Universidad de Alcalá
Escuela Politécnica Superior

Degree in Computer Engineering

Final Degree Project

Prototype on FreeRTOS and RISC-V of the application software of the EPD ICU onboard Solar Orbiter

Author: Manuel Urbano Rodríguez Schere

Tutor: Oscar Rodríguez Polo

2021

Court:

President: Pablo Parra Espada

1st vocal: Juan Ignacio García Tejedor

2nd vocal: Oscar Rodríguez Polo

Alternate: Antonio Da Silva Fariña

Table of Contents

List of figures

List of tables

Preface

My final degree project has been a test of everything I have been working on during my university degree. It has both made me go back to the basics of computer engineering and also challenged me in many new ways I haven't thought of before. It made me face a real-life challenge such as the porting of a software which is similar to what you would find in a job related to the field.

As a computer and high-tech freak, I have a great interest in world-changing technologies. Space exploration is one of this fields that is in the mouth of everyone, especially now during the advent of private space-related companies such as SpaceX and Blue Origin. NASA and the ESA also continue achieving great milestones in the field and Solar Orbiter is one of their jaw-dropping projects. I feel very fortunate on having the opportunity of working on something related to this field and I aim to learn much more about it.

Since the beginning of my degree my interest in different information technologies fields have been in a constant shift as new and exciting projects are born every day. My original intent was to focus my career on embedded systems related to space and aerospace technologies but the recent advancements in blockchain and distributed ledger technologies made me change my personal taste towards that direction. Nevertheless, these technologies unlock so many use cases and I'm sure space exploration will benefit greatly from making use of this new and exciting tech.

Acknowledgments

I want to thank my tutor Oscar Rodríguez Polo for putting through my countless emails and going out of the way to answer them and give constant feedback even during his summer vacation. His expertise was crucial for the development of this project, and I wouldn't be able to get these results without him. I also want to thank most of my teachers who gave me superb formation during my university degree and many of them shaped my interests and future professional choices.

Key words
Solar Orbiter, FreeRTOS, embedded systems, real time systems, port, EPD ICU.

Palabras clave
Solar Orbiter, FreeRTOS, sistemas empotrados, sistemas en tiempo real, puerto, EPD ICU.

Summary (English)
Solar Orbiter is an ambitious project developed by the European Space Agency which aims to investigate our sun more closely. Its Energetic Particle Detector ICU runs its application software with RTEMS, which is proven to be reliable. However, we aim to potentially improve it by building a port in FreeRTOS, which is a more popular and scalable RTOS and testing this software in a modern hardware architecture called RISC-V which is lightweight and fast. To do this, we will build a prototype and test it in the RV32M1-VEGA board developed by OpenISA.

Resumen (Español)
Solar Orbiter es un ambicioso proyecto desarrollado por la Agencia Espacial Europea que tiene como objetivo investigar más cercanamente nuestro sol. La ICU de su instrumento EPD corre su software de aplicación en RTEMS, el cual se ha demostrado que es confiable. Sin embargo, podemos tratar de mejorarlo desarrollando un puerto en FreeRTOS, el cual es más popular y puede proporcionar más escalabilidad. Este software correrá en una novedosa arquitectura llamada RISC-V. Para lograr esto, construiremos un prototipo y lo probaremos en la placa RV32M1-VEGA desarrollada por OpenISA.

Extended summary
Solar orbiter is a project developed by the European Space Agency (ESA) in conjunction with the National Space Agency (NASA) that aims to investigate the surface of the sun and gather information about its polar regions for the first time. Solar Orbiter contains a sophisticated instrument called the EPD (Energetic Particle Detector). This instrument is equipped with five sensors that can measure the various Solar Energetic Particles (SEP) and can operate in ranges from 2 keV to 200 MeV. These sensors are all connected to a Common Data Processing Unit (CDPU). SEP events are of great interest because of their effects on Earth which can vary from Solar Cell Damage to satellites on Earth's orbit to airline passenger radiation poisoning.

The current software of the EPD runs on the Real-Time Executive for Multiprocessor Systems (RTEMS) Real Time Operating System. It became available in 1993 and supports open standard Application Program Interfaces (APIs). It is used extensively in space flight, medical, networking and many other embedded systems. RTEMS is very versatile, supporting 18 processor architectures and around 200 Board Support Packages (BSPs).

The CDPU currently has a LEON2 based processor. It is a synthesizable VHDL model compliant with the SPARC V8 architecture and can be used both for research and commercial applications. SPARC stands for Scalable Processor Architecture and it's a big-endian RISC architecture designed by Sun Microsystems in 1985. It was the first open RISC architecture and as so, its design specifications are public. The LEON family of

processors are designed to be radiation-tolerant and are developed by the European Space Research and Technology Centre (ESTEC), part of the ESA.

The new architecture we will be using is RISC-V, an open-source architecture with a reduced instruction set which makes it extremely fast and resource-efficient. Its popularity has skyrocketed over the last few years and it is being adopted by the main chip manufacturers.

We aim to port a reduced educational version of this software to the Free Real Time Operating System (FreeRTOS) which came out in 2003. It was developed along a lot of the main chip manufacturers and is distributed freely under the MIT open-source license. It supports many architectures and it’s built to be reliable, feature rich, scalable and simple. The core files comply with the MISRA coding standard guidelines. It builds with many different compilers and provides ample support for testing.

The software will be tested on the OpenISA VEGAboard. The RV32M1-VEGA development board is a small, low-power and cost-effective evaluation and development board used for application prototyping. It offers an easy-to-use flash programmer, virtual serial port and standard programming and run-control capabilities. It integrates Bluetooth Low Energy (BLE), a radio transceiver, 2 ARM CPUs and 2 RISC-V CPUs. It also has various sensors including a 3D accelerometer and magnetometer, 4 integrated push-button switches and various status LEDs.

Chapter 1 - Introduction

1.1 Objectives

Our main objective, as stated above, is to port an educational version of the EPD’s ICU-ASW from one RTOS and architecture to another. This is motivated by the current surge in popularity of FreeRTOS and the increased scalability and support from chip manufacturers that it offers. This will modernize the project and open it to future upgrades. To achieve this, we will develop this port with a reduced and simplified version of the software, and test it on the RV32M1-VEGA, which will offer us a clear vision of how the system behaves and compare it to the original. We will need to follow a methodical approach by applying the core concepts of computer engineering and a rigorous testing process. To accomplish this, we must define different testing scenarios to evaluate how the system operates in different conditions and levels of integration. Performance must also be tested so that execution times of the different processes are according to the specifications.

1.1.1 Toolchain and development environment setup

The first step in the development process is setting up the toolchain to have the cross-compiler working correctly. Eclipse was chosen as the IDE as it easily integrates with the toolchain and has official support by OpenISA. To complete this setup, we will follow the official guide provided by OpenISA. Once configured, we will run some example demo applications to check that they build and run correctly. We can check the program output via the board’s UART via programs like Minicom or PuTTY.

1.1.2 Requirements analysis

ICU ASW has a series of components which create different tasks. These tasks should only take resources in the order that their priority allows them to and should complete their execution within their specified amount of time and use as little memory and hardware resources as possible without sacrificing speed or functionality.

1.1.3 Implementation and validation of the port

Once the port is developed it should be tested thoroughly and validated via extensive functional testing. Testing is an essential part of the development and validation of any space-related system, as any minor error in the software or hardware could cause catastrophic losses.

1.1.4 Evaluation and results analysis

The final version of the port should be evaluated to check if it fulfilled the objectives set in terms of handling, resource consumption and ease of use. By doing this, conclusions can be drawn to assess the final specifications of the port. The time requirements must be respected in the final version.

Chapter 2 - Theoretical description

In this theoretical description the main concepts and components of the project will be explained from a high-level view so that a clear picture is offered about the technologies involved in the project.

2.1 Solar Orbiter

Fig. 2.1: Artistic representation of the Solar Orbiter mission [3]

Solar Orbiter has been specifically designed to make measurements of on-site of plasma properties, electromagnetic fields and energetic particles of the inner heliosphere. It differs from other similar projects in that it has instruments that will allow it to directly correlate in-situ measurements and high-resolution imaging and spectrographic observations of the sun from a close view. It will also provide images of the Solar Poles for the first time in history. Solar orbiter is composed of the following instruments:

-Magnetometer (MAG)
-Radio and Plasma Wind Analyzer (SWA)
-Energetic Particle Detector (EPD)
-X-ray Spectrometer (STIX)
-Polarimetric and Helioseismic Imager (PHI)
-Extreme Ultraviolet Imager (EUI)
-Spectral Imaging of the Coronal Environment instrument (SPICE)
-Visible light and ultraviolet coronal imager (METIS)
-Heliospheric Imager (SoloHI)
-Radio and Plasma Waves (RPW)

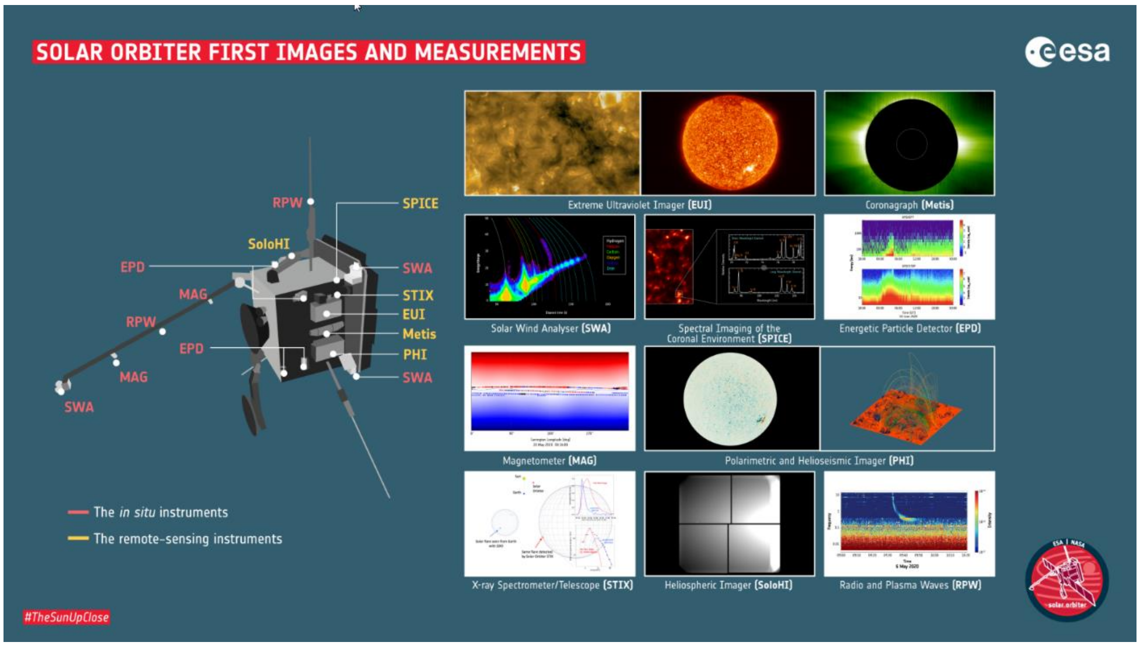

SOLAR ORBITER FIRST IMAGES AND MEASUREMENTS
esa
RPW
SPICE
SoloHI
EPD
SWA
MAG
STIX
EUI
RPW
Metis
EPD
PHI
MAG
SWA
SWA
The in situ instruments
The remote-sensing instruments
Extreme Ultraviolet Imager (EUI)
Coronagraph (Metis)
Solar Wind Analyser (SWA)
Spectral Imaging of the Coronal Environment (SPICE)
Energetic Particle Detector (EPD)
Magnetometer (MAG)
Polarimetric and Helioseismic Imager (PHI)
X-ray Spectrometer/Telescope (STIX)
Heliospheric Imager (SoloHI)
Radio and Plasma Waves (RPW)
#TheSunUpClose

Fig. 2.2: Solar Orbiter's instruments suite and measurements demonstration [3]

To achieve the mission objectives, these instruments need to coordinate at extreme conditions in order to obtain good quality results. Various groups back on Earth are tasked with recompiling and analyzing the data gathered so conclusions can be drawn. The EPD is one of the most important instruments, as it helps unveil the mysteries of solar energetic particles, one of the main drivers behind solar activity.

2.2 Solar Orbiter's EPD

The sun is the most powerful source of energetic particles in the Solar System. Solar phenomena are capable of energizing electrons and ions that escape into space along with magnetic field lines. The main sources of SEPs are solar flares and coronal mass ejections (CMEs). These events are of great relevance for space weather because of the radiation emissions they produce, which affect both crews in space, space electronics and even humans on Earth.

SEPs are usually classified between impulsive and gradual, depending on their acceleration mechanism. Gradual events are proton-rich and last longer. They are usually produced in fast and large CMEs and can be accompanied by type II radio bursts. Impulsive events are shorter, electron-rich and show higher charge states. They are usually seen along type III radio bursts and can be produced in solar flares, narrow CMEs and extreme-ultraviolet jets. Some SEP events have a hybrid composition in which both CMEs and solar flares are involved.

The EPD will try to answer a central question:

How do solar eruptions produce energetic particle radiation that fills the heliosphere?

This main question can be broken into three sub-questions:

1. Injection: *How and where are energetic particles injected at the sources and, in particular, what are the seed populations for energetic particles?*

2. Acceleration: *How and where are energetic particles accelerated at the Sun and in the interplanetary medium?*

3. Transport: *How are energetic particles released from their sources and distributed in space and time?*

EPD is equiped with a suite of sensors aimed to provide information about these particles and events. Its main components are:

- SupraThermal Electrons and Protons (STEP): Designed to measure protons and electrons at supra-thermal energies (low energy particles).

- Electron Proton Telescope (EPT): Two double-ended telescopes that separate and measure electrons and protons in a medium energy range.

- Suprathermal Ion Spectrograph (SIS): Provides observations of He and Fe in energy ranges just above solar wind.

- High Energy Telescope (HET): Measures electrons, protons and heavy ions in the high-energy ranges of the EPD.

The Instrument Control Unit (ICU) works as the interface between the spacecraft and the EPD sensors.

The EPT and HET units are bundled together and separated into 2 almost identical units EPT-HET1 and EPT-HET2. All of these units have evolved from previous implementations in similar missions and their designs have been updated to withstand the extreme conditions of the Inner Solar Orbit. Each EPD sensor unit has a heater to keep it in its functional temperature range.

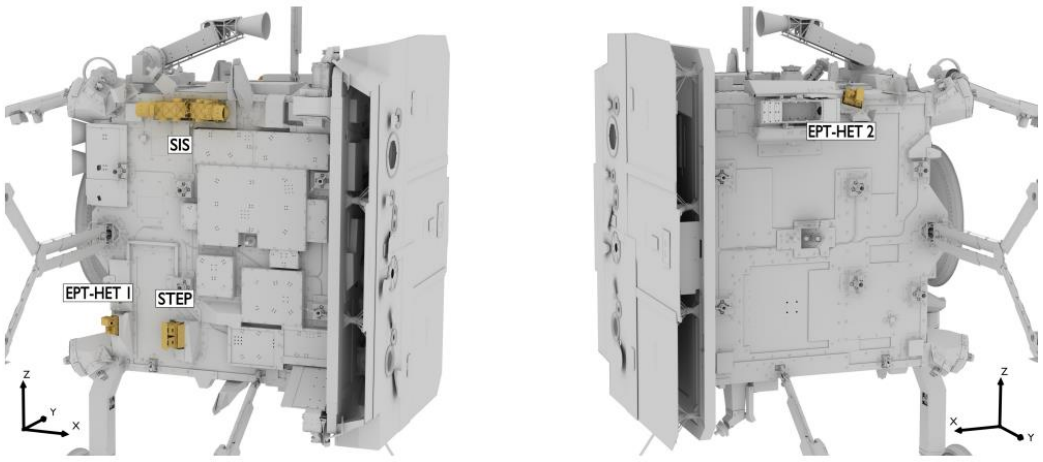


Fig. 2.3: Representation of the physical location of the EPD sensors onboard Solar Orbiter [4]

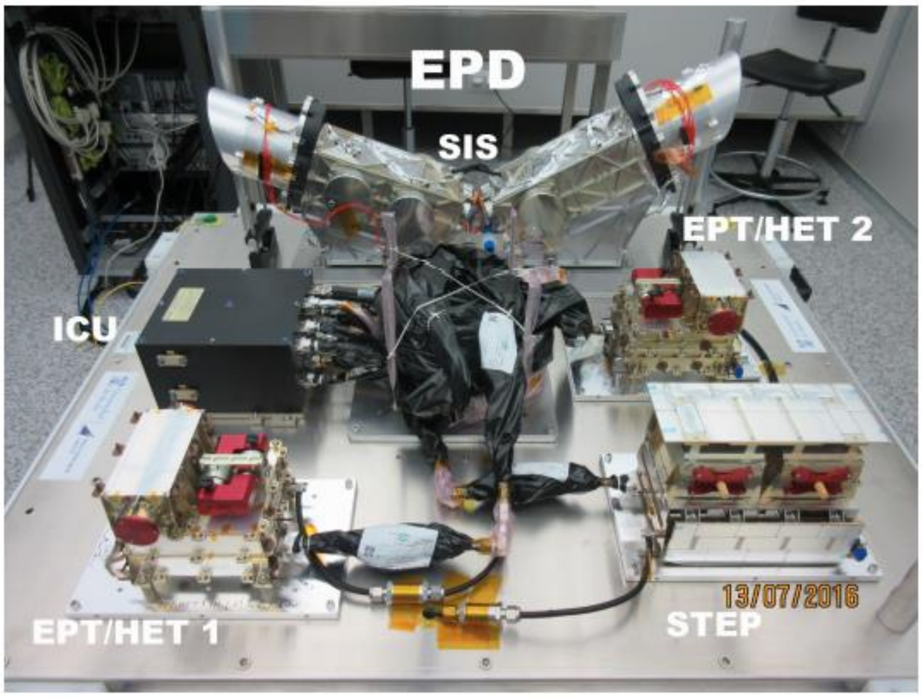


Fig. 2.4: Photo of the EPD instrument suite [4]

The nominal telemetry speed of the EPD is 3600 bps. This can be adjusted depending on the proximity to the Sun by changing the telemetry rates. Data can be classified into 4 categories: housekeeping, nominal, burst-mode and low-latency data. Housekeeping data has the highest priority and is received on Earth to monitor the instruments functionality and status. Nominal science data has the EPD-related information and can vary in resolution and size. To increase the resolution of the data, a burst mode can be activated by the operators on Earth, by a smart trigger or by other Solar Orbiter instruments.

The data latency depends on the distance of the satellite from Earth and on the state of the memory buffer. Small samples of scientific data are sent to show samples of what is being recorded and to plan accordingly. Low-latency data is automatically processed by a pipeline and sent back to Earth with a lower priority.

2.3 Solar Orbiter's EPD ICU

In this project, we will be working only with material related to the Instrument Control Unit. It functions as the interface between the EPD sensors and the rest of the spacecraft and provides the sensors with telecommand and telemetry communication capabilities, time synchronization, processing and power. It is composed of the Central Data Processing Unit (CDPU) and the low-voltage power supply (LVPS). Both of these units have a nominal and a redundant unit in case of failure, which makes 4 electronic boards in total. All the boards are packaged in a single case to simplify transport and connections. Parts have been tested to withstand 100 Krads.

The LVPS provides power to both the CDPU and sensors. Its main purpose is to provide a steady +28V power supply to all sensors. Latch-current limiters (LCLs) are used to turn off sensors in case of over or under voltage. It won't be addressed in more detail as it is outside the realm of computer engineering.

In this project we will be specifically working with software running on the CDPU.

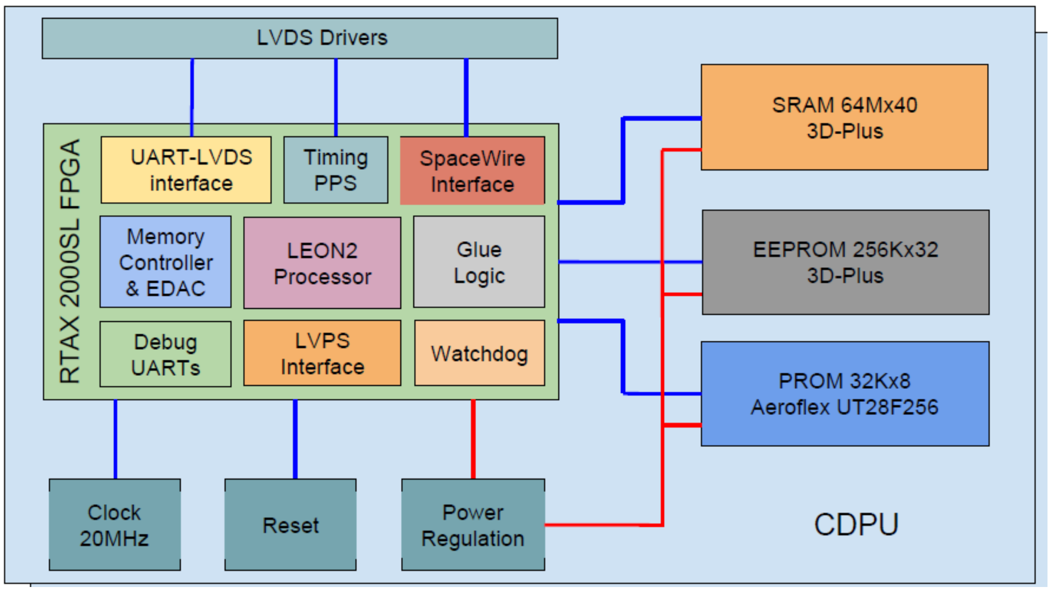


Fig. 2.5: Schematic of the ICU CDPU [4]

The CDPU currently runs in a board based on the LEON2 architecture, the Microsemi RTAX200 FPGA. Sensors communicate with the ICU via Universal Asynchronous Receiver Transmitter/Low Voltage Differential Signaling (UART/LVDS) interfaces operating at 115200 bauds. Sensors also have an interface to receive a pulse-per-second from the ICU, which is used to synchronize the data transmission, except for the SIS which transmits every 3s. The ICU also shares information with MAG, RPW and SWA to allow for burst-mode high-frequency data acquisition when the triggering conditions are met. Other functions of the CDPU include:

- Reception, storage and processing of sensor telemetries. Generation of CCSDS (Consultative Committee for Space Data Systems) science telemetries.

- Storage and managing of the sensor’s configuration tables. These tables contain configuration message sequences that can be modified by ground commands or autonomously by the EPD error recovery mechanism.

- Provision of Standard Packet Utilization Standard (PUS) services, such as telecommand verification, housekeeping or event reporting.

- Management of the accepted telecommands received from the spacecraft which must be either forwarded to the sensors or executed.

- Implementation of state logic required by the EPD.

- Fault detection, isolation and recovery mechanisms implementation. Examples of faults include storage corruption, fatal software errors or off-limit parameters. Some examples of recovery actions are activation of the safe mode, ICU reboot, ASW switching, sensor power-off or changing a sensor’s configuration.

2.4 FreeRTOS

FreeRTOS is an open-source Real Time Operating System intended for both hobbyists and professional developers working on commercial products. It is widely used for real-time applications with both hard or soft time requirements (critical for the functioning of the system or non-critical). Differently from a standard OS, an RTOS is designed to provide a deterministic execution pattern for processes. This is of great relevance for a system such us the EPD ICU, which requires tight execution planning for optimal operation and processing of the different sensor's data.

FreeRTOS is formed by the hardware layer, the device driver, the freeRTOS microkernel, the device driver and the tasks to be performed. Other RTOS advantages over a standard OS include increased maintainability, scalability, modularity, cleaner interfaces, easier testing, code reusability, greater efficiency, and easier control over peripherals.

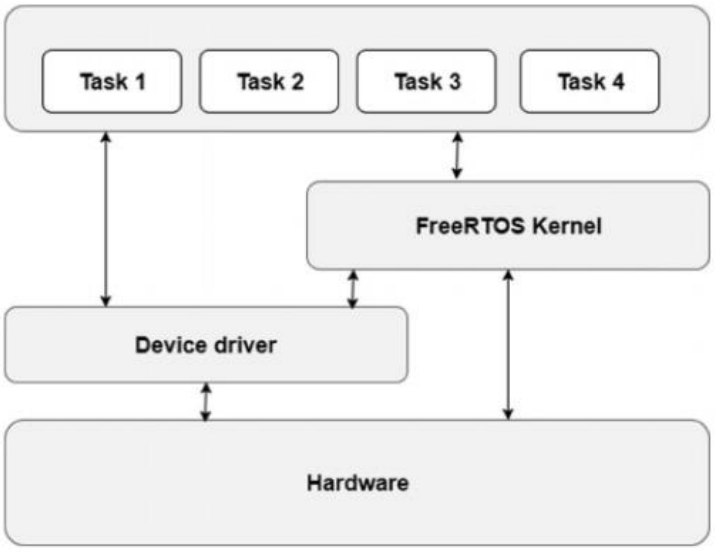


Fig. 2.6: Architecture of FreeRTOS [7]

In FreeRTOS, each thread is known as a task, and different priorities can be set to tasks depending on their importance, so the processing deadlines are met, and task execution is predictable. The scheduler used for the tasks can be configured to be the fixed prioritized preemptive or cooperative scheduler. We will be using the first one, as priorities should be absolute, and the highest priority task should always execute first.

It includes many useful tools and features to write time-critical applications such as notifications, queues, binary and mutex semaphores, timers, hook functions, stack-overflow checking and interrupt nesting, among others. Additionally, using the idle task hook, it can decrease power consumption of the device dramatically.

FreeRTOS can be built with twenty different compilers and can run on more than thirty different processor architectures. Each combination of a compiler and processor is considered a FreeRTOS port. Each port includes both common files with other ports and specific files. Each port includes a header file called FreeRTOSConfig.h. which adjusts the settings for the specific application. There should be an individual configuration file for each project.

In this project, we will be using a FreeRTOS port for the RV32M1 chip, which uses the RISC-V C and C++ cross-compiler to build programs runnable in the PULP RI5CY core.

2.5 RISC-V Architecture and the RI5CY core

RISC-V is an open instruction set architecture that is freely available for both academia and industry. It is an improvement of the well-known Reduced Instruction Set Computing or RISC architecture. It is the fifth edition of the RISC ISA designed at UC Berkley, California.

It is a load-store architecture: arithmetic instructions are register-only, load and store instructions are the only ones that interact with memory and data must be loaded into a register before it can be operated on. It supports both 32 and 64-bit processors, it has no branch-delay slots (unlike LEON), and uses little-endian byte ordering. As the current implementation uses LEON, this required changing the logic of telecommand processing, so the different fields were sent or read in little-endian. RISC-V can be used in many implementations as it has a base ISA and implementation-specific extensions. To provide better support for embedded systems, RISC-V uses a code compression extension called RVC. RVC substitutes the common 32-bit instructions with shorter 16-bit instruction encodings.

In this project we will be using a specific implementation developed by PULP called RI5CY. It is a 4-stage in-order 32-bit RISC-V processor core. It includes hardware loop support, post-increment load/store instructions and additional ALU instructions that are not present on the standard RISC-V ISA. These include simple bit manipulation/counting and min/max/avg instructions. It also provides optional floating-point operations support, ASIC/FPGA synthesis support and vectorial ALU operations.

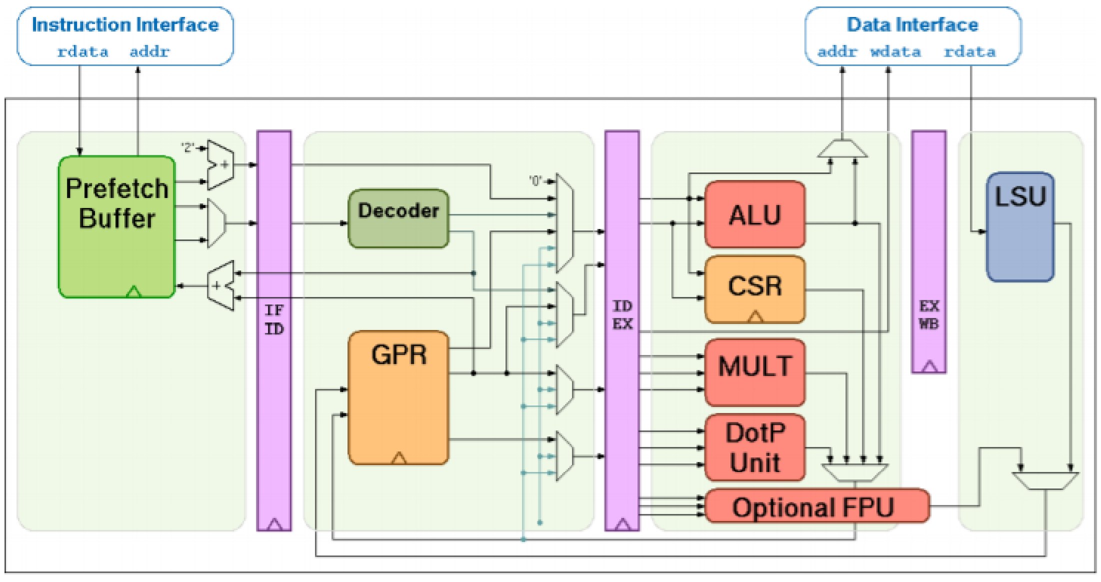


Fig. 2.7: Schematic of the RI5CY core [8]

2.6 The RV32M1 Vegaboard

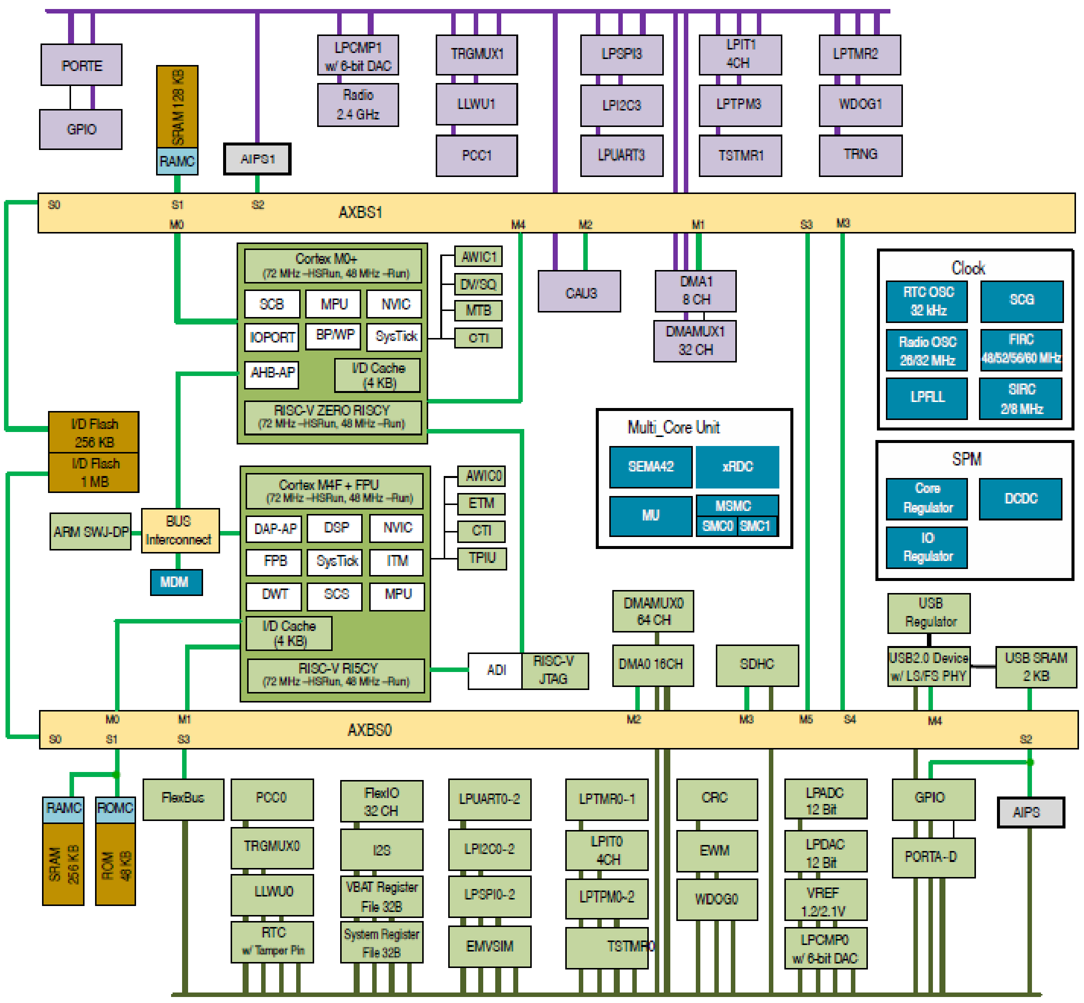


Fig. 2.8: RV32M1 Vegaboard modules and interfaces [9]

The OpenISA VEGAboard is a small, low-power and cheap development board aimed for application prototyping using the RV32M1 chip. It is what is known as an SoC or System on Chip. The single-chip device includes as its communications suite Bluetooth BLE, a flash serial programmer and virtual serial port. It has two RISC-V and two ARM CPUs, various peripherals, a 2.4 Ghz radio and programmable pins. Its RISC-V cores are based on the PULP platform and are the RI5CY "main" core and the ZERO-RI5CY secondary core, which is usually used as a coprocessor. They communicate via shared memory and messaging peripherals.

Specifications:
1 MiB flash and 192 KiB SRAM (RI5CY core)
256 KiB flash and 128 KiB SRAM (ZERO-RISCY core)
Low power modes
DMA support
Watchdog, CRC, cryptographic acceleration, ADC, DAC, comparator, timers, PWM, RTC, I2C, UART, SPI, external memory, I2S, smart card, USB full speed, uSDHC, and 2.4 GHz multiprotocol radio peripherals
On-board sensors and peripherals:
32 Mbit SPI flash
6-axis accelerometer, magnetometer, and temperature sensor (FXOS8700)
Ambient light sensor
RGB LED
MicroSD card slot
Antenna interface
Additional features:
Form-factor compatible with Arduino Uno Rev 3 expansion connector layout
UART via USB using a separate OpenSDA chip
RISC-V flash and debug using external JTAG dongle (not included) via 2x5 5 mil pitch connector

Table 1: RV32M1 VEGAboard specifications

The RISC-V cores use a slow internal reference clock (SIRC) to manage the system timer and a fast internal reference clock (FIRC) to generate the 48MHz core clock. The USB connector is linked to the OpenSDA chip which provides the serial USB interface via the LPUART0, used for both console and logging. It should be noted that on Linux systems LPUART0 is recognized as an ACM device while the rest are recognized as TTL devices.

Flashing and debugging cannot be made via the USB interface, instead, a separate hardware debug probe must be used. In our project, the J-Link EDU debug probe was chosen as the debugger with a Cortex-M adapter. The J-Link EDU is the low-cost version of the Segger J-Link, used for educational purposes. It offers the same functionality as its base version. It supports a wide range of microcontrollers and CPUs, unlimited breakpoints in flash memory and concurrent access to CPU by multiple applications.

Chapter 3 – Port development process

In this section we will describe the main work methodology that was adopted and the decisions that had to be made in order to carry on with the development of the project. It will be divided into a series of subsections explaining each step of the development and validation process.

3.1 Installation and setup of the development environment

The installation of the development environment was made following OpenISA's official guide. However, a series of problems arose, and some changes had to be made.

We can summarize the development environment setup in the following steps:

- Installing the Eclipse IDE for C/C++ Developers or Eclipse CDT.
- Installing the GNU MCU Windows Build Tools.
- Installing the RV32M1 GNU GCC Toolchain
- Installing OpenOCD
- Installing the RV32M1 SDK
- Configuring environment variables
- Configuring paths in Eclipse

A first attempt was made following the Windows guide, as it is the most complete and the one that has the most online support. After following it thoroughly, the Eclipse IDE and the toolchain were configured correctly, along with OpenOCD to connect to the Jlink debugger with its appropriate driver. One key part of this setup was the installation of the OpenOCD driver on the Jlink debugger. The standard Segger driver won't work with OpenOCD, so this separate driver must be installed in order for it to work accordingly. This was done successfully at first, however, some months later Windows 10 updated and the Jlink debugger was no longer detected by the OS. Attempts to install the OpenOCD driver back were futile as an error window popped up every time.

Development was moved to an Ubuntu 18.04 virtual machine instead, running on top of Arch Linux. Linux doesn't require any specialized driver for the debugger. However, this also proved to be problematic as the building process made by the toolchain was incorrect. The .ELF executable files couldn't be generated. To fix this, the toolchain had to be built from the Github source, a process which took several hours to complete. After that, projects began building correctly.

Days later, it was discovered that the debugger driver could be reinstalled in Windows using a software called Zadig, which is a simple driver installer for USB devices. After the successful installation of the driver, the debugger was detected by OpenOCD correctly. Development was moved back to Windows for convenience as it is the OS where FreeRTOS is developed and where most applications are tested.

3.2 Detailed installation guide for Windows:

The installation process is explained in more detail here, as it is crucial to do it in an orderly fashion to avoid future errors. This guide is based on OpenISA's official guide, but with a few additions and modifications. Only a Windows guide was made, as it is the most commonly used OS by users, and it has the most documentation from OpenISA.

3.2.1 Optional requirements

Some tools can be optionally installed if we wished to debug applications via the Telnet Client along with OpenOCD. To install it simply type the following command on the Windows console: **`pkgmgr /iu:"TelnetClient"`**

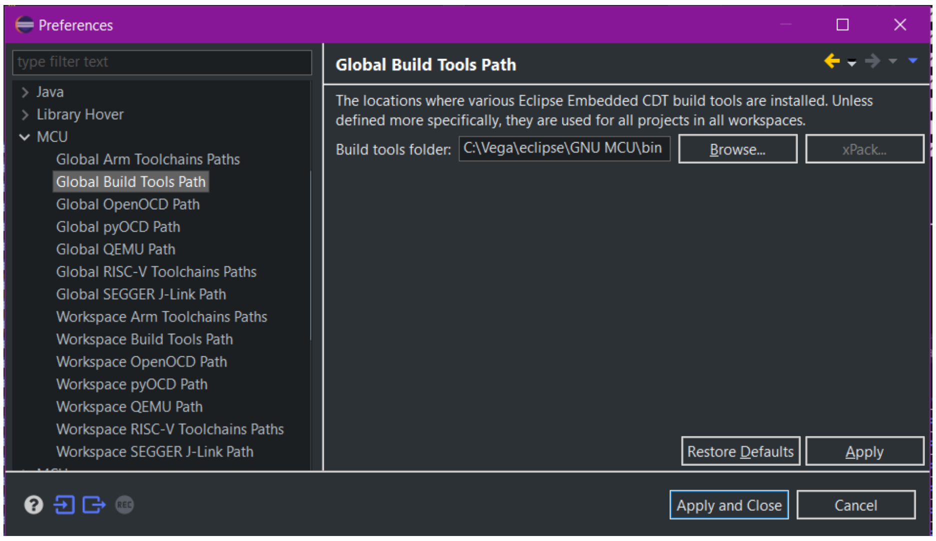
3.2.2 Installation of Eclipse CDT.

Simply head to
https://github.com/eclipse-embed-cdt/org.eclipse.epp.packages/releases and install the Windows version, selecting the x32 or x64 depending on your OS version.

3.2.3 Installation of the GNU MCU Windows Build Tools.

This step can be avoided if the Eclipse external builder is not used. The installation of these tools is necessary because the external builder uses commands such as make and rm.

1. Head to https://github.com/eclipse-embed-cdt/org.eclipse.epp.packages/releases and install the x32 or x64 version depending on your system.
2. Unzip the downloaded package in the Eclipse installation folder.
3. In Eclipse, go to Window → Preferences to set Global Build Tools Path to its installation location:



Global Build Tools Path

The locations where various Eclipse Embedded CDT build tools are installed. Unless defined more specifically, they are used for all projects in all workspaces.

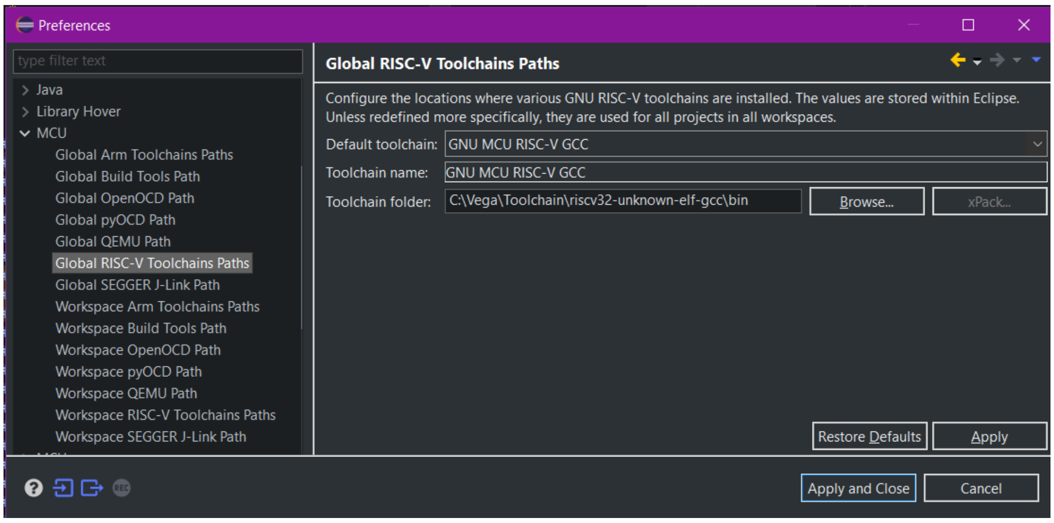
Build tools folder: C:\Vega\eclipse\GNU MCU\bin

Fig. 3.1: Global Build Tools Path Eclipse configuration

3.2.4 Installation of the RV32M1 GNU GCC Toolchain

OpenISA provides a pre-built GNU GCC toolchain for the RV32M1.

1. Download it from the official webpage: https://open-isa.org/downloads/
2. Unzip the file in the desired location.
3. Go to the Eclipse menu Windows → Preferences to set Global RISC-V Toolchains Paths to its installed location.


Preferences

Global RISC-V Toolchains Paths

Configure the locations where various GNU RISC-V toolchains are installed. The values are stored within Eclipse. Unless redefined more specifically, they are used for all projects in all workspaces.

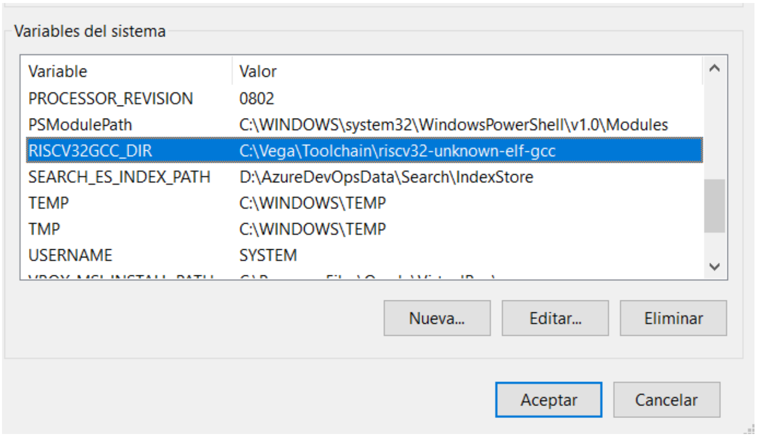
Default toolchain: GNU MCU RISC-V GCC

Toolchain name: GNU MCU RISC-V GCC

Toolchain folder: C:\Vega\Toolchain\riscv32-unknown-elf-gcc\bin

Fig. 3.2: Global RISC-V Toolchains Paths Eclipse configuration

4. Create an environment variable called RISCV32GCC_DIR and set it to the GNU GCC toolchain folder.


Variables del sistema

Variable	Valor
PROCESSOR_REVISION	0802
PSModulePath	C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
RISCV32GCC_DIR	C:\Vega\Toolchain\riscv32-unknown-elf-gcc
SEARCH_ES_INDEX_PATH	D:\AzureDevOpsData\Search\IndexStore
TEMP	C:\WINDOWS\TEMP
TMP	C:\WINDOWS\TEMP
USERNAME	SYSTEM

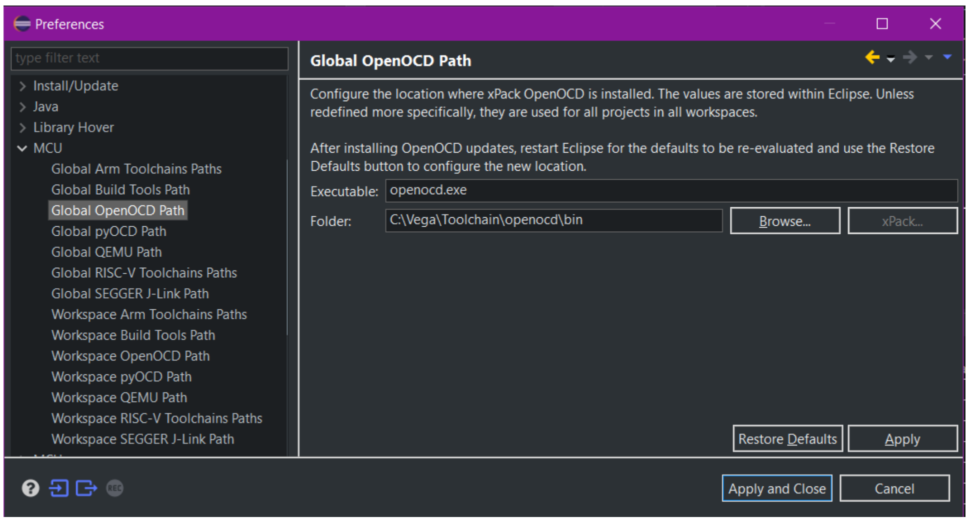
Fig. 3.3: RISCV32GCC_DIR enviroment variable configuration

5. Add the GNU GCC Toolchain executable files folder */riscv32-unknown-elf-gcc/bin* to the system PATH variable.

3.2.5 Installation of OpenOCD

OpenISA provides a pre-built OpenOCD binary file and the driver to support the J-link adapter hardware.

1. Download OpenOCD and the J-Link driver from https://open-isa.org/downloads/
2. Unzip the package in the desired location.
3. Go to the Eclipse menu Windows → Preferences to set Global OpenOCD Path to its installation location



Global OpenOCD Path

Configure the location where xPack OpenOCD is installed. The values are stored within Eclipse. Unless redefined more specifically, they are used for all projects in all workspaces.

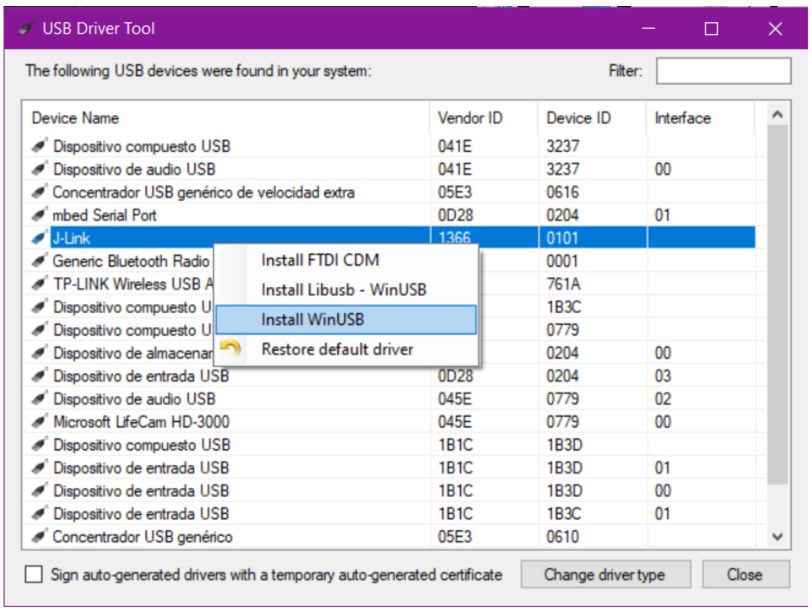
After installing OpenOCD updates, restart Eclipse for the defaults to be re-evaluated and use the Restore Defaults button to configure the new location.

Executable: openocd.exe

Folder: C:\Vega\Toolchain\openocd\bin

Fig. 3.4: Global OpenOCD Path Eclipse configuration

4. Add the path of OpenOCD.exe to the environment variable PATH.
5. Plug in the J-Link hardware debugger to your PC.
6. Run *\openocd\drivers\drivers\UsbDriverTool.exe*
7. Right click the "J-link driver" and select "Install WinUSB" so OpenOCD can recognize the device. The official segger driver will be uninstalled and can be reinstalled by selecting "Restore default driver".

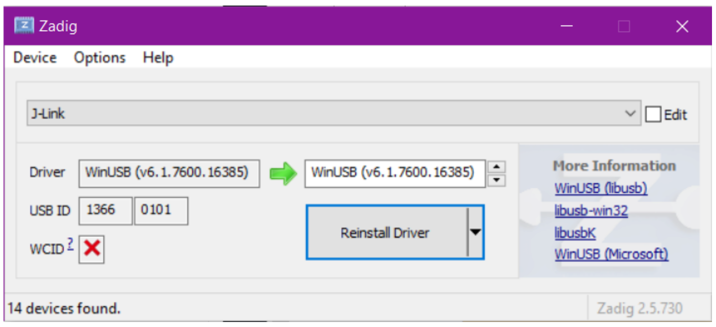


The following USB devices were found in your system:

Device Name	Vendor ID	Device ID	Interface
Dispositivo compuesto USB	041E	3237	
Dispositivo de audio USB	041E	3237	00
Concentrador USB genérico de velocidad extra	05E3	0616	
mbed Serial Port	0D28	0204	01
J-Link	1366	0101	
Generic Bluetooth Radio		0001	
TP-LINK Wireless USB A		761A	
Dispositivo compuesto U		1B3C	
Dispositivo compuesto U		0779	
Dispositivo de almacenar		0204	00
Dispositivo de entrada USB	0D28	0204	03
Dispositivo de audio USB	045E	0779	02
Microsoft LifeCam HD-3000	045E	0779	00
Dispositivo compuesto USB	1B1C	1B3D	
Dispositivo de entrada USB	1B1C	1B3D	01
Dispositivo de entrada USB	1B1C	1B3D	00
Dispositivo de entrada USB	1B1C	1B3C	01
Concentrador USB genérico	05E3	0610	

Fig. 3.5: Installing the WinUSB J-Link driver via the OpenOCD driver tool

8. (Optional) If the above step launches an error, head to https://zadig.akeo.ie and download the latest version of Zadig, a multipurpose driver installer.
9. (Optional) Run the executable as administrator and select "J-Link" from the drop-down list. Click on the "Install Driver" button.



J-Link

Driver WinUSB (v6.1.7600.16385) → WinUSB (v6.1.7600.16385)

USB ID 1366 0101

WCID ✗

14 devices found.

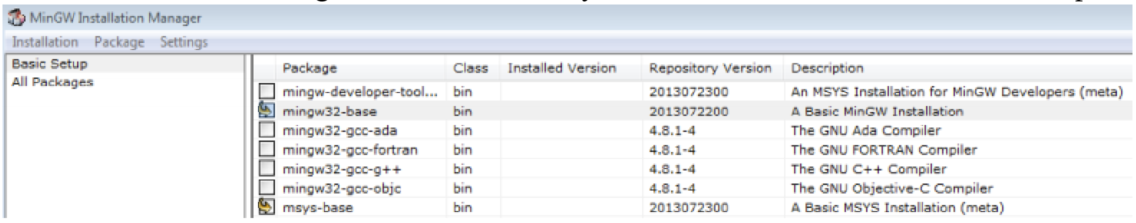
Zadig 2.5.730

Fig. 3.6: Installing the WinUSB J-Link driver via the Zadig driver tool

3.2.6 Installation the RV32M1 SDK and optional tools

1. Head to https://open-isa.org/downloads/ and download the SDK.
2. Unzip the file in your desired location.
3. (Optional) If you wish to use the command line to build, debug and run applications, install the following software:
   a. Install CMake from https://cmake.org/download/ by downloading the 32x or 64x version installer.
   b. Install the MinGW development tools from https://sourceforge.net/projects/mingw/files/Installer/

1. Ensure that the “mingw32-base” and “msys-base” are selected under Basic Setup.



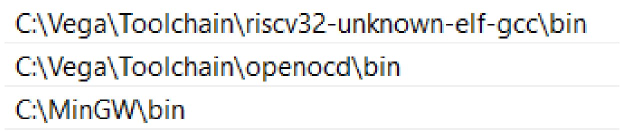
	Package	Class	Installed Version	Repository Version	Description
☐	mingw-developer-tool...	bin		2013072300	An MSYS Installation for MinGW Developers (meta)
✓	mingw32-base	bin		2013072200	A Basic MinGW Installation
☐	mingw32-gcc-ada	bin		4.8.1-4	The GNU Ada Compiler
☐	mingw32-gcc-fortran	bin		4.8.1-4	The GNU FORTRAN Compiler
☐	mingw32-gcc-g++	bin		4.8.1-4	The GNU C++ Compiler
☐	mingw32-gcc-objc	bin		4.8.1-4	The GNU Objective-C Compiler
✓	msys-base	bin		2013072300	A Basic MSYS Installation (meta)

Fig. 3.7: MinGW installation options

2. Click on Installation → Apply Changes and follow the remaining instructions to complete the installation.
3. Add <*install_dir*>\bin to the PATH environment variable

3.2.7 Environment Variables configuration

If the above tutorial was followed correctly you should have the following variables in your PATH:



```
C:\Vega\Toolchain\riscv32-unknown-elf-gcc\bin
C:\Vega\Toolchain\openocd\bin
C:\MinGW\bin
```

Fig. 3.8: PATH envrioment variable configuration

3.2.8 Additional Eclipse configurations

Some additional eclipse configurations and extensions can be helpful to share and debug the project.

Print Size:

To check the size of each of the program sections the Print Size tool is very useful and should be enabled in most low-level projects. The tool can be accessed via the binary file found in the Vega toolchain or via the eclipse GUI.

To enable it in Eclipse go to Project → Properties → C/C++ Build → Settings → Toolchains and check the “Print Size” checkbox. The Print Size tool will be invoked after each build and the output will be displayed on console.

Subclipse

If the project is to be developed alongside someone else, or needs supervision, the Subclipse extension is very useful for sharing and managing Subversion repositories. To install it, go to Help → Eclipse Marketplace and search for “Subclipse” on the searchbox. Press on “Install” and follow the on-screen steps.

Alternatively, a Github repository can be used to share the project. Both the console version and the desktop application are valid options to achieve this.

3.3 Development of the UART Port

The RV32M1 has 4 built-in Low Power UART interfaces called LPUART. The only one connected to the USB port is the LPUART0, which is used for logging, debugging and can be used for sending or receiving data. The following block diagrams show its structure:

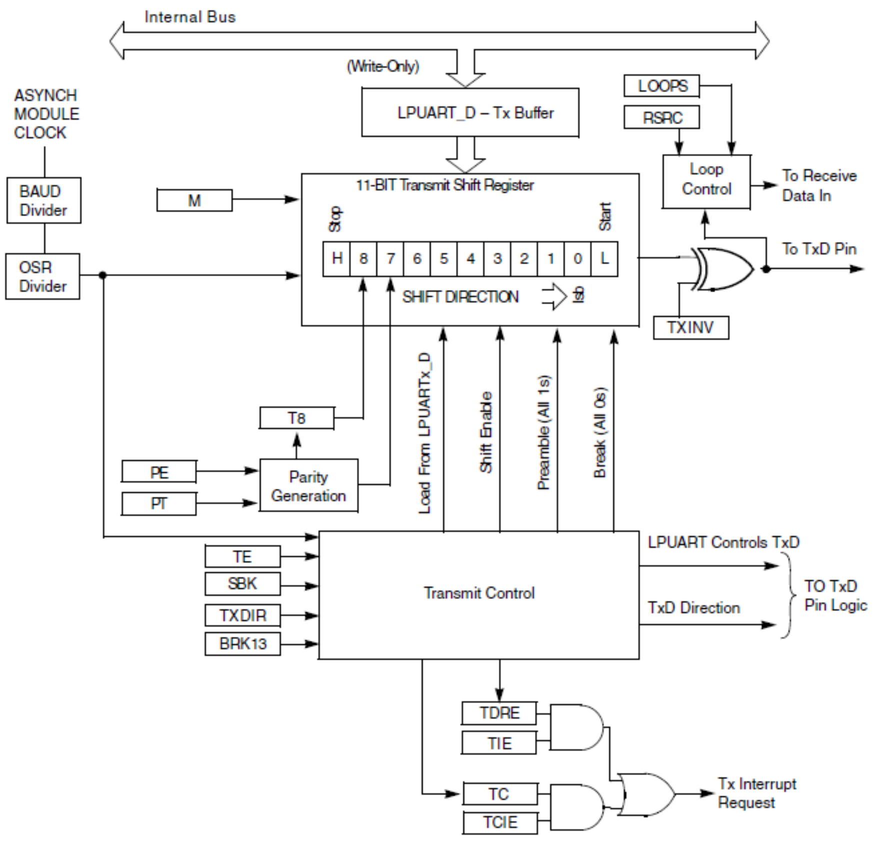


Fig. 3.9: LPUART transmitter block diagram [9]

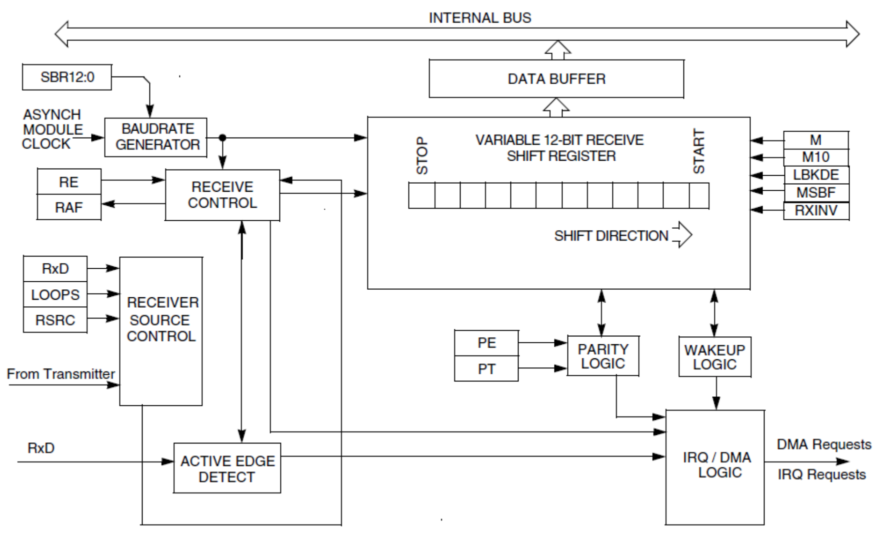
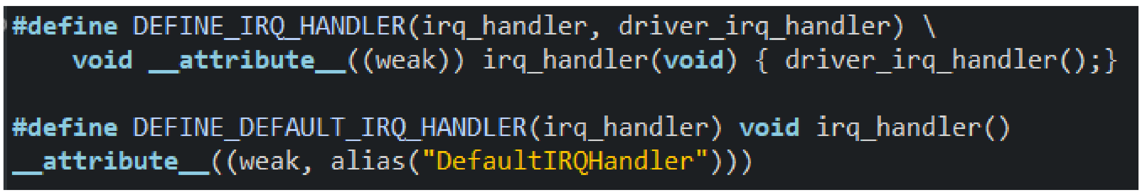


Fig. 3.10: LPUART receiver block diagram [9]

3.3.1 UART Port Testing

OpenISA's freertos_lpuart_ri5cy demo project includes a source file which exemplifies the handling and reception of data via LPUART0. This example was used as a base to create a test in which characters are received via Putty's serial interface and are then checked and processed to check their validity so they can be sent again via the UART if they form a correct data stream. Our testing program is formed by two tasks: recv_task and send_task, and the interruption handler: LPUART0_DriverIRQHandler. The IRQ vectors can be found at the RV32M1_ri5cy.h file and are defined as the default handlers in the system_RV32M1_ri5cy.c file:



```
#define DEFINE_IRQ_HANDLER(irq_handler, driver_irq_handler) \
    void __attribute__((weak)) irq_handler(void) { driver_irq_handler();}

#define DEFINE_DEFAULT_IRQ_HANDLER(irq_handler) void irq_handler()
__attribute__((weak, alias("DefaultIRQHandler")))
```

Fig. 3.11: Setting of the IRQ handlers

Our application requires installing a handler that manages the reception of data when the UART's RX signal activates. However the FreeRTOS port for the RV32M1 installs it's own handlers and it's not easy to use a custom handler used by the basic UART read and write character functions. Also, both the RX and TX signals use the same handler for UART interruptions. The default handler only activates with the LPUART_RTOS_Send and LPUART_RTOS_Receive functions, which sends a buffer instead of a single character and are the FreeRTOS-specific functions.
The solution that was found was to manually trigger the handler by using the LPUART_RTOS_Receive function with a buffer of only 1 character and, once inside the handler, send or receive characters with the LPUART_ReadByte and LPUART_WriteByte functions, which don't trigger the handler.

In the handler, we need to check that we receive the appropriate 4 byte header, then 2 bytes with the application data length and finally, the application data itself. The first 6 bytes represent the synchronization part of the packet that other components use to recognize it as an ASW packet. To check this, a simple state machine was used. After we've checked that all of this data is correct the handler won't process any new characters.

The recv_task will poll the handler via a stream_ok function and, if the stream is indeed correct, it will suspend itself and wake up the send_task, which initially was in a suspended state. This send_task will simply send the stream with the LPUART_RTOS_Send function. The handler will be triggered again, but won't do anything because of it will be at a "RX complete" state.

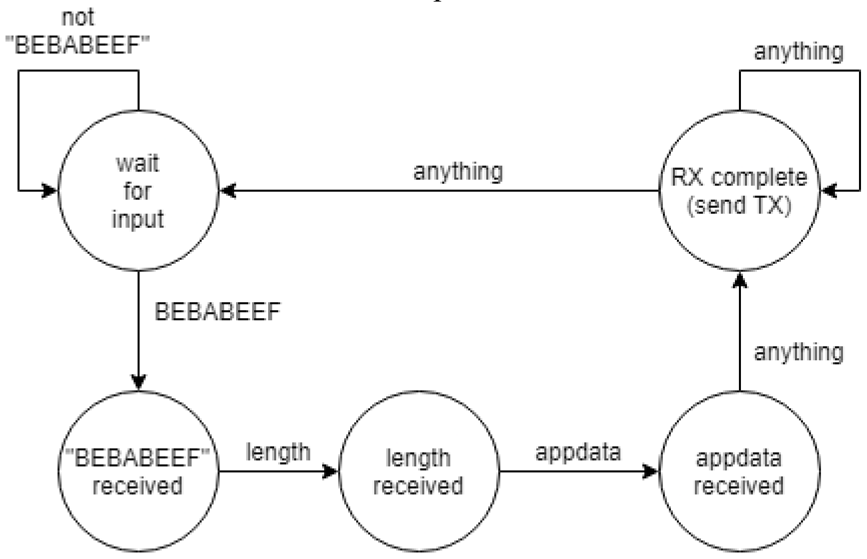


Fig. 3.12: State diagram of the UART test program

After the successful initial test, this program was ported to C++ to test its compatibility in this more complete language. Some errors in the FreeRTOS freescale header files made the program hard to compile initially. These includes were moved to a new file which included all the basic FreeRTOS include headers. This file was included in the edroom glue header file, which was then included in our test program. This proved to solve the problem. A minor change was also done to address a C++ struct declaration error, as it is different from standard C. After that, the program worked flawlessly. The C++ test was deemed successful.

The next upgrade to the test program, involved an improvement so it could accept streams in an infinite loop by resetting the handler's variables after each complete stream was sent. The receiving and sending tasks were modified so they would suspend and resume adequately. Freescale's lpuart file also needed to be modified in order to have this new functionality. A function to reset the handler was incorporated so it could reset its state flags and buffers. The new program worked as expected.

Other testing programs were tried that worked without using the FreeRTOS operating system. There is one for testing UART transmission and reception by the polling method using the LPUART_WriteBlocking and LPUART_ReadBlocking functions, which don't use interruptions and another one which uses the LPUART_TransferSendNonBlocking and LPUART_TransferReceiveNonBlocking, which uses interruptions and allows for custom handler installation. These LPUART functions are provided by the distribution. These programs were later used to test using 2 UARTs at the same time.

3.4 Development of the MCU ICU port

To start with the proper port of the MCU ICU, an old, outdated version of it was used which already run on an obsolete FreeRTOS distribution. This initial project was developed with EDROOM, a powerful tool used to build embedded system software with an easy to use interface. It had to be adapted to match the current FreeRTOS version by changing a lot of the functions and variables which had their syntax changed over the years. The process consisted of building the project and checking the Eclipse "problems" tab in order to identify and correct errors and warnings. At the beginning there were more than 400 errors. Some of them simply required changing a variable's deprecated type to the new version, such as the task handles, tick types, and enable and disable interrupts functions. Others involved changing a function's arguments, such as the task creation function's. The project also required a restructuring to eliminate the old FreeRTOS files which created conflicts with the new ones. Type conflicts also existed, especially because of matching definitions of integer types such as uint32_t. The basic types of the project were discarded in favor of universal GCC stdint types.

Project include paths and preprocessor variables had to be configured so all the files and macros necessary for the correct compilation of the project were found. Some flags were also added after the g++ command for C++ compilation to avoid a number of errors (-fno-rtti -fno-exceptions -lstdc++). Functions such as __dso_handle, __cxa_pure_virtual, new and delete had to be defined or redefined due to compilation errors.

3.4.1 Project Structure

The project structure is similar to the one in the original project, but additional folders with the FreeRTOS files were added following a structure similar to the one in the official examples. These new folders include board, debug, drivers, freertos, settings, source, startup and utilities. Some of these folders are linked resources as they are part of the SDK and their paths are preconfigured in the project settings. The hierarchy of files shouldn't be modified in order to avoid compilation errors. Folders belonging to the original project are components, config, dataclasses, edroom_glue, protocols, pswpackages, swinterfaces and swpackages.

Some of the main files of each folder will be enumerated:

Board:
board.c initializes the debug console in LPUART0 while board.h describes some important core board functionalities with macros. clock_config.c and clock_config.h allow tweaking the system clocks in order to over or underclock the board. pin_mux.c and pin_mux.h handle how the pins are configured and multiplexed as each header pin has multiple functions. More can be learnt in **annex 2** and in the official OpenISA documentation.

Components:
It includes the files that define the behavior of the different EDROOM components which are initialized at the start of the program.

Config:
config.h has custom defined preprocessor macros that change the behavior of the program.
Some important ones include CONFIG_EDROOMBP_DEPLOYMENT_NEED_TASK, which changes if the system deployment is done via a task or not, BEBASYNC 6 which

defines that the synchronization part of a packet are the first 6 bytes and USE_EDROOMBP_LPUART0_IRQ_HANDLER, which indicates that a custom handler will be used for LPUART0 interruptions.

Dataclasses:
The files in this folder define custom types that are used by the EDROOM components of the program. Some important ones include CDTMList, which is where sent telemetries are managed, CDTCDescriptor, where received telecommands are processed and Pr_time, the class that implements real-time functions.

Debug:
This folder holds the .o object files that form up the .ELF executable at the build process. These files are temporary, and they are destroyed upon cleaning the project.

Drivers:
The main FreeRTOS driver files are here. fsl_port.h and fsl_common.h define some of the core functionality such as status codes, events and interrupt logic. fsl_lpuart.c and fsl_lpuart.h are very important in this particular project as they describe the UART logic, which is critical for this port. These files were made by FreeScale semiconductor, which was acquired by NXP in 2015.

Edroom_glue:
It has some important header files that link different parts of the project based on EDROOM components. edroomdeployment.cpp and edroomdeployment.h describe the system deployment object structure, including the memory required by EDROOM components. An fsl_inclues.h file was added that holds the main port includes.

Freertos:
The files here describe the main data structures and components that FreeRTOS uses such as semaphores, queues, routines, lists and tasks. A very relevant file in this project has been heap.c, as it had to be redesigned to simplify the debug process after memory problems arose.

Protocols:
This folder has the models that describe the main ICU ASW communication protocols. They won’t be addressed in more detail as they are outside of this project’s aims and objectives

Pswpackages:
The original basic_types.h file can be found here, which was modified so there wouldn’t be type conflicts with the FreeRTOS files, which use the STD types.

Settings:
The RV32M1_ri5cy_flash.ld linker configuration file can be found here. This file initializes and describes the program memory areas and it can be tweaked to change the size and memory locations of structures such as the heap or the stack.

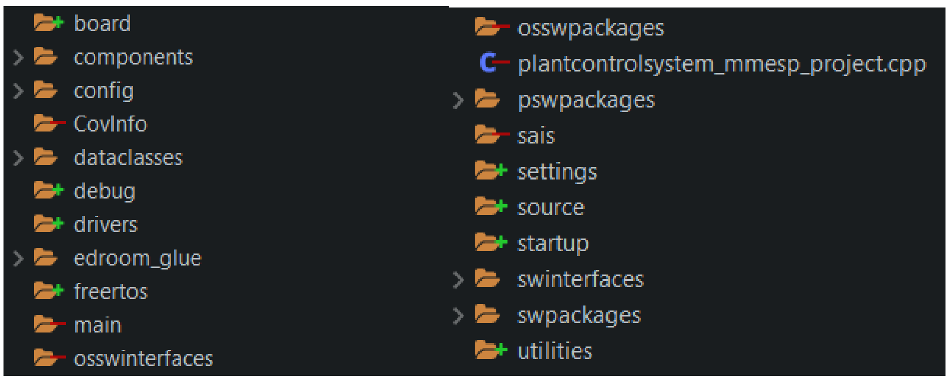
Source:
The file containing the main function, icuasw_mmesp_project.cpp, can be found in this folder along with FreeRTOSConfig.h, the FreeRTOS project configuration file. In the main function components are instantiated and connected with the systemDeployment object.

Startup
It contains three files with the same name and different extension named startup_RV32M1_ri5cy.S, .c and .h that are the boot files of the VEGAboard. The one ended in .S has the exception handlers written in assembly code that manage fatal CPU errors.

Swinterfaces:
Holds the main ASW ICU .h software interfaces which have the function declarations that are given functionality in the swpackages .cpp files.

Swpackages:
Most of the ICU ASW program code files are in this folder. The most heavily modified files in this project are in edroombp_slib_head, the abstraction layer of the RTOS, emu_sc_channel_drv_slib_head, the manager of the spacecraft communications, and icuasw_pus_services_slib_head, the implementation of the Packet Utilization Services (PUS) used by the ESA. A service is a set of related telecommands and telemetries. The ASW executes the received TCs and generates TMs which can either be autonomous or in response to TCs.

Utilities:
Contains some useful system utilities such as the fsl_log and fsl_debug_console tools.



```
board                  osswpackages
components             plantcontrolsystem_mmesp_project.cpp
config                 pswpackages
CovInfo                sais
dataclasses            settings
debug                  source
drivers                startup
edroom_glue            swinterfaces
freertos               swpackages
main                   utilities
osswinterfaces
```

Fig 3.13: Comparison between the initial Plant Control System project and final project's folders

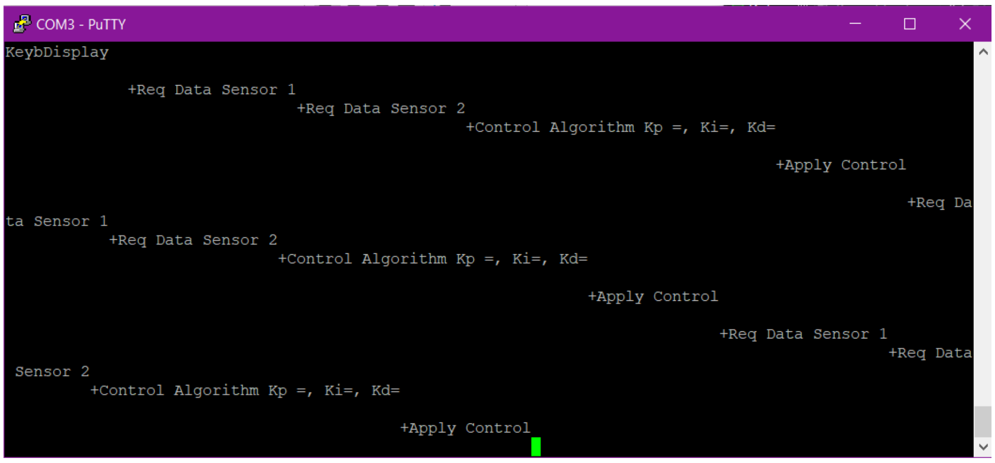
3.4.2 Plant Control System V0

The Plant Control System file is the main executable file that has been generated to complete this example port. It initializes 6 components and then the system deployment is configured and started. The components are CCControlPlant, CCLEDTask, CCHTTPServerTask, CCKeyScanTask, CCRS232Task and CCMonitor. Each component initializes several tasks, counting semaphores and mutex semaphores. This program proved to be very challenging to run with the initial setup and several changes had to be made. At first, a lot of errors were shown because of failure to initialize tasks, semaphores and mutexes. The program was running out of memory as the heap only had 10KB available. The FreeRTOS configuration file was modified so the heap could hold 40 KB. This proved to be a great improvement, as all the components until component 5 were initialized correctly. After component 5 was initialized, the program broke again, this time because of heap corruption. This error was very difficult to trace and debug, as the heap became corrupt very early on the program execution and no clear reason to why this happened was found.

Dynamic memory version with a new heap management algorithm

A new version of Plant Control System was tested, this time with components using dynamically allocated memory. For simplification of the debug process, a custom heap memory management program was made with custom malloc() and free() functions. This program was based on the public Github Gist file provided by Masahiro Hiramori **[14]** and adapted to comply with FreeRTOS heap format and functions. The new heap algorithm simply initializes a pointer at the start of the heap array and updates it when a function requests memory. An MCB or Memory Control Block manages each memory segment and specifies its size and availability. If no free block is found, a new one is created after the last valid memory address. If a memory element needs to be freed, it is set to being available. The program keeps track of how many heap bytes are remaining at every moment, this is very useful for debugging purposes. The amount of memory requested must be greater than 0 and lesser than the free memory remaining. There are three main pointers: one points at the heap start, other at the last valid address and another at the end of the heap. These pointers can be used to always keep track of the memory directions the heap is using and to check in Eclipse's memory view what's inside of the of it or if there are any conflicts with other parts of the program. The implementation can be seen in **annex 1.**

This new version proved to be successful, and all components initialized correctly, however, the system deployment start failed. The main task it created was very large as it requested 65.5KB and the system ran out of memory again. However, it was discovered that it was caused by a misallocation of memory present in the original program. It was abnormal that this task requested that much memory, so it was modified so it would only request 1KB. The heap size was newly increased to 70KB, which proved to be enough to create the task. This task also requested a very high priority of 7, which was more than the default max priority of FreeRTOS which was 5. The max priority was increased to 8 for the task to work correctly. After this modification the program ran correctly.



```
KeybDisplay

             +Req Data Sensor 1
                               +Req Data Sensor 2
                                                 +Control Algorithm Kp =, Ki=, Kd=

                                                                                  +Apply Control

                                                                                                +Req Da
ta Sensor 1
           +Req Data Sensor 2
                             +Control Algorithm Kp =, Ki=, Kd=

                                                              +Apply Control

                                                                            +Req Data Sensor 1
                                                                                              +Req Data
 Sensor 2
         +Control Algorithm Kp =, Ki=, Kd=

                                          +Apply Control
```

Fig. 3.14: Plant Control System UART output

3.4.3 Plant Control System V1

A different version of Plant Control System was also tested using static components. The differences with its dynamic counterpart are the same as the ones in ICU ASW V1, which was developed first, and it's **addressed in 3.4.4**. This version caused new problems as it instantly crashed during the initialization of the first component, so it was deemed a failure and abandoned as we already had a working version of Plant Control System and using static memory provided no visible advantages.

3.4.4 ICU ASW V0

The next program to port was the ICU ASW educational prototype. It has a similar structure to Plant Control System, but this time a new set of EDROOM components had to be configured and started.

These instances of the component classes are:

- ICUASW: Its instance component contains all other components, and its main functionality is to simulate the flow of time, and in these starting versions, simulate the reception of TCs without having a real system sending them.
- CCEPDManager: It implements the high-priority component which receives the TCs and either executes them or forwards them to other components according to their priority.
- CCTM_ChannelCtrl: Implements the very-high-priority component which is in charge of transmitting the TMs generated by the rest of the system components.
- CCHK_FDIRMng: Implements the medium-priority component which provides the Housekeeping service, as well as the ones related to the Fault, Detection, Isolation and Recovery (FDIR) system. The Housekeeping service (service 3 of the PUS standard) periodically notifies about relevant system parameters such as temperature, voltages or baudrates.
- CCBKGTCExec: Implements the low-priority component which executes the TCs with the less priority.

After solving the usual problems of type conflicts and include errors, a new large problem arose. The program was very large, occupying a large chunk of the m_data chunk of memory and our large heap caused an overflow of this region. The main culprit was the systemDeployment variable, responsible for deploying the tasks with a size of 148KB. This constituted a major problem as the available SRAM of the RV32M1 VEGAboard is 192KB. If the heap was reduced, components rapidly ran out of allocatable memory and tasks, semaphores etc. could not be correctly created. Increasing m_data's size by modifying the linker script was also unsuccessful, as writing in the new area was considered an illegal instruction by the CPU.

Memory Area	Origin	Length
m_text	0x00000000	0x000FFF00
m_vector	0x000FFF00	0x00000100
m_data	0x20000000	0x0002E800
rpmsg_sh_mem	0x2002E800	0x00001800
m_usb_sram	0x48010000	0x00000800

Table 2 RV32M1 memory areas

The next solution that was tested was to declare the heap as a pointer in a custom memory address. Several options were tested. It was first declared at the start of m_data (0x2000000). This option offered mixed results as the components were created, but the main task didn't start correctly and instead it waited indefinitely as if it was blocked by another task. The next option was to declare the heap at the end of m_data (0x2002E800 – HEAP_SIZE). This caused the heap's pointers to be corrupted when the main task was created, and it subsequently crashed.

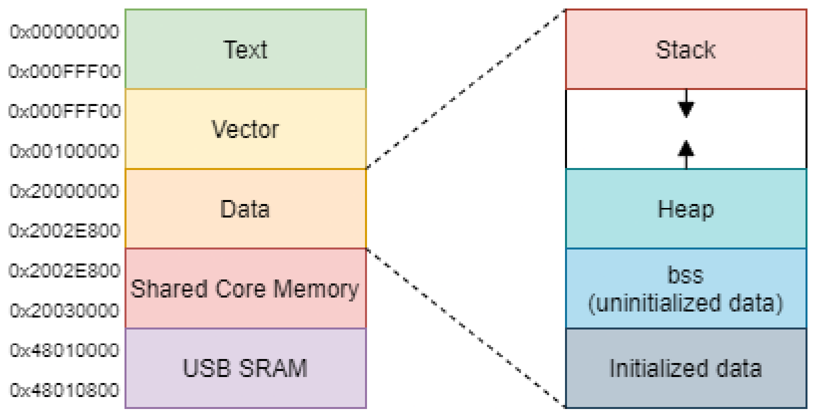


Fig. 3.15: Program memory representation

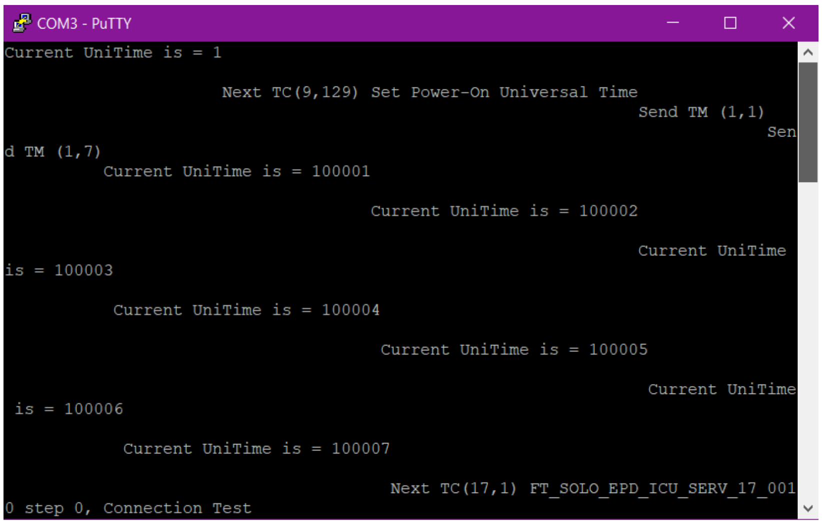
After that, a more methodical approach was tried. It required going into the entrails of the program and checking memory addresses to find available space. Eclipse's memory tab was used for this purpose. By checking the addresses pointed by the heap start and heap end pointers we could get a clear view of the content of this space. After a thorough analysis of the memory space, a valid address was found for the placement of the heap. Plant Control system worked perfectly with this new configuration. ICU ASW worked decently, as the kernel started successfully, but the tasks stalled soon after.

3.4.5 ICU ASW V1

A new version of the ICU ASW project was developed, this time completely prescinding from the heap. To change the general behavior, a simple option in the FreeRTOS configuration file had to be changed so dynamic memory wouldn't be used in favor of static memory. Functions like xTaskCreate were substituted by their static counterparts like xTaskCreateStatic. All components declaration were changed so the "new" function wouldn't be used. The new functions required passing the component stack as an argument. This stack could be initialized in the constructors of pr_task and pr_semaphore defined in the edroombp interface. The program worked well until the kernel started and failed as the LSU handler was triggered due to an illegal assembler instruction being run.

3.4.6 ICU ASW Lite

This third version of the ICU ASW was much lighter as it only used 3 components: ICUASW CCEPDManager and CCTM_ChannelCtrl. The array used as the heap was configured to be a static one like it originally was in the FreeRTOS heap management implementation and it could be 70KB as the data portion of the program was much lighter than in the full ICU ASW. After solving type and include errors, the program worked flawlessly.

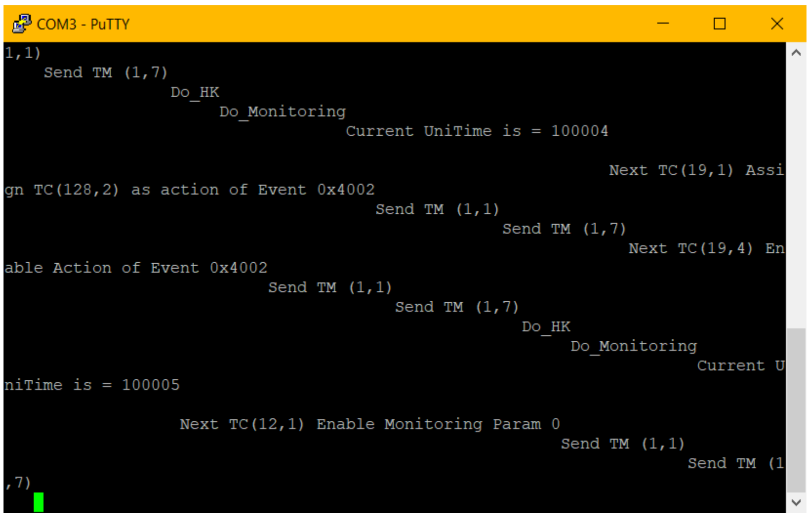


```
Current UniTime is = 1

                      Next TC(9,129) Set Power-On Universal Time
                                                                Send TM (1,1)
                                                                             Sen
d TM (1,7)
          Current UniTime is = 100001

                                     Current UniTime is = 100002

                                                                Current UniTime
is = 100003

           Current UniTime is = 100004

                                      Current UniTime is = 100005

                                                                 Current UniTime
 is = 100006

            Current UniTime is = 100007

                                       Next TC(17,1) FT_SOLO_EPD_ICU_SERV_17_001
0 step 0, Connection Test
```

Fig. 3.16: ICU ASW Lite UART output

3.4.7 ICU ASW V2

The third version of ICU ASW incorporated the static heap array which proved to be successful in the lite version and included a new important change: By changing the maximum number of telemetry messages that can be on the mempool at any given time, the amount of reserved memory used by the systemDeployment object could be reduced dramatically. This number was changed from 16 to 4 and systemDeployment went from being 148KB to just over 42KB. This allowed us to use the static heap array in this version, which proved to work fine and avoided memory conflicts with other variables. The size of the main task was also reduced like in previous versions, so an 80KB heap proved to be enough to compile and run successfully.



```
1,1)
    Send TM (1,7)
                 Do_HK
                      Do_Monitoring
                                   Current UniTime is = 100004

                                                              Next TC(19,1) Assi
gn TC(128,2) as action of Event 0x4002
                                      Send TM (1,1)
                                                   Send TM (1,7)
                                                                Next TC(19,4) En
able Action of Event 0x4002
                           Send TM (1,1)
                                        Send TM (1,7)
                                                     Do_HK
                                                          Do_Monitoring
                                                                       Current U
niTime is = 100005

                  Next TC(12,1) Enable Monitoring Param 0
                                                         Send TM (1,1)
                                                                      Send TM (1
,7)
```

Fig. 3.17: ICU ASW V2 UART output

3.4.8 Relevant changes made to the EDROOM generated files

The EDROOM library is divided into two layers:

- The bottom layer handles the basic functionality of the RTOS, that gives support to tasks, timers, semaphores and events. In this project it is implemented by endroombp.
- The top layer is independent of the platform and provides the interface for planning, communication, temporization, memory management and interruption

management that EDROOM components use. In this project it is implemented by endroomsl.

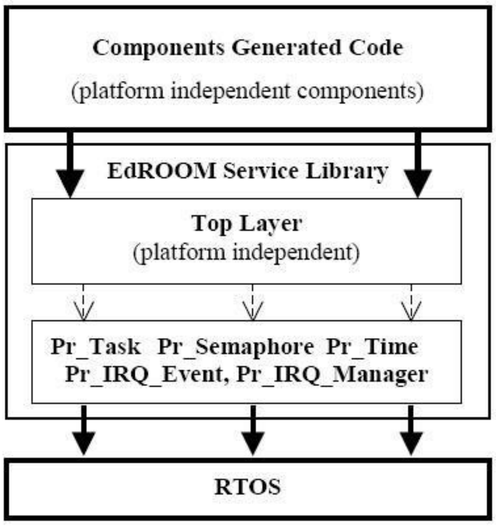

Components Generated Code
(platform independent components)
EdROOM Service Library
Top Layer
(platform independent)
Pr_Task Pr_Semaphore Pr_Time
Pr_IRQ_Event, Pr_IRQ_Manager
RTOS

Fig. 3.18: EDROOM library structure

Some changes had to be made to the bottom layer for the EDROOM library to work with this specific port. The most heavily modified file was edroombp_free_rtos.cpp. This file was very outdated as it used old FreeRTOS methods and data types that are no longer existent in the most recent version. One clear example is the change from the task handler type from xTaskHandle to TaskHandle_t:

```
//Changed from xTaskHandle to TaskHandle_t
lista_tarea & lista_tarea::add(TaskHandle_t handle,Pr_Task *pr_tarea)
```

Arguments from the task creation method also changed in newer FreeRTOS versions so this part in the Pr_Task constructor had to be modified as well:

```
 //Changed task creation function's arguments so they match new definition
 if (pdPASS == xTaskCreate((TaskFunction_t) _taskCode, (const char *) this->name,
         (uint16_t)_stackSize, NULL, (UBaseType_t)aux_priority, &taskID))
```

There were some type mismatches between the edroombp_iface header file definitions and the edroombp implementation which had to be corrected such as definitions using the uint32_t STD type and implementations using the unsigned integer type:

```
//Changed from unsigned int to uint32_t
Pr_Semaphore::Pr_Semaphore(uint32_t _value)
```

Some other changes were made in edroombp_iface, such as changing the outdated semaphore handle type xSemaphoreHandle to SemaphoreHandle_t:

```
 /** This attribute is the class mutex   */
 SemaphoreHandle_t xSemaphore; //Changed from xSemaphoreHandle to SemaphoreHandle_t
```

Like in edroombp, there were also some mismatches between definition and implementation which were changed to use the STD integer types:

```
/** Constructor for priority ceiling resource sempahore */
Pr_SemaphoreRec(const char *name, int32_t prioceiling); //Changed from int to int32_t
```

Port-specific Edroombp

There is another edroombp file specific for this port in a separate folder named vega_board. This file complements the other one by adding a function to install the IRQ handler for the LPUART0. It should be noted that this function can install any handler required for a specific use case. For example, a handler could also be installed for the I2C or SPI protocols if the application were to use those ports for communication with another device. Although some adaptions to the ASW program code should be made, in theory it would be entirely possible.

The LPUART0 handler function can also be found in this file. It was named LPUART0_DriverIRQHandler, the same name that the original one system has, so the RV32M1 system file could remain unchanged. This handler was declared as an extern "C" as it's a C function located in a C++ file. To choose between the custom edroombp handler and the system handler, a preprocessor macro was defined in the ASW configuration file.

There are also functions to uninstall handlers, create new interruption masks and enable or disable interruptions.

3.5 ICU ASW RISC-V (final version): Development and validation of the TM/TC serial port

The final phase of the project was to achieve simultaneous transmission of TCs and reception of TMs and passing a series of tests in the MASSIVA simulator, checking that the temporal requirements were being fulfilled.

Some changes were made to ICU ASW for the final testing with MASSIVA. Firstly, the SC Channel Driver was changed to accommodate to the RV32M1 Vega's LPUART functionality. A new function called Init_sc_channel() initializes the LPUART0 by setting the interruption priority of the associated interrupt number (17 in this case) to 5, setting the clock frequency, setting the configuration struct and initializing the port. CDTM lists were also adapted to send the desired telecommands. The program was changed so bitstreams would be sent instead of text and no telecommand reception would be simulated by the program itself. The incoming packet's processing logic was moved to the CDTC descriptor file. The interrupt handler was moved to edroombp along with the interruption installer method.

3.5.1 MASSIVA

For the development and validation of the TM/TC port, a simulator known as Monitoring and Analysis System for Software Inspection, Verification/Validation and Assessment (MASSIVA) was used for testing the correct functioning of the device. It is what is known as Ground Support Equipment or GSE, which are tools used by land crews to test and verify different parts of a system in development. This software was developed by the Universidad de Alcalá de Henares (UAH) for the testing of systems onboard spacecraft such as Solar Orbiter. The main objective of the tests is to receive a bitstream that our device should correctly interpret and subsequently produce the correct response. To achieve this, a virtual machine running Lubuntu was built and MASSIVA was installed in it.

The initial intention was to handle UART communication via the LPUART0, just like always until now, but it was initially believed that MASSIVA wasn't compatible with the ttyACM format so the program was reconfigured to work with LPUART1 which can be accessed by the Arduino-style pins which are recognizable by the simulator.

3.5.2 Functional tests adaption

Some changes had to be made to existing functional tests so they would work correctly with the VEGAboard. A small linux script was used to determine to what port the RV32M1 was connected to, in this case it was ttyACM0. A new MASSIVA workspace folder was created with the required configuration files so it would directly connect to the desired port. It was then believed that MASSIVA didn't support ttyACM type USB connections, only ttyUSB. This suggested that the LPUART0 was unusable by the simulator. An older version of MASSIVA, SRG GRS was tested on Windows to check if the output would be received there, but it wasn't the case.

A new approach was taken: to reroute the transmissions to LPUART1 and connect it via a UART to USB adapter. The adapter we had available was the StarTech RSR232. This required changing the pin multiplexing options so the pins D0 and D1 would work as the LPUART1 RX and TX respectively. To do this, the pin_mux.c file, which handles the pin configuration and multiplexing, was changed to reflect these changes.

3.5.3 Transmission

Some of the official testing programs were used to check on the correct functioning of the UART. The characters became corrupted upon transmission and were illegible. It was believed that it could be an endianness or baud rate issue, but after hours of testing it was discovered that it was the adapter itself which was incompatible. A new adapter was used, the PL2303. This adapter provided a clean transmission but only using the non-interrupt transmission method via polling with the LPUART1. The interruption method didn't work as it did in the LPUART0.

It was later discovered that there was an error in the transmission of the length field of the packet and was causing MASSIVA to not recognize packets. It was an endianness issue and was fixed by changing the packet transmission from big to little endian by parameterizing the bytes sent via a function. After that, MASSIVA correctly identified the packets, but it wasn't processing them correctly. This was caused by an issue in the MASSIVA config files, not in the transmission itself. It was quickly identified and corrected. With the transmission fixed, Massiva also recognized packets coming from the LPAURT0, via the ttyACM0 port, which was previously believed to be incompatible with the simulator. SRG GSS also detected packets in Windows. This was a great relief as the VEGAboard's USB port could be used again for communication and it supported UART interrupt transfer and reception.

In the final ASW RISC-V project, Init_sc_channel() was changed to initialize both the LPUART0 and LPUART1, so any of them could be used depending on the system to which the Board is connected to and its physical port compatibility. The initialization process was changed from the one typically used in FreeRTOS to a low-level one that interacted directly with the board instead of using the OS. MASSIVA was configured to connect to the LPUART0 via the ttyACM0 port. It correctly received the packets sent with the LPUART_WriteBlocking function and processed them correctly, incrementing the "packets received" counter along with the SID0 and SID1 counters. As you can see in the figure, next to the SID0 counter, there is a field indicating that the packet period is 2.007s. This is very close to the 2s hard temporal requirement, so we can confirm that the temporal restrictions are being fulfilled.

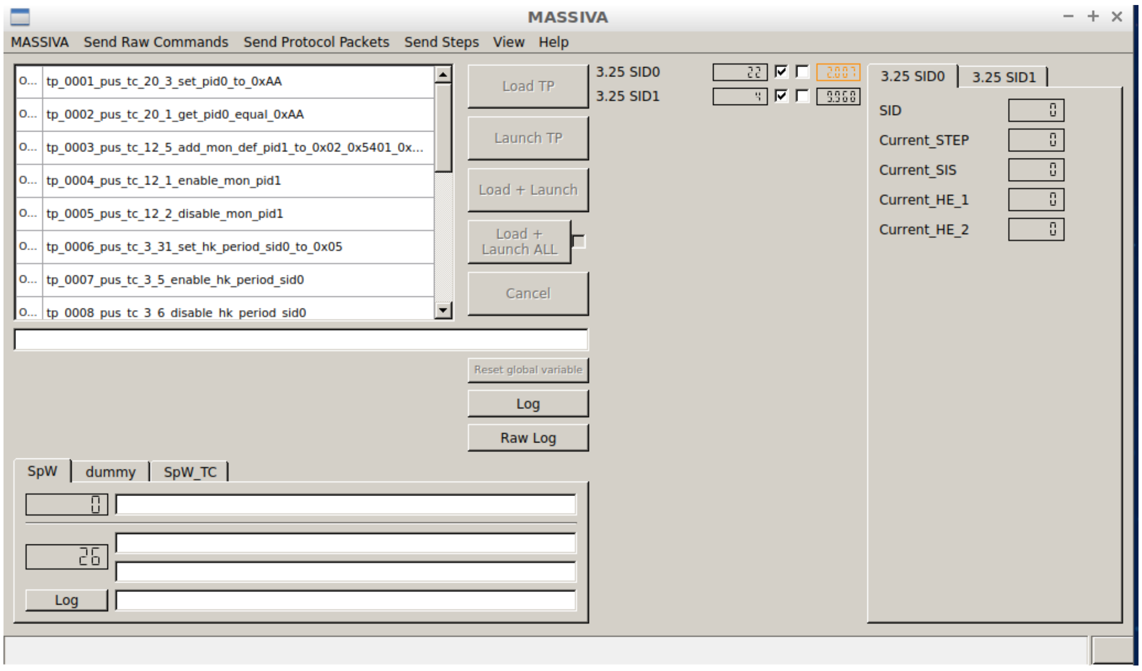
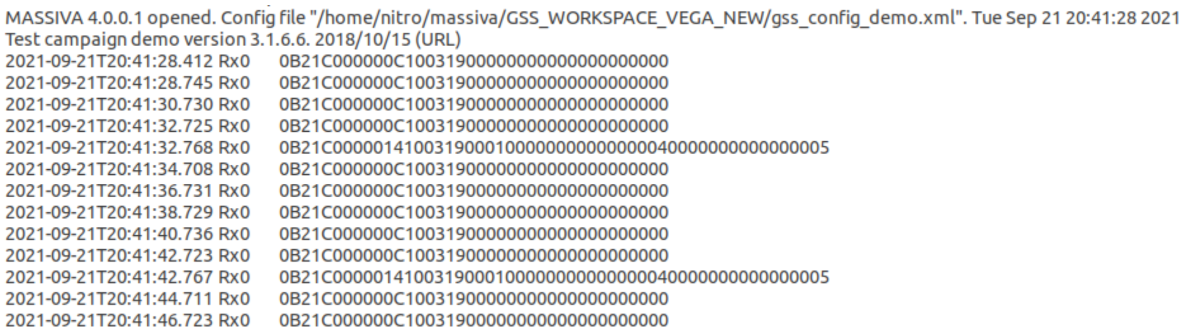

MASSIVA
MASSIVA Send Raw Commands Send Protocol Packets Send Steps View Help
tp_0001_pus_tc_20_3_set_pid0_to_0xAA
tp_0002_pus_tc_20_1_get_pid0_equal_0xAA
tp_0003_pus_tc_12_5_add_mon_def_pid1_to_0x02_0x5401_0x...
tp_0004_pus_tc_12_1_enable_mon_pid1
tp_0005_pus_tc_12_2_disable_mon_pid1
tp_0006_pus_tc_3_31_set_hk_period_sid0_to_0x05
tp_0007_pus_tc_3_5_enable_hk_period_sid0
tp_0008_pus_tc_3_6_disable_hk_period_sid0
Load TP
Launch TP
Load + Launch
Load + Launch ALL
Cancel
Reset global variable
Log
Raw Log
3.25 SID0
3.25 SID1
3.25 SID0
3.25 SID1
SID
Current_STEP
Current_SIS
Current_HE_1
Current_HE_2
SpW
dummy
SpW_TC
Log

Fig. 3.19: MASSIVA receiving packets via the LPUART0



```
MASSIVA 4.0.0.1 opened. Config file "/home/nitro/massiva/GSS_WORKSPACE_VEGA_NEW/gss_config_demo.xml". Tue Sep 21 20:41:28 2021
Test campaign demo version 3.1.6.6. 2018/10/15 (URL)
2021-09-21T20:41:28.412 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:28.745 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:30.730 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:32.725 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:32.768 Rx0     0B21C000001410031900010000000000000400000000000000005
2021-09-21T20:41:34.708 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:36.731 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:38.729 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:40.736 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:42.723 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:42.767 Rx0     0B21C000001410031900010000000000000400000000000000005
2021-09-21T20:41:44.711 Rx0     0B21C000000C100319000000000000000000
2021-09-21T20:41:46.723 Rx0     0B21C000000C100319000000000000000000
```

Fig. 3.20: MASSIVA raw log

Packet Structure:

Packets sent by the ASW ICU have a very specific structure that must be consistent at all times in order for the receptor system to process them correctly.

The structure of the PUS Service 03 packets is the following:

header	Tm length	Tm packID	Tm seqCtrl	packHeader length	PUS Version ACK	Tm service	Tm subservice	dummy	rest
0xBEBABEEF	0x0013	0x0B21	0xC000	0x000C	0x10	0x03	0x19	0x00	0x0

Table 3: ICU ASW PUS Service 03 Telemetry Structure

- The first 4 bytes are the header 0xBEBABEEF which identifies an ASW ICU packet.
- The next 2 bytes are the telemetry length 0x0013, 19 in decimal.
- The next 2 bytes are the telemetry packID which is always 0x0B21.

- The next 2 bytes are the telemetry seqCtrl which is always 0xC000.
- Los next 2 bytes are the packHeader length 0x000C, 12 in decimal.
- The next byte is PUS Version Ack 0x10.
- The next byte is the tm service 0x03.
- The next byte is the tm subservice 0x19.
- The last byte is a dummy with value 0x00.
- The next 9 bytes are all 0x0 for this specific telemetry.

The telemetry configuration can be modified in the PUS Service 3 file.

For the second telemetry, it has the following structure:

Enabled	TM ID	Period (s)	Dummy 0	N parameters	Parameters ID
true	0	2	0	1	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Table 4: ICU ASW PUS Service 03 Telemetry Configuration Example

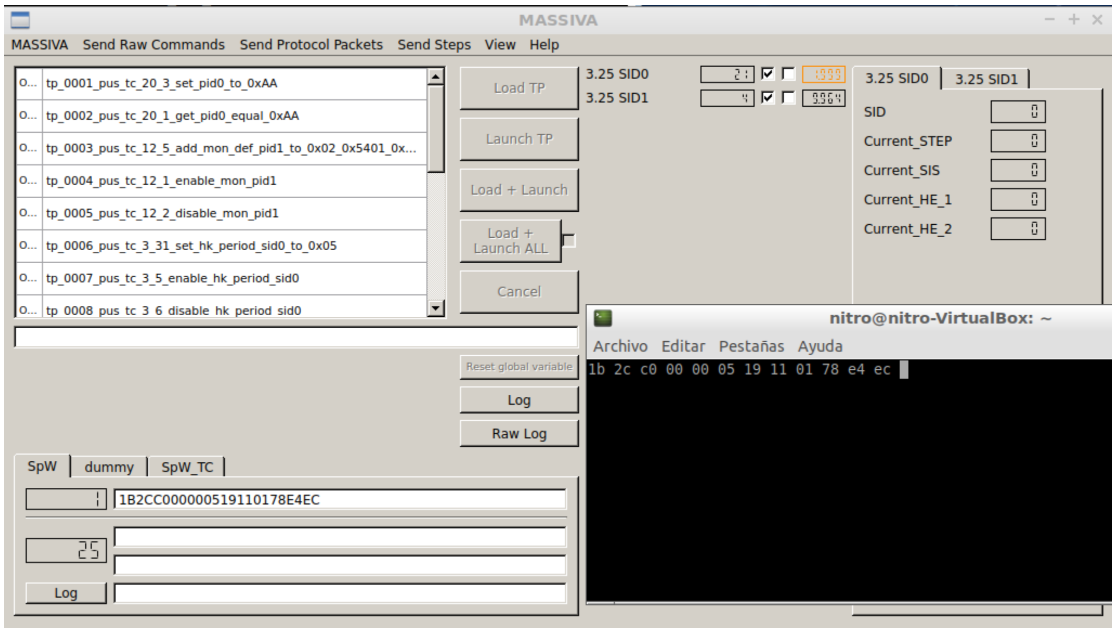
3.5.4 Reception

Reception was trickier to achieve that transmission. A variety of methods were tried in the final ASW RISC-V project until a successful one was found.

The first method tried was to use LPUART_TransferReceiveNonBlocking in the SendTM function and installing a handler like it is done in the official examples. This method wasn't successful as transmission couldn't complete in time and the resulting received packet was incomplete. The interruption handler is only triggered when the receiver buffer is full, so a 1-byte buffer had to be used, as we needed to process bytes one by one. The next method involved using LPUART_ReadBlocking with no handler at all, but again packets arrived incomplete and couldn't be processed. These first methods couldn't determine beforehand the packet's length so a bigger array had to be used for storage.

A new approach was taken to determine the packet's length after receiving the header and length fields which are always the first 6 bytes and, after that, processing and saving the packet. The original freeRTOS interrupt handler for the LPUART0 was used for this purpose. The problem was that it was hard to control the program flow by using this method. It involved initializing the LPAURT0 via the FreeRTOS LPUART_RTOS_Init function, which abstracted the program more. By combining this initialization process with a conditional that only triggered if a non-null byte was received, we could correctly trigger the handler and send back the received bytes without halting the transmission of packets.

To simplify this process, the FreeRTOS initialization method was discarded, and the low-level one was used again. A new custom handler was used, but this time going into the RV32M1 system files and changing the original LPUART0 handler function declaration to a new custom one. This allowed us to change this system handler to one we declared ourselves in the SC Channel Driver file. In this handler we could send back the received bytes without manually using a reception method and it would trigger automatically when a RX interrupt is detected. We later renamed the handler to have the same name as the system one, so the internal files wouldn't have to be modified and we could switch between the two via a macro in the ASW configuration file. The initialization process of the UART0 was changed so interrupts would only be triggered upon reception and not upon transmission. After dealing with some endiannes issues in the packet processing method of the custom handler, packets were received, processed and sent back correctly.



O... tp_0001_pus_tc_20_3_set_pid0_to_0xAA
O... tp_0002_pus_tc_20_1_get_pid0_equal_0xAA
O... tp_0003_pus_tc_12_5_add_mon_def_pid1_to_0x02_0x5401_0x...
O... tp_0004_pus_tc_12_1_enable_mon_pid1
O... tp_0005_pus_tc_12_2_disable_mon_pid1
O... tp_0006_pus_tc_3_31_set_hk_period_sid0_to_0x05
O... tp_0007_pus_tc_3_5_enable_hk_period_sid0
O... tp_0008_pus_tc_3_6_disable_hk_period_sid0

1B2CC000000519110178E4EC

```
1b 2c c0 00 00 05 19 11 01 78 e4 ec
```

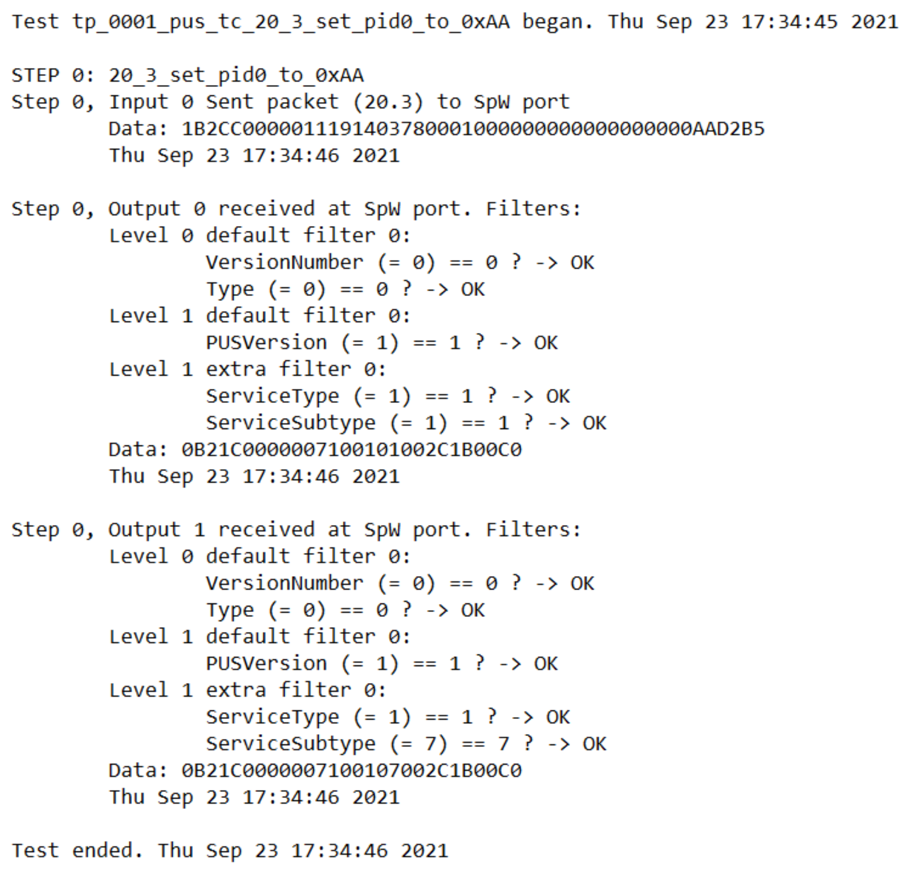
Fig. 3.21: Reception of the 17.1 protocol packet via the LPUART1

3.5.5 Port Validation

To check that the port behaves correctly and that it could hypothetically integrate with the other EPD components and sensors, a series of functional tests had to be successfully completed. After achieving simultaneous transmission and reception, the groundwork for the ICU ASW RISC-V project to support these tests was laid.

As we could see before, the temporal requirements were being respected so the port was working according to specifications. The final step to determine the success of the port was to launch and pass the tests. After launching all of them one after the other, they completed successfully.

One example is illustrated here with the tp_0001_pus_tc_20_3_set_pid0_to_0xAA test, which consists of modifying the value of the number 0 parameter of the ICU System Data Pool, so it takes the value 0xAA using the 20.3 subservice.



```
Test tp_0001_pus_tc_20_3_set_pid0_to_0xAA began. Thu Sep 23 17:34:45 2021

STEP 0: 20_3_set_pid0_to_0xAA
Step 0, Input 0 Sent packet (20.3) to SpW port
        Data: 1B2CC000001119140378000100000000000000000000AAD2B5
        Thu Sep 23 17:34:46 2021

Step 0, Output 0 received at SpW port. Filters:
        Level 0 default filter 0:
                VersionNumber (= 0) == 0 ? -> OK
                Type (= 0) == 0 ? -> OK
        Level 1 default filter 0:
                PUSVersion (= 1) == 1 ? -> OK
        Level 1 extra filter 0:
                ServiceType (= 1) == 1 ? -> OK
                ServiceSubtype (= 1) == 1 ? -> OK
        Data: 0B21C0000007100101002C1B00C0
        Thu Sep 23 17:34:46 2021

Step 0, Output 1 received at SpW port. Filters:
        Level 0 default filter 0:
                VersionNumber (= 0) == 0 ? -> OK
                Type (= 0) == 0 ? -> OK
        Level 1 default filter 0:
                PUSVersion (= 1) == 1 ? -> OK
        Level 1 extra filter 0:
                ServiceType (= 1) == 1 ? -> OK
                ServiceSubtype (= 7) == 7 ? -> OK
        Data: 0B21C0000007100107002C1B00C0
        Thu Sep 23 17:34:46 2021

Test ended. Thu Sep 23 17:34:46 2021
```

Fig. 3.22: p_0001_pus_tc_20_3_set_pid0_to_0xAA testing procedure

Chapter 4 - User guide

This comprehensive guide will explain how to run or debug in Windows any of the projects addressed in this work.

1. Follow the Detailed Windows installation guide, which you can find above in **3.2**
2. Open Eclipse CDT and go to File → Open Projects from File System… Select the import source and click on the Finish button.
3. A safe practice is to click on Project → Clean… after importing a project and rebuilding the project after that. This will regenerate .o object files and the .ELF executable.
4. Connect your Segger J-Link debugger probe to the OpenISA VEGAboard with a 9-Pin Cortex-M Adapter like how it is shown in the image. After that, connect your micro-USB cable to the J55 connector of the VEGAboard and plug in both the board and the debugger to your PC.

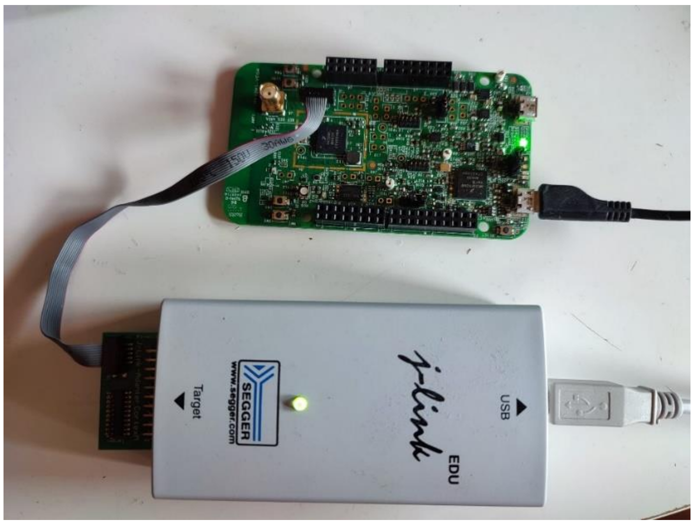
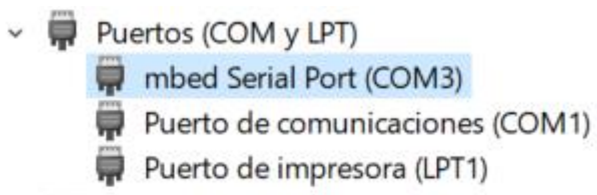


Fig. 4.1: VEGAboard connections

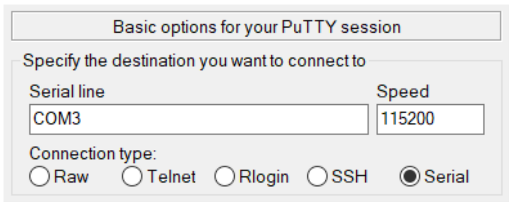
5. Go to the Windows Device Manager and click on the Ports tab to figure out to which port is the Debugger connected. The correct port is shown as “mbed Serial Port”.



Puertos (COM y LPT)
mbed Serial Port (COM3)
Puerto de comunicaciones (COM1)
Puerto de impresora (LPT1)

Fig. 4.2: Windows device manager

6. Open your preferred serial monitor, in my case I’m using Putty, which can be downloaded at https://www.putty.org/. Configure a new serial connection by selecting the “serial” checkbox, setting a speed of 115200 bauds and selecting the appropriate serial port. A new window will open if the connection was successful.


Basic options for your PuTTY session

Specify the destination you want to connect to

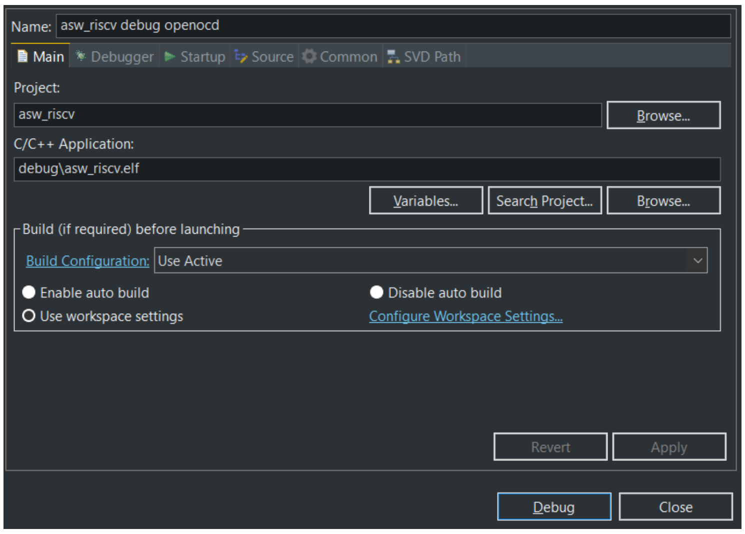
Serial line: COM3

Speed: 115200

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Fig. 4.3: Putty serial connection

7. Back in Eclipse, go to Run → Debug configurations and either run an existing Debug Configuration or create a new one. Fill in the fields as required in accordance with the project. Click on the "Apply" button after finishing your configuration and then on the "Debug" button. It should look similar to this:


Name: asw_riscv debug openocd

Main | Debugger | Startup | Source | Common | SVD Path

Project: asw_riscv

C/C++ Application: debug\asw_riscv.elf

Build (if required) before launching

Build Configuration: Use Active

Enable auto build

Disable auto build

Use workspace settings

Configure Workspace Settings...

Fig. 4.4: Debug configuration

8. Once the debugging process starts, control the execution process freely by using the Resume, Step Into, Step Over or Step Back buttons in Eclipse (the keyboard shortcuts are F8, F5, F6, F7 respectively). You can also restart or terminate the process. breakpoints can be set and viewed in the Breakpoints window. Alternatively, if you only need to flash and run the program, go to Run → Run Configurations and select and your desired configuration.
9. If the program requires input via the LPUART, you can send characters or commands via the Putty serial monitor, and also check the program output.

(Optional) Using another UART for end-to-end communication:

1. If you wish to use another UART for transmission or reception, make sure that the pin_mux.c file of the project has the pertinent changes, so the board is configured with the appropriate pin logic.
2. Connect your PL2303 or similar serial-to-USB adapter to the VEGAboard by connecting the white wire (RX) to the TX pin and the green (TX) wire to the RX pin. Refer to **annex 2** and to the OpenISA official Board User Guide for more info.

(Optional) Using another protocol for end-to-end communication:

1. If you wish to use another UART or an entirely different communication protocol, make the pertinent changes to the pin_mux.c file of the project so the board is configured with the appropriate pin logic.
2. Program and install the correct interrupt handler in edroombp_free_rtos.c. Change the SendTM and Init_sc_channel functions in emu_sc_channel_drv.c to initialize the new port and transmit with it.
3. Refer to the official FreeRTOS and OpenISA to check on the available communication protocols and their behaviour.

For info on how to use the SRG GSS or MASSIVA simulators, refer to their respective official guides.

Chapter 5 - Conclusions

We can conclude that the project was a success as all of its main objectives were fulfilled. We successfully developed a port of the ICU ASW on the RV32M1 VEGAboard that can transmit and receive packets simultaneously and pass all the required functional tests as well as the original version. The project paves the way for future extensions and enhancements to the RISC-V port of the ICU ASW.

Porting software to new platforms can be a daunting task. Not only because of the software changes which have to be made, but because of the physical differences and limitations of the new hardware to which it is being ported. It also has the added difficulty of working with software made by someone else that you may not know well. This is why a good documentation is an essential part of any project. Although the RV32M1 VEGAboard is an excellent testing board, bigger programs like the ICU ASW are very hard to run on it because of memory space limitations. Fortunately, with enough time and effort, these problems can be solved by saving memory space in some non-critical areas. Debugging the program after the kernel starts is also very difficult as no program-state information is provided. The initial incompatibilities between MASSIVA and our hardware and software also posed additional problems, but fortunately, they could be solved, and it proved that the testing software can be adapted to other kinds of boards and architectures.

This project provided a challenge in the area of computer engineering, as it required working at a very low level by checking memory addresses, hardware configurations and preprocessor macros. This field of computer science is challenging, but it is nevertheless very interesting as it is what we can always find in the entrails of every computer and mobile application.

Low-level applications, such as this one, require a clear understanding of how memory allocation works, where each type of variable and component is stored in the memory, knowledge of the different hardware resources available and how to manage them. As a student of computer engineering, I never really experienced these kinds of challenges before, as most subjects cover these areas very lightly and don’t face you with real problems that you would find in the day-to-day work of computer engineering.

Space exploration is one of the big topics of our era. With private space-related companies on the rise, having a clear understanding of what is around us is critical to carry on successful space missions and truly elevating humankind to its next stage. Solar Orbiter and the European Space Agency are precisely helping us clear the fog of uncertainty that covers space.

Lastly, understanding the solar climate could provide more information about climate change that is currently unavailable. We can’t be certain of how much solar flares and coronal mass ejections affect the climate here on Earth and if they could be partly responsible for the current rising temperatures in our planet.

All in all, the project has been a real challenge to me as a computer engineer and will likely affect how I deal with new problems in the future, even in other fields of information technologies.

Glossary

API: Application Program Interface
ASIC: Application Specific Integrated Circuit
ASW: Application Software
BLE: Bluetooth Low Energy
BSP: Board Support Packages
CCSDS: Consultative Committee for Space Data Systems
CDPU: Central Data Processing Unit
CME: Coronal Mass Ejection
EPD: Energetic Particle Detector
ESA: European Space Agency
ESTEC: European Space Research and Technology Centre
FDIR: Fault, Detection, Isolation and Recovery
FIRC: Fast Internal Reference Clock
FPGA: Field Programmable Gate Array
ICU: Instrument Control Unit
IDE: Integrated Development Environment
ISA: Instruction Set Architecture
LCL: Latch Current Limiter
LED: Light Emitting Diode
LVDS: Low Voltage Differential Signaling
LVPS: Low Voltage Power Supply
MASSIVA: Monitoring and Analysis System for Software Inspection, Verification / Validation and Assessment
MCU: Memory Control Unit
NASA: National Space Agency
OCD: On Chip Debugger
OS: Operating System
PUS: Packet Utilization Standard
RISC: Reduced Instruction Set Computer
RTEMS: Real Time Executive for Multiprocessor Systems
RTOS: Real Time Operating System
SEP: Solar Energetic Particle
SIRC: Slow Internal Reference Clock
SoC: System on Chip
SPARC: Scalable Processor Architecture
STD: Standard
UART: Universal Asynchronous Receiver Transmitter

Bibliography

[1] [Online] Yihua Zheng and Rebekah M. Evans - Solar Energetic Particles (SEPs)
URL:
https://ccmc.gsfc.nasa.gov/RoR_WWW/SWREDI/2014/SEP_YZheng_20140602.pdf
(Accessed: June 2021)

[2] [Online] Stack Overflow Forum (Various answers)
URL: https://stackoverflow.com/
(Accessed: June 2021)

[3] [Online] European Space Agency
URL: https://www.esa.int
(Accessed: June 2021)

[4] [Online] (Various authors) - The Energetic Particle Detector, Energetic particle instrument suite for the Solar Orbiter mission
URL: https://www.aanda.org/articles/aa/pdf/2020/10/aa35287-19.pdf
(Accessed: June 2021)

[5] [Online] (Various authors) - The Energetic Particle Detector Suite for Solar Orbiter
URL: https://www.iac.es/es/ciencia-y-tecnologia/publicaciones/energetic-particle-detector-suite-solar-orbiter
(Accessed: June 2021)

[6] [Online] Pablo Parra Espada - The Instrument Control Unit (ICU)
URL: http://espada.uah.es/epd/ICU.php
(Accessed: June 2021)

[7] [Online] Mahmoud H. Qutqut, Aya Al-Sakran, Fadi Almasalha, Hossam S. Hassanein – Comprehensive survey of the IoT open-source OSs
URL:
https://www.researchgate.net/publication/328031606_Comprehensive_Survey_of_the_IoT_Open_Source_OSs
(Accessed: August 2021)

[8] [Online] Sam Leonard – A dive into RI5CY core internals
URL: https://www.embecosm.com/2019/08/13/a-dive-into-ri5cy-core-internals/
(Accessed: August 2021)

[9] [Online] OpenISA Official webpage and official documentation
URL: https://open-isa.org/
(Accessed: July 2021)

[10] [Online] RTEMS Official webpage
URL: https://www.rtems.org/
(Accessed: July 2021)

[11] [Online] FreeRTOS Official webpage
URL: https://www.freertos.org/
(Accessed: July 2021)

[12] [Online] RISC-V Official webpage
URL: https://riscv.org/
(Accessed: July 2021)

[13] [Online] Pulp Platform Official webpage
URL: https://pulp-platform.org/
(Accessed: July 2021)

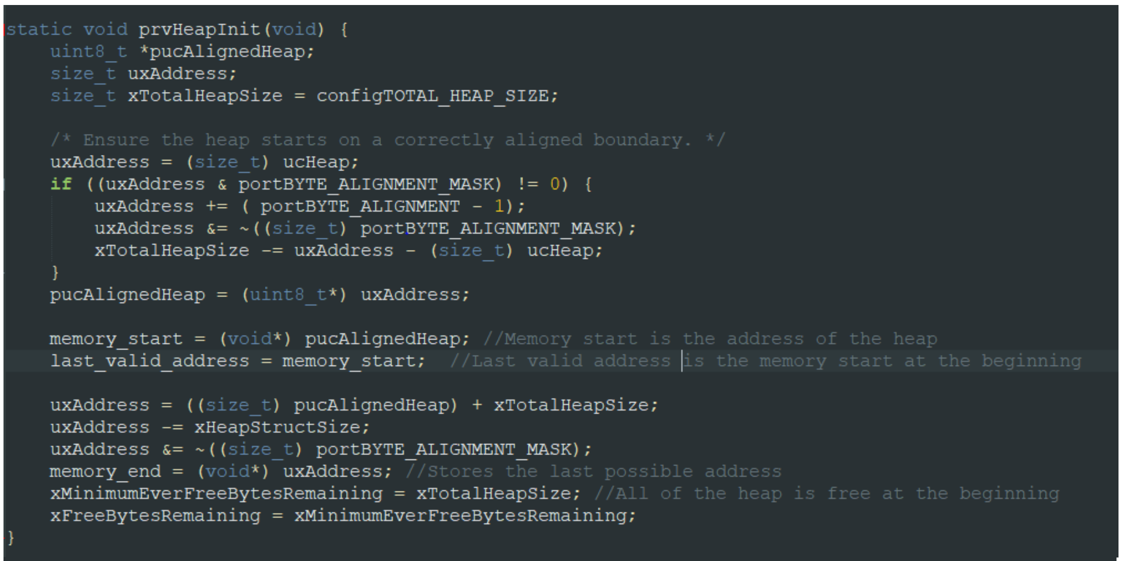
[14] [Online] Masahiro Hiramori – Simple malloc implementation
URL: https://gist.github.com/mshr-h/9636fa0adcf834103b1b
(Accessed: August 2021)

[15] [Online] Nthatisi Hlapisi - Understanding RISC-V Architecture and Why it could be a Replacement for ARM
URL: https://circuitdigest.com/article/understanding-risc-v-architecture-and-why-it-could-be-a-replacement-for-arm
(Accessed: July 2021)

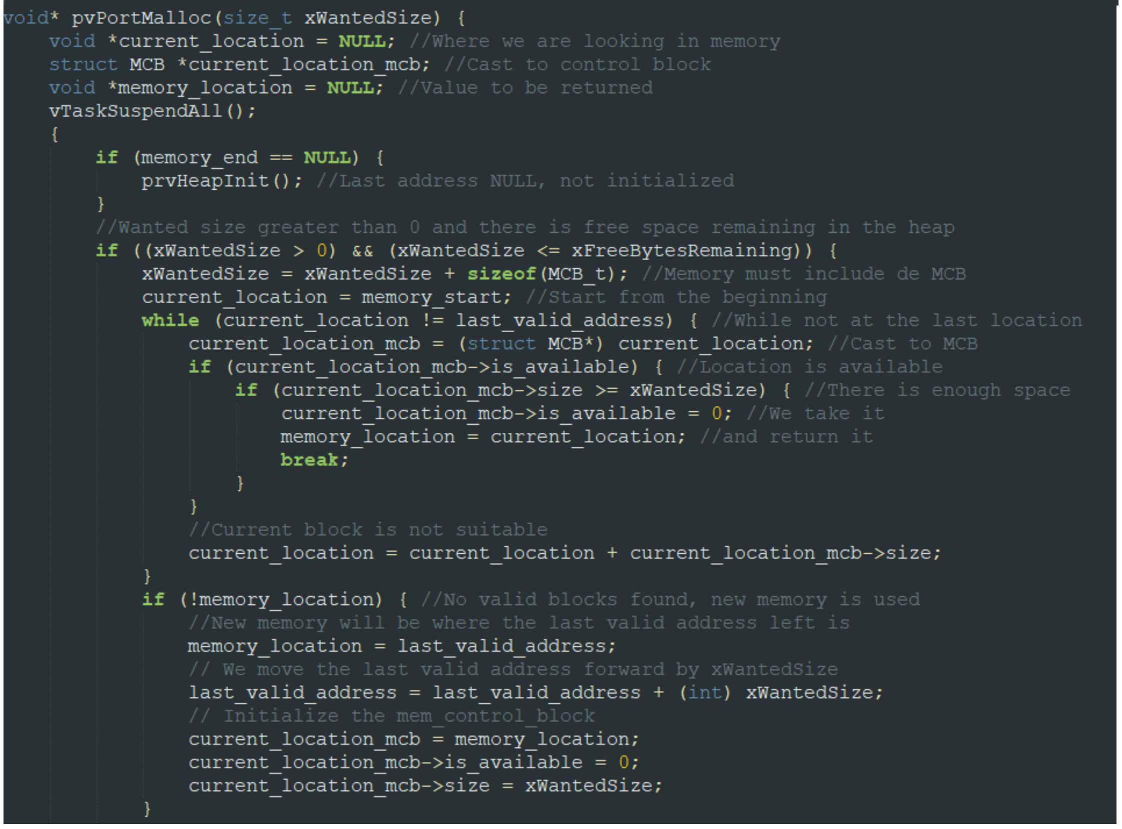
[16] [Online] Zephyr Project Documentation – OpenISA VEGAboard
URL: https://docs.zephyrproject.org/latest/boards/riscv/rv32m1_vega/doc/index.html
(Accessed: July 2021)

Annexes

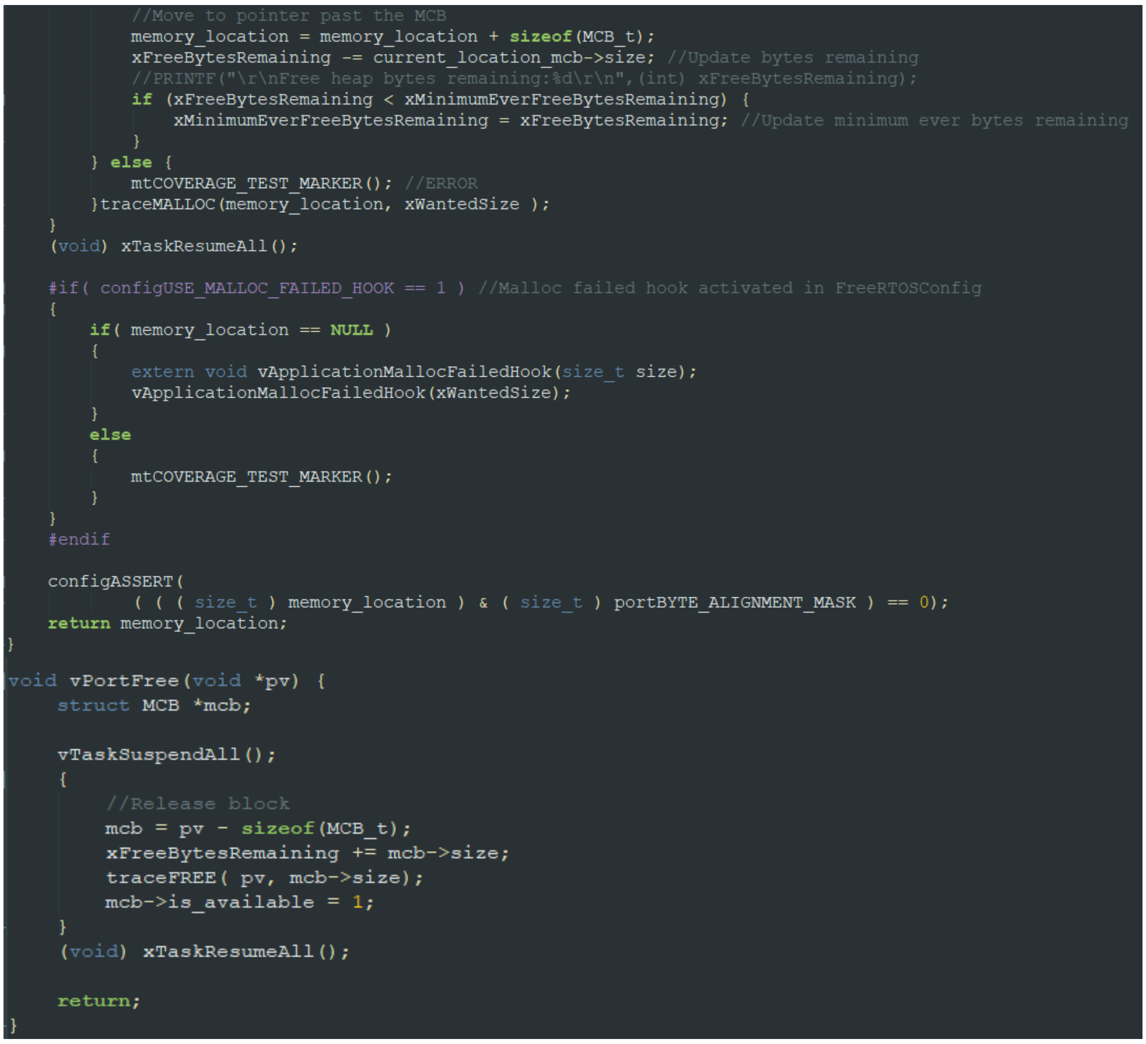
Annex 1: heap.c main functions



```c
static void prvHeapInit(void) {
    uint8_t *pucAlignedHeap;
    size_t uxAddress;
    size_t xTotalHeapSize = configTOTAL_HEAP_SIZE;

    /* Ensure the heap starts on a correctly aligned boundary. */
    uxAddress = (size_t) ucHeap;
    if ((uxAddress & portBYTE_ALIGNMENT_MASK) != 0) {
        uxAddress += ( portBYTE_ALIGNMENT - 1);
        uxAddress &= ~((size_t) portBYTE_ALIGNMENT_MASK);
        xTotalHeapSize -= uxAddress - (size_t) ucHeap;
    }
    pucAlignedHeap = (uint8_t*) uxAddress;

    memory_start = (void*) pucAlignedHeap; //Memory start is the address of the heap
    last_valid_address = memory_start;  //Last valid address is the memory start at the beginning

    uxAddress = ((size_t) pucAlignedHeap) + xTotalHeapSize;
    uxAddress -= xHeapStructSize;
    uxAddress &= ~((size_t) portBYTE_ALIGNMENT_MASK);
    memory_end = (void*) uxAddress; //Stores the last possible address
    xMinimumEverFreeBytesRemaining = xTotalHeapSize; //All of the heap is free at the beginning
    xFreeBytesRemaining = xMinimumEverFreeBytesRemaining;
}
```

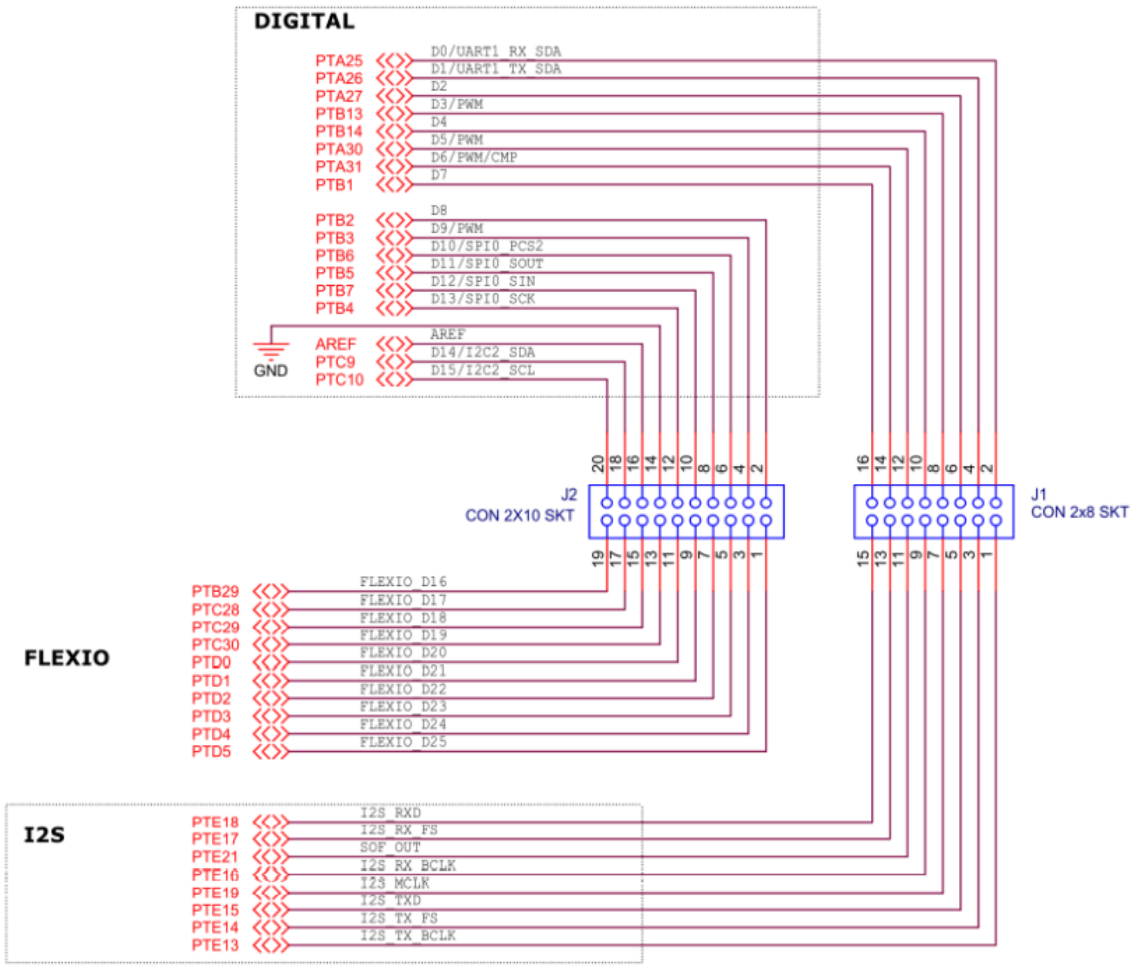


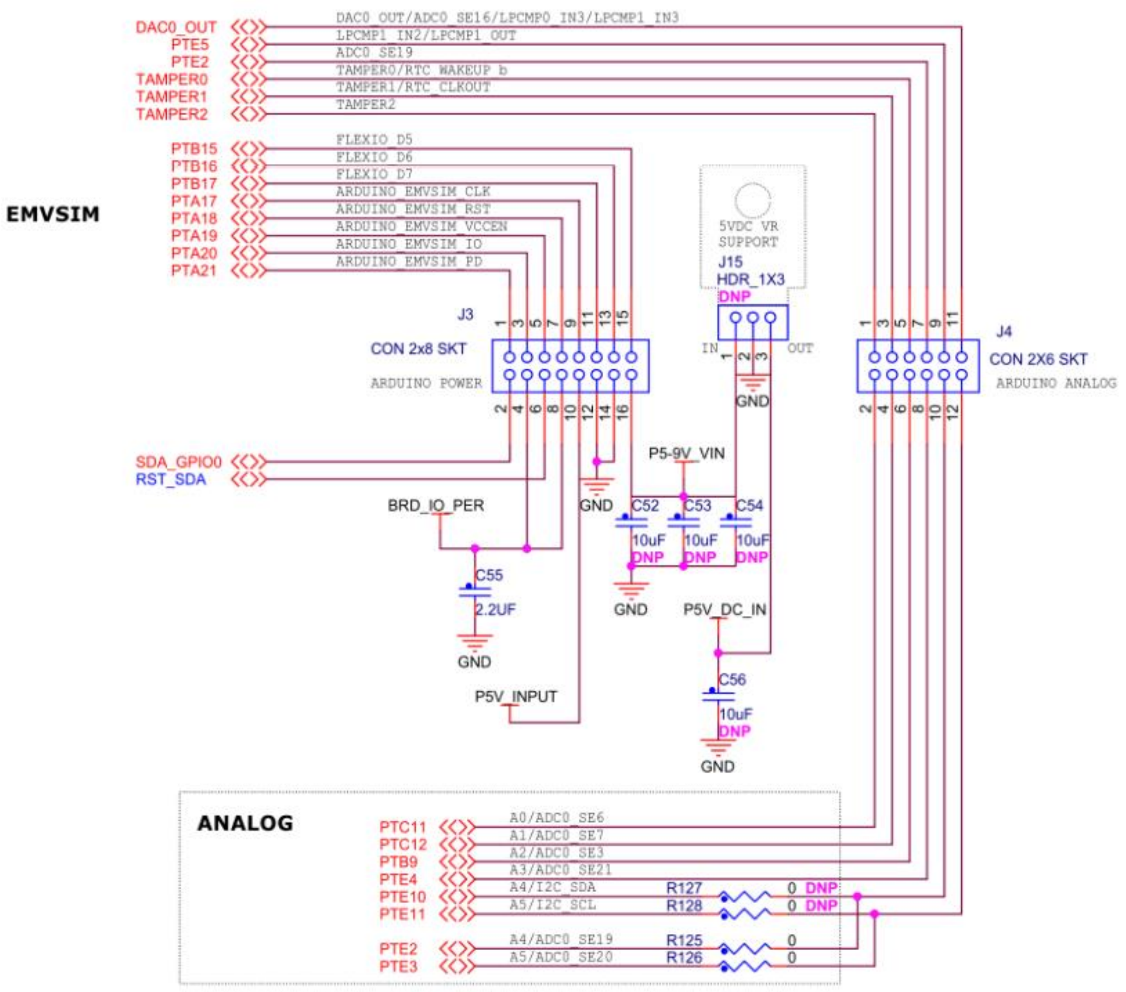
```c
void* pvPortMalloc(size_t xWantedSize) {
    void *current_location = NULL; //Where we are looking in memory
    struct MCB *current_location_mcb; //Cast to control block
    void *memory_location = NULL; //Value to be returned
    vTaskSuspendAll();
    {
        if (memory_end == NULL) {
            prvHeapInit(); //Last address NULL, not initialized
        }
        //Wanted size greater than 0 and there is free space remaining in the heap
        if ((xWantedSize > 0) && (xWantedSize <= xFreeBytesRemaining)) {
            xWantedSize = xWantedSize + sizeof(MCB_t); //Memory must include de MCB
            current_location = memory_start; //Start from the beginning
            while (current_location != last_valid_address) { //While not at the last location
                current_location_mcb = (struct MCB*) current_location; //Cast to MCB
                if (current_location_mcb->is_available) { //Location is available
                    if (current_location_mcb->size >= xWantedSize) { //There is enough space
                        current_location_mcb->is_available = 0; //We take it
                        memory_location = current_location; //and return it
                        break;
                    }
                }
                //Current block is not suitable
                current_location = current_location + current_location_mcb->size;
            }
            if (!memory_location) { //No valid blocks found, new memory is used
                //New memory will be where the last valid address left is
                memory_location = last_valid_address;
                // We move the last valid address forward by xWantedSize
                last_valid_address = last_valid_address + (int) xWantedSize;
                // Initialize the mem control block
                current_location_mcb = memory_location;
                current_location_mcb->is_available = 0;
                current_location_mcb->size = xWantedSize;
            }
```



```c
            //Move to pointer past the MCB
            memory_location = memory_location + sizeof(MCB_t);
            xFreeBytesRemaining -= current_location_mcb->size; //Update bytes remaining
            //PRINTF("\r\nFree heap bytes remaining:%d\r\n",(int) xFreeBytesRemaining);
            if (xFreeBytesRemaining < xMinimumEverFreeBytesRemaining) {
                xMinimumEverFreeBytesRemaining = xFreeBytesRemaining; //Update minimum ever bytes remaining
            }
        } else {
            mtCOVERAGE_TEST_MARKER(); //ERROR
        }traceMALLOC(memory_location, xWantedSize );
    }
    (void) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 ) //Malloc failed hook activated in FreeRTOSConfig
    {
        if( memory_location == NULL )
        {
            extern void vApplicationMallocFailedHook(size_t size);
            vApplicationMallocFailedHook(xWantedSize);
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif

    configASSERT(
            ( ( ( size_t ) memory_location ) & ( size_t ) portBYTE_ALIGNMENT_MASK ) == 0);
    return memory_location;
}

void vPortFree(void *pv) {
    struct MCB *mcb;

    vTaskSuspendAll();
    {
        //Release block
        mcb = pv - sizeof(MCB_t);
        xFreeBytesRemaining += mcb->size;
        traceFREE( pv, mcb->size);
        mcb->is_available = 1;
    }
    (void) xTaskResumeAll();

    return;
}
```

Annex 2: RV32M1-VEGA I/O header pinout




EMVSIM
DAC0_OUT
PTE5
PTE2
TAMPER0
TAMPER1
TAMPER2
DAC0_OUT/ADC0_SE16/LPCMP0_IN3/LPCMP1_IN3
LPCMP1_IN2/LPCMP1_OUT
ADC0_SE19
TAMPER0/RTC_WAKEUP_b
TAMPER1/RTC_CLKOUT
TAMPER2
PTB15
PTB16
PTB17
PTA17
PTA18
PTA19
PTA20
PTA21
FLEXIO_D5
FLEXIO_D6
FLEXIO_D7
ARDUINO_EMVSIM_CLK
ARDUINO_EMVSIM_RST
ARDUINO_EMVSIM_VCCEN
ARDUINO_EMVSIM_IO
ARDUINO_EMVSIM_PD
J3
CON 2x8 SKT
ARDUINO POWER
5VDC VR
SUPPORT
J15
HDR_1X3
DNP
IN
OUT
GND
J4
CON 2X6 SKT
ARDUINO ANALOG
SDA_GPIO0
RST_SDA
GND
P5-9V_VIN
BRD_IO_PER
C52
10uF
DNP
C53
10uF
DNP
C54
10uF
DNP
C55
2.2UF
GND
GND
P5V_DC_IN
C56
10uF
DNP
GND
P5V_INPUT
ANALOG
PTC11
PTC12
PTB9
PTE4
PTE10
PTE11
A0/ADC0_SE6
A1/ADC0_SE7
A2/ADC0_SE3
A3/ADC0_SE21
A4/I2C_SDA
A5/I2C_SCL
R127
0
DNP
R128
0
DNP
PTE2
PTE3
A4/ADC0_SE19
A5/ADC0_SE20
R125
0
R126
0