimpleSPI SPI Core

SPI is required for communicating with the accelerometer and the radio module. SPI cores are added in the project, on top of the system controller, for the task. Figure 28 presents these cores within the project scope.

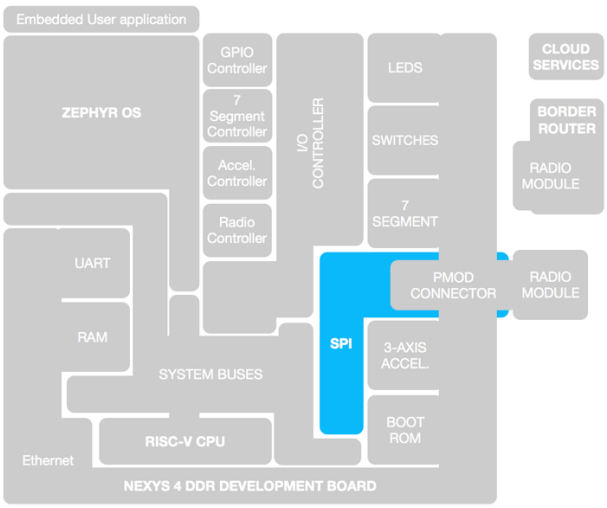


Figure 28: SPI Cores within the project scope (own elaboration)

[43] **MCU**: MicroController Unit

SweRVolf Nexys uses a dedicated SPI core from Opencores [37], the "Simple SPI" core. It is defined as *spi0* and it is tied to the SPI 128Mbit flash chip for bootrom usage. Two more SPI cores have been added to the project, *spi1 and spi2,* and their control and data signals have been connected to the ADXL362 accelerometer and to the PMOD A connector through the project constraints file. They are mapped into available memory ranges of the multicon interface and attached to the Wishbone bus. The SimpleSPI documentation [38], marked as *Rev 0.1 preliminary*, dates back from 2003 and it lacks information on some of the functionality of the current core. The core has even evolved to support elements recognized as not supported in the documentation.

SPI is a single-master/multiple-slaves serial synchronous protocol developed by Motorola and introduced in 1979 for its MC68HC11A8 microcontroller. It allows for a full-duplex communication between physically close circuits at speeds up to 100MHz[44]. In its standard configuration, it uses four wires: MOSI (Master Out, Slave In), MISO (Master In, Slave Out), SCK (Clock) and CS (Chip Select). CS is alternatively named SS (Slave Select). All signals but MISO are driven by the Master device at any given time. SPI operates in 8 bits where bits are shifted at the same time between the master and the slave through shift registers connected to the MISO and MOSI lines. This behavior may be seen as a *virtual 16-bit circular shift register*, where half of the register resides in the Master device and the other half in the slave device. Figure 29 illustrates this *virtual circular register* concept.

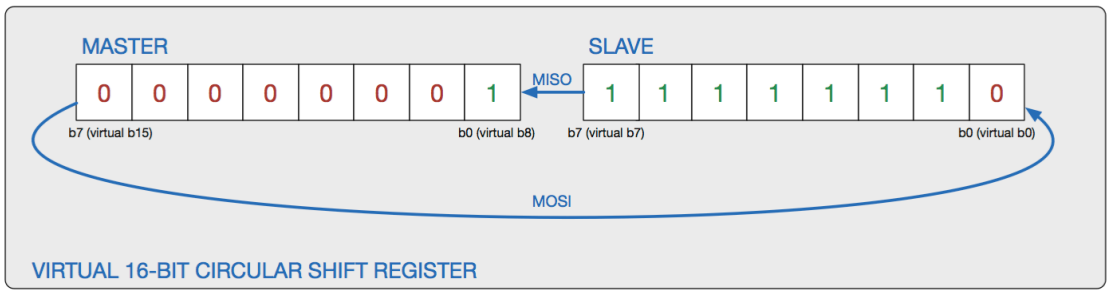


Figure 29: SPI Virtual 16-bit circular shift register concept (own elaboration, [39])

On top of these registers, data registers, often called shadow registers or *buffers*, are implemented to hold the data to be sent and the actual received data. The communication is started by the Master device writing to the buffer register. A copy is made into its shift register and a total of 8 clock pulses are generated as the data is being shifted (MSb first) and transferred between the master's and the slave's registers. Note that a full communication cycle, request-response pattern, would require at least two complete shifts of the registers (i.e. 16 clocks cycles). Nevertheless, a pipeline of data is often used, only requiring an extra 8-bit shift to finish the transaction. This is the case of the ADXL362 Accelerometer, which uses 3-byte transmissions for reading a register. A diagram of the Master-Slave connection is shown in Figure 30.

[44] Although most SPI buses run under 10Mhz, LTC2376 [78] from Analog Devices supports a 100Mhz SPI bus

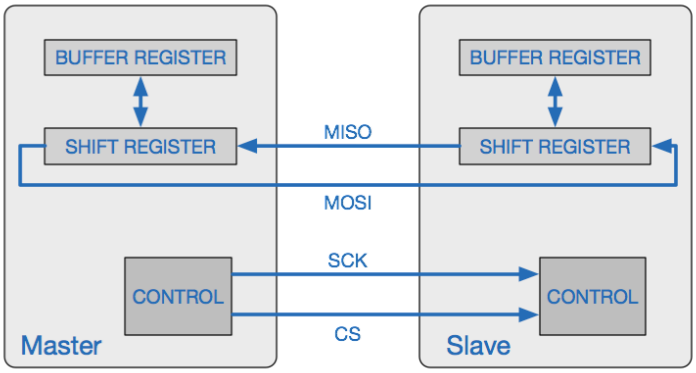


Figure 30: SPI communication diagram (own elaboration [39])

A wishbone interface is used to connect the to the RISC-V core; given that SweRV uses an AXI4 bus, a Wishbone-AXI bridge is also included in the SoC. The signals presented in Table 13 must be implemented and mapped into the SweRVolf SoC level controller, axi_multicon.v.

SimpleSPI implements a series of registers to manage SPI communications. It only supports acting as a master device, which is enough to comply with the project requirements. The Buffer register is called SPDR and a list of all registers and their description is presented in Table 14. Note that the original core uses 8-bit registers, addressable in consecutive positions, but since the SweRV EH1 core, on which SweRVolf is based, uses 64-bit transfers through the AXI bus, the author decided on a quick workaround by enlarging those registers to occupy 64 bit each. Since RISC-V is little endian, 8, 16, or 32-bit words have the least significant byte at the first one position

Port	Width	Direction	Description
clk_i	1	Input	Master clock input
rst_i	1	Input	Asynchronous active low reset
inta_o	1	Output	Interrupt request signal
cyc_i	1	Input	Valid bus cycle
stb_i	1	Input	Strobe/Core select
adr_i	2	Input	Lower address bus bits
we_i	1	Input	Write enable
dat_i	8	Input	Data input
dat_o	8	Output	Data output
ack_o	1	Output	Normal bus termination

Table 13: Wishbone signals for Simple SPI [38]

Name	adr_i[1:0]	Width	Access	Description
SPCR	0x00	8	R/W	Control Register
SPSR	0x01	8	R/W	Status Register
SPDR	0x02	8	R/W	Data Register
SPER	0x03	8	R/W	Extensions Register

Table 14: Simple SPI registers missing SPCS – Chip Selection register [38]

Register SPCS is notably missing from the documentation. This register, mapped right after SPER, at 0x05, is used to assert or deassert the CS line to the slave. SPCR is used for

enabling/disabling the SPI operations, setting SPI mode, interrupt generation mode and, alongside SPER, defining the SCK clock frequency as a division of the *clk_i* clock input from the Wishbone bus. The SPSR status register contains useful information on the communication events. In this project SPI works in a polled mode, so Bit 7 of the SPSR, called Serial Peripheral Interrupt Flag or SPIF, is heavily used. SPIF can be read or written and it is set when the transfer has ended. This flag must be manually deleted by software by writing a 1 to SPIF. Once the peripheral is activated, the steps to initiate an SPI transfer from a program running on the SoC are:

6. Clear SPIF by writing a 1 to bit 7 of SPSR
7. Initiate data transmission by writing a byte value to SPDR
8. Poll for bit 7 of SPSR and wait until its value is 1
9. Read SPDR data for getting the data just arrived from the slave device

Code 3 shows a RISC-V assembler program performing the described steps for a single byte transmission and reception.

```
# Function: Send byte through SPI and get the slave data back
# call:  by call ra, spiSendGetData
# inputs: data byte to send in a0
# outputs: received data byte in a1
# destroys: t0, t1
# ------------------------------------------------------------------------------
spiSendGetData:

internalSpiClearIF:             # internal clear interrupt flag
        li t1, SPSR             # status register
        lb t0, 0(t1)            # clear SPIF by writing a 1 to bit 7
        ori t0,t0,0x80
        sb t0, 0(t1)

internalSpiActualSend:
        li t0, SPDR             # data register
        sb a0, 0(t0)            # send the byte contained in a0 to spi

internalSpiTestIF:
        li t1, SPSR             # status register
        lb t0, 0(t1)
        andi t0, t0, 0x80
        li t1, 0x80
        bne t0,t1,internalSpiTestIF   # loop while SPSR.bit7 == 0. (tx in progress)

internalSpiReadData:
        li t0, SPDR             # data register
        lb a1, 0(t0)            # read the message from SPI

        ret
```

Code 3: RISC-V Assembler SPI byte transfer function (own elaboration)

5.6 Ethernet Support - MAC Layer core (Unimplemented)

Enabling the Ethernet interface in the SoC requires a MAC layer core. A Xilinx MAC IP Core and its required adaptor is covered in this chapter, notwithstanding the fact that ethernet networking is not included in the project scope. For clarification, Figure 31 depicts the lack of inclusion of this core in the project. It would lay on top of the system buses as marked in blue, connecting the Ethernet PHY device with the RAM, using DMA, and being accessed by the CPU through the System buses.

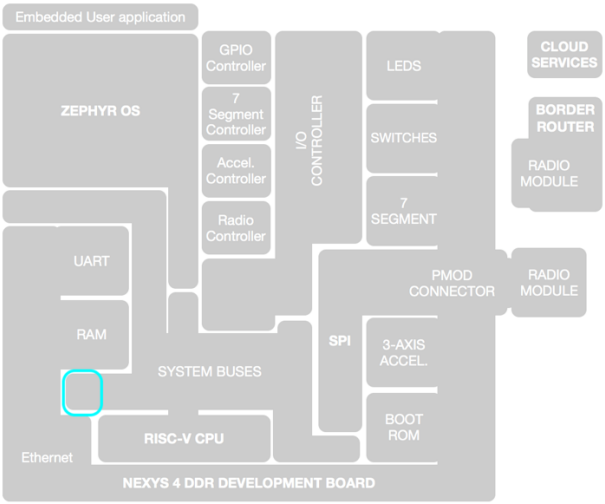


Figure 31: Lack of MAC core in the project scope and its would-be position (own elaboration)

Adding support for the on-board ethernet PHY interface was one of the original goals of this project. However, just the extension of the modifications required in the SweRVolf Core and in the Zephyr configuration exceeds by far the whole 270 [40] allocated hours for this thesis and, although the implemented project has clearly stretched this allocated work time, including this module was simply not timely possible. Additionally, Zephyr's IP stack is, most likely, too heavy for the SweRVolf core as it is configured, attending to the project performance on much simpler tasks. These reasons tilted the decision towards the NRF24L01+ radio option, which also enabled a broader IoT configuration by the addition of a border router.

Two weeks before the deadline of the project, the author of the SweRVolf core, upon request, pushed a branch of an old core [41] supporting the on-board Ethernet PHY, and an almost obsolete Zephyr OS version ad-hoc branch [42]. Future work for this project may be extending the SoC to support Ethernet with current versions of both the SweRVolf Core and Zephyr OS.

For completeness reasons, an overview of the EthMAC IP core from Xilinx and the required RMII to MII adaptor, as used in the aforementioned branch, is included in this document. Being a free-of-charge but commercial IP core, its 261 pages documentation [43] is extensive and detailed.

The core supports RGMII[45], GMII[46] and MII Physical layer interfaces, therefore an additional MII to RMII core is required because the on-board LAN8220A PHY only

[45] **RGMII**: Reduced Gigabit Management

[46] **GMII**: Gibabit Media-Independent Interface

supports RMII. Xilinx also provides this core as a discontinued IP. Source code is provided, without support as of Q4 2019, as announced in Xilinx AR# 71457 [44]. Documentation on this core [45] is also detailed, with 30 pages including a configuration walkthrough. The core sits in between the MAC Core and the Physical interface, adapting messages to the right protocol bidirectionally. A diagram of the interface is presented in Figure 32.

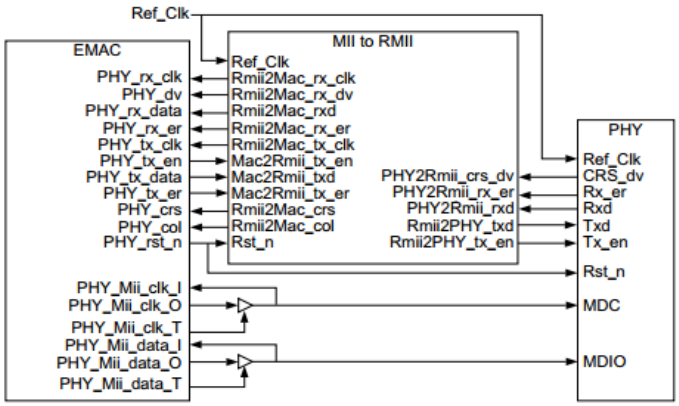


Figure 32: MII to RMII Core block diagram [45]

The MAC core is still in production and included into the available IP cores in Vivado. Therefore, it is offered within an encrypted RTL. It may optionally include quality of service rules for audio/video management, however this requires a paid license. It features an AXI4-Lite wrapper to connect it to an AXI-Lite bus, while it also may be connected to a full AXI4 bus. A block diagram of the core is presented in Figure 33 below.

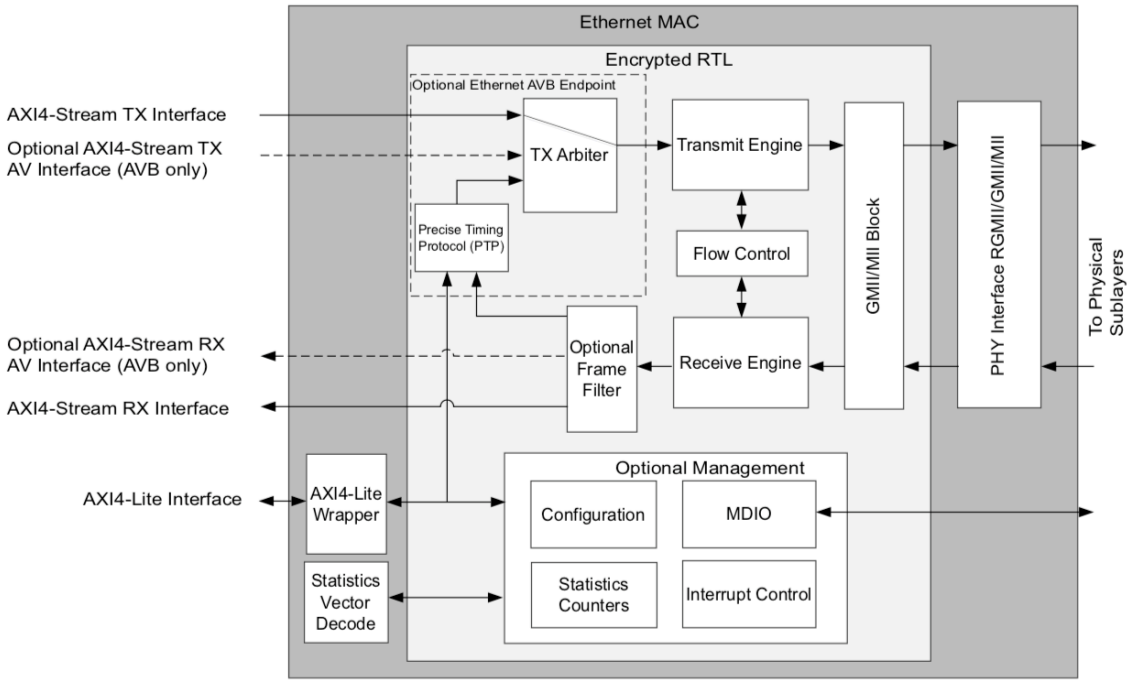


Figure 33: Xilinx Tri-Mode ethernet (ETHMAC) block diagram [43]

Although the MAC Core is functionally-complete, neither frame filtering nor AVB endpoint are required for this application. The management block utilization will very much depend on the deployed implementation of all the remaining Extended SoC elements and Zephyr's configuration. The relevant required blocks are listed below:

- **AXI4-Lite Wrapper**: allows the core to be connected to a AXI4-Lite bus.
- **PHY Inteface**: Connects to the PHY layer (or MII to RMII core).
- **Transmit Engine**: Pushes the AXI4 incoming data to the physical layer, adding preamble and frame check sequence fields and padding the data.
- **Receive Engine**: Takes PHY layer incoming data and checks for compliance with the 802.3 specification. It presents the unpadded data to the AXI bus.
- **Flow Control**: Compliant with 802.3-2008 may be configured to send pause frames.
- **GMII/MII Block**: Converts 4-bit data of the PHY layer to 8-bit data used by the receiver and the transmitter.

This Core supports different deployment types such as Router, Ethernet switch or up to 2.5G baseX (fiber-channel) physical media. However the desired deployment type includes DMA support for communicating with an embedded processor. Figure 34 shows the typical application configuration suggested by Xilinx. Note that for the Nexys 4 DDR and the Nexys A7 boards, a RMII to MII, not shown in the diagram, is required.

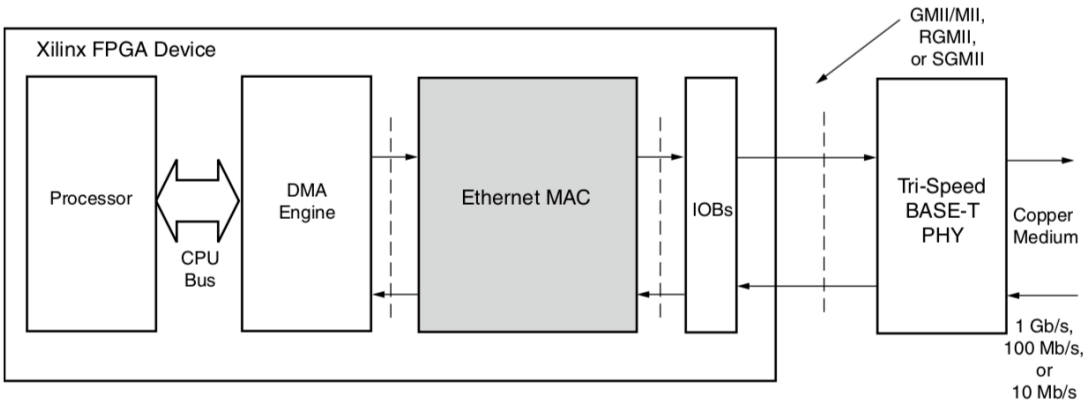


Figure 34: Typical application of Ethernet for Embedded processor [43]

Note that the old SweRVolf core implements the DMA engine for Ethernet communication as suggested in the Xilinx guide.

5.7 NodeMCU v3 board

This development board is used for implementing the border router. Figure 35 presents its situation within the project scope.

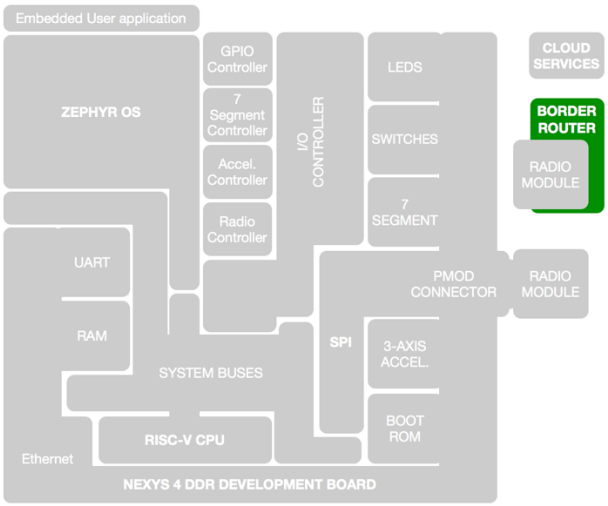


Figure 35: NodeMCU board within the project scope (own elaboration)

The NodeMCU is a development board based on the ESP8266 SoC from Espressif [46], which is built around the Tensilica L106 microcontroller from Cadence [47], a modified Harvard 32-bit RISC CPU[47] running at 80 or 160MHz. The system includes WIFI 802.11 b/g/n communication in the 2.4GHz band, including WEP[48], WPA[49] and WPA2[50] security. The SoC lacks program memory, so there are several modules built around it and including an SPI program memory chip.

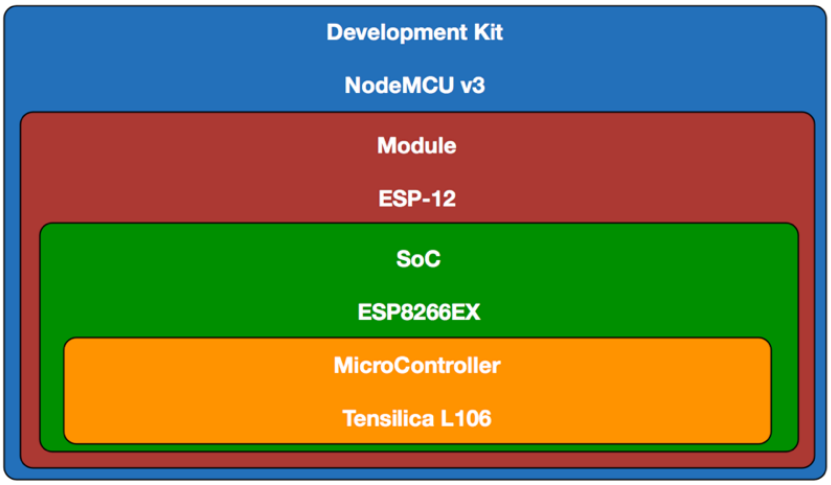


Figure 36: Structure of the NodeMCU (own elaboration)

[47] **CPU**: Central Processing Unit
[48] **WEP**: Wired Equivalent Privacy
[49] **WPA**: Wireless Protected Access
[50] **WPA2**: Wireless Protected Access version 2

The NodeMCU includes an ESP-12 module with 4Mbytes of program memory. Figure 36, in the previous page, shows the integration of the different components into the NodeMCU development board.
The ESP-12 module adds the program memory, a Led diode, a 26Mhz crystal and a WIFI antenna. The NodeMCU packages the Module into a board with accessible, through-hole pins, two buttons, a 3.3V regulator and a USB port for serial communications and power. Although Cadence offers a development environment for the SoC, the initial versions were launched to the market, by third parties, with a built in firmware for Hayes command interaction through serial, aiming at the DIY[51] and electronics hobbyist community.

This approach had moderate success, but a few months afterwards, the Arduino IDE included community-built support for the ESP8266 board, which became the number one board for small connected electronics test environments. The SoC is now replaced by the ESP-32 with a dual core processor and support for BLE[52], but the ESP8266 remains a platform of choice for its low cost and simplicity. A NodeMCU is only one of the many ESP8266EX based development kits that can be bought for less than 4€ in the market.
The SoC includes several hardware peripherals, besides WIFI. This work uses SPI, UART, and GPIO. Figure 37 shows a diagram of the available pins at the nodeMCU V3.

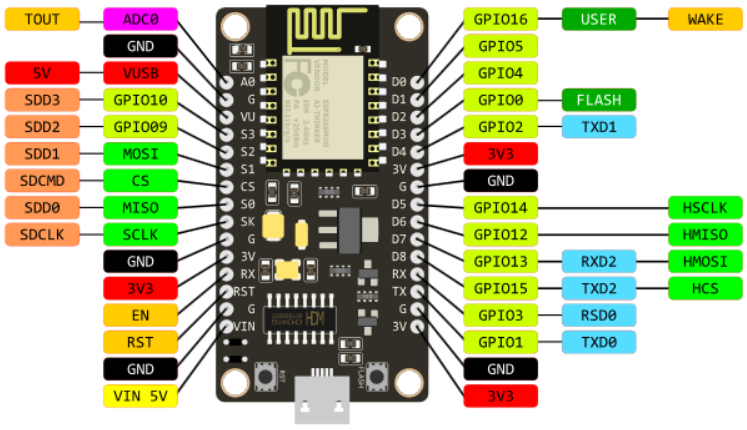


Figure 37: NodeMCU v3 pinout [48]

Note that the left side SPI port (green tags) is the one used for accessing the program memory so, unless the code is placed in Ram with a bootloader, it should remain unused. Also, the pins on the ESP8266 are not 5V-tolerant. As stated in the datasheet [49] for the SoC, a maximum of 3.6V may be applied for a logical high signal to any given pin. The USB serial communications is built around the CH340G chip from Nanjing Qinheng Microelectronics [50], a popular substitute of the FTDI [51] provided chips.
The higher ESP32 version was also evaluated for the border router, but its enhanced capabilities where not required for just one node. It is interesting though as the software can be compiled, without changes, for this board enabling a quick and painless upgrade for adding extra nodes in a star topology. Additionally, ESP32 is supported by Zephyr OS. For reference, a head to head comparison on the main features is shown in Table 15.

[51] **DIY**: Do It Yourself
[52] **BLE**: Bluetooh Low Energy

	ESP8266	ESP32
MCU	Xtensa Single-core 32-bit L106	Xtensa Dual-Core 32-bit LX6 with 600 DMIPS
802.11 b/g/n Wi-Fi	HT20	HT40
Bluetooth	**No**	Bluetooth 4.2 and BLE
Typical Frequency	80 MHz	160 MHz
SRAM	**No**	**Yes**
Flash	**No**	**Yes**
GPIO	17	36
Hardware /Software PWM	None / 8 channels	None / 16 channels
SPI/I2C/I2S/UART	2/1/2/2	4/2/2/2
ADC	10-bit	12-bit
CAN	**No**	**Yes**
Ethernet MAC Interface	**No**	**Yes**
Touch Sensor	**No**	**Yes**
Temperature Sensor	**No**	**Yes**
Hall effect sensor	**No**	**Yes**
Working Temperature	-40°C to 125°C	-40°C to 125°C

Table 15: ESP8266 vs. ESP32 comparison [52]

6. Zephyr OS

Zephyr OS is the selected RTOS for the node for many reasons:

- It has extensive support from the Linux Foundation and other key players
- It bets on RISC-V as one of the key supported platforms
- It has a convenient Apache 2.0 license
- It implements a complete set of IoT-related network protocols and security
- It is gaining momentum rapidly, with massive updates every other day

Figure 38 highlights Zephyr OS within the project scope.

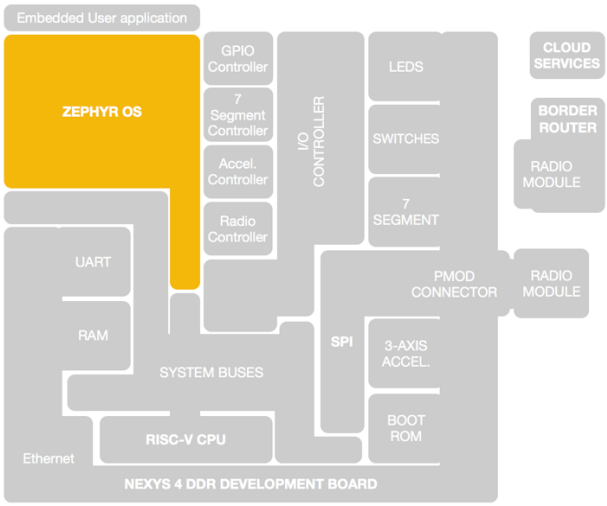


Figure 38: Zephyr OS within the project scope (own elaboration)

"The Zephyr™ Project strives to deliver the best-in-class RTOS[53] for connected resource-constrained devices, built be secure and safe".

This is the vision for Zephyr OS, and the first thing shown in the project website [8]. The Zephyr OS is an opensource, Apache 2.0 licensed, real time operating system. It is managed under the Linux Foundation and supports multiple platforms, including RISC-V 32, Intel x86, ARM Cortex M, ARC, Tensilica Xtensa and NIOS-II. Membership to the project is tiered, being the current platinum members (top level): Intel, Nordic, NXP and Oticon. Other members are Adafruit, Bose, SiFive, Texas Instruments and the Eclipse Foundation. It also has extensive QEMU [53] support and, currently, supports over 200 boards out-of-the-box. It's current version, updated on June, 9th 2020 is 2.3.0.

Zephyr was launched in February 2016 and its development cycle is frantic, with over 40.000 commits on github by May, 2020 [54]. Up to 6 versions have been released during this project, so keeping up has required multiple adaptations. This project is, however, locked into 2.3.0 rc1 release, the antepenultimate version at the time of writing. The Zephyr project presents a comprehensive release plan [55] for upcoming versions.

[53] **RTOS**: Real Time Operating System

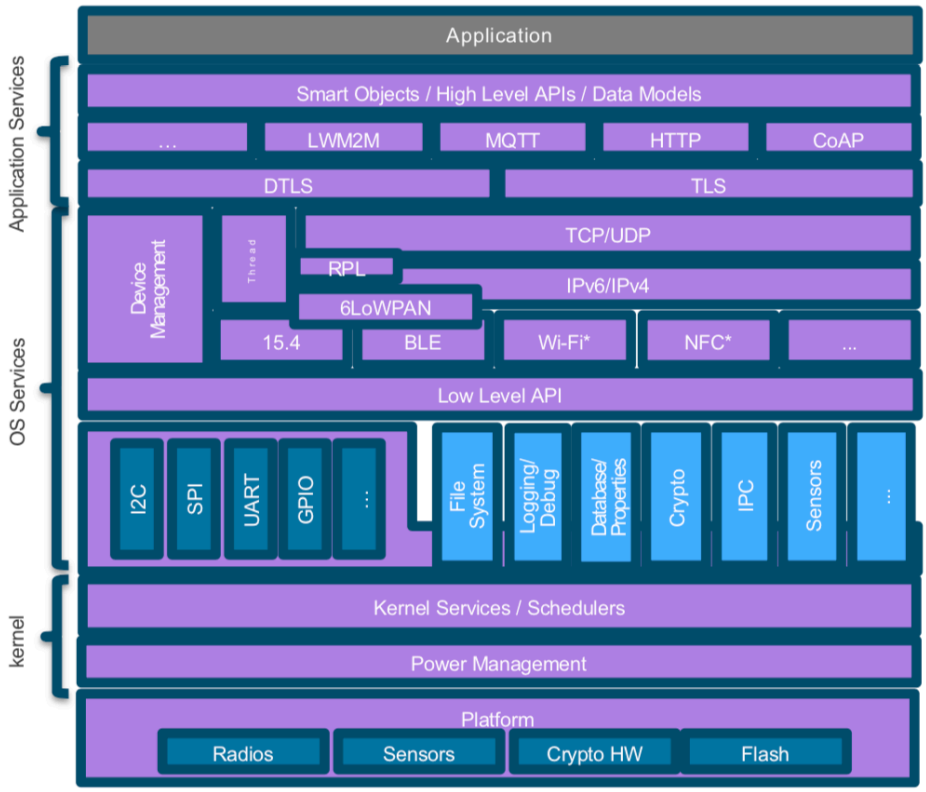


Figure 39: Zephyr OS networking architecture [58]

The OS implements a Kernel/HAL[54], OS services, including networking, and application services. Its architecture is presented in Figure 39 above.

Being focused on IoT devices, it offers footprints as low as 8KB, and also features extensive networking support including:

- MQTT[55], CoAP[56], LWM2M[57]
- BSD Socket support
- 6LowPAN, RPL
- TLS/DTLS (experimental)
- UDP and TCP
- IPV4 and IPV6, that may be active simultaneously
- 802.15.4, Bluetooth Low Energy (BLE), Ethernet
- Serial Line IP (SLIP), CanBUS, LoRa
- LoRaWAN, Thread, Bluetooth Mesh

Some forks to the project add more networking capabilities. This is the case of Nordic Semi nrfxlib [56], a fork that includes its own SDK to support multiple networking SoCs of the company, extending BLE and BSD support, adding a ZigBee Stack and Near Field Communications (NFC) and many other proprietary protocols. It also includes a full-

[54] **HAL**: Hardware Abstraction Layer
[55] **MQTT**: MQ Telemetry Transport
[56] **CoAP**: Constrained Application Protocol
[57] **LWM2M**: Lightweight Machine to machine

feature DTLS/TLS stack and an alternative IP stack. This fork, however, does not include drivers for the project's NRF24L01+ Radio, otherwise it would have been a candidate to consider instead of the main Zephyr branch, which is the one used.
Besides offering multitasking, both collaborative and preemptive, inter-process communication or filesystem support, it also implements drivers for an extensive list of sensors and radios. An extensible framework for adding new hardware elements and software features is presented to the developer. The main two elements for configuring the system are:

- **Device Tree:** Describes the system hardware and its default configuration at boot. As an example, the Extended SweRVolf Core defines its UART base address, boot baudrate and configuration here.
- **KConfig:** Describes included drivers (software features) and related configuration. This can be manually edited or assisted by tools such as guiconfig or menuconfig. Enabling the software drivers or leaving them out, regardless of the actual installed hardware, is done here. In this project, the software driver for the ADXL362 accelerometer has had to be excluded as explained later in the work package 2 description.

One important tool, extensively used in this project, is west [57], a meta-tool that allows managing multiple repositories, compiling and building, flashing, debugging, binary signing or accessing the KConfig helper applications. Zephyr guides strongly recommend using this tool and, although alternative tools may be used, they "imply extra effort and expert knowledge".

The Zephyr project started by releasing just the Kernel, but it has expanded since, to offer not only Zephyr OS, but SDK, tools, middleware and device Management, including bootloaders. A Zephyr community is also built around the Zephyr project, providing 3rd party libraries and new hardware support. The nrfxlib project introduced before, is an example of this. All these elements conform the "Zephyr Ecosystem" as presented in Figure 40.

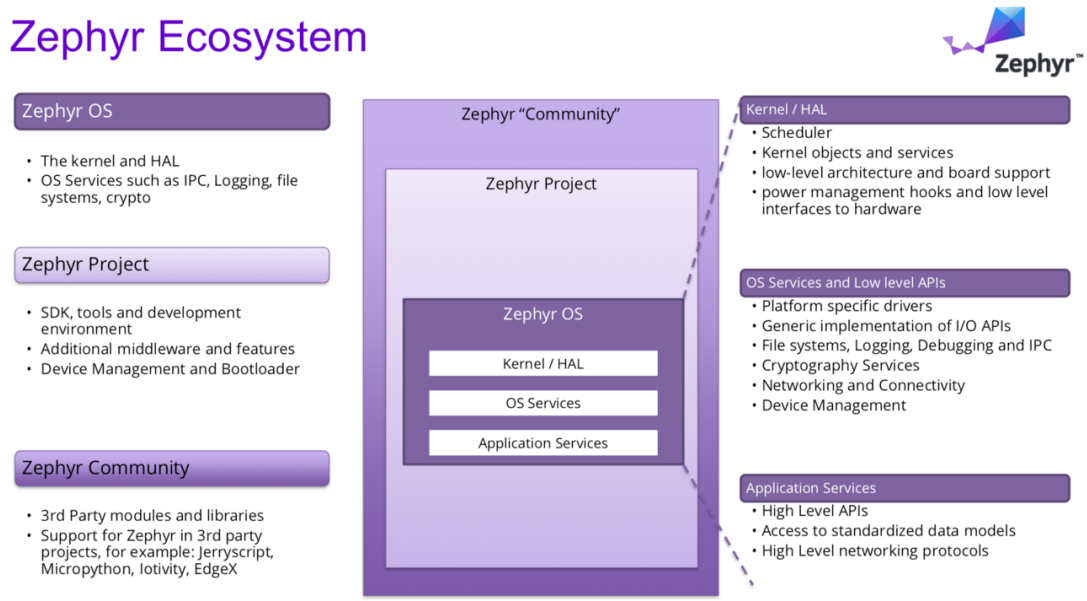


Figure 40: Zephyr EcoSystem [58]

This thesis project has just scratched the surface of the Zephyr project. While it is a very powerful and capable constrained operating system, its cycle of development, including radical changes and deprecations, makes very difficult for any project to keep up. Furthermore, no common base can usually be found when working with many components from different sources. Anyhow, Zephyr OS comes to a market in flux, facing a redefinition of the scope and reach of embedded, RTOS and IoT operating systems. This is a quite fragmented field too, with many contenders, and it is currently led by embedded Linux, in-house developments and FreeRTOS [59], as presented in Figure 41.

Figure 41: Top three most used OSs for embedded projects [59]

7. Project development

This project has required extensive work in many different areas, such as hardware implementation over an FPGA using Verilog, assembler programming for RISC-V, Zephyr OS programming of low-level drivers and applications, programming of the border router, also including low-level drivers for the radio, and cloud services configuration and integration with SMS services. Many work hours have been invested in setting up the different development environments and tools needed to accomplish the different tasks. Even more in studying the Nexys 4 DDR board, RISC-V architecture, SweRVolf SoC, Zephyr OS driver interface and programming APIs and the Nordic Semi RF radio. Three different computers were used to have as many screens as possible. Even so, there is always the need for a bigger working canvas. A picture of the working environment is presented in Figure 42. The necessary tasks have been divided in four work packages with several stages each. Their content is:

- **Work package 1**: **Development environment and platform setup**. A virtual machine was installed with the required software tools. Base SweRVolf SoC and Zephyr OS were downloaded, installed and tested.

- **Work package 2: Isolated SoC**. Its main target was having a SoC built around a RISC-V core with the required peripherals to read acceleration data from the Nexys 4 DDR board, read the switches and have access to the UART for sending messages, to the seven segments displays and to the Leds. It comprises several tasks: from the extension of the SweRVolf SoC to being able to run Zephyr OS in this Extended SoC. This work package also includes the development of BareMetal and Zephyr drivers to support the accelerator among other sensors and actuators.

- **Work package 3: Connected SoC.** This work package aimed at having end-to-end connectivity between the SoC and Internet. It includes extending the SoC for radio device support, the developing of the border router, the Zephyr and Arduino low level drivers for the Radio and the development of a basic test application to test the connectivity.

- **Work package 4: Full proof-of-concept environment.** This work package comprises the development of a Zephyr OS application, using the full-set of SoC components, that interacts with the border router for end-to-end communication with a cloud services provider (ThingSpeak [60]). It also includes all the cloud services configuration and setting up the independent boot process in the node.

Figure 42: Working environment (by the end of the project) with SoC development station to the left, border router development system to the right and documentation and cloud configuration, over two screens, at the back (own capture)

Using the project's block diagram, Figure 43 presents the evolution of the project between the four work packages.

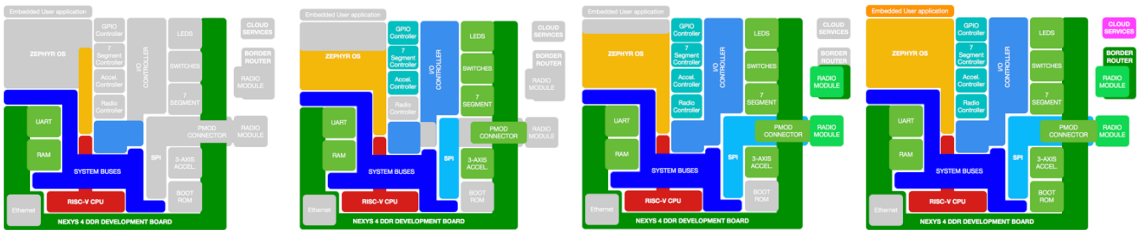


Figure 43: Progress of the project as addressed in the four work packages (own elaboration)

7.1 Work Package 1: Development environment and platform setup

This chapter describes the modifications required to the standard installation procedures in order to achieve a stable configuration. Note that many different configurations and tools may be valid and this guide intent is not, by any means, claiming that this is the only possible configuration. Links to the vendor installation procedures are included when required as well as release versions and potential patches or requisites. This work package was addressed during February, 2020, but had to be extended up until June for adapting the tools and environment to the latest software and core versions.

7.1.1 Stage 1: Create virtual machine with SoC tools and base SweRVolf Nexys

First task is to install a virtual environment. For this project a 2011 MacBook Pro with a 4-core i7 processor, 16GB of RAM and a 512GB SSD disk has been used. Table 16 describes all the elements and commands needed.

Step	Software	Release	Link / command	Description
1	Vmware Fusion	11.5.2	https://www.vmware.com/es/products/fusion.html	Desktop virtualization
2	Ubuntu	18.04.4 LTS	https://releases.ubuntu.com/18.04.4/	Operating System
3	Vivado WebPack	2019.2	https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2019-2.html	Hardware design for Xilinx platforms
4	Nexys	"new"	https://reference.digilentinc.com/vivado/installing-vivado/start#installing_digilent_board_files https://github.com/Digilent/vivado-boards/	Definitions for digilent boards
5	fusesoc	1.10	https://github.com/chipsalliance/Cores-SweRVolf pip3 install fusesoc fusesoc init fusesoc library add fusesoc-cores → https://github.com/fusesoc/fusesoc-cores	core generation tool
6	SweRVolf	0.6	Create directory and cd to it fusesoc library add swervolf → https://github.com/chipsalliance/Cores-SweRVolf	RISC-V based SoC
7	SweRVolf Vivado	-	fusesoc run --target=nexys_a7 swervolf	Synthezises and implements the core and creates vivado project
8	Openocd	0.10.0	sudo apt-get install openocd	On-chip-debugger
9	U-Boot Tools	2020.04-1	sudo apt-get u-boot-tools	Image maker for flash boot
10	GTKTerm	0.99.7	sudo apt-get install gtkterm	Serial terminal

Table 16: Tools for setting up the SoC development environment (own elaboration)

The following considerations were made in each of the steps described:

1. VMWare Fusion is a commercial software priced at $79. There are other non-paid virtualization environments such as VirtualBox [61] from Oracle. The decision for VMWare was based on its excellent external hardware support and proven track record on Vivado-on-Linux deployments.
2. This Long Term Support version will be supported until 2023.
3. Vivado WebPack is a limited-device version with no license cost. A Xilinx account must be created and the preferred option is getting the unified installer for a web-based installation of the software. There is an excellent guide [62] for the installation and first steps to integrate with the Nexys 4 DDR.

4. There are two sets of definitions called “old” and “new”. The correct installation is the folder under “new”. These files must be copied to the Vivado folders. Chapter 3 of the Digilent’s installing Vivado guide details the process.
5. Python3 and pip3 must be installed in Ubuntu. Otherwise, they must be installed by issuing a “sudo apt-get install python3 pip3”.
6. Optionally, the code may be downloaded by a git clone command.
7. This process actually builds the binary file for the target platform. In the process, a Vivado project file is created. The containing folder is: $WORKSPACE/build/swervolf_0.6/nexys_a7-vivado/ and the files are swervolf_0.6.bit for the binary file and swervolf_0.6.xpr for the Vivado project. The build process may take a long time, depending on the host computer. With the aforementioned hardware configuration, this takes around 30 minutes. Note that building the SoC needs at least 8GB of RAM assigned to the virtual machine, otherwise the process may abort. In this case, expanding the SWAP file or partition to a 10GB size is recommended.
8. OpenOCD is used for programming the RISCV software programs into the SoC running on the Nexys 4 DDR. [63]
9. Installed version is 2019.07
10. Minicom may be used instead.

After these steps, fusesoc is no longer needed for building. All modifications to the SoC and subsequent builds can be done within the Vivado WebPack software.

7.1.2 Stage 2: Add Zephyr OS and SDK toolchain and configure SoC support

Zephyr installation is directly supported on Ubuntu. It should be done following the guide [64] available at the Zephyr Project documentation site. The following fixes and considerations are made to the guide:

1. Select the Ubuntu OS, but do not update as requested, as it may affect the previously installed software.
2. Install dependencies using apt. Cmake is not up to date on Ubuntu 18.04.4, so it must be updated. Instead of using the procedure suggested on the Zephyr guide, (they are now updated) the following steps were taken:
    a. Download the latest package, in this case, *cmake-3.17.2-Linux-x86_64.sh* from https://cmake.org/download/
    b. Sudo install on /usr/bin
    c. Create a symbolic link at /usr/bin/cmake pointing to /usr/bin/cmake/cmake.
3. Installed Zephyr version is 2.3.0 rc1
4. Version 0.11.3 of the SDK did not work well with the project’s environment. Version 0.11.1 were installed instead. Installation shall be manually copied to /opt/zephyr-sdk as the installation script does not ask for a target directory. Note that Zephyr claims there is no Crosstool (toolchain) support for RISCV32, so “go.sh” may not work. The option is using “west” instead for building.
5. Nexys 4 DDR is not supported by Zephyr, thus some modifications must be made before compiling for this target.

a. SweRVolf provides three folders within the “zephyr” folder: “boards”, “dts” and “soc”. Content must be copied to the same route under Zephyr installation folder. The “soc” folder contains a “KConfig” that overwrites the original zephyr/soc/KConfig. A backup is made to KConfig.old
b. Edit zephyr/arch/KConfig and modify line 53, right below the “config RISCV” tag, from *bool* to *bool “DL swervolf”*. Any other text should work, but this change is mandatory, otherwise Zephyr programs will not build for the swervolf_nexys platform.
c. Edit the recently copied file zephyr/soc/risc/swervolf/soc.c. Change the gpio include directive to *#include <drivers/gpio/gpio_mmio32.h>*.
d. Also edit soc.h in the same directory. Change the inclusion of “generated_dts_board.h” to “devicetree.h”

7.1.3 Stage 3: Install Arduino IDE and required card support and libraries

This process is quite straightforward and takes just 6 steps:

1. Install the Arduino IDE from www.arduino.org. The installed version for this project is 1.8.12 for MacOS.
2. Launch the IDE and get into preferences. Add http://arduino.esp8266.com/stable/package_esp_8266com_index.json to the “Additional Boards Manager URLs” field.
3. Get into the “Tools” menu and click on the “Board:…” extensible option. Select “Boards Manager”
4. Enter esp8266 in the search field and install “ESP8266 Community”. The version installed for this project is 2.7.1.
5. Get into the “Tools” menu and click on “Manage Libraries” and search for “pubsubclient”. Install PubSubClient by Nick O’Leary. This project uses version 2.8.0. There are several other MQTT libraries available that may be used.
6. Drivers for the USB to serial adapter were installed from the manufacturer website. http://www.wch-ic.com/search?q=CH340&t=downloads

Configuration of the nodeMCU is done, under the tools menu, as presented in Figure 44.

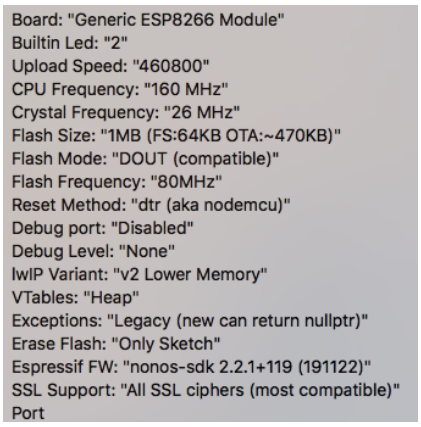


Figure 44: Configuration for the NodeMCU border router in the arduino IDE (own capture)

7.1.4 Stage 4: Test installation – sample hello world application for all elements.

The border router environment is tested by using a built-in example of the ESP8266 card support library. The chosen example is under Files → Examples → ESP8266WebServer → Hello Server. The program was compiled, uploaded to the board and executed, resulting in the expected outcome of an HTTP version of the Hello World program.

For testing the Vivado Webpack installation and its communication with the Nexys 4 DDR, the previously fusesoc-generated Vivado project was built again from the Vivado environment and then transferred to the board using the suite's Hardware Manager. This is done by setting JP1, the jumper for the FPGA load mode[pag.24] , in "JTAG" position. Figure 45 presents the hardware manager screen once the SoC has been transferred to the FPGA.

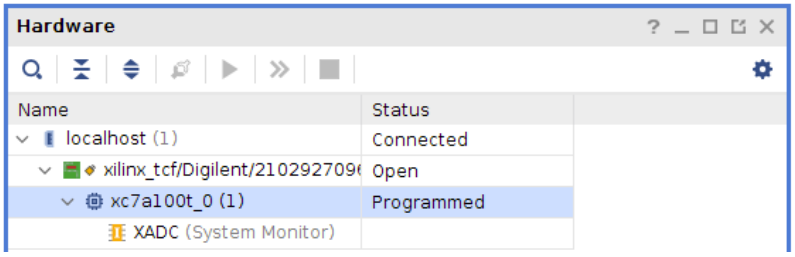


Figure 45: Vivado Hardware Manager (own capture)

There is, however, the chance to save the generated bitstream file into a USB or microSD card and then make the Nexys board load it from there. This has been the preferred option for this work as it leaves the USB port available for other tasks, because Vivado locks the serial port and it has to be connected and disconnected every time. The same jumper has a position USB/CD that must be selected. Then, JP2, close to the USB port, selects whether the load should be done from USB or SD. A PIC24 microcontroller is responsible for performing the programming if any of these options is selected.

The RISC-V toolchain included in the SweRVolf installation is tested by slightly modifying the hello.S program contained into the folder *Cores-SweRVolf/sw.* A modification may be needed in the Makefile of the folder, changing the *TOOLCHAIN_PREFIX*, at the beginning of the file, *from risc64-unknown-elf* to *risc32-unknown-elf.* The program is built by issuing a "make hello.elf".
Now, a series of steps must be followed for uploading a program to be executed by the SoC.

8. Execute: *openocd –f <swervolf dir>/src/OpenOCD_Scripts/swervolf_nexys_debug.cfg*
9. In another terminal, launch: *telnet localhost 4444*
10. A debugger prompt is presented. Enter: *reset halt*
11. Enter: *reg pc 0*
12. Enter: *load_image hello.elf*
13. Enter: *resume*

The outcome is shown in a terminal as presented in Figure 46.

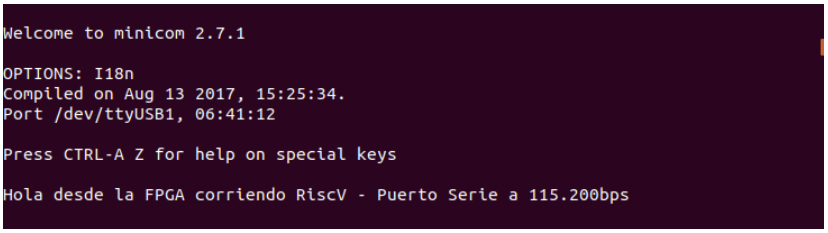


Figure 46: Hello World running on SweRVolf over the Nexys 4 DDR FPGA (own capture)

Testing Zephyr SDK must be done within the *zephyr* folder within the user's home. Entering a *wall build -d riscv-swervolf samples/hello_world* generates a build/zephyr folder with the compiled code, *zephyr.elf*. Using the same procedure followed in the previous test, the compiled Zephyr program may be uploaded to the board with the outcome shown in Figure 47.

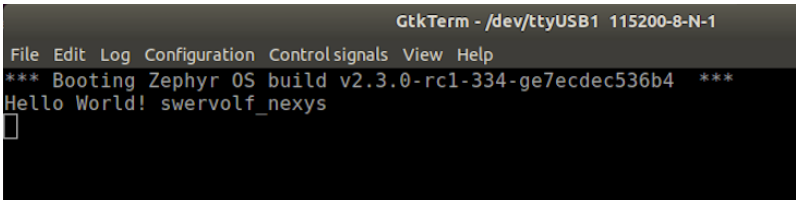


Figure 47: Hello world from Nexys running Zephyr (own capture)

Now, all the required software, tools and connections are installed and tested, so everything is ready for developing the project.

7.2 Work Package 2: Isolated SoC

This chapter describes the additions and modifications performed to the SweRVolf Core and Zephyr aimed at obtaining a system that can work with accelerometer data and present the information internally or log it through the serial port. The work done in this part extended from the beginning of March up to the first week of May, 2020. Figure 48 shows the final hierarchical structure of the SweRVolf Core as presented in Vivado's project manager view. Most of the work was performed within the axi_multicon.v file, the system controller presented in Figure 12 in chapter 4 - SwerVolf RISC-V Core, although the constraints file, linking the defined ports to actual pins of the FPGA, was also edited. The *spi_accel*, *spi_pmod*, *gpio** and *SegDispl_timer* shown in Figure 48 are all part of this work package.

The RISC-V architecture has memory-mapped I/O, i.e: Peripherals are to be mapped in the memory address space, and they are accessed by reading or writing to these addresses. In this project, all of the original and extended multicon peripherals use partial address decoding, meaning that a peripheral can be accessed in many different memory space addresses. While this approach locks much of the address space to a specific peripheral, the decoding is much simpler and requires less LUTs. Partial decoding is pervasive in the computer industry, with some notable examples such as the

1982 Sinclair ZX Spectrum ULA[58] internal I/O peripheral [65], which took half of the I/O address space by doing a partial decoding on bit 0. All even addresses were decoded as valid ULA addresses. Partial decoding is used for all the added peripherals in this project. The complete multicon I/O mapping of the extended SweRVolf Core is presented in Table 17.

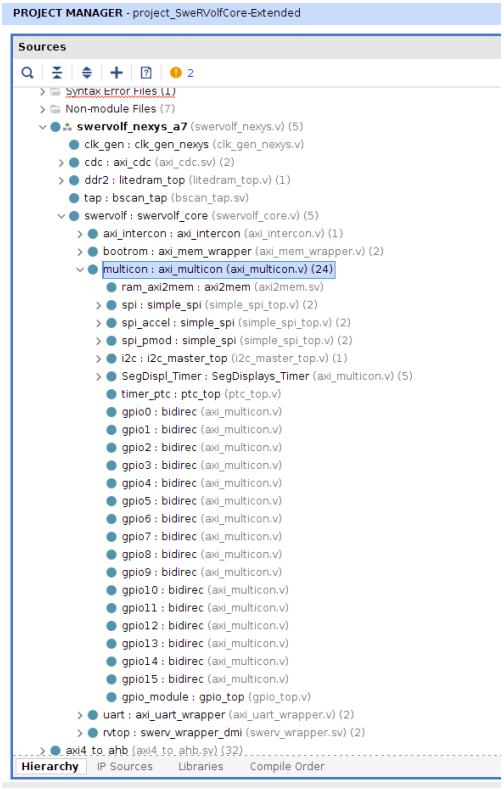


Figure 48: Hierarchical Project manager view in Vivado (own capture from project data)

lower 16 decoded bits	Preferred address	Peripheral
0001 0000 0000 0xxx	0x80001000	Version Information
0001 0000 0000 1xxx	0x80001008	Initialization information
0001 0000 0001 0xxx	0x80001010	Old GPIO
0001 0000 0010 0xxx	0x80001020	mtime from RISC-V privileged specification
0001 0000 0010 1xxx	0x80001028	mtimecmp from RISC-V privileged specification
0001 0000 0011 0xxx	0x80001030	Enable 7-segment displays
0001 0000 0011 1xxx	0x80001038	7-Segment data
0001 xxxx x1xx xxxx	0x80001040	BootROM SPI
0001 xxxx 1xxx xxxx	0x80001080	i2c peripheral
0001 xxx1 xxxx xxxx	0x80001100	Accelerometer SPI
0001 xx1x xxxx xxxx	0x80001200	Timer
0001 x1xx xxxx xxxx	0x80001400	Enhanced GPIO
0001 1xxx xxxx xxxx	0x80001800	PMOD A - NRF24L01 Radio SPI

Table 17: Partial address decoding on axi_multicon.v (own elaboration)

[58] **ULA**: Uncommited Logic Array

Using this decoding scheme, on reading requests, all peripherals write to the 64 wires *rdata*, connected to the wishbone interface.
This assignment of data is shown in Figure 49, where reg_addr is the 32-bit address arriving at the system controller, and each wb_* signal is generated by the specific peripheral implementation.

```
  case (reg_addr[5:3])
0 : reg_rdata <= {32'h`VERSION_SHA, version};
1 : reg_rdata <= {46'd0, i_ram_init_error, i_ram_init_done, 16'd0};
2 : reg_rdata <= 64'd0;
4 : reg_rdata <= mtime;
5 : reg_rdata <= mtimecmp;
  endcase

  assign rdata = reg_addr[6] ? {8{wb_spi_rdt}} :
                 (reg_addr[7] ? {8{wb_dat_o}} :
                 (reg_addr[8] ? {8{wb_spi_rdt_accel}} :
                 (reg_addr[9] ? {8{wb_dat_o_ptc}} :
                 (reg_addr[10] ? {wb_dat_o_gpio[31:0]} :
                 (reg_addr[11] ? {8{wb_spi_rdt_pmod}} :
                 reg_rdata)))));
```

Figure 49: data written from the multicon peripherals to the wishbone interface (own elaboration).

Besides the data registers, each peripheral may have additional registers that can be read from and, if applicable, registers to write to. These are implemented in specific modules within the Verilog code and they will be covered in each peripheral section of this chapter.
Bit 11 control on the reg_rdata assignment has been added to the address decoding, requiring a 0 for the peripherals to be enabled.

7.2.1 Stage 5: Extend SoC to include bidirectional GPIO and 7-Segment displays

These two extensions have not been created by this thesis author. They come from the ongoing development of the RVfpga UCM-Imagination project, but they make a very good addition to the final SoC, allowing for standalone clear data visualization.

SweRVolf includes a very basic, ad-hoc, implementation of GPIO, but it lacks the flexibility of a more general GPIO port array. It includes 64 i/o ports mapped in the address range 0x80001010, 0x80001017, although only 32 of those are physically connected to actual FPGA pins in the Vivado constraints file. 16 of these ports, with addresses 0x80001010 and 0x80001011 are connected to the on-board Leds. Another 16 ports are connected to the on-board switches, with addresses 0x80001012 and 0x80001013. However the direction of these ports is always fixed, meaning that the core has 16 output ports and 16 input ports. No direction change nor tri-state operation is allowed.

Although the project could work with these basic GPIOs, it seemed appropriate to actually have a set of full-fledged GPIOs, provided they were available, therefore this new GPIO module was added. Since the original switches ports are read during boot for deciding on the boot media, either SPI flash, serial or RAM, the original addresses for the switches are left with a constant response of 0x0000 (all off), indicating the desired operation for the developed SoC: Boot from SPI flash. Finally, the writing of the output signals (intended for the Leds) were commented in the axi_multicon.v file. Leaving the driving of the o_gpio wires to the new implemented module.

The new GPIO module is at address 0x80000400. It uses partial decoding, so anything intended for multicon addresses (0x80001xxx) with its bit 10 set, will be seen as a request to this module. Figure 50 shows the definition of this module within axi_multicon.v.

```
wire [31:0] wb_dat_o_gpio;
wire [31:0] en_gpio;
wire [31:0] i_gpio_temp;
wire [31:0] o_gpio_temp;

bidirec gpio0 (en_gpio[0], clk, o_gpio_temp[0], i_gpio_temp[0], io_data[0]);
bidirec gpio1 (en_gpio[1], clk, o_gpio_temp[1], i_gpio_temp[1], io_data[1]);
bidirec gpio2 (en_gpio[2], clk, o_gpio_temp[2], i_gpio_temp[2], io_data[2]);
bidirec gpio3 (en_gpio[3], clk, o_gpio_temp[3], i_gpio_temp[3], io_data[3]);
bidirec gpio4 (en_gpio[4], clk, o_gpio_temp[4], i_gpio_temp[4], io_data[4]);
bidirec gpio5 (en_gpio[5], clk, o_gpio_temp[5], i_gpio_temp[5], io_data[5]);
bidirec gpio6 (en_gpio[6], clk, o_gpio_temp[6], i_gpio_temp[6], io_data[6]);
bidirec gpio7 (en_gpio[7], clk, o_gpio_temp[7], i_gpio_temp[7], io_data[7]);
bidirec gpio8 (en_gpio[8], clk, o_gpio_temp[8], i_gpio_temp[8], io_data[8]);
bidirec gpio9 (en_gpio[9], clk, o_gpio_temp[9], i_gpio_temp[9], io_data[9]);
bidirec gpio10 (en_gpio[10], clk, o_gpio_temp[10], i_gpio_temp[10], io_data[10]);
bidirec gpio11 (en_gpio[11], clk, o_gpio_temp[11], i_gpio_temp[11], io_data[11]);
bidirec gpio12 (en_gpio[12], clk, o_gpio_temp[12], i_gpio_temp[12], io_data[12]);
bidirec gpio13 (en_gpio[13], clk, o_gpio_temp[13], i_gpio_temp[13], io_data[13]);
bidirec gpio14 (en_gpio[14], clk, o_gpio_temp[14], i_gpio_temp[14], io_data[14]);
bidirec gpio15 (en_gpio[15], clk, o_gpio_temp[15], i_gpio_temp[15], io_data[15]);

assign wb_cyc_i_gpio = reg_req & reg_addr[10];
assign wb_gpio_dat = !reg_req ? wb_gpio_dat_r : reg_wdata[31:0];

gpio_top gpio_module(
    .wb_clk_i     (clk),
    .wb_rst_i     (~rst_n),
    .wb_cyc_i     (wb_cyc_i_gpio | wb_cyc_i_gpio_r),
    .wb_adr_i     (wb_ptc_adr[7:0]),
    .wb_dat_i     (wb_gpio_dat[31:0]),
    .wb_sel_i     (4'b1111),
    .wb_we_i      (reg_we | reg_we_r),
    .wb_stb_i     (wb_cyc_i_gpio | wb_cyc_i_gpio_r),
    .wb_dat_o     (wb_dat_o_gpio),
    .wb_ack_o     (),
    .wb_err_o     (),
    .wb_inta_o    (),

    // External GPIO Interface
    .ext_pad_i    ({i_gpio[15:0],16'd0}),
    .ext_pad_o    (o_gpio_temp),
    .ext_padoe_o  (en_gpio)
```

```
module bidirec (input wire oe, input wire clk,
                input wire inp, output wire outp,
                inout wire bidir);

reg     a;
reg     b;

assign bidir = oe ? a : 32'bZ ;
assign outp  = b;

// Always Construct

always @ (posedge clk)
begin
    b <= bidir;
    a <= inp;
end

endmodule
```

```
wb_gpio_dat_r <= wb_gpio_dat;
```

Figure 50: Advanced GPIO implementation (ArTeCS [66])

The 7-segment module uses a counter that keeps each of the eight displays active for 2 milliseconds, resulting in a cycle time of 16ms. Due to the persistence of the light emitted by the displays in the human retina, the numbers would appear as being always on when, in reality, each one is on for two milliseconds and off for 14 milliseconds.
This is the standard way of driving 7-segment displays, as all digits share the 8 data (seven segments plus a dot) lines. By feeding the right number configuration on the data lines and enabling the appropriate display, the desired multi-number output is achieved. The details of this implementation are out of the scope of this project, but a summary is included for reference.

Two registers, at addresses 0x80001030 (8-bit enable) and 0x80001038 (32-bit data), control the operation of this module. A display can be individually enabled or disabled by writing a 0 or a 1 in the respective bit of the enable register. This bit, and the one coming from the timer, are then pass through an AND gate to actually enable the driving of a specific display.
The module implements an abstraction of the seven segments, accepting hexadecimal data to display. A 32-bit data register holds eight groups of 4 bits to represent the 0 to F hexadecimal number. No driving of the display dot or not-hexadecimal characters is allowed.
The signals are propagated through the system controller up to swervolf_nexys_a7, where they must be assigned to the actual FPGA pins connecting to the 7-segment displays, requiring adding these to the constraints file.

7.2.2 Stage 6: Add SPI to SoC - PMOD BareMetal testing

SPI is a key element of this SoC. It is used to communicate with the on-board accelerometer and with the external NRF24L01+ radio. The core for a standard SPI (Simple SPI) was already available in the SweRVolf Core implementation, since it´s used to access the flash SPI used for booting. Therefore the goal was to implement a new instance of the core within the system controller, propagate its signals up to the SweRVolf Core and, from there, to the desired hardware pins of the FPGA. The first test was performed by routing a new SPI module to the PMOD A connector of the Nexys 4 DDR board. The specific steps are detailed below, while a diagram of the PMOD connector is shown in Figure 51.

Pmod JA	Pmod JB	Pmod JC	Pmod JD	Pmod XDAC
JA1: C17	JB1: D14	JC1: K1	JD1: H4	JXADC1: A13 (AD3P)
JA2: D18	JB2: F16	JC2: F6	JD2: H1	JXADC2: A15 (AD10P)
JA3: E18	JB3: G16	JC3: J2	JD3: G1	JXADC3: B16 (AD2P)
JA4: G17	JB4: H14	JC4: G6	JD4: G3	JXADC4: B18 (AD11P)
JA7: D17	JB7: E16	JC7: E7	JD7: H2	JXADC7: A14 (AD3N)
JA8: E17	JB8: F13	JC8: J3	JD8: G4	JXADC8: A16 (AD10N)
JA9: F18	JB9: G13	JC9: J4	JD9: G2	JXADC9: B17 (AD2N)
JA10: G18	JB10: H16	JC10: E6	JD10: F3	JXADC10: A18 (AD11N)

Figure 51: Nexys 4 - DDR PMOD Connectors (Modified, [29])

7.2.2.1 axi_multicon.v changes:

- Instantiate a new SPI module and connect it to the SPI interface and to the wishbone slave interface, decoding the address for reading data from the bus.
- Code 4 shows the required Verilog code.
- Add the newly created wires to the axi_multicon module interface. In this case:
  - Output wire o_sclk_pmod – for the SPI clock signals
  - Output wire o_cs_n_pmod – for the SPI chip select signals
  - Output wire o_mosi_pmod – for the SPI Master to slave signals
  - Input wire i_miso_pmod – for the SPI Slave to master signals
- Create wire and reg for wishbone signal latching and assign them on the rising edge of the clock.
  - Wire wb_spi_cyc_pmod
  - Reg wb_spi_cyc_r_pmod
  - Wb_spi_cyc_r_pmod <= wb_spi_cyc_pmod

- Create an 8-line wire for wishbone data and assign it to *rdata*, as seen before, based on the decoded address of the peripheral.
    - Wire [7:0] wb_spi_rdt_pmod
    - See reg_addr[11] assignment in Figure 49, a few pages above.

```
// SPI-2

   assign wb_spi_cyc_pmod = reg_req & reg_addr[11];

   simple_spi spi_pmod
     (// Wishbone slave interface
      .clk_i  (clk),
      .rst_i  (~rst_n),
      .adr_i  (wb_spi_adr),
      .dat_i  (wb_spi_dat),
      .we_i   (reg_we | reg_we_r),
      .cyc_i  (wb_spi_cyc_pmod | wb_spi_cyc_r_pmod),
      .stb_i  (wb_spi_cyc_pmod | wb_spi_cyc_r_pmod),
      .dat_o  (wb_spi_rdt_pmod),
      .ack_o  (),
      .inta_o (),
      // SPI interface
      .sck_o  (o_sclk_pmod),
      .ss_o   (o_cs_n_pmod),
      .mosi_o (o_mosi_pmod),
      .miso_i (i_miso_pmod)   );
```

Code 4: SPI module instantiated in multicon (own elaboration)

7.2.2.2 swervolf_core.v changes

- Create external wires at the interface level:
    - Output wire o_pmod_sclk – for the SPI clock signals
    - Output wire o_pmod_cs_n – for the SPI chip select signals
    - Output wire o_pmod_mosi – for the SPI Master to slave signals
    - Input wire i_pmod_miso – for the SPI Slave to master signals
- Propagate the signals by adding the required connections in the multicon module definition:
    - .o_sclk_pmod (o_pmod_sclk)
    - .o_cs_n_pmod (o_pmod_cs_n)
    - .o_mosi_pmod (o_pmod_mosi)
    - .i_miso_pmod (i_pmod_miso)

7.2.2.3 swervolf_nexys_a7.v changes

- Create external wires at the interface level:
    - Output wire PMOD_SCLK – for the SPI clock signals
    - Output wire PMOD_CSN – for the SPI chip select signals
    - Output wire PMOD_MOSI – for the SPI Master to slave signals
    - Input wire PMOD_MISO – for the SPI Slave to master signals
- Propagate the signals by adding the required connections in the swervolf_core module definition:
    - .o_pmod_sclk (PMOD_SCLK)
    - .o_pmod_cs_n (PMOD_CSN)
    - . o_pmod_mosi (PMOD_MOSI)
    - . i_pmod_miso (PMOD_MISO)

7.2.2.4 swervolf_nexys.xdc changes

Connect the signals coming from swervolf_nexys_a7 to the desired pins of the FPGA by using information from the general constraints file for the Nexys 4 DDR board [67] and modifying in the lines included in Code 5.

```
set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports { PMOD_MOSI }]; #IO_L20N_T3_A19_15 Sch=pmod_mosi
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { PMOD_MISO }]; #IO_L16P_T2_A28_15 Sch=pmod_miso
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { PMOD_SCLK }]; #IO_L16N_T2_A27_15 Sch=pmod_sck
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { PMOD_CSN }]; #IO_L21P_T3_DQS_15 Sch=pmod_cs
```

Code 5: Constrain definition for SPI on PMOD A (own elaboration)

7.2.2.5 Test the newly created SPI module

For testing the new SPI interface, a Logic 8 analyzer from Saleae [68] was connected to the PMOD signals and a MOSI-MISO loopback was created with a physical wire. An ad-hoc RISC-V asm program was developed to read eight switches and their corresponding Led if the switch was in on position. The switches information was also send through the PMOD SPI interface and read back. This read value was then presented in the other array of 8 Leds. If both groups of 8 Leds were to show the same information, the MISO-MOSI connection would work as intended. Also, the logic analyzer was capturing the data and presenting the waveform. Note that this test was done following the Simple SPI documentation, so no *chip select* signal was captured at this point. Code 6 shows the RISC-V assembler test program and a picture of the test setup is shown in Figure 52.

```
        # SPI Loopback test PMOD
                # March 2020. Daniel León UCM
                #---------------------------------

                #define LEDs_ADDR       0x80001010
                #define SWs_ADDR        0x80001012
                #define SegEn_ADDR      0x80001070
                #define SegDig_ADDR     0x80001080
                #define SPCR            0x80001800
                #define SPSR            0x80001808
                #define SPDR            0x80001810
                #define SPER            0x80001818

                .globl _start
                _start:
                spiconf:
                        li x1, SPCR # control register
                        li t0, 0x53 #01010011 no ints, core enabled, reserved, master, cpol=0,
                        # cha=0, clock divisor 11 for 4096
                        sb t0, 0(x1)
                        li x1, SPER # extension register
                        li t0, 0x02 # int count 00 (7:6), clock divisor 10 (1:0) for 4096
                        sb t0, 0(x1)
                spiclearif:
                        li x1, SPSR # status register
                        lb t0, 0(x1) # clear SPIF by writing a 1 to bit 7
                        ori t0,t0,0x80
                        sb t0, 0(x1)
                readswitches:
                        li  x1, SWs_ADDR # read switches
                        lb  t0, 0(x1) # read only 0-7

                # deassert CS missing if using spi slave device. Should be here

                spisenddata:
```

```
        li x1, SPDR # data register
        sb t0, 0(x1) # send the switches 0-7 status
spitestif:
        li x1, SPSR # status register
        lb t0, 0(x1)
        andi t0, t0, 0x80
        li t1, 0x80
        bne t0,t1,spitestif # loop while SPSR.bit7 == 0. (tx in progress)

# assert CS missing if using spi slave device. Should be here

spireaddata:
        li x1, SPDR # data register
        lb t0, 0(x1) # read the message from SPI
             # (the switches 0-7 status if a loopback)
writeleds:
        li x1, LEDs_ADDR
        sb  t0, 0(x1)

j spiclearif # cycle

.end
```

Code 6: RISC-V Assembler SPI test code over PMOD A (own elaboration)

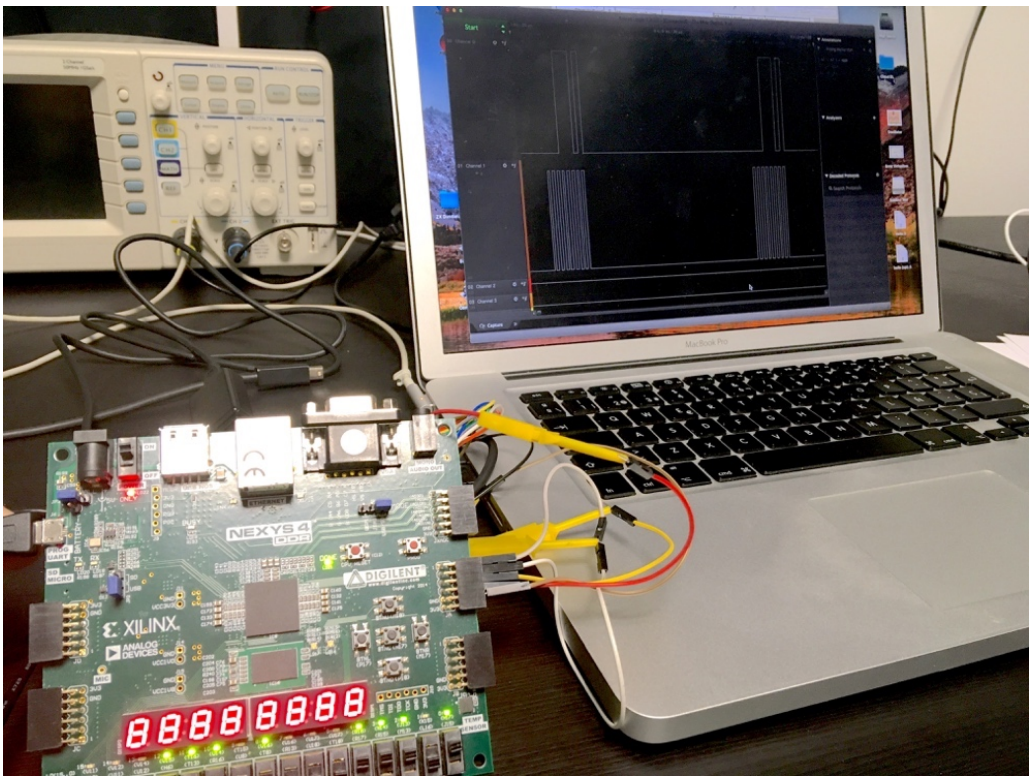


Figure 52: SPI core test setup (own capture)

7.2.3 Stage 7: Add ADXL362 BareMetal support

This stage covers the development of a baremetal, RISC-V assembler, driver for the ADXL362. A second SPI module was added to multicon, propagated up to swervolf_nexys_a7.v and then connected to the FPGA pins that are physically attached to the accelerometer in the Nexys 4 DDR board. The steps for this procedure are the same as the ones described in the previous chapter. This module was placed in address 0x80001100. Code 7 shows the SPI driver library and the UART driver library, followed

by the main program that reads the X axis acceleration data and sends it to the serial port.

```
# ------------------------------------------------------------------------------------------------
# SPI Basic Library for SwervOlf
# Apr. 2020 - Daniel Leon
# FDI / UCM
# ------------------------------------------------------------------------------------------------

# Register addresses for SPI Peripheral
# ------------------------------------------------------------------------------------------------
#define SPCR    0x80001100
#define SPSR    0x80001108
#define SPDR    0x80001110
#define SPER    0x80001118
#define SPCS    0x80001120

# Function: Initialize SPI peripheral
# call:  by call ra, spiInit
# inputs: None
# outputs: None
# destroys: t0, t1
# ------------------------------------------------------------------------------------------------
spiInit:
    li t1, SPCR # control register
    li t0, 0x51 #01010011 no ints, core enabled, reserved, master, cpol=0, cha=0, clock divisor
11 for 4096
    sb t0, 0(t1)
    li t1, SPER # extension register
    li t0, 0x01 # int count 00 (7:6), clock divisor 10 (1:0) for 4096
    sb t0, 0(t1)
    ret
# ------------------------------------------------------------------------------------------------

# Function: Pull CS Line to either high or low - Provides quick calls spiCSUp and spiCSDown
# call:  by call ra, spiCS
# inputs: CS status in a0 (0 is low, 1 is high)
# outputs: None
# destroys: t0
# ------------------------------------------------------------------------------------------------
spiCS:
        li t0, SPCS # CS register
        sb a0, 0(t0) # Send CS status
        ret
# ------------------------------------------------------------------------------------------------
spiCSUp:
        li a0, 0x00
        j spiCS
spiCSDown:
        li a0, 0xFF
        j spiCS

# Function: Send byte through SPI and get the slave data back
# call:  by call ra, spiSendGetData
# inputs: data byte to send in a0
# outputs: received data byte in a1
# destroys: t0, t1
# ------------------------------------------------------------------------------------------------
spiSendGetData:

internalSpiClearIF: # internal clear interrupt flag
        li t1, SPSR # status register
        lb t0, 0(t1) # clear SPIF by writing a 1 to bit 7
        ori t0,t0,0x80
        sb t0, 0(t1)
internalSpiActualSend:
        li t0, SPDR # data register
        sb a0, 0(t0) # send the byte contained in a0 to spi
internalSpiTestIF:
        li t1, SPSR # status register
        lb t0, 0(t1)
        andi t0, t0, 0x80
```

```
        li t1, 0x80
        bne t0,t1,internalSpiTestIF # loop while SPSR.bit7 == 0. (transmission in progress)
internalSpiReadData:
        li t0, SPDR # data register
        lb a1, 0(t0) # read the message from SPI
        ret
# ------------------------------------------------------------------------------------------------


# ------------------------------------------------------------------------------------------------
# UART Basic Library for SwervOlf
# Adapted from hello_uart.S
# Apr. 2020 - Daniel Leon
# FDI / UCM
# ------------------------------------------------------------------------------------------------

# Register addresses for Uart Peripheral
# ------------------------------------------------------------------------------------------------
#define CONSOLE_ADDR 0x80001008
#define HALT_ADDR    0x80001009
#define UART_BASE    0x80002000

#define REG_BRDL (4*0x00)  # Baud rate divisor (LSB)
#define REG_IER (4*0x01)   # Interrupt enable reg.
#define REG_FCR (4*0x02)   # FIFO control reg.
#define REG_LCR (4*0x03)   # Line control reg.
#define REG_LSR (4*0x05)   # Line status reg.
#define LCR_CS8 0x03       # 8 bits data size
#define LCR_1_STB 0x00     # 1 stop bit
#define LCR_PDIS 0x00      # parity disable

#define LSR_THRE 0x20
#define FCR_FIFO 0x01      # enable XMIT and RCVR FIFO
#define FCR_RCVRCLR 0x02   # clear RCVR FIFO
#define FCR_XMITCLR 0x04   # clear XMIT FIFO
#define FCR_MODE0 0x00     # set receiver in mode 0
#define FCR_MODE1 0x08     # set receiver in mode 1
#define FCR_FIFO_8 0x80    # 8 bytes in RCVR FIFO


# Function: Initialize UART peripheral
# call:  by call ra, uartInit
# inputs: None
# outputs: None
# destroys: t0, t1
# ------------------------------------------------------------------------------------------------
uartInit:
        li t0, UART_BASE
        li t1, 0x80               # Set DLAB bit in LCR
        sb t1, REG_LCR(t0)
        li t1, 27                 # Set divisor regs
        sb t1, REG_BRDL(t0)
        li t1, LCR_CS8 | LCR_1_STB | LCR_PDIS # 8 data bits, 1 stop bit, no parity, clear DLAB
        sb t1, REG_LCR(t0)
        li t1, FCR_FIFO | FCR_MODE0 | FCR_FIFO_8 | FCR_RCVRCLR | FCR_XMITCLR
        sb t1, REG_FCR(t0)
        sb zero, REG_IER(t0)      # disable interrupts
        ret


# Function: Send byte through UART
# call:  by call ra, uartSendByte
# inputs: a0, byte to be sent
# outputs: None
# destroys: t0, t1
# ------------------------------------------------------------------------------------------------
uartSendByte:
        li t1, UART_BASE
        lb t0, REG_LSR(t1)  # Check for space in UART FIFO
        andi t0, t0, LSR_THRE
        beqz t0, uartSendByte
        sb a0, 0(t1)
        ret
```

```
# Function: Send string through UART (terminated by \0)
# call:  by call ra, uartSendString
# uses: uartSendByte
# inputs: a0, address of first character of string to be sent
# outputs: None
# destroys: t0, t1, t2
# ------------------------------------------------------------------------------------------
uartSendString:
        li t1, UART_BASE
        add t2,zero,ra # save caller address
        add a1,zero,a0 # use a1 as index

        lb a0, 0(a1)   # Load first byte
internalNextChar:
        call ra, uartSendByte
        addi a1, a1, 1
        lb a0, 0(a1)
        bne a0, zero, internalNextChar

        add ra,zero,t2 # restore caller address
        ret
# ------------------------------------------------------------------------------------------


# ------------------------------------------------------------------------------------------
# ADXL362 Accelerometer data for SweRVolf
# Apr. 2020 - Daniel Leon
# FDI / UCM
# ------------------------------------------------------------------------------------------
.globl _start
_start:
configureHW:
        call ra, spiInit
        call ra, uartInit
sayHello:
        la a0, welcome
        call uartSendString
        call spiCSDown # Quick Enable Sensor
        li a0,0x0A
        call spiSendGetData
        li a0,0x2D
        call spiSendGetData
        li a0,0x02 # enable measurement
        call spiSendGetData
        call spiCSUp
getXaxis:
        call spiCSDown
        li a0, 0x0B
        call spiSendGetData
        li a0, 0x08
        call spiSendGetData
        li a0, 0xFF
        call spiSendGetData
        call spiCSUp
        add a0,zero, a1
        call uartSendByte
        j getXaxis # cycle

#include "libSpi.asm"
#include "libUart.asm"
.section .data
welcome:
.string "SPI Accelerometer Test - Daniel Leon Apr. 2020\n\n"
.end
```

Code 7: RISC-V SPI and UART Libraries and ADXL362 baremetal support (own elaboration)

7.2.4 Stage 8: Add ADXL362 support for Zephyr

Zephyr OS supports the SPI-Simple core and the ADXL362 accelerometer. Adding it requires enabling the drivers in Zephyr's kernel using *west build -t guiconfig*. Within "Device drivers", "SPI hardware bus support /SPI port 1" and "Sensor Drivers/ADXL362 sensor" must be checked. Figure 53 shows the *guiconfig* user interface with the selected options. Additionally, two files have to be edited, within the zephyr folder, to connect these selected drivers with the actual implemented hardware.

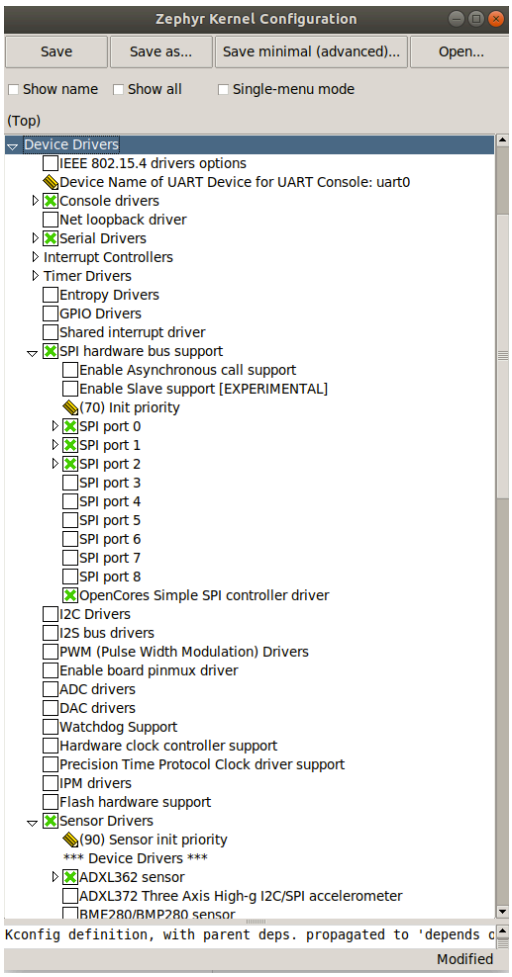


Figure 53: Zephyr's kernel configuration with guiconfig (own capture)

File "zephyr/dts/riscv/riscv32swerv.dtsi" must include the code shown in Code 8, while "boards/riscv/swervolf_nexys/swervolf_nexys.dts" must be modified to include the code shown in Code 9.

```
                spi1: spi@80001100 {
                        compatible = "opencores,spi-simple";
                        reg = <0x80001100 0x40>;
                        reg-names = "control";
                        label = "SPI1";
                        status = "enabled";
                        #address-cells = <1>;
                        #size-cells = <0>;
                };
```

Code 8: riscv32swerv.dtsi modifications for SPI1 (own elaboration)

```
                &spi1 {
                status = "okay";

                adi_adxl362: adi_adxl362@0 {
                        compatible = "adi_adxl362";
                        label = "adi_adxl362";
                        reg = <0>;
                        spi-max-frequency = <2000000>;
                };
        };
```

Code 9: swervolf_nexys.dts modifications for adxl362 support (own elaboration)

However, while the SPI driver was working, the ADXL362 driver failed to compile with the following error:

```
zephyr/include/generated/devicetree_unfixed.h:603:36:
error: 'DT_N_S_soc_S_spi_80001100_S_adi_adxl362_0_P_spi_max_frequency' undeclared here (not in a
function); did you mean 'DT_N_S_soc_S_spi_80001040_S_flash_0_P_spi_max_frequency'?
```

The *spi-max-frequency* was declared exactly the same as the parameter for the recognized bootrom SPI-Flash, so a deeper reason is probably involved. Unfortunately, although this error was reported in Zephyr supports groups and an issue was created on the SweRVolf github [69], no solution was found, so a port of the baremetal low level driver for SPI was performed and a driver for the ADXL362 accelerometer was developed from scratch. Both drivers are included in Code 12, within the Stage 13 chapter of work package 4. While this approach required a bigger effort, it works perfectly. A capture of the output for a Zephyr test program using this driver is shown in Figure 54.

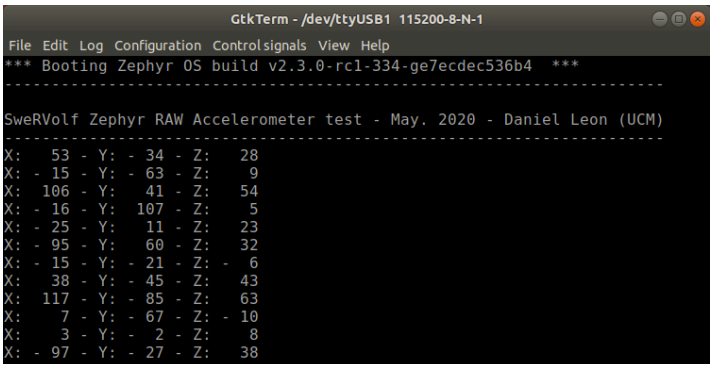


Figure 54: Zephyr test program output for the ADXL362 driver (own capture)

7.3 Work Package 3: Connected SoC

The goal of this work is to connect the SoC to the Internet to enable an upstream and a downstream communication with an MQTT broker service. Since adding ethernet support was postponed for future work of the project, the NRF24L01+ radio approach was used. This radio device does not contain a general purpose microcontroller and must be managed by an external system. The project's extended SoC does effectively configure and use the radio to achieve a two-way communication with an NRF radio-enabled border router. The work package extended from mid-May to early June 2020.

7.3.1 Stage 9: Extend the SoC to support the NRF24L01+ radio

The NRF24L01+ radio offers an SPI management interface, so the original PMOD-attached SPI module in the extended SoC was reenabled to connect to the external hardware. The connections from the radio module to the PMOD A header on the Nexys board, and the connections from a second radio module to the NodeMCU border router are presented in Table 18. The radio module fits nicely into the PMOD header using the VCC[59] and GND[60] pins. Figure 55 shows the radio module attached to the Nexys board.

Wire Color	Function	NRF24L01 Pin	PMOD Pin	NodeMCU Pin
Black*	GND	Pin 1 - GND	Pin 5 - GND	GND
White*	3.3V	Pin 2 - VCC	Pin 6 - 3V3	3V
Purple	CE	Pin 3 - CE	Pin 2	D2
Grey	CS	Pin 4 - CSN	Pin 3	D8 - HCS
Green	CLOCK	Pin 5 - SCK	Pin 7	D5 - HSCLK
Blue	MOSI	Pin 6 - MOSI	Pin 1	D7 - HMOSI
Yellow	MISO	Pin 7 - MISO	Pin 8	D6 - HMISO
Orange	NRF IRQ	Pin 8 - IRQ	Pin 9 (n/c)	D1

* NRF module power pins are directly plugged into the PMOD connector

Table 18: Radio module connections to the Nexys board and to the NodeMCU border router (own elaboration)

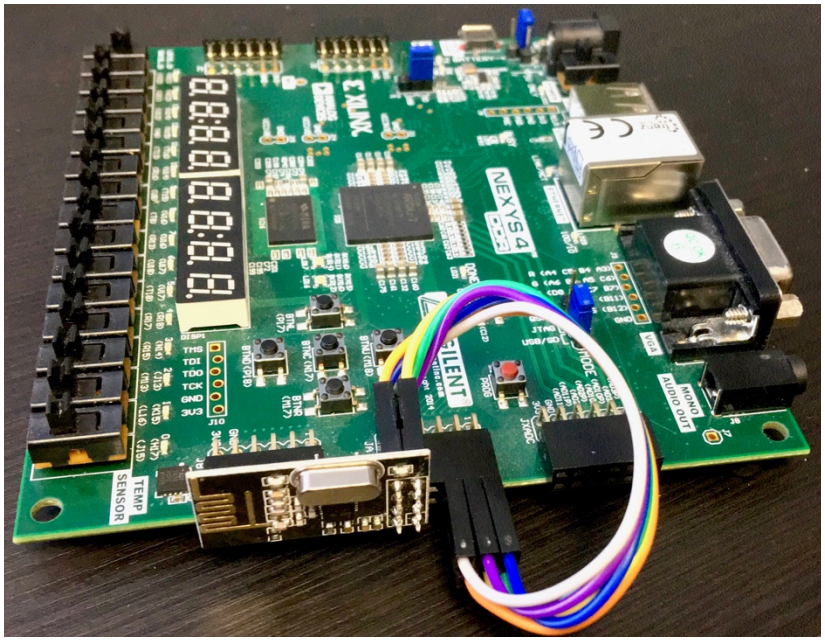


Figure 55: NRF24L01+ radio module attached to PMOD A in the nexys board (own capture)

Although the orange wire (NRF IRQ) is connected to the pin 9 of the PMOD A header, this signal is not used in the project and, therefore, it is not routed to the core.

[59] **VCC**: Voltage common collector. Power input of a circuit
[60] **GND**: Ground. Ground reference of a circuit

A GPIO port (GPIO15), originally connected to the leftmost Led of the Nexys 4 DDR board, was repurposed por acting as the NRF CE control signal. The constraints file was edited to change the port assignment to PMOD A pin 2, and the driving code for the Leds was adjusted to avoid modifying the value of that GPIO port.

At this point of the project, the SoC was fully defined. Its logic core drains as little as 19mW as shown in Figure 56. To put this into perspective, the ESP8266 logic core drains 50mW [70] at roughly the same MHz clock.

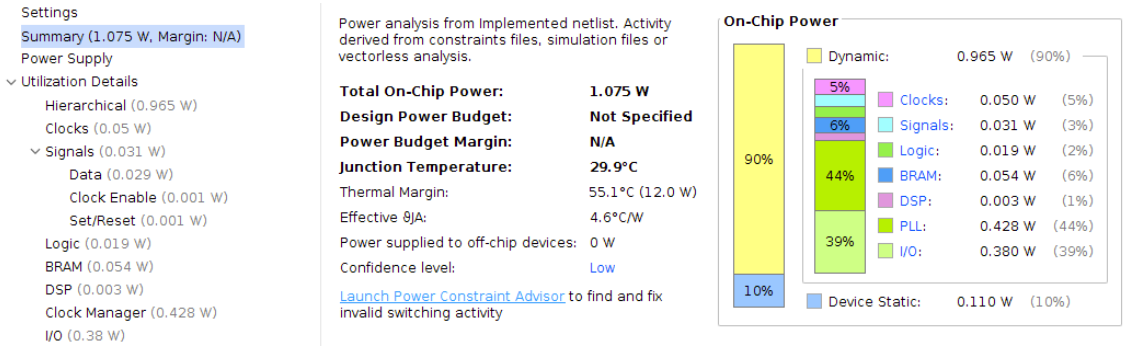


Figure 56: Power analysis of the complete SoC (own capture)

7.3.2 Stage 10: Develop the border router for SoC-router Communication

The border router is implemented using the Arduino IDE on a NodeMCU (ESP8266 SoC) platform. While there are libraries for communicating with the NRF24L01+ module, none offered the required message piggybacking as demanded by the project, so a new radio library was developed from scratch. The border router configures the radio as a primary reception station (PRX) and enables the communication pipe 0 to receive data from the Nexys configured radio address. It enables the interrupt line on the radio module and offers an ISR[61] to manage incoming radio data from the node. In this interrupt service routine, and having the auto-acknowledgement, dynamic payload, and ack payload flags enabled, it inserts downstream data to be sent back to the Nexys node if any information is available. The implementation of this library is included in the complete code, as presented in Code 10. The NodeMCU connection to the radio is shown in Figure 57.

[61] **ISR**: Interrupt Service Routine

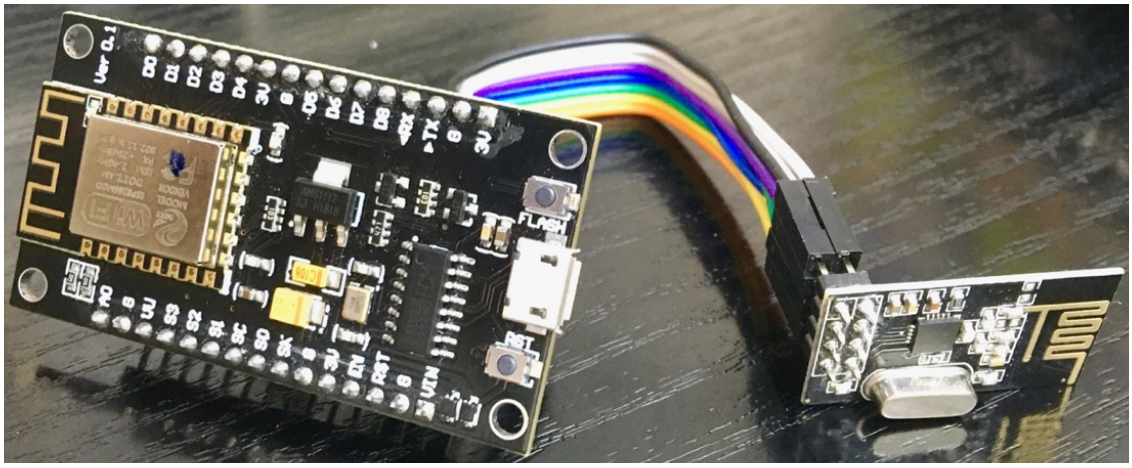


Figure 57: Border router with a NRF24L01+ radio module attached (own capture)

7.3.3 Stage 11: Develop Zephyr low level drivers for the radio

This stage outcome is a working Zephyr library for communication with the NRF24L01+ radio module. It uses the previously ported low-level SPI library. The Extended SoC is configured as a primary transmitter (PTX) on pipe 0 and the auto-acknowledgement support and dynamic payload is enabled. Since the communication is initiated by this node, no IRQ support from the radio module is needed, instead, it checks whether the returning ack package contains attached data and, in this case, retrieves it from the RX FIFO of the radio module. Lost ACK packages are logged to the console as shown in Figure 58. This driver's code is included in the full Zephyr application code presented later in Code 12.

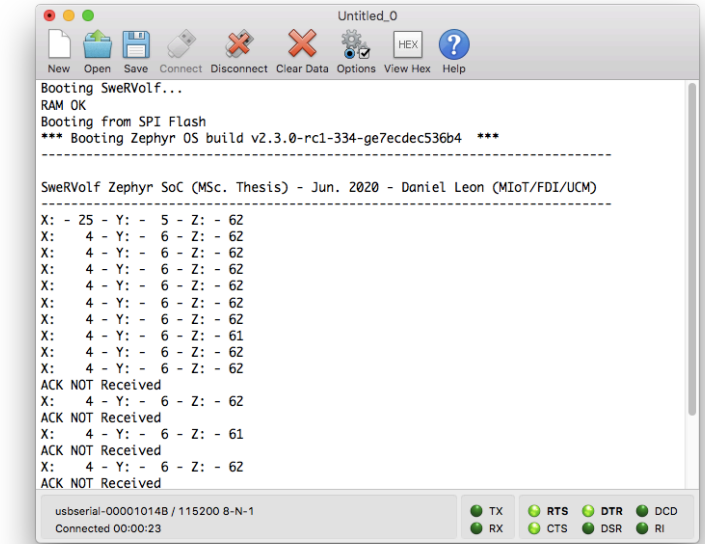


Figure 58: lost ACK packages logged by the Zephyr application (own capture)

7.3.4 Stage 12: Enable MQTT support on the border router for the Internet side

The border router is configured as a WIFI station and connects to a network access point. By means of the PubSubClient MQTT library, a subscription to the "field5" topic of the

ThingSpeak Channel is performed. (See Stage 15 / ThingSpeak channel configuration, later in the document). A callback on MQTT received data enables the piggybacking of the subscription received message into the radio ACK packet. On the other hand, the border router receives all messages sent from the Nexys node. Publishing of those messages is limited by the maximum update rate in ThingSpeak, that for the paid version used in this project, this update rate is set to one second. The border router uses two MQTT publishing strategies to meet this limitation and minimize Internet traffic:

- It uses a timer to enable MQTT publishing. This timer is set by the THINGSPEAK_UPDATE_RATE parameter, currently set to 1000ms. Every time a message is published, a flag denying further publishing is set, and the timer is launched. The callback for the timer re-enables MQTT publishing by clearing the flag. All Nexys node messages arriving between updates are discarded, but logged to the serial console if present.
- It uses sensitivity thresholds to decide whether to publish acceleration data or not. This is managed by the SENSITIVITY parameter and it is currently set to 1. The border router stores the latest published acceleration data and only publishes new messages if the difference of acceleration data, for any given axis, is higher than the SENSITIVITY parameter. In this case, the latest published data information stored in the border router is also updated.

The border router, however, logs all incoming and outgoing messages, as well as connection information, to the attached console, if present. The full border router code is included in Code 10: Full border router code. A sample output of the border router console is shown in Figure 59.

```
#define THINGSPEAK_UPDATE_RATE 1000
#define SENSITIVITY 1

const unsigned char ADDRESS[5] = {0x51, 0xA2, 0x53, 0xA4, 0x28};
#define PL_SIZE 3
int8_t payload[PL_SIZE];
int8_t oldPayload[PL_SIZE];
int8_t dataBack;

#define W_TX_PAYLOAD   0xA0
#define W_ACK_PAYLOAD  0xA8 // Ack Payload on pipe 0
#define R_RX_PAYLOAD   0x61
#define W_REGISTER     0x20
#define FLUSH_RX       0xE2
#define FLUSH_TX       0xE1
#define NRF_NOP        0xFF

// NRF24L01 REGISTER MAP
#define CONFIG         0x00
#define EN_AA          0x01
#define EN_RXADDR      0x02
#define SETUP_AW       0x03
#define SETUP_RETR     0x04
#define RF_CHANNEL     0x05
#define RF_SETUP       0x06
#define STATUSREG      0x07
#define RX_ADDR_P0     0x0A
#define RX_ADDR_P1     0x0B
#define RX_ADDR_P2     0x0C
#define TX_ADDR        0x10
#define RX_PW_P0       0x11
#define RX_PW_P1       0x12
#define RX_PW_P2       0x13
#define DYNPD          0x1C
```

```
#define FEATURE        0x1D
#define REGISTER_MASK  0x1F

#define NRF_CSN 15  // D8
#define NRF_CE 5    // D1
#define NRF_INT 4   // D2

#include <SPI.h>
#include "Ticker.h"
#include <PubSubClient.h>
#include <ESP8266WiFi.h>

Ticker ThingSpeakTimeOut;
uint16_t TSTimeOut = THINGSPEAK_UPDATE_RATE;
bool tsIsCleared = true;
bool intReceived = false;
WiFiClient wifiClient;
PubSubClient mqttClient;
#define STASSID "TFM_MIOT"
#define STAPSK  ""

#define topic  "channels/1074528/publish/2L5ASPHAI8CDMX7H"
#define subTopic  "channels/1074528/subscribe/fields/field5/A6ZY2JBS4PGZ57EG"
const char* mqttServer = "mqtt.thingspeak.com";
const uint16_t mqttPort =1883;

void write_register(unsigned char NRF_register, unsigned char value) {
    unsigned char s;
    digitalWrite(NRF_CSN,LOW);
    delay(10);
    s = W_REGISTER | (REGISTER_MASK & NRF_register);
    SPI.transfer(s);
    SPI.transfer(value);
    digitalWrite(NRF_CSN,HIGH);
    delay(10);
}

void clear_rx_int(void){
    unsigned char NRF_Status;
    digitalWrite(NRF_CSN,LOW);
    NRF_Status=SPI.transfer(W_REGISTER | (REGISTER_MASK & STATUSREG)); // Status register
    NRF_Status|=0x70;// write 1 to bit 6 to reset RX_DR, set INT low, clear all INTs
    SPI.transfer(NRF_Status);
    digitalWrite(NRF_CSN,HIGH);
}

void write_register_multi(unsigned char NRF_register, unsigned char *data, unsigned char size) {
    digitalWrite(NRF_CSN,LOW);
    delay(10);
    SPI.transfer(W_REGISTER | (REGISTER_MASK & NRF_register));
    while (size--) {
        SPI.transfer(*data++);
    }
    digitalWrite(NRF_CSN,HIGH);;
    delay(10);
}

unsigned char read_nrf(void)
{
    unsigned char r;
    digitalWrite(NRF_CSN,LOW);;
    delay(10);
    r = SPI.transfer(NRF_NOP);
    digitalWrite(NRF_CSN,HIGH);
    delay(10);
    return r;
}

void flush_rx(void)
{
   digitalWrite(NRF_CSN,LOW);
    SPI.transfer(FLUSH_RX);
    digitalWrite(NRF_CSN,LOW);
}
```

```
void flush_tx(void)
{
   digitalWrite(NRF_CSN,LOW);
    SPI.transfer(FLUSH_TX);
    digitalWrite(NRF_CSN,LOW);
}

void executeReceiveData() {
   uint8_t plCnt;
   intReceived = false;
   digitalWrite(NRF_CSN,LOW);
   SPI.transfer(R_RX_PAYLOAD);   // READ fifo
   for (plCnt=0; plCnt<PL_SIZE; plCnt++) {
     payload[plCnt]=SPI.transfer(NRF_NOP);
   }
    digitalWrite(NRF_CSN,HIGH);

    delayMicroseconds(10);  // ACK PiggyBack Payload
    if (dataBack !=0) {
      digitalWrite(NRF_CSN,LOW);
      SPI.transfer(W_ACK_PAYLOAD);
      SPI.transfer(dataBack);
      digitalWrite(NRF_CSN,HIGH);
      dataBack=0;
    }
    clear_rx_int();    // Clear all interrupt flags
    Serial.print("Datos recibidos: ");
    for (plCnt=0; plCnt<PL_SIZE; plCnt++) {
      Serial.print(payload[plCnt]);
      Serial.print(" ");
    }
    Serial.print("\n");
    tsMqttPublish();
}

ICACHE_RAM_ATTR void nrfIsr() {
 executeReceiveData();
}

void clearTS() {
  tsIsCleared = true;
}
void tsMqttPublish() {
  char mqttPayload[100];
  if ( (abs(payload[0]-oldPayload[0]) >SENSITIVITY )
    || (abs(payload[1]-oldPayload[1])>SENSITIVITY)
    || (abs(payload[2]-oldPayload[2])>SENSITIVITY)) {
    memcpy(oldPayload, payload,PL_SIZE);
    sprintf(mqttPayload,
"field1=%d&field2=%d&field3=%d&field4=%d",payload[0],payload[1],payload[2],-payload[1]);
    if (tsIsCleared) {
      mqttClient.publish(topic, mqttPayload);
      tsIsCleared = false;
      ThingSpeakTimeOut.once_ms(TSTimeOut,clearTS);
      Serial.print("Published to ThingSpeak: ");
      Serial.println(mqttPayload);
    }
  }
}

void mqttCallback(char* t, byte* p, unsigned int l) {
   Serial.print("Recibido dato de subscripción: ");
   p[l]=0;
   dataBack = atoi((const char*)p);
   Serial.println(dataBack);
}
void setup() {
  pinMode(NRF_CE,OUTPUT);
  pinMode(NRF_CSN,OUTPUT);
  pinMode(NRF_INT, INPUT_PULLUP);

  digitalWrite(NRF_CE,HIGH);
  digitalWrite(NRF_CSN,HIGH);
```

```
  SPI.begin();
  Serial.begin(115200);
  Serial.println("\n\n\n\n\n\n\n\nRouter Border Starting : TFM Daniel Leon 2020");
  Serial.println("-----------------------------------------------");
  WiFi.mode(WIFI_STA);
  WiFi.begin(STASSID,STAPSK);
  Serial.print("Connecting to Wifi ");
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println(" Wifi Connected");
  mqttClient.setClient(wifiClient);
  mqttClient.setServer(mqttServer,mqttPort);
  mqttClient.connect("BorderRouter","DL_TFM","68UT66A52JEUMBHS");
  Serial.println("MQTT connected to ThingSpeak");
  mqttClient.subscribe(subTopic);
  mqttClient.setCallback(mqttCallback);
  Serial.println("Subscribed to field5");

  write_register(SETUP_AW,0x03); // 5 bytes address
  write_register_multi(RX_ADDR_P0, (unsigned char*)ADDRESS, 5);
  write_register_multi(TX_ADDR, (unsigned char*)ADDRESS, 5);
  write_register(RX_PW_P0, PL_SIZE);// Payload of size PL_SIZE (overridden by dynamic payload)
  write_register(EN_AA,0xFF); // Enable AutoACK on Pipe 0
  write_register(FEATURE,0x06); // Enable Dynamyc Payload and payloads with ACK
  write_register(DYNPD, 0x01); // Enable dinamic Payload on pipe 0 - needed for payload-ack
  write_register(CONFIG, 0x3B); //Power UP, Receive Mode, mask all IRQ but RX

  attachInterrupt(digitalPinToInterrupt(NRF_INT), nrfIsr, FALLING);
}

void loop() { mqttClient.loop(); }
```

Code 10: Full border router code (own elaboration)

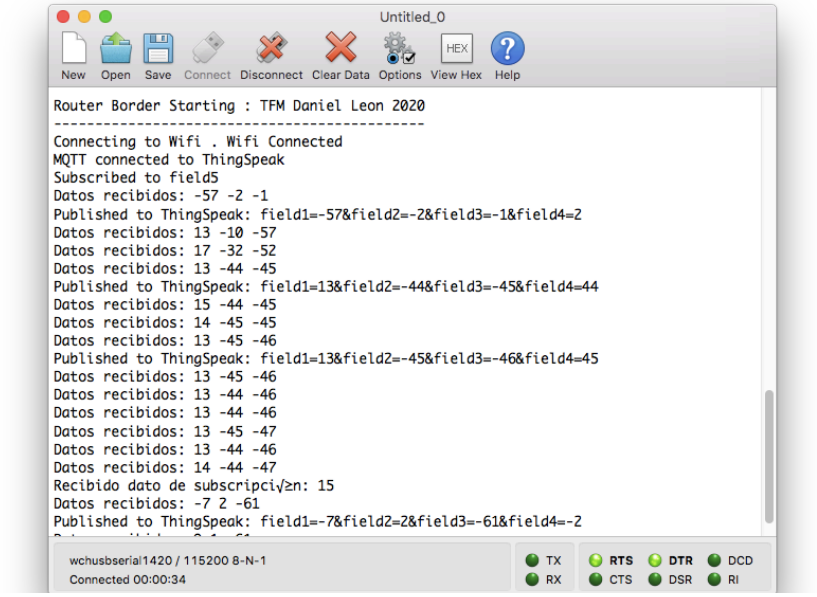


Figure 59: border router console output (own capture)

7.4 Work Package 4: Full proof-of-concept environment

This work package includes two stages and was addressed during June, 2020. It is an extension of the Connected SoC work package, where the SoC application has been polished and the cloud services integrated. Additionally, the Zephyr application is stored

in Flash memory for autonomous booting. A full schematic of the proof-of-concept environment is presented in the Figure 60 below.

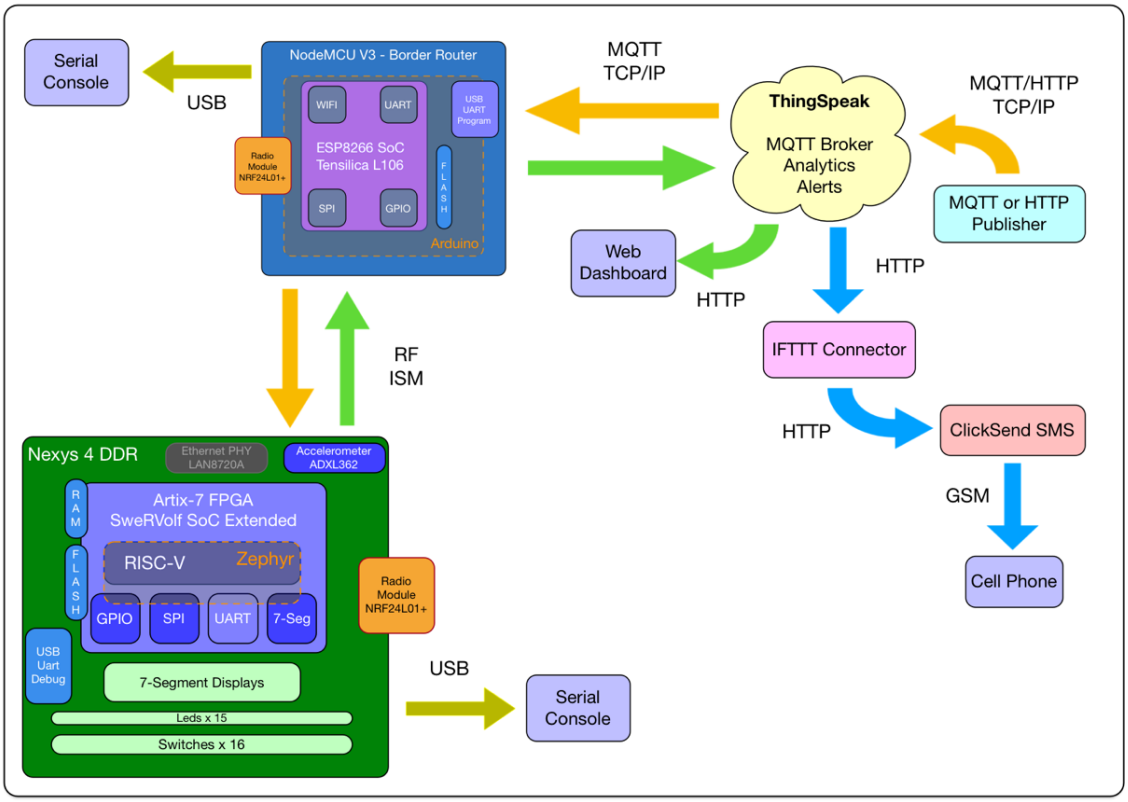


Figure 60: Full proof of concept environment (own elaboration)

The green arrows represent the accelerometer data originated at the node, while the yellow ones show the data flow originated by the external publisher and pushed down to the node. Serial console lines are a mix of colors because data from both sources is sent to the consoles. Finally, the blue lines indicate the flow of the action triggered by the ThingSpeak server that finalizes with an SMS delivery.

The Nexys 4 DDR board runs Zephyr OS over the extended SweRVolf Core, by loading it into the FPGA from an SD file and then loading the Zephyr compiled code from the SPI flash. It is executing a program that checks the acceleration data from the on-board accelerometer. Besides showing the information on the local 7-segment display and simulating a leveler effect on the Leds, the node sends this acceleration information through the Nordic Radio module to a border router. It also sends useful information to a serial console that may be connected to the USB port.

The border router, based on an Espressif ESP8266EX SoC, runs a custom software to receive information from the nodes (up to 6 may be configured in a single-hop, star topology) and routes it, using the MQTT protocol, to a specific advanced broker: ThingSpeak.

The ThingSpeak channel shows the acceleration information in a dashboard. Also, it has a specific data field (“DataBack”, channel field 5) that can be updated by a third-party

MQTT or HTTP publisher. Upon publishing a value in this topic/field, the border router, which is subscribed to it, receives the information and piggybacks the data in the RF radio packet reception ACK from the Nexys node, effectively allowing downstream traffic to reach the node and achieving an end-to-end communication.
A FIFO of data is enabled for a store-and-forward approach, ensuring the node gets the information. This technique has been chosen to minimize the radio usage and let the node decide when to initiate the communication, which is the standard procedure in power-saving IoT nodes. The code to update the channel's field 5 using http is presented in Code 11. MQTT updating key and ThingSpeak MQTT broker address is included in Code 10: Full border router code.

```
curl 'https://api.thingspeak.com/update?api_key=2L5ASPHAI8CDMX7H&field5=15'
```

Code 11: HTTP updating code for field 5 "DataBack" using curl. Pushed data is '15' in the example (own elaboration)

Once the Nexys node receives the downstream data, it shows it into the 7-segment display and logs the reception to the console, simulating an action on a potential connected actuator.

ThingSpeak also performs an action on the received data, starting a process that finalizes with an SMS sent to a cell phone if certain conditions are met. In this case, the condition is that the Nexys node lies upside-down. This simulates a cloud-triggered event on the data, or the lack thereof, from a node.
The environment allows for third-party actors to interact with the system as detailed further below. For the thesis dissertation, committee members are invited to participate as third-parties to the simulation, by receiving SMS messages and/or updating the data field that will reach the Nexys node, acting as MQTT or HTTP publishers.

7.4.1 Stage 13: Develop a full Zephyr proof-of-concept application

The Zephyr application was built using code snippets from previous work packages code, cleaning it and expanding the functionality for using several hardware elements of the Nexys 4 DDR. It is presented in Code 12 below.

```
//-----------------------------------
// TFM Full application Concept
// June 2020. Daniel León MIoT/FDI/UCM
// 0.6 SweRVolf version
//-----------------------------------

#include <zephyr.h>
#include <sys/printk.h>
#include <drivers/spi.h>

// Leds and switches using GPIO
#define LEDs_ADDR   0x80001408
#define SWs_ADDR    0x80001400
#define GPIO_INOUT  0x80001410

// Leds and switches using GPIO
#define SegEn_ADDR  0x80001030
#define SegDig_ADDR 0x80001038

// Timer peripheral
```

```
#define RPTC_CNTR   0x80001200
#define RPTC_HRC    0x80001208
#define RPTC_LRC    0x80001210
#define RPTC_CTRL   0x80001218

int8_t dataBackHex = 0; // Global var for holding the cloud incoming data

// SPI LIBRARY - Low Level (DL)
// ------------------------------------------------------

// SPI 1
#define SPI1      0x80001100
#define SPCR1     0x80001100
#define SPSR1     0x80001108
#define SPDR1     0x80001110
#define SPER1     0x80001118
#define SPCS1     0x80001120

// SPI 2
#define SPI2      0x80001800
#define SPCR2     0x80001800
#define SPSR2     0x80001808
#define SPDR2     0x80001810
#define SPER2     0x80001818
#define SPCS2     0x80001820

//SPI REGISTER OFFSETS
#define SPCR      0x00
#define SPSR      0x08
#define SPDR      0x10
#define SPER      0x18
#define SPCS      0x20

void spiInit(uint32_t spiIf) {      // Inits SPI mapped at address spiIf
   sys_write8(0x53, spiIf+SPCR);    // #01010011 no ints, core enabled, reserved, master,
cpol=0, cha=0, clock divisor 10
   sys_write8(0x02, spiIf+SPER);    // int count 00 (7:6), clock divisor 10 (1:0) for 2048
}

void spiCs(uint32_t spiIf, uint8_t upDown) { // Changes the SPI CS line mapped at address spiIf
   sys_write8(upDown, spiIf+SPCS);           // (0x00 is up, 0xFF is down)
}

void spiCsDown(uint32_t spiIf) { // Quick Access to spiCS for pulling the signal LOW
   spiCs(spiIf,0xFF);
}
void spiCsUp(uint32_t spiIf) {   // Quick Access to spiCS for pulling the signal HIGH
   spiCs(spiIf,0x00);
}

uint8_t spiSendGet(uint32_t spiIf, uint8_t dataByte) {   // SPI transaction
uint8_t tmp;
   tmp = sys_read8(spiIf+SPSR);  // Clear SPI interrupt Flag by writing a 1 to bit 7 of the SPI
Status Register SPSR
   tmp |= 0x80;
   sys_write8(tmp,spiIf+SPSR);
   sys_write8(dataByte,spiIf+SPDR); // Write Byte to data register

   do {                          // loop while SPSR.bit7 == 0. (transmission in progress)
     tmp = sys_read8(spiIf+SPSR);
     tmp &=0x80;
   } while (tmp == 0);
   k_sleep(K_USEC(1));           // Allow the SPDR to be available
   return sys_read8(spiIf+SPDR); // read the message from SPI
}
// END OF SPI LIBRARY
// ------------------------------------------------------

// ADXL362 LIBRARY - Low level over SPI (DL)
// ------------------------------------------------------
//adxl362 registers for accelerometer data
#define XAXIS     0x08
#define YAXIS     0x09
#define ZAXIS     0x0A
```

```
#define ACCREAD   0x0B
#define ACCWRITE  0x0A

uint8_t aclGetReg(uint32_t spiIf, uint8_t regAddr) { //Get value of a register in ADXL mapped on
spiIf
 uint8_t value;
   spiCsDown(spiIf);
   spiSendGet(spiIf, ACCREAD);
   spiSendGet(spiIf, regAddr);
   value = spiSendGet(spiIf, 0x00);
   spiCsUp(spiIf);
   return value;
}

void aclSetReg(uint32_t spiIf, uint8_t regAddr, uint8_t data) { //Set value of a register in
ADXL mapped on spiIf
   spiCsDown(spiIf);
   spiSendGet(spiIf,ACCWRITE);
   spiSendGet(spiIf, regAddr);
   spiSendGet(spiIf,data);
   spiCsUp(spiIf);
}

void aclGetAxes(uint32_t spiIf, int8_t *xyz){ // Get three 8-bit values for X, Y and Z
acceleration
   xyz[0] = (int8_t)aclGetReg(spiIf,XAXIS);
   xyz[1] = (int8_t)aclGetReg(spiIf,YAXIS);
   xyz[2] = (int8_t)aclGetReg(spiIf,ZAXIS);
}

void aclInit(uint32_t spiIf) {  // Initialize Accelerator
   aclSetReg(spiIf,0x1F, 0x52); // Reset Accelerometer
   k_sleep(K_MSEC(1));          // Minimum 0,5ms to allow Accelerometer reset
   aclSetReg(spiIf,0x2C, 0x03); // 100Hz sample
   aclSetReg(spiIf,0x2D, 0x02); // Enable measurement mode
}
// END OF ADXL362 LIBRARY
// ------------------------------------------------------

// NRF24L01+ LIBRARY - Low level over SPI (DL)
// ------------------------------------------------------
#define W_TX_PAYLOAD   0xA0
#define R_RX_PLWID     0x60
#define R_RX_PAYLOAD   0x61
#define W_REGISTER     0x20
#define FLUSH_RX       0xE2
#define NRF_NOP        0xFF

// NRF24L01 REGISTER MAP
#define CONFIG         0x00
#define EN_AA          0x01
#define EN_RXADDR      0x02
#define SETUP_AW       0x03
#define SETUP_RETR     0x04
#define RF_CHANNEL     0x05
#define RF_SETUP       0x06
#define STATUSREG      0x07
#define RX_ADDR_P0     0x0A
#define RX_ADDR_P1     0x0B
#define RX_ADDR_P2     0x0C
#define TX_ADDR        0x10
#define RX_PW_P0       0x11
#define RX_PW_P1       0x12
#define RX_PW_P2       0x13
#define DYNPD          0x1C
#define FEATURE        0x1D

#define REGISTER_MASK  0x1F

void nrfCeDown(uint32_t gpioAddr) { // Uses repurposed GPIO for pulling NRF CE signal LOW
   uint16_t gpioStatus;
   gpioStatus=sys_read16(gpioAddr) & 0x7FFF;  // Mask gpioADDR write to avoid affecting other
bits
```

```
    sys_write16(gpioStatus,gpioAddr);
}

void nrfCeUp(uint32_t gpioAddr) { // Uses repurposed GPIO for pulling NRF CE signal HIGH
    uint16_t gpioStatus;
    gpioStatus=sys_read16(gpioAddr) | 0x8000;  // Mask gpioADDR write to avoid affecting other
bits
    sys_write16(gpioStatus,gpioAddr);
}

void nrfSetReg(uint32_t spiIf, uint8_t nrfReg, uint8_t value) { // Sets value on a NRF register
    uint8_t s;
    spiCsDown(spiIf);
    s = W_REGISTER | (REGISTER_MASK & nrfReg);
    spiSendGet(spiIf,s);
    spiSendGet(spiIf,value);
    spiCsUp(spiIf);
}

void nrfSetMultiReg(uint32_t spiIf, uint8_t nrfReg, uint8_t* data, uint8_t size) { //sets multi-
byte register
    uint8_t s;
    spiCsDown(spiIf);
    s = W_REGISTER | (REGISTER_MASK & nrfReg);
    spiSendGet(spiIf,s);
     while (size--) {
         spiSendGet(spiIf,*data++);
     }
    spiCsUp(spiIf);
}

uint8_t nrfRead(uint32_t spiIf) { // Read Data from NRF24L01. If not in operation, it returns
the status register
    uint8_t r;
    spiCsDown(spiIf);
    r = spiSendGet(spiIf,NRF_NOP);
    spiCsUp(spiIf);
    return r;
}

void nrfClearInts(uint32_t spiIf) { // Clear all three interrupt flags at NRF Status register
    uint8_t status;
    status = nrfRead(spiIf);
    status |=0x70;
    nrfSetReg(spiIf,STATUSREG, status);
    spiCsDown(spiIf);
    spiSendGet(spiIf,R_RX_PAYLOAD);
    spiCsUp(spiIf);
}

void nrfFlushRX(uint32_t spiIf) { // Flush reception FIFO
    spiCsDown(spiIf);
    spiSendGet(spiIf,R_RX_PAYLOAD);
    spiCsUp(spiIf);
}

uint8_t nrfRXPayload(uint32_t spiIf) { // Get byte from reception FIFO
    uint8_t r,s;
    r=dataBackHex;
    spiCsDown(spiIf);
    s = nrfRead(spiIf);        // read status
    spiCsUp(spiIf);

    if ((s&0x20) != 0x20) {    // bit 5 of STATUS is asserted when ACK is received in AUTOACK
       printk("ACK NOT Received\n"); // No ACK = No connection to Border Router
       dataBackHex = 0;
    }

    if ((s&0x0E) == 0x00){     // Data ready to be read in PIPE0
       spiCsDown(spiIf);       // Read Data from RX Fifo
       spiSendGet(spiIf,R_RX_PAYLOAD);
       r = nrfRead(spiIf);
       spiCsUp(spiIf);
       printk("RX ACK-Piggibacked Data: %d\n",r);
```

```
      nrfFlushRX(spiIf);      // Flush other potential unnatended receptions
   }
   nrfClearInts(spiIf);       // Clear interrupts, in particular STATUS bit 5
   return r;
}

int8_t nrfTXPayload(uint32_t spiIf, uint32_t gpioAddr, uint8_t* data) // Sends payload
{
   uint8_t r;
   spiCsDown(spiIf);
   spiSendGet(spiIf,W_TX_PAYLOAD);
   spiSendGet(spiIf,data[0]); // X Axis
   spiSendGet(spiIf,data[1]); // Y Axis
   spiSendGet(spiIf,data[2]); // Z Axis
   spiCsUp(spiIf);

   nrfCeUp(gpioAddr);         // Begin transmission by CE pulse
   k_sleep(K_USEC(20));       // 20 uS recommended pulse
   nrfCeDown(gpioAddr);
   k_sleep(K_MSEC(1));        // Allow for Piggybacked data to arrive
   r= nrfRXPayload(spiIf);    // Read downstream data
   return (int8_t)r;
}

void nrfInit(uint32_t spiIf, uint32_t gpioAddr) { // Initialize NRF Radio
   const unsigned char ADDRESS[5] = {0x51, 0xA2, 0x53, 0xA4, 0x28}; // Network
   nrfCeDown(gpioAddr);             // CE LOW & CS HIGH for normal operation
   spiCsUp(spiIf);
   k_sleep(K_MSEC(100));            // boot time allowance
   nrfSetReg(spiIf,SETUP_AW,0x03);  // 5 bytes address
   nrfSetMultiReg(spiIf,RX_ADDR_P0, (uint8_t*)ADDRESS, 5); // Set both reception and
transmission in the same address...
   nrfSetMultiReg(spiIf,TX_ADDR, (uint8_t*)ADDRESS, 5);    // ... Which is needed for auto ACK
   nrfSetReg(spiIf,RX_PW_P0,3);     // Payload of size 3, X, Y and Z axes
   nrfSetReg(spiIf,EN_AA,0xFF);     // Enable Auto ACK on all pipes
   nrfSetReg(spiIf,FEATURE,0x06);   // Enable Dynamic Payload and payloads with ACK
   nrfSetReg(spiIf,DYNPD,0x01);     // Enable Dynamic payload on pipe 0. Needed for payload-ack
   nrfSetReg(spiIf,CONFIG,0x3A);    // Enable Radio in TX mode with 1 byte CRC
}
// END OF NRF24L01+ LIBRARY
// ---------------------------------------------------------

uint16_t ledSet(uint32_t gpioAddr, uint16_t leds) { // Set leds, excluding gpio15 address,
dedicated to NRF CE signal
   uint16_t gpioStatus;
   gpioStatus=sys_read16(gpioAddr) & 0x8000;
   leds&=0x7FFF;
   leds|=gpioStatus;
   sys_write16(leds,gpioAddr);
   return leds;
}

// MAIN PROGRAM
// ---------------------------------------------------------
void main(void) {
#define CONTLOOPS 0        // Skip value for sending radio data. 0 = No skip
  int8_t aclData[3];       // Holder for acceleration data
  uint16_t leds;           // leds shadow register
  uint16_t switches;       // switches shadow register
  int8_t tmpAcc;           // temporal var
  uint32_t dat7s;          // 7-segment 32 bit, 8 number shadow register
  uint32_t cont = 0;       // counter up to  CONTLOOPS
  int8_t dispData = 0;     // Manipulated dataBack for presenting decimal numbers in Hex Display
  struct device *spi;      // SPI Zephyr driver for potential native ADXL362 driver integration
  struct spi_config spi_cfg = {};

  spi = device_get_binding("SPI1"); // Not needed yet, until SwerRVolf issue is responded, but
instantiates the SPI1 hardware from Zephyr
  if (!spi) printk("Could not find SPI driver SPI 1\n");

  printk("-------------------------------------------------------------------------------\n\n");
  printk("SweRVolf Zephyr SoC (MSc. thesis) - Jun. 2020 - Daniel Leon (MIoT/FDI/UCM)\n");
  printk("-------------------------------------------------------------------------------\n");
```

```
  sys_write16(0xFFFF,GPIO_INOUT);   // Enable Leds
  sys_write8(0x24, SegEn_ADDR);     // Enable 7Seg 00_00_00, two displays disabled
  sys_write32(0x00000000, SegDig_ADDR); // Write 0 to all 7S displays

  spiInit(SPI1);                    // Init SPI1
  spiInit(SPI2);                    // Init SPI2
  aclInit(SPI1);                    // Init Accelerometer on SPI1
  nrfInit(SPI2,LEDs_ADDR);          // Init Radio on SPI2 and GPIO LED15 (repurposed)

  while (1) {                       // Main loop

    switches=sys_read16(SWs_ADDR+2);// Get switches status
    if ((switches&0x8000) == 0x8000) {  // Disable actions if Switch 15 is off
      aclGetAxes(SPI1,aclData);     // Get accelerometer data
      if ((switches&0x2000) == 0x2000) if (cont>=CONTLOOPS) printk ("X: %4i - Y: %4i - Z:
%4i\n",aclData[0],aclData[1],aclData[2]); // Log Acc. Data
      if ((switches&0x1000) == 0x1000) if (cont>=CONTLOOPS) printk ("Switches: %4X\n",switches);
// Log Switches status
      if ((switches&0x4000) == 0x4000) cont++;
      leds = ledSet(LEDs_ADDR, 0x0180); // Set the two central leds
      tmpAcc = aclData[1] ;         // Get Y Axis
      if (tmpAcc > 0) {             // Modify to perform a leveler effect with the leds
      if (tmpAcc > 63) tmpAcc = 63; // Adjust for over-acceleration
      leds = leds << ((tmpAcc>>3) & 0x07);
      }
      else {
  if (tmpAcc < -63) tmpAcc = -63;
        leds = leds >> ((-tmpAcc>>3) & 0x07);
      }
      ledSet(LEDs_ADDR, leds);

      if ((cont>=CONTLOOPS) && ((switches&0x4000) == 0x4000)) { // Only send radio data each
CONTLOOPS iterations if switch 14 is on
  dataBackHex= nrfTXPayload(SPI2,LEDs_ADDR,aclData);           // Send Accelerometer information
        if ((dataBackHex < 0) || (dataBackHex > 99)) {         // Check dataBack within range
           printk ("Data out of range (0-99) : %d\n",dataBackHex);
    dataBackHex = 0;
  }
  dispData=dataBackHex+(dataBackHex / 10) *6; // Adjust number for decimal values in Hex display
  cont=0;
      }

      dat7s  = (aclData[0] << 24) & 0xFF000000;                // Compose 7Segment display Data
      dat7s |= (dispData << 12) & 0x000FF000;
      dat7s |= aclData[2] & 0x000000FF;
      sys_write32(dat7s,SegDig_ADDR);                          // Update 7Segment
    }
  }
}
```

Code 12: Full Zephyr OS Application running on the SoC. Project proof of concept (own elaboration)

7.4.2 Stage 14: Store the Zephyr application in the SoC flash

Up until this point of the development, all baremetal and Zephyr applications were loaded to the SoC via USB, using OpenOCD. For the required standalone prototype, the application must be stored in the Nexys on-board SPI Flash memory. SweRVolf provides a core-embedded Bootrom program that, when switches are configured in a specific way, checks for a valid RISC-V program in flash, copies it to RAM and jumps to it.
Figure 61 shows the required switch configuration for the different boot options. Note that, since the original GPIO has been disconnected from the hardware switches, these will have no effect on the boot process. As a remainder, the original GPIO was configured in *axi_multicon.v* to return the expected switches configuration for booting from the SPI Flash.

sw15	sw14	Boot mode
off	off	Boot from SPI Flash
off	on	Boot from address 0 in RAM
on	off	Boot from serial
on	on	Undefined

Figure 61: SweRVolf boot mode [22]

The Zephyr application must be adapted to the Bootrom code requirements. Once this step is done, it must be stored into the SPI Flash. Note that writing to the SPI Flash is a two-step process, where the FPGA must be configured first in order to expose the Flash writing commands. The whole process is accomplished following a series of steps:

- Get into the build Zephyr directory: *zephyr/build/zephyr*
- Execute: *mkimage -A riscv -C none -T standalone -a 0x0 -n ´´-d zephyr.bin zephyr.ub*
- Execute: *objcopy -I binary -O verilog zephyr.ub zephyr.hex*
- Copy the *zephyr.hex* file to the *$WORKSPACE* folder
- Get a bit image for the FPGA Flash writing configuration from https://github.com/quartiq/bscan_spi_bitstreams/blob/master/bscan_spi_xc7a100t.bit and place it in the *$WORKSPACE* folder
- From the *$WORKSPACE* folder, execute: *openocd -c "set BINFILE ./zephyr.hex" ../cores/Cores-SweRVolf/data/swerolf_nexys_write_flash.cfg*

7.4.3 Stage 15: Integrate cloud services

ThingSpeak is an IoT analytics platform own by MathWorks [71] that streamlines data capture and visualization. It includes data collection and visualization as well as cloud-triggered actions. For the PoC scenario, ThingSpeak is used to receive data using MQTT and visualize information, in real time, on a web page. A cloud-triggered action has been configured too: When the Nexys board is upside-down, ThingSpeak will communicate with an SMS provider to send a message to a predefined phone. The connection uses IFTTT [72], a well-established interconnector of applications and devices.

In this stage, ThingSpeak has been configured to include a channel to visualize the acceleration data in an integrated dashboard, as shown in Figure 62. A React App and a ThingsHTTP App is configured to connect to IFTTT and link it to the ClickSend [73] SMS provider.

This chapter documents the configuration, in detail, for all related services, so it can be used as a guide for replicating the environment. It covers:

- ThingSpeak Channel configuration
- ClickSend configuration
- IFTTT Applet configuration using ClickSend API
- ThingSpeak Apps configuration: React, ThingHTTP

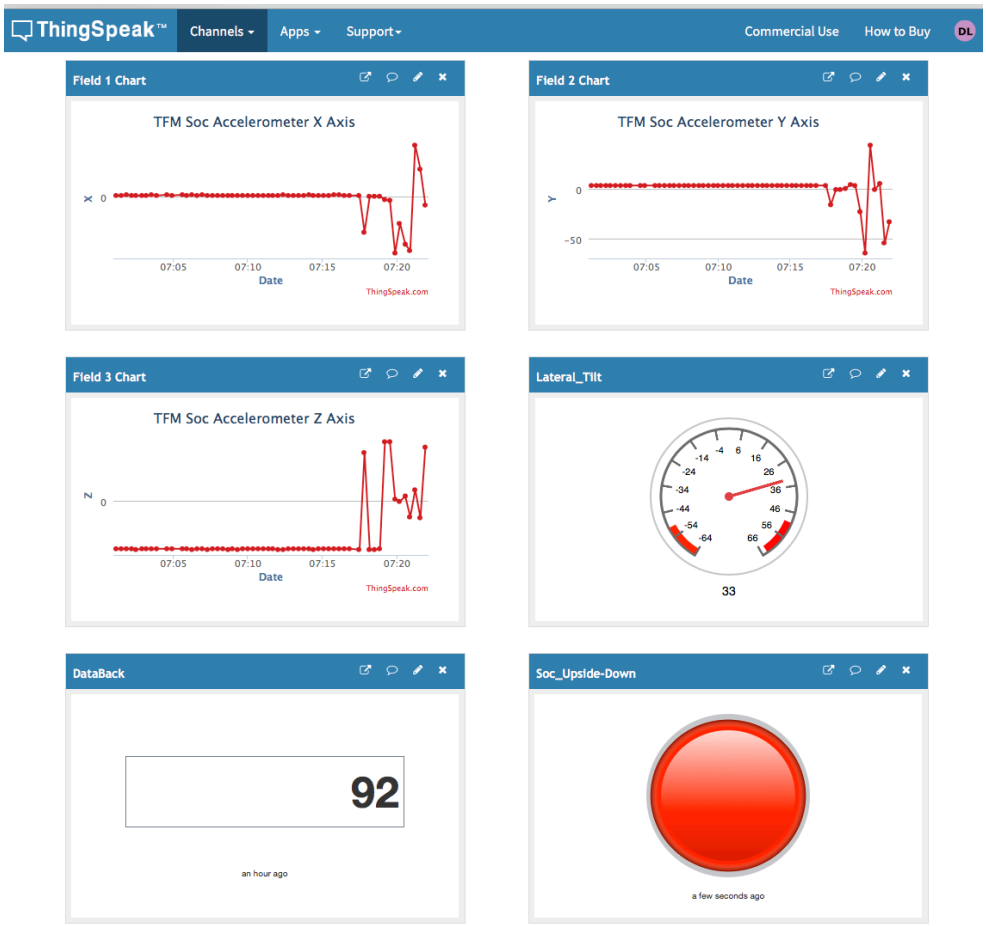


Figure 62: ThingSpeak DashBoard for the demo environment (own capture)

7.4.3.1 ThingSpeak Channel configuration

ThingSpeak offers limited free accounts into their service. The most prominent limitation is that updates may only take place every 15 seconds. This limitation is removed in the paid version, allowing updates each second. There are student packages starting at €66 per year, which also enhance connectivity options and offer more channels. For this project, a student paid subscription is used, however, all steps are documented considering a free account.

After creating a MathWorks account and logging into it, a new channel is created by clicking into the "Channels" menu and pressing the "New Channel" button. The configuration of the fields is shown in Figure 63. The channel is then made public by navigating to the "Sharing tab" and selecting the appropriate option.
Although there is much information that can be entered for a channel, the only important fields are the channel name and the active fields. In this case, 5 fields are created:

- **X:** holding X acceleration information
- **Y:** holding Y acceleration information
- **Z:** holding Z acceleration information
- **Tilt_Y:** Is the negative of Y for gauge visualization
- **DataBack:** Holds the downstream information to the node

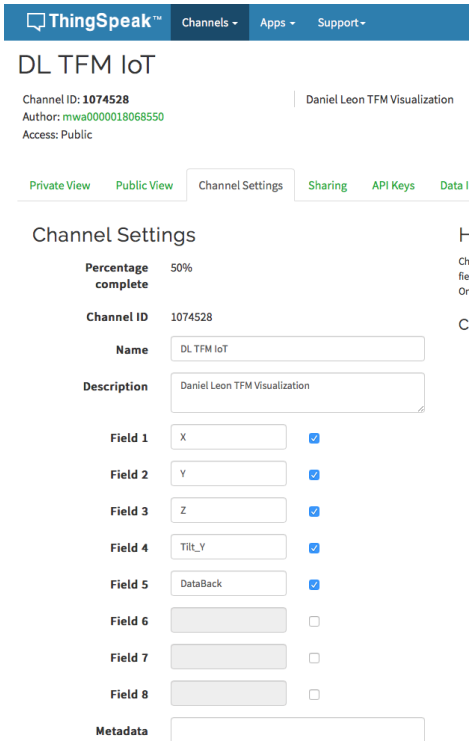


Figure 63: ThingSpeak channel configuration (own capture)

The channel ID uniquely identifies the channel in the ThingSpeak's ecosystem. This specific channel public's view is at https://thingspeak.com/channels/1074528.

Moving into the Public View Tab, 5 charts are presented, one per field. Charts can be slightly customized or removed from the view. As shown in Figure 62 above, the view includes three different graphics added by the "Add Widgets" button (Figure 64).

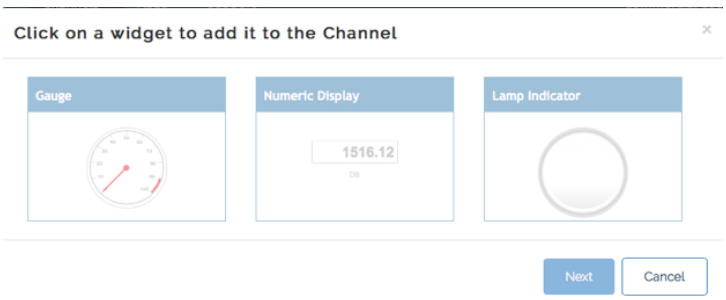


Figure 64: Standard widgets available for ThingSpeak (own capture)

There are some customizable details on each widget, such as thresholds or timeouts for the lamp indicator, decimal places for the numeric display and ranges for the gauge. All dashboards may be configured differently for private and public views

7.4.3.2 ClickSend Configuration

ClickSend is an Australian based SMS service with a cost, in Spain, of 0.044 cents per message. Signing up is a quick process and 1 euro credit is offered free of charge. The interface is intuitive and allows the creation of both email and FAX campaigns as well as SMS, MMS and voice campaigns. Figure 65 depicts the main screen with the account balance, SMS sending statistics and access to the main functions. No further configuration is needed.

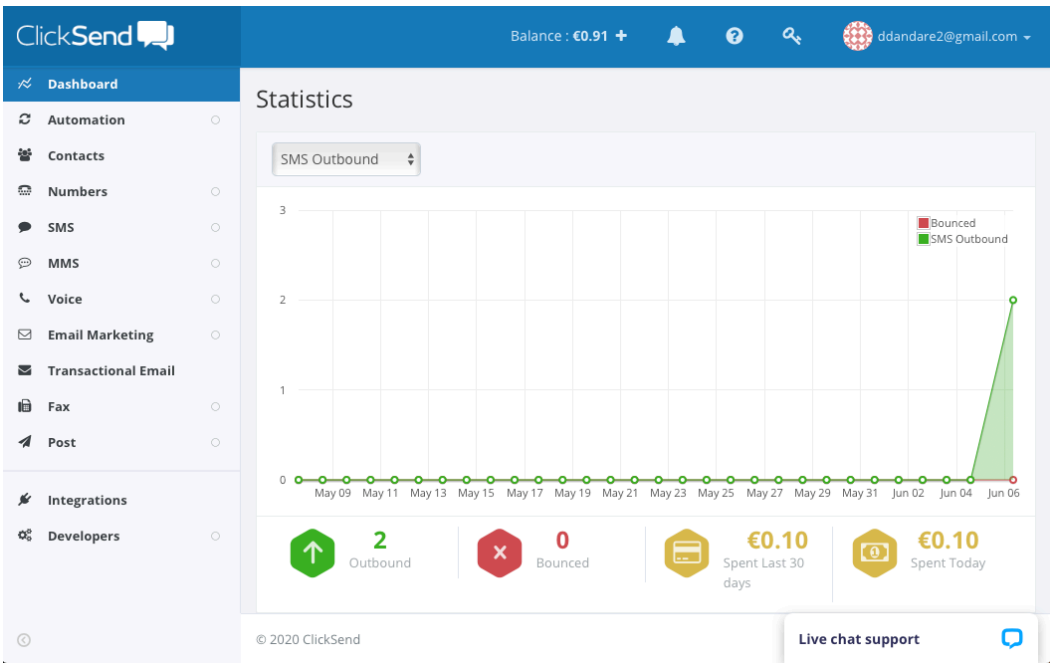


Figure 65: ClickSend control Panel (own capture)

7.4.3.3 IFTTT Applet configuration using the ClickSend API

The actions in IFTTT "If This then That", are called Applets and comprise a pair or services each. Trigger services, which are the ones that launch the action, the "This" part, and action services, which are the ones that are executed when the action has been triggered, the "That" part.

Signing up for IFTTT.COM is a quick process and can be done using a google account. Once inside the service, the "create" menu is used to build the desired applet.
Pressing on the big plus sign on the “This” part leads to a menu of trigger services. Since the goal is to connect to ThingSpeak, the trigger action needs to be “WebHooks”, which can be launched by a ThingsHTTP action from ThingSpeak. At this point the trigger only asks for a name. After selecting the “That” part, a menu of action services is presented.

ClickSend → Send SMS action must be selected. Once connected by entering your ClickSend user and password, it is possible now to fill in a target phone number and the *from* field, that may be a text. This action allows to pass certain values, although this functionality is not required for the project. Figure 66 presents the configuration details

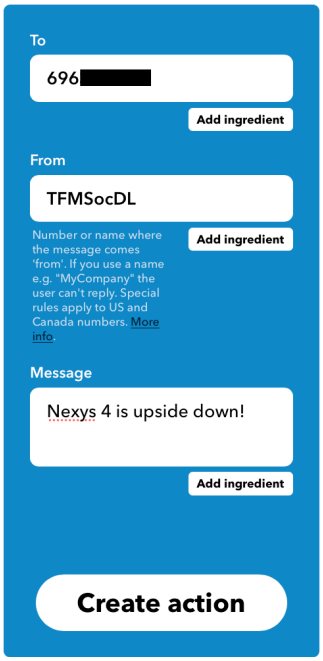


Figure 66: ClickSend Action configuration in IFTTT (own capture)

To launch a WebHook, a key within IFTTT is required. This key is used to build a URL containing the name of the Applet and the key. Navigate to "my services", click on Webhooks and then on settings. The provided URL redirects to a page showing the details on the URL construction.

The created trigger launches on this URL:
https://maker.ifttt.com/trigger/TFM_SoC_UpsideDown/with/key/gTWmYiYPtlsKop

7.4.3.4 ThingSpeak Apps configuration: React and ThingHTTP

Once IFTTT is configured, a ThingHTTP App should be created in ThingSpeak with the URL for the event to be launched. From the Apps menu, choosing ThingHTTP presents a menu where a name and the IFTTT URL must be filled in. As IFTTT supports both POST and GET methods, no other information is required.

The last step is creating a React app. This is done by clicking on the App Menu and selecting React. This specific React monitors the Z acceleration field for values greater than 50, indicating that the Nexys Board is upside down. This option is configured to run only the first time, so the action will only launch on a change from "no condition" to "condition", so the ThingHTTP action will not be executed again until the Nexys 4 DDR board is returned to its natural position and back upside down again. Other options are periodic checks, from 10 minutes to 1 hour or reacting with every data insertion. The full configuration fields are presented in Figure 67.

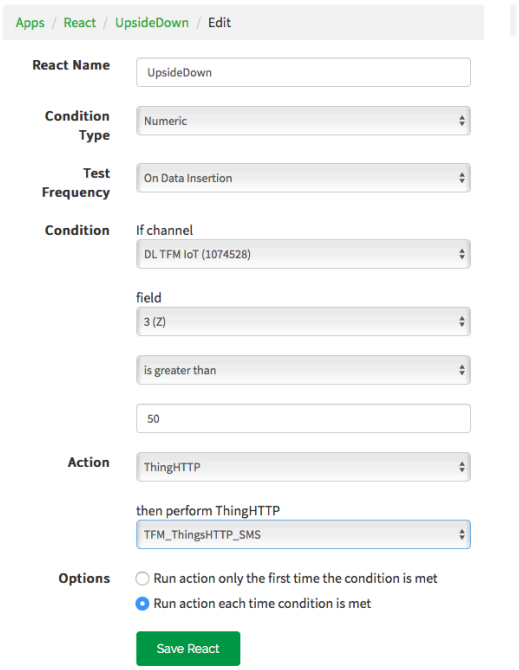


Figure 67: React Configuration at ThingSpeak (own capture)

7.4.4 Stage 16: Demo script for the MSc. thesis defence

This PoC environment will be presented live during the thesis defence. DACYA/ArTeCS member and IoT Master coordinator, Ignacio Gómez has been so kind to participate in the demonstration by remotely interacting with the system and observing acceleration changes, sending downstream data to the node and receiving SMS alerts. A successful test run was performed on June 12th.

A minimalistic slide-based presentation will also support the thesis defence as well as a pre-recorded video with the demo. Presentation and answers to any question may be given in either English or Spanish, based on the committee members preference. Nevertheless all materials and code are prepared in English.

8. Conclusions

Industrial IoT, or IIoT, that comprises all IoT deployments in manufacturing, energy, healthcare and transportation, constitutes a growing market with an impressive economic projection in the upcoming five years.
IIoT devices are massively being deployed in environments where multiple sensor readings and rapid response actions are required. Therefore these devices must rely on very fast processors, while keeping a low energy demand and being as inexpensive as possible, resulting in a minimal TCO[62] that impacts directly on the bottom line of the businesses using them.
In a scenario where Moore's Law [74] appears to be no longer valid [75], specific-purpose computers, usually integrated in SoCs, are replacing general-purpose microcontrollers or computers, offering a customized and, therefore, optimized solution for the task at hand.
A new and adaptable ISA, RISC-V, is claiming its space in this market where, also, several operating systems are competing in a fast-changing market, which is redefining the reach and positioning of embedded, real time and IoT solutions.

Based on this scenario and premises, the goal of the project was to create a prototype node for Industrial IoT and to implement a proof-of-concept deployment of an end-to-end communication between a node and a cloud services provider through a border router.

These are the conclusions arising from this project:

- RISC-V is a very promising ISA for IIoT. Its modular approach enables fine adjustments of the processing capabilities and it also has outstanding support by leading companies, providing a plethora of different implementations, many of them, opensource.
- Western Digital's SweRV EH1 implementation offers a very optimized balance between performance/MHz and power consumption. Adding the support provided by Chips Alliance and its open licensing, SweRV is a family of cores to seriously consider in any IIoT project.
- While using a pre-built minimal SoC, and adapting it for the required needs, streamlines and accelerates the project, the available SoCs are usually not optimal for the project goals, so a perfect balance is difficult to achieve. This compromise is a valid one for prototyping, but a full-fledged, commercial project, should build a customized SoC from the base processor Core.
- Hardware description languages for implementing designs on FPGAs are a massively step forward in hardware design. However, the currently available IP cores quality is very variable. While commercial IP cores are optimized and provide very good documentation, most opensource cores have defective documentation and take shortcuts to "get the job done" without much optimization. This is particularly visible in the many bus adaptations required for

[62] **TCO**: Total Cost of Ownership

their interconnection and in the hundreds or thousands of warnings, or even critical warnings, of these cores during their synthesis process.

- Creating a System-on-chip from opensource components is a very manual job too, requiring multiple tools and steps as well as manual HDL code editing. Commercial IP cores are moving towards a graphical design approach and, most likely, an opensource project will arise in the upcoming years to fulfil this void, very much the same as KiCad EDA [76] did in the physical hardware design arena a few years ago.
- Zephyr is a very capable and adaptable IoT operating system. Although it arrived late to the market, only four years ago, it has evolved very rapidly. This fast pace of updates is, however, something to consider in a project. During the short time of this project, there were five different releases of the Zephyr OS, some of them with critical changes and deprecations, such as the devicetree API change in the 2.3.0 release, requiring a redefinition of all custom drivers. Nevertheless LTS[63] versions exists and should be considered for commercial projects.
- Although there are many radio standards that enable interoperability between different vendors, this project approached communications by using a proprietary radio from Nordic Semiconductor. This decision resulted in incorporating a device that, compared to BLE, requires half the power at twice the communication speed while having a price 4 times lower. Interoperability is then achieved by the inclusion of a custom border router, resulting in a very targeted system. While this may not be the best approach for open IoT deployments where multiple devices from multiple vendors may co-exist, a close and controlled IIoT deployment could benefit from it.
- The ESP8266 platform is a very reasonable option for fast prototyping at a very low cost. Its wide acceptance is well deserved. Development of the border router was quick and painless.
- ThingSpeak offers a reasonable set of features at a fair price. It even provides limited free-of-charge packages. IoT analytics and cloud-triggered actions can be rapidly deployed and, even though this project does not use this capability, the option of executing Matlab code on the cloud makes it a very powerful platform.

I believe the goals of the Project have been fulfilled: a valid IIoT node has been designed and implemented and it has been integrated into a PoC environment simulating a potential deployment scenario. The full process has been documented in detail and future lines of work identified.

On a personal note, working in this project has been challenging, given the breadth of the scope, but very satisfying and educative. Although the project has taken considerably more than the 270 projected hours, I would have liked for this thesis to have had twice the time assigned to it, up to 24 ECTS[64], so I could have gone deeper into several aspects, most of them outlined in the future work chapter.

[63] **LTS**: Long Time Support
[64] **ECTS**: European Credit Transfer and accumulation System

9. Future work

The work done in this project opens many future work opportunities, by expanding the scope, tightening the integration of the different elements or moving into different wireless protocols such as 802.15.4 or Bluetooth Low Energy. A list of the suggested lines of future work is outlined, alongside a small description of their projected scope.

- **Tighter Zephyr integration**: In this project, hardware is directly accessed, at low level, from Zephyr OS. Using the devicetree API and the KConfig environment, these hardware elements could be integrated into the Zephyr's HAL and the code changed to be detached from hardware implementations details. These modifications would cover the GPIO ports, the 7-segment display, the SPI peripherals, the ADXL362 accelerometer and, of course, the NRF24L01+ radio.
- **Ethernet support**: Using EthMac and MII to RMII IP cores from Xilinx, expand the Extended SoC to support Ethernet at the hardware level. Then, connect the newly created hardware to Zephyr and enable the network stack. Initial information for this future project is presented in chapters 5.3 and 5.6 of this document.
- **Node power management**: Current implementation of the node does not use power saving features available in the connected hardware elements, such as the micro-watt standby mode for the radio or the disconnection of the accelerometer. The code could be enhanced to take power saving modes into consideration. A more ambitious approach will also include RISC-V CPU speed throttling by enabling sleep modes and a custom SoC implementation over the SweRV EH1 RISC-V Core.
- **Alternative Radio**: The current NRF24L01+ Radio uses a proprietary "Enhanced Shockburst" air protocol. While this is a fast, fairly-ranged, capable and inexpensive device, future work could replace this radio with a 802.15.4 or a BLE transceiver, such as the ST Microelectronics BlueNRG-2 [77], to extend the interoperability with third-party nodes.
- **Border router network enhancements**: This project is configured for one connected node to the border router, but it lies the foundation for a multi-node local network. For future work, the border router code could be enhanced to enable a 6-node star topology and allow peer-to-peer communication between nodes. A set of store-and-forward FIFO buffers should be added to ensure the delivery of the messages even when end nodes are disconnected for a long time.
- **Edge computing**: This project implements node-driven actions and cloud-driven actions. In a multi-node deployment, specific decisions could be offloaded to the border router, that contains information from all connected nodes, thus minimizing the node's power consumption and the overall network traffic.
- **MQTT Broker enhancements**: The MQTT broker used in the project (ThingSpeak) imposes a 1-second update rate limitation in its used paid version (free version is 15 seconds). This causes that some messages may be ignored if not meeting the constraints. In a worst case scenario, the MQTT connection is dropped altogether. Adding a full-fledged MQTT broker and using broker bridging with ThingSpeak may address these issues, while keeping all the benefits of the ThingSpeak platform.

10. Index of FIGURES

11. Index of TABLES

12. Index of CODE

13. References

[1] "IoT Market report," [Online]. Available: https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307.

[2] I. M. -. Fortunly, "Laptop," 5 November 2019. [Online]. Available: https://fortunly.com/blog/lap-top-market-share/.

[3] Newzoo via VentureBeat, "2018-2022 Global Games Market," 18 June 2019. [Online]. Available: https://venturebeat.com/2019/06/18/newzoo-u-s-will-overtake-china-as-no-1-gaming-market-in-2019/.

[4] F. M. -. JATO, "Carmaker's revenue remained stable 2019," 23 May 2019. [Online]. Available: https://www.jato.com/carmakers-revenue-remained-stable-but-they-earned-less-money-in-q1-2019/.

[5] Orange, "Internet of Things - 29 buenas prácticas en grandes empresas nacionales e internacionales," May 2019. [Online]. Available: https://www.orange.es/static/pdf/InternetOfThingsGrandesEmpresas.pdf.

[6] Industrial Internet Consortium, "https://www.iiconsortium.org," [Online].

[7] "RISC-V International," [Online]. Available: https://riscv.org.

[8] "Zephyr Project," [Online]. Available: https://www.zephyrproject.org.

[9] "Linux Foundation Home page," [Online]. Available: https://www.linuxfoundation.org.

[10] "Artix-7 Cost and Transceiver Optimized FPGAs," [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

[11] D. Patterson and A. Waterman, The RISC-V Reader, Strawberry Canyon, 2017.

[12] Z. Z. Bandic, R. Golla, J. Rahmeh, O. Shinaar, M. Smittle and P. Pattel, "High Performance RISC-V enbedded SweRV EH1 core: microarchitecture, performance and SSD controller implementations," 20 May 2020. [Online]. Available: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/SweRV%20core%20roadmap%20white%20paper.pdf.

[13] OpenHW Group, "OpenHw Group RI5CY Github," [Online]. Available: https://github.com/openhwgroup/cv32e40p.

[14] OpenHW Group, "RI5CY User Manual," December 2019. [Online]. Available: https://github.com/openhwgroup/core-v-docs/blob/master/cores/cv32e40p/CV32E40P_and%20CV32E40_Features_Parameters.pdf.

[15] Western Digital Incorporated, "WD's Innovations: RISC-V," [Online]. Available: https://www.westerndigital.com/company/innovations/risc-v.

[16] Chips Alliance, "Chips Alliance home page," [Online]. Available: https://chipsalliance.org.

[17] T. Marena, "RISC-V: high performance embedded SweRV core microarchitecture, performance and CHIPS Alliance," 2 April 2019. [Online]. Available: https://content.riscv.org/wp-content/uploads/2019/04/RISC-V_SweRV_Roadshow-.pdf.

[18] Chips Alliance, "EH1 SweRV RISC-C Core 1.6 from Western Digital Github repository," [Online]. Available: https://github.com/chipsalliance/Cores-SweRV.

[19] W. Digital, "RISC-V SweRV EH1 Programmer's Reference Manual revision 1.6," 15 May 2020. [Online]. Available: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf.

[20] T. Newsome, M. Wachs and SiFive, "RISC-V External Debug Support. Version 0.13.2," 22 March 2019. [Online]. Available: https://riscv.org/specifications/debug-specification/.

[21] "Pulp Platform," [Online]. Available: https://pulp-platform.org.

[22] O. K. -. C. Alliance, "SweRVolf Github repository," [Online]. Available: https://github.com/chipsalliance/Cores-SweRVolf.

[23] "Qamcom homepage," [Online]. Available: https://www.qamcom.com.

[24] "FOSSi Foundation," [Online]. Available: https://fossi-foundation.org/organization.

[25] O. Kindgren, "Fusesoc github page," [Online]. Available: https://github.com/olofk/fusesoc.

[26] "OpenRISC Project homepage," [Online]. Available: https://openrisc.io.

[27] OpenCores, "WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores," 7 September 2002. [Online]. Available: https://opencores.org/cdn/downloads/wbspec_b3.pdf .

[28] EnjoyDigital, "LiteDRAM Github page," [Online]. Available: https://github.com/enjoy-digital/litedram.

[29] Digilent, "Nexys 4 DDR Reference Manual," [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual.

[30] Xilinx, "Vivado Design Tools," [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[31] Analog Devices, "ADXL362 Micropower, 3-Axis Digital Output MEMS Accelerometer," 2019. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf..

[32] Microchip, "LAN8720A / LAN8720I Small Footprint RMII 10/100 Ethernet Transceiver with HP Auto-MDIX Support," 7 July 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/00002165b.pdf.

[33] "Nordic Semiconductor homepage," [Online]. Available: https://www.nordicsemi.com.

[34] "RF24Ethernet - TCP/IP over RF24Network 1.6b," [Online].

[35] Nordic Semiconductor, "nRF24L01+ Single Chip 2.4GHz transceiver Product Specification v1.0," September 2008. [Online]. Available: https://www.nordicsemi.com/-/media/DocLib/Other/Product_Spec/nRF24L01PPSv10.pdf.

[36] DevicePlus; Iyer, Rahul, "nRF24L01+ RF Module Tutorial," 13 April 2017. [Online]. Available: https://www.deviceplus.com/arduino/nrf24l01-rf-module-tutorial/.

[37] "OpenCores homepage," [Online]. Available: https://opencores.org.

[38] Opencores and R. Herveille, "SPI Core Specifications 0.1," 7 January 2003. [Online]. Available: https://opencores.org/websvn/filedetails?repname=simple_spi&path=%2Fsimple_spi%2Ftrunk%2Fdoc%2Fsimple_spi.pdf.

[39] D. León and R. Gil, "Design of a test platform for studying radiation effects on SPI FRAM and I2C CMOS non-volatile memories," June 2019. [Online]. Available: https://eprints.ucm.es/58860/1/1137343455-453707_DANIEL_LEÓN_GONZÁLEZ_Memoria_TFG._DESIGN_OF_A_TEST_PLATFORM_FOR_STUDYING_RADIATION_EFFECTS_ON_SPI_FRAM_AND_I2C_CMOS_NON-VOLAT_1158801573.pdf.

[40] UCM, "Normativa para trabajos de fin de máster," 2020. [Online]. Available: https://informatica.ucm.es/normativa-trabajos-de-fin-de-master.

[41] O. Kindgren, "Old SweRVolf core with Ethernet github," [Online]. Available: https://github.com/chipsalliance/Cores-SweRVolf/tree/eth.

[42] O. Kindgren, "Obsolete Zephyr OS branch with SweRVolf Ethernet Support," [Online]. Available: https://github.com/olofk/zephyr/tree/ethoc_irq.

[43] Xilinx, "Tri-Mode Ethernet MAC v9.0 - LogiCORE IP Product Guide Vivado Design Suite," 4 April 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v9_0/pg051-tri-mode-eth-mac.pdf.

[44] Xilinx, "AR# 71457 Ethernet MII_to_RMII core deprecated," [Online]. Available: https://www.xilinx.com/support/answers/71457.html.

[45] Xilinx, "MII to RMII v2.0 LogiCORE IP Product Guide - Vivado Design Suite," 5 December 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mii_to_rmii/v2_0/pg146-mii-to-rmii.pdf.

[46] "Espressif homepage," [Online]. Available: https://www.espressif.com.

[47] "Cadence homepage," [Online]. Available: https://www.cadence.com/en_US/home.html.

[48] "Shop of Things Internet Store," [Online]. Available: https://shopofthings.ch.

[49] Espressif, "ESP8266EX Datasheet V6.4," December 2015. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf.

[50] "Nanjimg Qinheng Microelectronics homepage," [Online]. Available: http://www.wch-ic.com.

[51] "Future Technology Devices International Ltd. homepage," [Online]. Available: https://www.ftdichip.com.

[52] I. Andrew and W. Community, "ESP8266 vs ESP32: What's the difference?," October 2018. [Online]. Available: https://community.wia.io/d/53-esp8266-vs-esp32-what-s-the-difference.

[53] "QEMU, The FAST! processor emulator homepage," [Online]. Available: https://www.qemu.org.

[54] "Zephyr hits 40.000 commits on Github," 16 May 2020. [Online]. Available: https://www.zephyrproject.org/zephyr-project-marks-remarkable-milestone-proving-were-on-to-something-big/.

[55] "Zephyr OS release plan," [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/projects/9.

[56] "Nordic Semiconductor nRF Connect SDK (nrfxlib) for Zephyr OS," [Online]. Available: http://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/zephyr.html.

[57] "Zephyr's metatool WEST," [Online]. Available: https://github.com/zephyrproject-rtos/west.

[58] R. (. Qian and K. (. L. F. Stewart, "Zephyr Project Overview," 2018. [Online]. Available: https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Zephyr-Overview-Presentation.pdf.

[59] ASPENCORE, "2019 Embedded Markets Study," March 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[60] "ThinkSpeak homepage," [Online]. Available: https://thingspeak.com.

[61] Oracle, "Virtualbox homepage," [Online]. Available: https://www.virtualbox.org.

[62] "Digilent's guide for Vivado installation and Digilent boards," [Online]. Available: https://reference.digilentinc.com/vivado/installing-vivado/start.

[63] "OpenOCD homepage," [Online]. Available: http://openocd.org .

[64] "Getting Started with Zephyr - Installation guide," [Online]. Available: https://docs.zephyrproject.org/latest/getting_started/index.html.

[65] "comp.sys.sinclair FAQ - ZX Spectrum Hardware ports," [Online]. Available: https://worldofspectrum.org/faq/reference/ports.htm.

[66] UCM ArTeCS Group, "ArTeCS Group Website," [Online]. Available: https://artecs.dacya.ucm.es.

[67] N. 4. D. c. file. [Online]. Available: https://github.com/Digilent/digilent-xdc/blob/master/Nexys-4-DDR-Master.xdc.

[68] "Saleae Logic Analyzer homepage," [Online]. Available: https://www.saleae.com.

[69] "ADXL362 Zephyr Issue on SweRVolf Github page," 29 May 2020. [Online]. Available: https://github.com/chipsalliance/Cores-SweRVolf/issues/14.

[70] "ESP8266 power compsuption," 12 January 2015. [Online]. Available: https://bbs.espressif.com/viewtopic.php?t=133.

[71] "MathWorks homepage," [Online]. Available: https://www.mathworks.com.

[72] "IFTTT "If This Then That" homepage," [Online]. Available: https://ifttt.com.

[73] "ClickSend SMS homepage," [Online]. Available: https://www.clicksend.com/es/.

[74] Encyclopaedia Britannica, "Moore's Law," [Online]. Available: https://www.britannica.com/technology/Moores-law.

[75] CNET, "CES 2019: Moore's Law is dead, says Nvidia's CEO," 9 January 2019. [Online]. Available: https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/.

[76] "KICad EDA homepage," [Online]. Available: https://kicad-pcb.org.

[77] "BlueNRG-2 product page at ST.com," [Online]. Available: https://www.st.com/en/wireless-transceivers-mcus-and-modules/bluenrg-2.html.

[78] Analog Devices; Linear Technology, "LCT2376-20 20-bit, 250ksps, Low Power SAR ADC with 0.5ppm INL datasheet," 2013. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/237620fb.pdf.

[79] "Complutense University of Madrid homepage," [Online]. Available: https://www.ucm.es.

[80] "Imagination Technologies," [Online]. Available: https://www.imgtec.com.

[81] UCM, "Artículo 83 L.O.U. (contratos con empresas)," [Online]. Available: https://www.ucm.es/otri/financiacion-articulo-83-lou.

14. Glossary

Alphabetically ordered list of all footnote terms.

ACK: Acknowledgement communication packet
ASIC: Application-Specific Integrated Circuit
BLE: Bluetooh Low Energy
CAGR: Compound Annual Growth Rate
CAPEX: Capital Expense
CMT: Clock Management Tile
CoAP: Constrained Application Protocol
CPU: Central Processing Unit
CRC: Cyclic Redundancy Check. An error-detecting code
DIY: Do It Yourself
DMA: Direct Memory Access
ECE: Electrical and Computer Engineering
ECTS: European Credit Transfer and accumulation System
FAT32: File Allocation Table 32 bit FileSystem
FDI: Facultad De Informática
FIFO: First In First Out. List queue operation format
FPGA: Field Programmable Gate Array
GMII: Gibabit Media-Independent Interface
GND: Ground. Ground reference of a circuit
GPIO: General Purpose Input Output
GSK: Gaussian Frequency-Shift keying
HAL: Hardware Abstraction Layer
HDL: Hardware Description Language
HVL: Hardware Verification Language
I2C: Inter Integrated Circuit
I2S: Integrated InterChip Sound
IIoT: Industrial Internet of Things
IoT: Internet of Things
ISA: Instruction Set Architecture
ISM: Industrial, Scientific and Medical frequency allocation band.
ISR: Interrupt Service Routine
JTAG: Join Test Action Group
LTS: Long Time Support
LUT: Look-Up Table
LWM2M: Lightweight Machine to machine
MAC: Medium Access Control Layer.
MCU: MicroController Unit
MDC: Management Device Clock
MDI: Medium Dependent Interface.
MDIO: Management Device Input Output
MDIX: MDI with crossed transmission and reception wiring
MEMS: Micro Electrical Mechanical System
MII: Media Independent Interface

MQTT: MQ Telemetry Transport
PHY: Physical layer (layer 1) of the OSI network model
PoC: Proof-of-Concept
PWM: Pulse Width Modulation
RGMII: Reduced Gigabit Management
RMII: Reduced Media-Independent Interface
RTOS: Real Time Operating System
SMI: Serial Management Interface
TCO: Total Cost of Ownership