E.ON Energy Research Centre

TECHNICAL REPORT

August 2022

Embedded Rust on ZCU106 Evaluation Board

PREPARED BY
Venkatakrishnan Tirupattur Ramakrishnan
Bharath Kartha A.
Ramees Zainudheen Puzakkal

CONTENTS

1. INTRODUCTION 3
2. BACKGROUND 3
    2.1 Rust Programming Language 3
    2.2 ZCU106 Evaluation Board 4
        2.2.1 Real-time Processing Unit 6
        2.2.2 JTAG and Raspberry-Pi Interface 6
    2.3 Bare-metal programming in Rust 6
3. METHODOLOGY 8
    3.1 IDE and Toolchains 8
        3.1.1 Vitis 8
        3.1.2 Rust development in Vitis 9
    3.2 Calling RUST code from C program 10
    3.3 Calling C code from RUST program 11
    3.4 Interfacing with custom FPGA Peripheral 12
    3.5 Scheduler 12
    3.6 Experimental Work 14
        3.6.1 Print function with Rust 14
        3.6.2 Print function using impl Write 15
4. RESULT AND DISCUSSION 17
5. REFERENCES 18

1. INTRODUCTION

Rust is a promising programming language that has vast potential in the embedded system domain. In this project, the use of Embedded Rust on ZCU106 Evaluation Board is studied. The key objectives of the project are to understand Rust as a programming language, implement the toolchain for cross-compilation, and run Rust code on the embedded device. A sample C project in the same environment is studied thoroughly to get acquainted with software architecture and various interfaces. Rust code is introduced in stages to the C project to finally have the main function running from the Rust side. Additional investigations like implementing a scheduler, writing a print function purely in Rust, and interfacing CAN port are also studied.

2. BACKGROUND

2.1 Rust Programming Language

Rust is a language that was created by researchers from Mozilla, primarily while they were working on Servo experimental browser engine and the Rust Compiler. It has been noted as a language with one of the most growing user bases in the past few years thanks to its reliability and strong compile-time checks.

Rust is a general-purpose programming language that emphasizes concurrency, type safety, and performance. It supports multiple paradigms like most other programming languages. Low-level driver codes are often prone to an assortment of hidden bugs, which in most other languages can be caught only through multiple levels of testing and attentive code review by senior developers, can be more or less avoided in Rust thanks to its excellent compiler checks. In Rust, the compiler also takes up the role of a gatekeeper by only allowing the compilation of code without any elusive bugs, including possible concurrency bugs making it a very reliable language for system programming.

Main features of Rust:

1. **Memory Safety**:
   The memory safety features of Rust make sure that all references are always pointing towards valid memory locations thus eliminating things like garbage collector or reference counting mechanisms that are present in other programming languages such as C. Rust does not permit null pointers, dangling pointers, or data races. Initialization of data values can only be done through a fixed set of forms and these require already initialized inputs.

2. **Memory Management**:
   Memory and other resources are managed through the “resource acquisition is initialization” (RAII) convention. This is not a concept native to Rust but borrowed from C++. In RAII, a resource is tied to a variable and memory is allocated during creation (acquisition) by the constructor and then deallocated using destructor once the variable goes out of scope. This makes sure that we never have to manually free memory and don’t have to worry about any memory

leaks.

3. **Ownership**:
   Ownership is one of Rust's most unique features.This mechanism in Rust makes sure that all values mentioned in the code have a unique owner and the scope of the value matches the scope of the owner. We make sure that at all times, there are either multiple immutable references or just one mutable reference. This makes the concept of 'readers-writer lock' unnecessary in Rust, as only one reference can be modified by any threat at a time.

4. **Types and polymorphism**:
   Rust uses 'type inference' to recognize the variables defined using the 'let' keyword. Variables that are to be modified later on in the code need to be defined with an additional keyword 'mut' to make it mutable. The type system in rust supports a mechanism called 'traits' which annotate types defining common behavior between different types. For example, the 'Add' trait can be implemented by both int and float data types because they can both be added. This feature is called ad hoc polymorphism. Rust allows the definition of functions with generic parameters that can implement specific traits. This allows the same function to be applied to different data types.

5. **Macros for language extension:**
   Rust language can be extended by using the procedural macro mechanism.

6. **Interface with C and C++:**
   The Foreign Function Interface (FFI) allows calling code written in languages such as C or C++ in Rust and also code written in Rust in other programming languages. In this project, the principles of FFI have been widely utilized for exploring the usage of Rust application code with existing C functionalities.

2.2 ZCU106 Evaluation Board

The ZCU106 Evaluation board from Xilinx is a multiprocessing-capable general-purpose evaluation board comprising a Zynq® UltraScale+™ MPSoC EV device and other major peripherals and interfaces. The MPSoC device in turn has a quad-core Arm Cortex-A53 processing system and a dual-core Arm Cortex-R5F real-time processor. It also facilitates flexible programming by providing an FPGA fabric for custom design. The peripheral interfaces contained in the evaluation board include CAN, UART, SPI, etc. The top-level block diagram of the board is given in Figure 1[5].

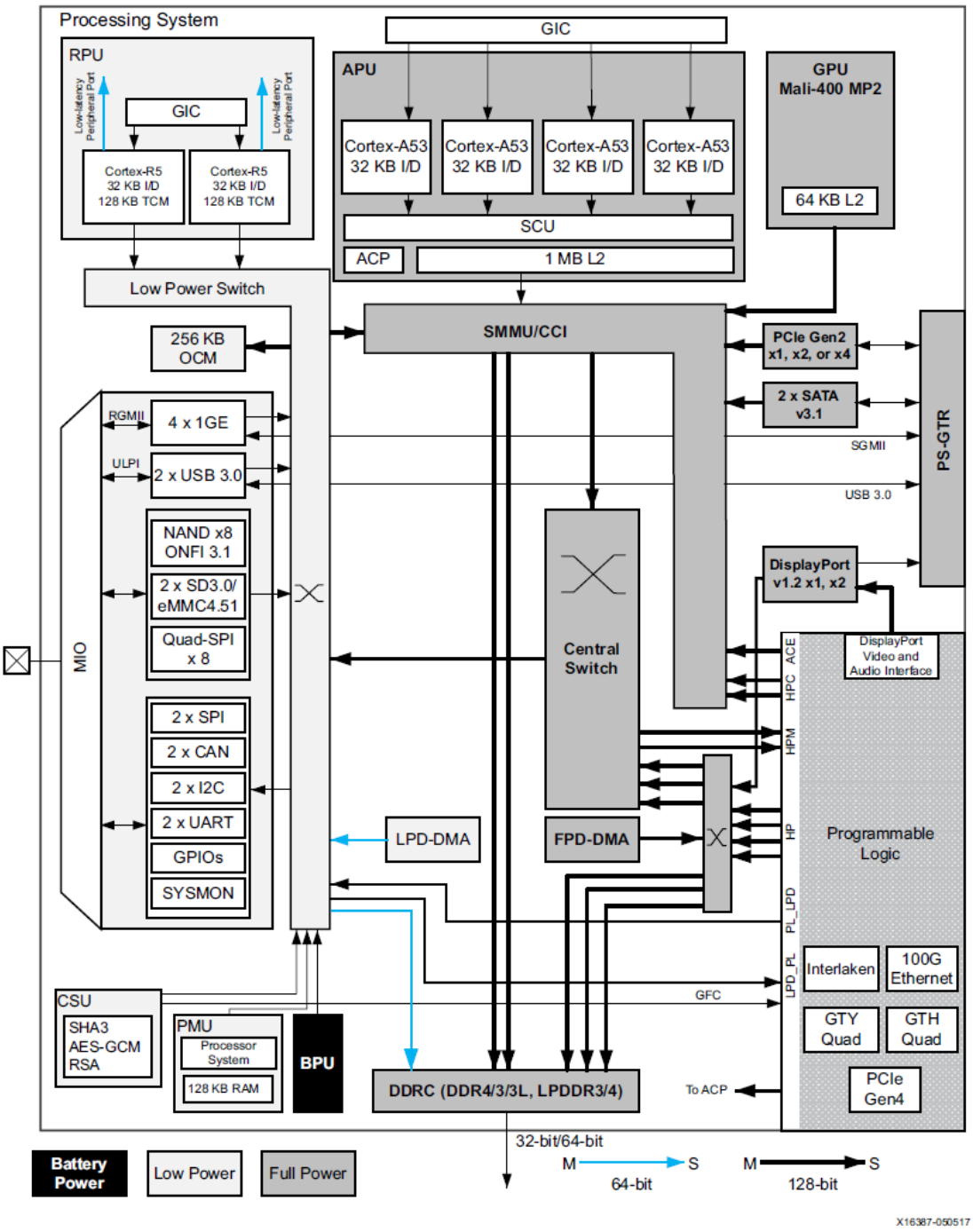
Processing System
RPU
Low-latency Peripheral Port
GIC
Cortex-R5 32 KB I/D 128 KB TCM
Cortex-R5 32 KB I/D 128 KB TCM
APU
GIC
Cortex-A53 32 KB I/D
Cortex-A53 32 KB I/D
Cortex-A53 32 KB I/D
Cortex-A53 32 KB I/D
SCU
ACP
1 MB L2
GPU Mali-400 MP2
64 KB L2
Low Power Switch
256 KB OCM
SMMU/CCI
PCIe Gen2 x1, x2, or x4
2 x SATA v3.1
PS-GTR
SGMII
USB 3.0
RGMII
4 x 1GE
ULPI
2 x USB 3.0
NAND x8 ONFI 3.1
2 x SD3.0/ eMMC4.51
Quad-SPI x 8
MIO
2 x SPI
2 x CAN
2 x I2C
2 x UART
GPIOs
SYSMON
Central Switch
DisplayPort v1.2 x1, x2
DisplayPort Video and Audio Interface
ACE
HPC
HPM
HP
PL_LPD
LPD_PL
Programmable Logic
LPD-DMA
FPD-DMA
Interlaken
100G Ethernet
GTY Quad
GTH Quad
PCIe Gen4
GFC
CSU
SHA3
AES-GCM
RSA
PMU
Processor System
128 KB RAM
BPU
DDRC (DDR4/3/3L, LPDDR3/4)
To ACP
32-bit/64-bit
M
S
64-bit
M
S
128-bit
Battery Power
Low Power
Full Power
X16387-050517

Figure 1

2.2.1 Real-time Processing Unit

In this project, the Real-time Processing Unit (RPU) is used as the target for the Rust application. The RPU consists of a dual-core Arm Cortex-R5F processor interfaced with a Generic Interrupt Controller (GIC) and low latency peripheral ports. As opposed to Cortex-A processors, the Cortex-R processor[2] series has enhanced error management and extended functional safety. This makes it ideal for use in real-time and safety-critical systems. Figure 2[2] shows the general structure of the Cortex-R5 processor.

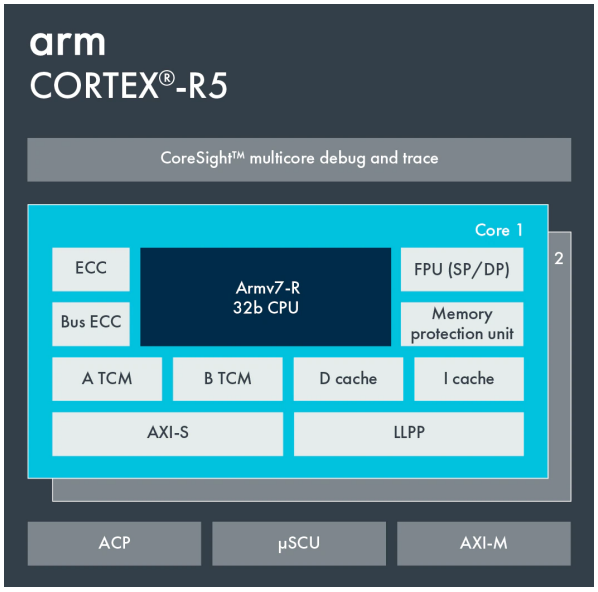

Figure 2

2.2.2 JTAG and Raspberry-Pi Interface

We use a Xilinx HW-FTDI-TEST FT232H 42382 JTAG connector for debug communication channels from the cortex r5 controller to a Raspberry Pi. The Raspberry Pi then forwards the debugging information to authorized virtual machines which are used by the developers to work independently. Connection of the Raspberry pi to the virtual machine is done via configuring the remote access with the lab server "miob-3.acs-lab.eonerc.rwth-aachen.de".

2.3 Bare-metal programming in Rust: (Venkat)

Rustc is the compiler used for the Rust code and is also invoked by **"Cargo"** which is the package manager and build tool for the Rust language.

The following attributes are used in developing bare-metal code from Rust.

#![no_std] - The crate level attribute ‘no_std’ has to be used to prevent rust from linking to the Rust standard library which requires a low-level Software or OS abstraction to run on.

#![no_main]- Attribute used to indicate that this program won't use the standard `main` interface.

Cross-Compilation using Cargo:

(1) The information about the target architecture for which the code is to be built and the corresponding rust flags to be used are mentioned in the config.toml file used by the Cargo tool.

(2) The information about different memory regions and memory sections to be used for the target `armv7r-none-eabihf` is provided in the **link.x** linker file. This linker file is used by the Cargo build tool to cross-compile and provide the bare metal code for the corresponding architecture.

```
# .cargo./config.toml
rustflags = [
   # LLD (shipped with the Rust toolchain) is used as the default
linker
   "-C", "link-arg=-Tlink.x",
]
[build]
target = "armv7r-none-eabihf"
```

(3) The information about the dependency crates for the project and the creation of a static or dynamic library to be generated after compilation, the following information is provided in the **Cargo.toml** file used by Cargo.

```
# Cargo.toml
[lib]
name ="rt"
crate-type=["staticlib"]
[dependencies]
libc = "0.2"
```

3. METHODOLOGY

3.1 IDE and Toolchains

3.1.1 Vitis

Vitis is an eclipse-based IDE used to develop, build and launch projects for the Xilinx Zynq Evaluation Board. A sample project in this IDE consists of a platform project and an application project. The platform project is a combination of hardware information (XSA exported from Vivado) and software runtime environment (Domains/Board Support Packages, First Stage Bootloader, etc.). A layout of the software architecture is shown in Figure 3. Vitis IDE originally supports development in the C programming language.

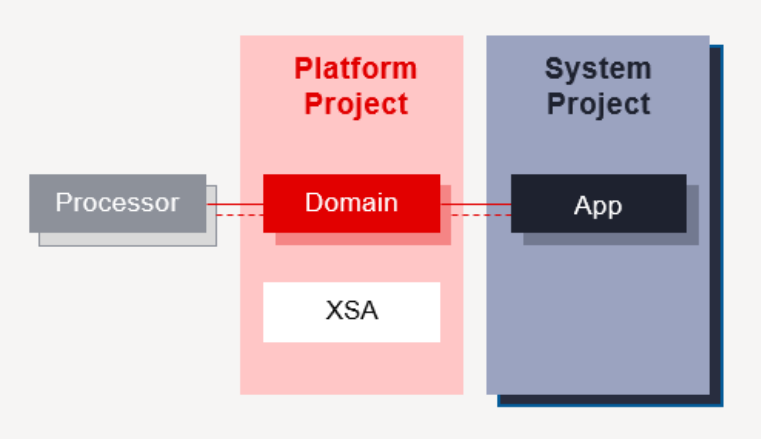

Figure 3

A sample application project is created to understand the IDE workflow. Steps from the Cortex-R-Programming-Guide[4] are followed for setting up the project. During the build process of the application project, makefiles are automatically generated by the IDE according to the source files present in the project. A successful build process results in the generation of two *.elf* files- one for the application project and another for the platform project respectively.

The download of the application is carried out by the IDE by executing a series of steps present in a *tcl* script. The log for the script execution can be found under the *Vitis Log* tab within the IDE.

3.1.2 Rust development in Vitis (Ramees)

In order to facilitate development in Rust in the Vitis IDE, the Rust cross-compiler for the specific target is to be installed and build configurations changed. To install the Rust toolchains, firstly, the following command[6] is run from the terminal:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Once the rust compiler is installed, use the below command to add the target-specific cross-compiler:

```
rustup target add armv7r-none-eabihf
```

With the above steps, the *rustc* cross-compiler is ready to use.

As mentioned in the previous section, the sample application project in Vitis is a makefile project. Here a project named *Rustapp* is created. This creates two subfolders in the workspace *Rustapp_System* and *Rustapp*.

Now new Rust source files can be created under the directory */Rustapp/src* with .rs extension(Eg:*rustcode.rs*)

To obtain the template for the makefile, the C project is built once. The makefiles are automatically generated by the IDE */Rustapp/Debug* folder when the build button is pressed. Once the template for the makefile is obtained, the automatic generation of the makefiles are disabled with the following step:

Rightclick Rustapp(project) from workspace->Properties->C/C++ Build->Uncheck 'Generate makefiles automatically'

Now the makefiles can be modified without being overwritten. The compiler invocations are present in *subdir.mk* file under */Rustapp/Debug/src.* The Rust target object files are added under *OBJS* section (Use the same name as the source file):

```
OBJS += \
./src/helloworld.o \
./src/platform.o \
./src/rustcode.o
```

The following section is added at the end to compile the rust source files into the target object files: (See full code in Appendix <appendixno>)

```
src/%.o: ../src/%.rs
	@echo 'Building file: $<'
	@echo 'Invoking: Rustc Cross-Compiler for ARMv7R'
	rustc --emit=obj --target=armv7r-none-eabihf -o "$@" "$<"
	@echo 'Finished building: $<'
	@echo ' '
```

3.2 Calling RUST code from the C program

To make a RUST sub-function callable from a C program, a 'C-friendly' API has to be created and the output files integrated into the C build toolchain.

To build a C-friendly API, the following points are to be noted:

(1) Use **#[no_mangle]** attribute: Rust compiler mangles symbols in a way different from what is expected by the native linkers. This attribute allows the names of the functions written in Rust to be discovered during the linking process.

(2) Use **extern "C"** keyword: Normally, Rust uses its own Application Binary Interface for all the functions in the program. However, system ABI is to be used to allow the function to be called from another module (in a different language) by using a Foreign Function Interface. This keyword ensures that system ABI is used so that there is smooth communication between modules from C and Rust.

Considering the above two points, the function header can be rewritten as:

```
#[no_mangle]
pub extern "C" fn rust_func() -> i32 {
```

In order to verify the proper working of the interaction between C and Rust modules, a sample code (`rustcode.rs`) is written in Rust and compiled to obtain the object file (`rustcode.o`) file from the terminal. Rust cross compiler is used for this purpose as shown below :

```
rustc rustcode.rs -o rustcode.o --emit=obj --target=armv7r-none-eabihf
```

--*emit* flag is used to specify the type of the generated output file. Here, *obj* denotes the native object (.o) file.

--*target* flag is used to select the target triple. A target triple is a string to inform the compiler about the architecture of the target processor. Since Cortex-R5F has a ARMV7 instruction set with a dedicated floating-point unit, the target triple used here is: *armv7r-none-eabihf* (Bare ARMv7-R, hard-float). It is to be noted that using *armv7r-none-eabi* (Bare ARMv7-R, soft-float) as a target would create build errors if the use of *hard-float* ABI is not consistent throughout the project.

The resultant object file is linked with the *main.o* (compiled from *main.c*) using the GCC Linker. Makefile is modified such that the *rustcode.o* file is included during the linking process.

Once the callability of the function from C is verified, the compiling of the Rust source files is integrated into the makefile of the project as mentioned in section 3.1.2

3.3 Calling C code from the Rust program

The process of calling a "C" function from the main function of the rust code involves the use of the **"unsafe"** block of code. According to the rust language description, the keyword "**unsafe**" has to be used for the following cases:

(1) To declare the existence of the contracts the compiler cannot check.
(2) To declare that the programmer has checked these contracts have been upheld.

The "unsafe" block declares code or interfaces whose memory safety cannot be verified by the type system. The Unsafe rust block can be used to implement these features:

(1) Dereference raw pointers
(2) Implement unsafe traits
(3) Call unsafe functions
(4) Mutate statics
(5) Access fields of unions

Since the usage of the external functions developed by "C" code involves the part of code which cannot be checked by the Rust compiler, the call to these functions has to be given inside the unsafe code blocks.

To start with, the function prototype of the target function to be called is defined in an extern block as shown below:

```
extern {
      fn print(ptr: *const str);
      fn init_platform();
      fn cleanup_platform();
      }
```

Then the unsafe block is written where it is intended; Eg: in main function:

```
pub extern "C" fn main() -> i32 {
      unsafe {
      init_platform();
      print("Calling Print fn from Rust");
      cleanup_platform();
      }
      return 0;
}
```

In this part of the code, since it involves the initiation and cleaning up of the platform code base written in C by implementing calls to **init_platform()** and **cleanup_platform(),** the function calls had to be implemented inside an unsafe block in the rust's main function.

3.4 Interaction with a custom FPGA peripheral (Ramees)

An interrupt-based timer is created in Vivado from scratch by referring to the document Zynq UltraScale+ MPSoC Embedded Design Tutorial[7]. The block diagram of the final design is given in Figure 4.

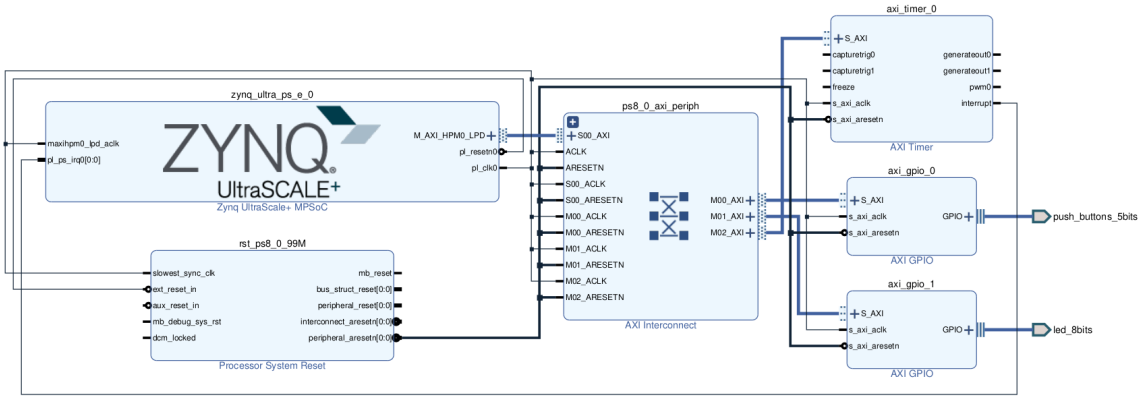

Figure 4

The design is synthesized, implemented and the bitstream generated. Then XSA+bitstream file is exported from Vivado and a new platform project is created in Vitis with it. An empty application project is created for the Cortex R5 and the code[8] for the interrupt-based timer is added and modified to facilitate calls from Rust. A Rust wrapper is created to call the timer function with the number of led blinks as an argument. The makefiles are modified to include the new rust code and the target is built. During the launching process, the FPGA is programmed with the bitstream file and the application is run. It is observed that the function calls from Rust work from the console output.

3.5 Scheduler (Venkat)

This work involves the construction of a basic scheduler function in Rust to implement the scheduling of real-time tasks in the cortex-r5 core. The scheduler uses the value of the inbuilt sleep timer present in the Zynq evaluation board. The program in Rust uses the built-in Triple-Time-Counter(TTC) module as a timer for scheduling the tasks.

Non-preemptive Priority scheduling:

The Scheduler uses the Non-preemptive priority scheduling algorithm with the usage of three priority levels for the task. P1 is set as the highest priority level while P3 being the lowest.

The tasks to be scheduled are initially added to the Primary task table which contains the entire list of tasks to be scheduled and the corresponding Priority task table which contains the list of tasks to be scheduled in the task's priority level.

The scheduler then checks for the list of tasks to be executed in the highest priority queue according to their periodicity and then in the next priority level queue and so on.
The scheduler run function implements the safety check to check if the index of the task arrays are below the array bounds to prevent the core from panicking in case of out of array bounds access.

Attributes of Rust used in the code development:

Use of extern functions:

Extern blocks are part of the Rust's Foreign function interface used to access the functions that are not part of the current crate. In this case of the scheduler code written in Rust, accesses are made to the C library functions for calling the print functions implemented in the Platform project in C.

Unsafe Blocks:

Unsafe blocks are implemented in this Rust code since it involves the dereferencing of raw pointers, usage of mutable static variables and calls to unsafe functions.

Usage of Panic Handlers:

Since it is a no_std application the behavior during the occurrence of a panic event can be expressed using a Panic Handler function.

```
#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {} //Print panic message
}
```

Usage of Copy and Clone traits:

The code involves the usage of Copy and Clone traits to create the initial number of task descriptors statically.

3.6 Experimental work

3.6.1 Print function in Rust (Bharath)

A print function using the debug communication channel has been implemented in C as part of Xilinx platform code. An attempt to recreate this print function using purely rust code was made.

Analysis of the C code revealed that the function uses a char pointer as input and writes one character at a time into the Data Transfer Register of the Debug communication channel (DCC). This corresponds to Co-Processor 14 (CP14) which is called using inline assembly code within the c code using __asm() functionality. In Rust, a similar implementation of print function using inline assembly can be performed by using the **!asm()** functionality provided by the core crate.

```
#[no_mangle]
fn xcoresightps_dccsendbyte_rust(data: u8){
      while (xcoresightps_dccgetstatus_rust() & (1<<29)) ==1 {
            _dsb();
            unsafe{asm!("nop");}
      }
      unsafe {
            asm!("mcr p14, 0, {0}, c0, c5, 0", in(reg) data);
      }
      _isb();
}
```

In the latest Rust Versions, the asm!() function uses volatile as default. So we don't have to explicitly mention the 'volatile' keyword like in C language.

Before writing the data into the Data Transfer Register (DTR) of the cortex r5 DCC , we are required to check the RXfull flag which is the 30th Bit in the Debug Status and Control Register (DBGDSCR). We have implemented the rust function xcoresightps_dccgetstatus_rust**()** to do this. The system needs to make sure that this bit is cleared and make sure that there is no data left to be read and that the processor is ready to read CP14 DTR.

```
#[no_mangle]
fn xcoresightps_dccgetstatus_rust() -> u32 {
      let mut status : u32 = 0;
      unsafe {
            asm!("mrc p14, 0, {0}, c0, c1, 0", out(reg) status);
      }
      return status;
}
```

We also need to use functions _dsb() and _isb() for synchronization purposes.

_dsb() : Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes only when both:

- Any explicit memory access made before this instruction is complete.
- All cache and branch predictor maintenance operations before this instruction complete

```
#[inline(always)]
pub fn _dsb() {
        unsafe {
            asm!("dsb");
            }
}
```

_isb() : Instruction Synchronization Barrier flushes the pipeline in the processor so that all instructions following the `ISB` are fetched from cache or memory after the instruction has been completed.

```
#[inline(always)]
pub fn _isb() {
        unsafe {
            asm!("isb");
            }
}
```

The implementation is unstable currently and more debugging will be needed to produce a reasonably useful library to support print functionality to the virtual system through DCC.

3.6.2 Print function using impl Write (Bharath)

Currently the Xilinx platform for cortex r5 does not provide a proper print function that can be used to print string and variables together. This can be done in Rust by utilizing the write!() core crate functionality. Also, the feature of “impl” in Rust can be used to implement functions for structures.

An implementation that utilizes these features of rust can be used to develop an efficient print function as shown below.

```
#[derive(Debug)]
struct WriteBuf{
    data: [char; 64],
    count: usize
}
```

```
#[no_mangle]
impl Write for WriteBuf {
    fn write_str(&mut self, s: &str) -> core::fmt::Result {
        for c in s.chars() {
            self.data[self.count] = c;
            self.count += 1;
            self.count %= self.data.len();
        }
        Ok(())
    }
}

#[no_mangle]
unsafe fn print_rust(data: &[char;64], size: usize) {
      let mut datau8: u8;
      for count in 0..data.len(){
            datau8 = data[count] as u8;
            XCoresightPs_DccSendByte(0x1234,datau8);
      }
}
```

Firstly the WriteBuf structure is created where we can store a string data and its length. The structure is passed through write!() function along with any other variables that we need to print. These variables in turn get appended to the data of the object “buf” of the structure WriteBuf. Finally, the whole string in buf is written into the DCC using print_rust function which writes byte by byte into the DCC by calling the Xilinx platform C function XCoresightPs_DccSendByte().

The print functionality can be executed or called in the main function as shown in the below example.

```
let mut buf = WriteBuf{data: ['\0'; 64], count: 0};
let i = 6 * 5;
write!(buf, "a number: {} and more", i );
print_rust(&(buf.data),buf.count);
```

This should ideally print the string “a number: 30 and more” in the Vitis IDE if the DCC is configured to be displayed there.
This code cannot be provided as a library package right now as it is unstable with the compiler flags used during implementation. It currently causes errors during the linker stage related to “undefined references” of core crate functionalities. Further in-depth analysis of Rust core crate behaviors during GCC linker stage is required to completely fix this.

4. RESULT AND DISCUSSION

A basic development infrastructure for Embedded Rust is set up during the due course of this project. Various methods to interface modules written in Rust and C code are learned, implemented, and verified. Integration of Rust module into the native C build system of Vitis IDE is also achieved. Another aspect that was studied is the creation of custom peripherals in FPGA fabric and its interaction. It is verified that the custom interrupt-based timer created in the Programmable logic part of the MPSoC runs and the iteration of the runs can be controlled by the Rust code.

In the future, to allow more control of various system components from the Rust-side, exporting of user-defined types (Eg: interrupt, timers, GPIO structs) into Rust can be done. The dependent C headers can be exported into Rust-readable format using bindgen tool[9]. Since most of the user-defined types used in the current platform code have a recursive dependency on other user-defined types, the use of bindgen is recommended with a cargo project. This also requires some reconfigurations in the currently implemented build system. These points can be addressed in the future in order to create a more versatile infrastructure for Embedded Rust development.

5. REFERENCES

[1]https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Workspace-Structure-in-the-Vitis-Software-Platform.
[2]https://developer.arm.com/Processors/Cortex-R5
[3]https://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-ev-block.PNG
[4]https://github.com/RWTH-ACS/miob/blob/main/docs/Cortex-R-Programming-Guide.md
[5]https://www.xilinx.com/support/documents/boards_and_kits/zcu106/ug1244-zcu106-eval-bd.pdf
[6]https://rustup.rs/
[7]https://xilinx.github.io/Embedded-Design-Tutorials/docs/2022.1/build/html/docs/Introduction/ZynqMPSoC-EDT/7-design1-using-gpio-timer-interrupts.html#
[8]https://github.com/Xilinx/Embedded-Design-Tutorials/blob/master/docs/Introduction/ZynqMPSoC-EDT/ref_files/design1/timer_psled_r5.c
[9]https://rust-lang.github.io/rust-bindgen/introduction.html
[10]https://doc.rust-lang.org/book/
[11]https://docs.rust-embedded.org/book/
[12]https://github.com/rust-lang/rustlings/