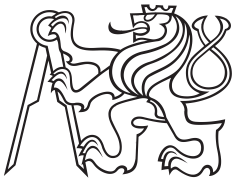
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Zephyr RTOS

Marek Vedral

Supervisor: Ing. Michal Sojka, Ph.D.
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Vedral Marek** Personal ID number: **499096**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Zephyr RTOS

Bachelor’s thesis title in Czech:

RTOS Zephyr

Guidelines:

1. Make yourself familiar with the Zephyr real-time operating system.
2. Run Zephyr on a supported ARM or ARM64 board (e.g. Raspberry Pi Pico) or in an emulator (ARM Cortex-A53 Emulation (QEMU))
3. Create several sample programs and use them to compare the features of the Zephyr RTOS and VxWorks. Focus on interacting with programs using the serial line and on debugging capabilities (interactive debugging, tracing).
4. Add support for MicroZed (Xilinx Zynq 7k) and MZAPO hardware to the Zephyr RTOS. Allow the Zephyr applications to be debugged without hardware debugger (this will probably require extending Zephyr GDB stub for the ARM architecture). Evaluate your port with the sample programs developed in the previous step. Compare Zephyr and VxWorks scheduling latency with the benchmark used in the PSR course.
5. Document the results and if possible, submit at least some of the developed code for inclusion to the open-source Zephyr project.

Bibliography / sources:

[1] Zephyr Project Documentation, online: https://docs.zephyrproject.org/latest/
[2] M. Sojka: Measuring of Response Time to Asynchronous Events, PSR lab instructions, online: https://rtime.felk.cvut.cz/psr/cviceni/latency/
[3] Avnet: MicroZedTM Zynq® Evaluation Kit and System on Module Hardware User Guide, Version 1.7, 2017
[4] Xilinx, Zynq-7000 All Programmable SoC Technical Reference Manual – UG585 (v1.10) February 23, 2015

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Michal Sojka, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to express my sincere gratitude to Ing. Michal Sojka, PhD. for his guidance and invaluable assistance throughout the work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

V Praze, 24. May 2023

Abstract

Zephyr is a new, free and open source real-time operating system that intends to provide a secure, feature-rich application platform on a wide range of supported boards. This work describes the process of porting the system to Avnet MicroZed board and adding support for remote debugging using GDB. Our board port builds upon and expands the configuration of another officially supported device. Additionally, we develop an extension to the existing GDB stub implementation for the 32-bit ARM architecture. The performance of Zephyr on the MicroZed board is currently much worse than Linux, as the CPU caches need to be disabled for proper functionality. When combined with the limited troubleshooting capabilities, we conclude that a lot of work is still required to replace established operating systems, such as Linux or VxWorks, with Zephyr.

Keywords: systems programming, RTOS, embedded systems, Zephyr, GDB, MicroZed

Supervisor: Ing. Michal Sojka, Ph.D.
CIIRC,
Jugoslávských partyzánů 1580/3,
Praha 6

Abstrakt

Zephyr je nový, svobodný operační systém reálného času, který slibuje poskytovat bezpečnou platformu pro vývoj aplikací na širokém spektru podporovaných desek. Tato práce popisuje proces portování tohoto systému na desku Avnet MicroZed a přidání podpory vzdáleného ladění pomocí GDB. Náš port je založen na a rozšiřuje konfiguraci jiné, oficiálně podporované desky. Dále je vytvořeno rozšíření stávající implementace GDB stub pro 32 bitovou architekturu ARM. Výkon systému Zephyr na desce MicroZed je zatím výrazně horší, než Linux, protože pro správnou funkcionalitu musí být CPU cache vypnuta. Pokud toto spojíme s omezenou možností ladění, tak usuzujeme, že je stále potřeba velké množství práce, aby mohl Zephyr nahradit zavedené operační systémy jako VxWorks nebo Linux.

Klíčová slova: systémové programování, RTOS, vestavěné systémy, Zephyr, GDB, MicroZed

Překlad názvu: RTOS Zephyr

Contents

Figures

Tables

Chapter 1

Introduction

Embedded devices have become increasingly popular in recent years as a result of surging demand for smart home devices, gadgets or self driving vehicles. With these technologies also rises the demand for real-time operating systems as platforms for reliable and responsive applications. Zephyr OS aims to be a feature-rich, free and open source real-time system targeted for a wide variety of microcontrollers and architectures. This work aims to port the system to Avnet MicroZed, a development board based on Xilinx Zynq-7000 SoC. Additionally, we add support for debugging without the use of a hardware debugger by extending the current implementation of GDB stub to ARM.

Furthermore, the thesis evaluates the potential of using Zephyr for school programming courses. Some subjects at FEE, CTU have been using the proprietary VxWorks RTOS on the MicroZed boards for assignments. The main disadvantage of this approach is the licensing and distribution of VxWorks. The board has to be connected to a laboratory LAN with a TFTP server to download and boot the image. It is also necessary to develop on computers with the Wind River IDE. Using Zephyr would allow students to tinker with the OS on their own hardware at home, to inspect the source code and to use any development environment they prefer.

The thesis begins in chapter 2 by outlining the fundamental features and application development on Zephyr. The following chapters describe the process and the challenges of adapting Zephyr to MicroZed and enabling key peripherals. Additionally, we document the expansion of GDB stub for ARM and the issues with our implementation. Finally, we assess whether the system is suitable as a platform for practical assignments.

Chapter 2
Background

This chapter describes the main terms and technologies used throughout this work.

2.1 Embedded systems

An embedded system is generally a computer system integrated in the technology it controls. These systems tend to sacrifice universal applicability, a feature of traditional consumer-oriented operating systems, for dedicated aspects such as power consumption, physical size or performance in specific tasks. Embedded computer systems therefore usually run bare-metal software or real-time operating systems. However, resource-heavier operating systems with established API, such as Linux, are commonly used too.

Embedded systems are heavily used today. Such computers may be found in smart home appliances, gadgets, IoT sensors as well as vehicles, industrial robots and deep space probes.

2.2 Real-time OS

Real-time operating system (RTOS) is an operating system, which is generally designed to react within precise time constraints to its requests. Such systems are expected to provide deterministic behaviour even in the worst case scenario and to run continuously without human intervention. A failure of this system could be inconvenient for its users (in case of a **soft**-deadline task) or it may have catastrophic consequences (**hard**-deadline task).

Compared to conventional, consumer-oriented operating systems, real-time systems are not designed to provide a *fair* computing environment for its tasks, or even a graphical interface. Instead, the resources are reserved for tasks with the highest **priority**.

Real-time operating systems are heavily used for a wide variety of applications, including:

- Automotive control systems
- Avionics and space computers

- Medical instruments
- Video game engines and multimedia applications
- Industrial automation
- Nuclear power plant control

There are dozens of real-time operating systems used in production today. Some examples of the main ones with a brief description are the following:

- **VxWorks** – A proprietary, Unix-based OS developed by Wind River. It has many safety certifications and conforms to the POSIX standard. The system comes with its set of development tools, such as an IDE, debugger and a simulator.
- **RTLinux** – One of the first real-time variations of Linux kernel. A modified Linux kernel is treated as a low priority, preemptable thread with a real-time scheduler. The applications run at higher priority and are meant to be separated from the kernel as much as possible. It is distributed as open-source version or a commercial Wind River Real-Time Core for Linux [1].
- **Linux + PREEMPT_RT** – An optimizing patch to the Linux kernel. It makes the majority of the kernel preemptable mostly by replacing spinlocks with mutextes.
- **FreeRTOS** – A free and open source RTOS distributed under the MIT license. The system can be thought of a thread execution environment with focus on small memory and simplicity. It supports a wide range of architectures and embedded boards, from AVR 8-bit microcontrollers to Cortex-A ARM chips. Two commercial derivatives, OpenRTOS and SafeRTOS, are available, which provide warranty or safety certification [2].

Real-time operating systems often run on embedded, application specific devices, such as a small IoT microprocessors, wearable devices or aircraft computers.

2.3 Devicetree

Devicetree specification [3] defines a structure to describe hardware components of a computer system. This structure is loaded at boot time to system memory and the bootloader passes a pointer to the structure to the operating system. Device drivers then access these properties and configure the devices accordingly.

The fundamental part of a devicetree is a tree data structure, devicetree **source** (DTS), which defines **nodes**. Each node describes a hardware device, such as a CPU core or LED diode, or an abstraction of other components, like pin control blocks or a group of FPGA registers. Each node may have multiple *properties*, which are key-value pairs specifying parameters of the device. The tree can also represent interrupt routing, where some components represent interrupt controllers and others point at them, which forms a logical interrupt tree. Listing 2.1 shows a simple example of a DTS file.

Listing 2.1: Part of DTS file for Raspberry Pi Pico

```
/ {
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu0: cpu@0 {
        compatible = "arm,cortex-m0+";
        reg = <0>;
    };
    cpu1: cpu@1 {
        compatible = "arm,cortex-m0+";
        reg = <1>;
    };
};
soc {
    sram0: memory@20000000 {
        compatible = "mmio-sram";
        reg = <0x20000000 DT_SIZE_K(264)>;
    };
    flash0: flash@10000000 {
        compatible = "soc-nv-flash";
        write-block-size = <1>;
    };
    gpio0: gpio@40014000 {
        compatible = "raspberrypi,pico-gpio";
        reg = <0x40014000 DT_SIZE_K(4)>;
        interrupts = <13 RPI_PICO_DEFAULT_IRQ_PRIORITY>;
        gpio-controller;
        #gpio-cells = <2>;
        status = "disabled";
        ngpios = <30>;
    };
};
};
```

All nodes in the DTS are checked with corresponding devicetree **bindings**, or a collection of rules that describe required properties, values or value data types for a device.

Finally, the devicetree is *compiled* into a devicetree **blob** (DTB), which represents the structure in a linear, binary file. This file is copied to RAM by the bootloader.

Devicetree is used mostly on microcontrollers and embedded systems. Platforms such as x86 much more often use interfaces like ACPI, which allows the OS kernel to dynamically configure connected devices.

2.4 Avnet MicroZed

The MicroZed is a system on module (SoM) developed by Avnet and based on the AMD Xilinx Zynq-7000 SoC. In this work, we use the **revision F** of the SoM, which has the following main features [4]:

- Dual-core Cortex-A9 CPU, version 7Z020
- 123 programmable MIO pins, FPGA
- 1 GB DDR3 memory, 128 MB QSPI flash
- 10/100/1000 Ethernet
- Two 100-Pin Microheader pins
- USB-UART bridge via USB 2.0 port
- 33.333 MHz on-board oscillator
- MicroSD card slot
- User LED and push button

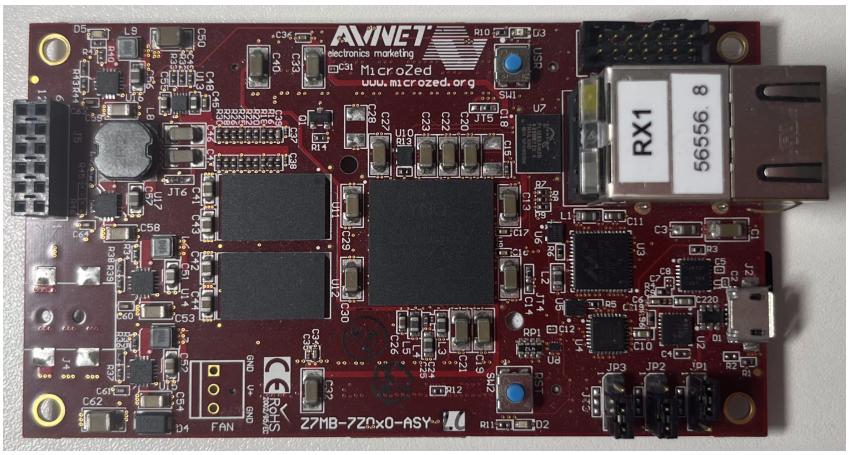


Figure 2.1: Avnet MicroZed

The board has been used for teaching at FEE with an additional shield designed by Ing. Petr Porazil [5]. The shield connects to the MicroZed board using the two Microheader pins and provides many additional peripherals, such as a display, input knobs, PMOD connectors or an audio jack. This kit, shown on picture 2.2, is often referred to as **MZAPO**. See the section 6.2 for more details about the kit.

In chapter 5, we work with the standalone board, whereas chapter 6 utilizes the shield.

Figure 2.2: MZAPO kit

Chapter 3
Zephyr OS

This chapter describes the integral subsystems and properties of the Zephyr RTOS.

3.1 Overview

Zephyr is a free and open source real-time operating system licensed using the Apache 2.0 license. It is designed primarily as a small-footprint runtime environment for applications on microcontrollers and embedded systems. Key features of Zephyr include [6]:

- Support for many architectures, such as x86, RISC-V or ARM
- Modularity and flexible configuration
- Various on-line scheduling algorithms
- Hardware configuration with devicetree
- User mode threads and basic memory protection
- Limited support for POSIX API

This section describes a couple of the main features of the system that I experimented with. Refer to the official documentation for more details.

3.2 Kernel

Kernel is essentially a collection of code libraries, which provide fundamental functions to the operating system and its applications, such as scheduling, virtual memory management and hardware abstraction. This section describes the key aspects and properties of the Zephyr kernel.

3.2.1 Threads

Thread provides a single execution context for a program. Each thread has its own stack and registers, meaning that different threads can run asynchronously at the

same time on a multi-core processor. Threads can also communicate and share data in parallel applications using shared memory.

Contrary to general-purpose operating systems, Zephyr does not support **processes**. A process is a distinctive program in execution, which has its own virtual memory and encompasses the threads created by that process. Zephyr system therefore can be thought of as one process.

Zephyr supports theoretically unlimited number of threads and is limited only by the amount of RAM. Each is given a *priority*, a positive or negative integer value, during its creation. The OS can define arbitrary bounds of the priority value. Based on the value of priority, Zephyr distinguishes three thread *classes* — preemptive, cooperative and meta-IRQ [7]. Zephyr also defines six **runlevels**, which are groups of functions that are called different stages of the boot process. From earliest execution to the latest, the groups are `EARLY`, `PRE_KERNEL__1`, `PRE_KERNEL__2`, `POST_KERNEL`, `APPLICATION` and `SMP` (if SMP is enabled). Function is assigned with a particular priority to a given group at compile time using a macro.

Zephyr also defines two **system threads**, **main** and **idle**, which are started automatically during boot. The main thread initializes the OS kernel and calls the **main** function, assuming that it is defined. If the user application does not include this function, it can either assign a function to a runlevel or define a new thread with a given entry function. The idle is executed if there is no pending thread and the system has simply nothing else to do. This thread may activate various power saving features of the processor or wait in a loop.

Zephyr provides multiple kernel objects, which implement commonly used **synchronization** tools, such as mutexes, condition variables, semaphores and atomic services. Zephyr can be also configured to run **without threads** [8]. In that case, all synchronization objects and the thread scheduler will be removed entirely from the system executable. Core functionality such as interrupt handling, timers and memory management are expected to work. On the other hand, Zephyr subsystems other than UART, GPIO and Flash are **not supported** in this configuration.

3.2.2 Scheduling

A **scheduler** is an algorithm in operating system kernel that decides to which thread to assign the CPU. The scheduler may be called periodically or when a running thread changes its state, for example when running thread goes to sleep (these points are sometimes called *reschedule points*).

Zephyr supports the following on-line scheduling algorithms:

- **Static priority scheduler** — This is the default algorithm. The scheduler simply executes the thread with the highest static priority. In case two threads with the same priority are ready, the one that has been waiting longer is chosen.
- **Earliest deadline scheduler** (EDF) — This scheduler may be optionally selected. In case two threads have the same static priority, the scheduler executes the thread with the earliest deadline — the priority does **not** depend solely on the deadline and the static priority is still the primary indicator! A thread with a higher static priority and later deadline would be executed before a thread with lower priority and early deadline.

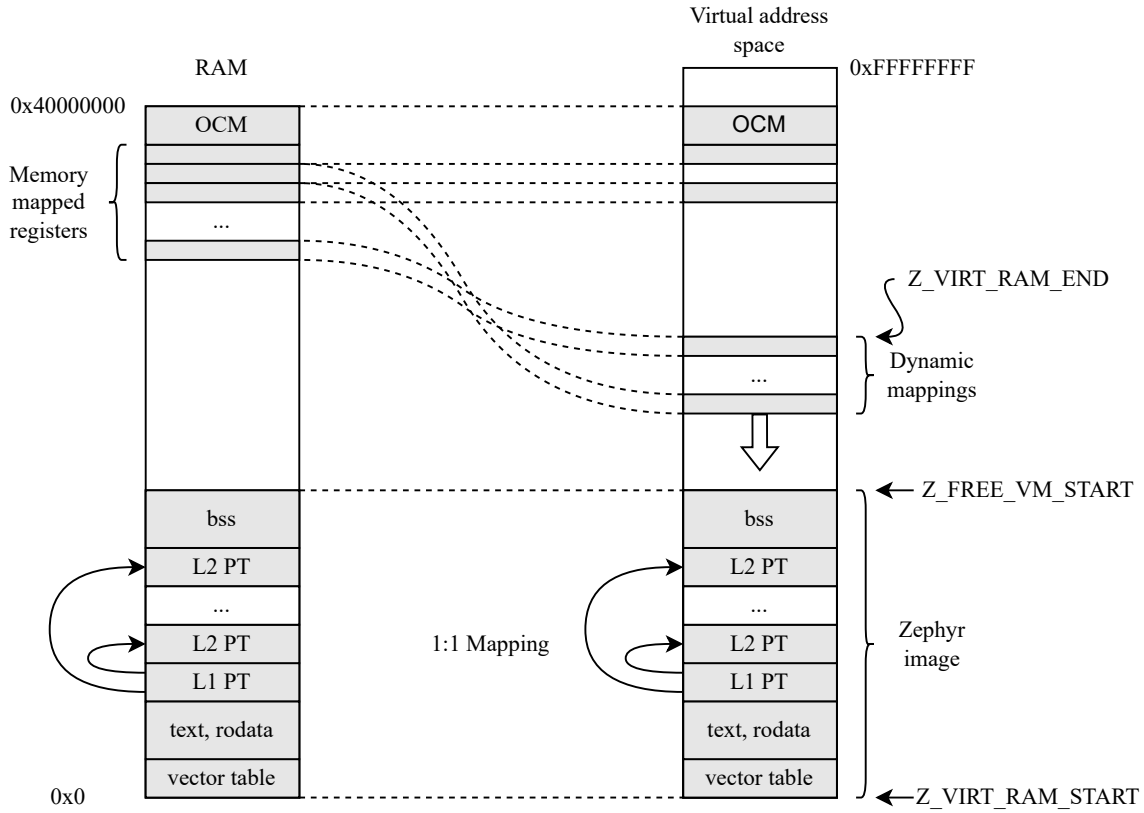


Figure 3.1: Zephyr memory model on Zynq-7000

Zephyr also supports different data structures, which implement the **ready queue**, such as a red-black tree or a linked-list. Each structure has different performance and memory overhead [9]. In case a running thread preempts itself, it is placed at the back of the ready queue of its priority.

3.2.3 Memory management

According to the documentation, the memory protection design of Zephyr has been developed for processors with an **MPU** (Memory Protection Unit). Even though the system supports architectures, which support a fully-featured **MMU** (Memory Management Unit), the system treats it as an **identity page MPU** [10]. However, this is not entirely accurate. The memory management library for Cortex-A series SoC does indeed support MMU with multi-level page tables, access permissions and TLB (Translation Lookaside Buffer). While the kernel image and some memory mapped registers are mapped identically (one-to-one), Zephyr supports dynamic memory allocation, which many device drivers and the heap API use at runtime. Diagram 3.1 shows a high level overview of the Zephyr memory model.

The Zephyr image is generally composed of the program instructions, static variables and other data structures. The *section* **bss** (Block Starting Symbol) contains static variables, which have been declared in the code, but have not been assigned a value

yet. The sections **text** and **rodata** are the program instructions and the initialized variables (constants and literals) respectively. I should again emphasize that the diagram is a significant simplification of the real memory map. All of the sections and the instructions may be inspected by disassembling the output ELF file. The three sections mentioned take up the vast majority of the final Zephyr image size. The system maps these sections at boot, along with a few other memory areas, such as the **interrupt vector table**. The SoC also defines some memory mapped registers, which are mapped one-to-one and at boot time.

The constant `Z_VIRT_RAM_START` defines the start of the kernel address space. By default, this is the same as the base of the memory region assigned to `zephyr,sram` property in the `chosen` node — **0** for us. The value `Z_FREE_VM_START` defines the end address of the mapped Zephyr image. In case the system maps all physical RAM pages at boot, which Zephyr on ARM does not, this value equals the RAM size. In our case, it is set at build time to the end address of the image and that of course depends on the final size of the compiled system. The macro `Z_VIRT_RAM_END` is defined as `Z_VIRT_RAM_START` + `Z_VIRT_RAM_SIZE`, where the latter constant is configurable — by default to **0x800000** [11] or roughly 8.3 MB. Since this address represents the upper bound of the kernel address space, the remaining virtual memory region is typically used for identity mapping of some memory mapped registers.

The region between `Z_VIRT_RAM_SIZE` and `Z_FREE_VM_START` is used for runtime memory mappings, which are used by dynamic memory allocation or by drivers for memory mapped IO. The virtual pages are allocated from the highest address downward, as visualized in the diagram. It is important to note that if the Zephyr image is larger than the `Z_VIRT_RAM_SIZE` constant, there is no space for dynamic memory mapping and the system **crashes** at boot.

The L1 PT block is the **level 1 page table**, which, simply put, contains address locations of all **level 2 page tables** (visualized by arrows) and some translation properties. These tables in turn have entries pointing to the desired **memory page**. More specifically, Zephyr uses the following memory configuration:

- The page size is **4 kB** by default. Although the size is configurable, Zephyr recommends using the smallest possible value [12]. In our case, Cortex-A9 additionally supports 64 kB large pages, 1 MB sections and 16 MB supersections [13].
- The L1 page table has **4096** entries (first-level descriptors). Although this is configurable when initializing the MMU, the length is fixed in Zephyr.
- The L2 page table has **256** entries (second-level descriptors) — the first 12 bits of the 20 bit memory page indexes the L1 PT, as $2^{12} = 4096$, and the remaining 8 bits index the L2 table. This length is therefore hard-coded in Zephyr as well. By default, the system uses **64** of these tables, but the number is adjustable [14].

3.2.4 User mode

Zephyr allows to use an optional, non-privileged CPU state called **User mode**, which implements many additional safety features. The system aims to protect

mainly against the following errors [15]:

- User-level thread memory access — Non-privileged threads have by default access only to their stacks and read-only program data.
- Stack overflow attack
- Incorrect kernel API parameters — The system checks argument types, nonsensical values or access rights to passed objects.
- Unauthorized re-entry to supervisor mode

These properties may be overridden during initialization of user level threads. Zephyr does not protect against any of these errors during supervisor mode or against other types of attacks, such as CPU starvation.

It can be also argued that compile time memory allocation is a form of prevention against run time errors. Zephyr defines many C macros for allocating kernel objects during compilation, such as threads, synchronization objects or atomic variables. It is also possible to define a fixed-size memory pools for dynamic allocation at runtime.

3.3 Subsystems

3.3.1 Shell

Zephyr provides API to implement a Unix-like shell and functions for both compile-time and run-time command definition [16]. The shell supports features such as wildcard expansion, CLI option parsing (getopt), multiple instances or command completion. Zephyr also allows *minimal shell* configuration to reduce memory footprint or to hand-pick additional features.

The system support multiple IO subsystems for the shell, including UART, Telnet, USB or Segger RTT. The particular device used for the shell is specified in the board devicetree.

3.3.2 Console

Console in Zephyr is a high-level wrapper for serial communication over UART and provides IO functions such as `getchar` or `putchar`. These methods use lower-level *TTY* (serial port object) interface, which provides API for buffered, unbuffered, interrupt-driven and polling serial communication. Some of these functions support adjustable receive and transmit timeouts, which are infinite by default. UART drivers are called from the TTY object. Zephyr provides a wide range of formatted output APIs. The console subsystem is intended primarily to simplify porting of software that uses the fully-featured C standard library.

It is important to note that some functions of this subsystem are **not compatible** with each other [17]! For example, the function `console\_getchar()` and `console_getline()` may be used only after a `console_init()` function or `console_getline_init()` have been called - each of these sets up a different Zephyr

subsystem to interact with the UART driver. The `getchar` and `putchar` functions (and also POSIX-like functions `console_read()` or `console_write()`) serve simply as wrappers around the TTY interface. The `console_getline` function uses a more complex **console handler** subsystem, which provides other functionality on top of the UART driver, such as cursor handling or pluggable transport for **MCUmgr**. MCUmgr defines an open and portable[1] system for remote OS management [18]. If we wanted to use both of these subsystems at once, we would have to use **different UART** modules and define this mapping in the devicetree.

Similarly, the shell subsystem uses its own utilities on top of the UART driver and is not compatible with either `console_getchar()` or `console_getline()` functions.

[1] At the time of writing, MCUmgr is implemented only by Zephyr and Apache Mynewt

Chapter 4

Zephyr application examples

Zephyr OS provides official support for many development boards and IoT sensors. This section briefly documents the process of developing, building and running applications on Zephyr.

I tested the applications on a Raspberry Pi Pico board and QEMU x86 emulator, both of which are officially supported [19].

4.1 System configuration

Zephyr uses the **Kconfig** configuration system to manage and set-up the system properties [20]. As a matter of fact, Kconfig is also used by Linux.

Briefly speaking, Kconfig in Zephyr mainly consists of configuration files named **Kconfig** scattered throughout the source tree. These files define Kconfig **options**, which are properties corresponding to a certain subsystem or functionality of the system. Some examples of Kconfig options may include `CONFIG_CONSOLE`, which includes console drivers to Zephyr, or `CONFIG_NO_OPTIMIZATIONS`, which instruct the compiler to not perform any optimizations. Each option may be one of two types:

- **Visible** — These may be configured directly by the user. Zephyr includes the `menuconfig` and `guiconfig` programs, which manage these options with a graphical interface. Each board may also define board-specific assignments of these options in its `[board]_defconfig` (MicroZed does too) and applications often override global options in `prj.conf` files. See the documentation for more information about how the final configuration is derived [21].
- **Invisible** — Invisible symbols are set automatically based on the configuration of other symbols. Many symbols thus form a dependency tree. The files `Kconfig.defconfig` are used to configure the board-specific, invisible symbols with certain conditions.

Additionally, the build system, **CMake**, needs to define which source files correspond to which symbols. For example, the following entry in `drivers/ethernet/CMakeLists.txt` adds the Ethernet driver code for a Xilinx SoC to Zephyr in case the `CONFIG_ETH_XLNX_GEM` option is set:

Listing 4.1: CMake command for Xilinx Ethernet driver

```
zephyr_library_sources_ifdef(CONFIG_ETH_XLNX_GEM
    eth_xlnx_gem.c
    phy_xlnx_gem.c
    )
```

It is also important to note that each symbol has a certain type, such as a **bool** (with values **y** or **n**), **string** or **int**.

4.2 Building

To build Zephyr applications, it is necessary to install the Zephyr SDK and a number of other packages, such as a devicetree compiler or the CMake build system. Code building is done in Python virtual environments, so a working installation of Python 3 is required as well. All steps to set up the development environment are in the getting started guide [22].

The SDK also includes a few tools for running and building Zephyr, chief among which is **west**. West is a command-line tool developed for Zephyr, which mainly provides convenient access to build, flash or run applications. It is also able to track multiple git repositories and cryptographically sign the system images. By default, west uses **ninja** to build the project files generated by CMake, but GNU Make may be used too [23].

The following example shows the process of building sample Zephyr application for QEMU x86 emulator:

Listing 4.2: Building a sample application using west

```
~/zephyrproject/.venv/bin/activate
west build -p always -b qemu_x86 zephyr/samples/hello_world
```

The first command is of course not necessary if west is not installed in a Python virtual environment.

Executable output files of the build process are ELF binaries and other formats depending on the target platform. For example, west generates UF2 firmware image for Raspberry Pi.

4.3 Raspberry Pi Pico

Raspberry Pi Pico is a small microcontroller board designed by Raspberry Pi Foundation. It uses their ARM-based RP2040 chip and has the following key features [24]:

- Dual Cortex-M0+ running at 133MHz
- 264kB on-chip SRAM
- 2MB QSPI flash

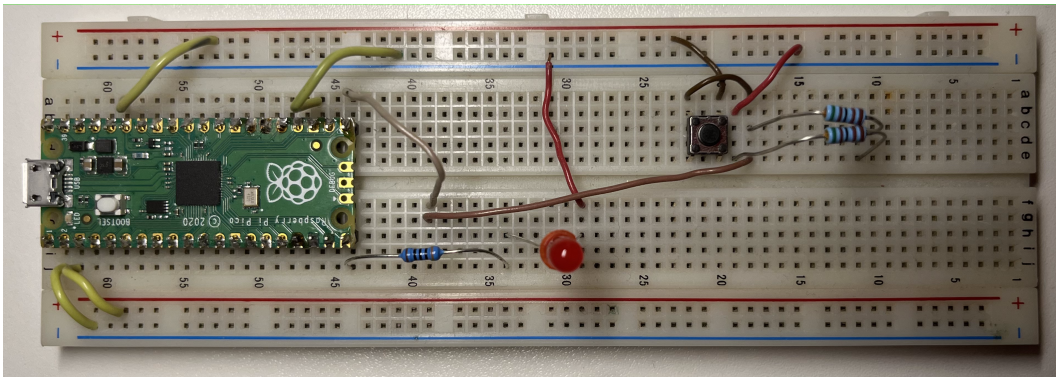


Figure 4.1: Raspberry Pi Pico on a breadboard

- 26 multipurpose GPIO pins
- 2 UART controllers, 16 PWM channels
- USB 1.1 controller with on-chip programmer

As mentioned in the previous section, west also generates the UF2 image when building for this board. The microcontroller is then flashed by simply moving this image onto the board if it is connected as a mass storage device.

I have developed and run several simple Zephyr applications on the board to test various subsystems and capabilities, such as the console, shell or interaction with physical peripherals.

The console and many buffered IO functions can connect to a computer with serial connection over UART. The RX and TX pins of either UART controller can be simply wired to a USB-to-serial converter and connected to a workstation.

Zephyr provides a simple API to directly control physical GPIO pins or generally any device defined in the board devicetree. I have added several devicetree nodes representing LED diodes and push buttons at their respective pins. The API then allows to interact with these nodes in the application code. For example, configuring a device as output allows to set the assigned pin to logical high or low value. Input pins may be connected to a *callback function* (an ISR) for a particular logical level or a level change, such as a rising edge.

4.4 QEMU

QEMU (Quick Emulator) is a widely used, free-and-open-source emulator. It is included in the Zephyr SDK for various architectures, such as x86, ARM (Cortex M0, M3 or R5), RISC-V and others. Each of these supported architectures is represented in Zephyr as another **board** with its devicetree and default kernel configuration options. These QEMU builds use system emulation (as opposed to QEMU user-mode).

Once an application is compiled for the emulator (shown in listing 4.2), the following command opens QEMU on the host computer and runs the system binary:

Listing 4.3: Running an application on QEMU

```
west build -t run
```

Behind the scenes, it expands to this command:

Listing 4.4: QEMU call options

```
/bin/sh -c "cd /home/user/zephyrproject/applications/QEMUTest/\
build && /home/user/zephyr-sdk-0.15.0/sysroots/x86_64-pokysdk-\
linux/usr/bin/qemu-system-x86_64 -m 4 -cpu qemu32,+nx,+pae \
-device isa-debug-exit,iobase=0xf4,iosize=0x04 -no-reboot \
-nographic -no-acpi -net none -pidfile qemu.pid \
-chardev stdio,id=con,mux=on -serial chardev:con \
-mon chardev=con,mode=readline -icount shift=5,align=off,\
sleep=off -rtc clock=vm -kernel/home/user/zephyrproject \
/applications/QEMUTest/build/zephyr/zephyr.elf"
```

We tried running some example applications on the included QEMU-x86 emulator, from simple character IO applications to more complex multi-threading applications. While many Zephyr API functions worked as expected, there seems to be an issue with hardware timer emulation in QEMU that in turn causes some methods to misbehave. More specifically, we have tested both *HPET* and *APIC* timers, both are supported in Zephyr and neither worked correctly.

HPET (High Precision Event Timer) is a modern hardware timer specification for x86. It is commonly used for real-time multimedia streaming, scheduling or general time sampling. One such chip may provide up to 32 *comparators* and one main counter. The timer is programmed using memory-mapped registers at predefined offsets. The *Main Counter Value Register* is a 32 or 64-bit read-write register that stores the current counter value and increments the value at a specified frequency [25]. On Zephyr, HPET is 64-bit long and is incremented every 10 ns (100 MHz) by default.

Once initialized and started during Zephyr boot, this counter would often increase its value by huge amount. This meant that system timer interrupts, which are set to 100Hz on Zephyr, would be generated significantly more often, at nearly 90000 interrupts per second. The system was therefore flooded with timer interrupts and spent all CPU resources on handling the HPET ISR and rescheduling the comparator registers. This resulted mainly in the following issues:

- The OS had no resources to handle other ISRs with. For example, callback routine for getchar function would not execute and character input over UART therefore does not work.
- Time perceived by the system would speed up dramatically. As a result, waiting functions such as k_msleep return immediately. On the other hand, methods for busy-waiting, like k_busy_wait would work correctly.

My thesis supervisor, Dr. Michal Sojka, has submitted an issue to Zephyr GitHub repository regarding these problems on QEMU. We have tested the same application on different QEMU versions as well as with older releases of Zephyr. We also tried

using a standard QEMU releases instead of the emulator that comes with Zephyr SDK. However, none of these attempts have solved the issue and at the time of writing, there seems to be no result on the GitHub issue either.

APIC (Advanced programmable interrupt controller) is a modern specification for interrupt controllers on x86 processors. Among other functionalities, it defines a per-CPU-core local interrupt controllers (LOAPIC), which manage local vector tables (LVT) and handle incoming interrupts. The LVT is also able to handle events such as internal clock and can therefore define a timer bound to the specific CPU core.

Zephyr may be configured to use APIC instead of HPET. I tested the same applications as I did with HPET and there appears to be a similar issue. I have not tested this configuration nearly as much as HPET, but character input over serial console worked correctly. However, neither busy waiting or classic delay functions worked.

Finally, we have also tested a simple application on an desktop computer with an Intel Core i5 Ivy Bridge CPU. The program printed a couple of "Hello world" messages in a loop with a delay of 1s. The application worked as expected, indicating that there is likely no HPET issue in Zephyr, but in the QEMU emulator.

4.5 Debugging

Zephyr supports various ways to debug the application or the system itself. This section describes two of these approaches using *GDB* (GNU Debugger), which I tested and are universally applicable. The system also supports and recommends different commercial hardware debuggers, none of which I tested or have access to.

4.5.1 QEMU hardware debugging

Hardware debugging refers to program debugging using the host CPU resources. For example, setting dedicated registers to an address may act as a breakpoint and stop execution when program counter of the CPU reaches that address.

I have tested this lower-level approach with QEMU x86 on a few applications. This may not classify as "hardware" debugging, but the GDB client communicates directly with QEMU instead of the OS that is running on the emulator. Once QEMU is started with this option, it waits for a remote connection from GDB on a predefined port. GDB can then connect to this remote target.

To enable hardware debugging in QEMU, you can add the following code to CMakeLists.txt file in your project:

Listing 4.5: Modification of CMakeList.txt

```
if(BOARD MATCHES "qemu_x86")
    list(APPEND QEMU_EXTRA_FLAGS -s -S)
endif()
```

These commands append the options `-s -S` to the command shown in listing 4.4. After starting the simulator, open a new terminal window and enter:

Listing 4.6: Connecting GDB to QEMU

```
gdb build/zephyr/zephyr.elf
target remote localhost:1234
```

The first command loads GDB with symbol table of the Zephyr system executable. The second, entered to GDB shell, attaches the debugger to QEMU.

I have not found any problems with this configuration. When used with QEMU monitor, this is a very convenient and powerful debugging tool.

4.5.2 GDB stub

GDB stub is an implementation of the GDB Remote Serial Protocol (RSP). It defines a communication protocol between a server, a system or application that implements RSP, and a client, which is the GNU Debugger. Zephyr provides the GDB stub, which, at the time of writing, supports only the serial backend and two architectures — **x86** and **Xtensa**. It should be noted, that starting Zephyr on QEMU with *west* sets up several communication interfaces and the serial connection used for the stub is redirected to a networking port (5678 by default).

I have tested GDB stub with a couple of applications only on the QEMU emulator, as it does not appear to be supported on the Raspberry Pi Pico yet.

The debugger works well for simple applications, but sometimes gets stuck and unresponsive. For example, when stepping deep inside the system through nested function calls or calling a synchronization method in a multithreaded application, the system freezes. I tried debugging the GDB stub code with QEMU hardware debugging, but have not found any bugs. The issue could be related to the HPET problem mentioned before.

Chapter 5

Adding support for MicroZed board

This chapter describes the process of porting Zephyr to the MicroZed board. As mentioned in the introduction, this board is not officially supported by the OS, but the Zynq-7000 SoC is.

The documentation recommends to base a new board on one that is already supported and has the same SoC. At the time of writing, Zephyr officially supports one board with the Zynq chip, **Digilent Zybo** [26]. Our port thus derives from its configuration, modifies it and adds new functionality.

5.1 Zynq-7000 clocks

To configure some subsystems on the SoC, we need to manually specify various frequency-related values in the devicetree. Drivers parse these values at boot and adjust *clock dividers* to run the devices at a particular frequency. This section describes basic properties of the Zynq-7000 clock subsystem and the frequencies that our system is going to run with.

The clock subsystem in the SoC is driven by an external clock source connected to the PS_CLK pin, in our case a **33.333 MHz** oscillator. This clock signal drives three programmable **Phase locked loops** (PLL), electrical circuits, which are used to multiply the frequency of the source clock. Each PLL has multiple parameters adjustable by a memory-mapped register. One of these parameters is **feedback divisor**, which specifies the clock multiplier. The table 5.1 shows the PLL configuration on our board, along with their divisor values and output frequencies:

PLL	Feedback divider	Output frequency [MHz]
ARM PLL	40	1333
DDR PLL	32	1067
IO PLL	30	1000

Table 5.1: PLL configuration on MicroZed

It is important to note that Zephyr **does not adjust** any of these PLL registers or other core clock divisors. The values above are either the default register values or values that were later in the boot process adjusted by a second stage bootloader, such as U-Boot. Although Zephyr does provide a *clock control* API [27], no driver is implemented for the Zynq-7000 SoC yet.

All clocks in the SoC are derived from one of the PLLs by dividing their frequency. One of the most important clock *clock domain* on the SoC is the **CPU clock domain**. It is driven by one of the PLLs. The module divides the input frequency by an even number before dividing the frequency further to four outputs frequencies in a given ratio, either 6:3:2:1 or 4:2:2:1 (abbreviated **6:2:1** and **4:2:1** respectively). Layout of the CPU clock generator is shown in figure 5.1:

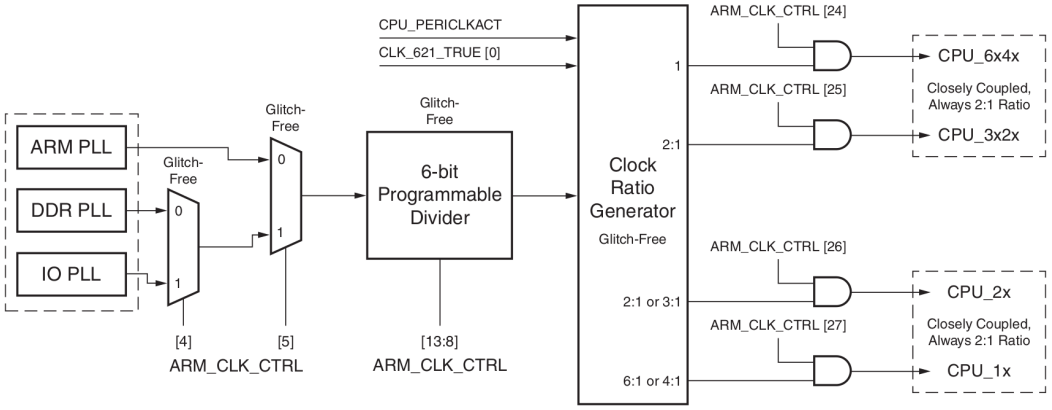


Figure 5.1: CPU clock divisor

In our case, the clock is routed from **ARM PLL**, the divisor in register ARM_CLK_CTRL is set to **2** and the clock ratio is **6:2:1**. All four outputs are enabled, table 5.2 outlines their frequencies.

CPU Clock	Ratio	Output frequency [MHz]
CPU_6x4x	6	666
CPU_3x2x	3	333
CPU_2x	2	222
CPU_1x	1	111

Table 5.2: CPU domain clock frequencies

The CPU clock handles the ARM processor and many subsystems, such as timers, on-chip memory controllers or interconnect busses [28].

5.2 The board directory

At this point, we can start to add various files that describe the MicroZed board to the Zephyr kernel tree. Here, I follow the board porting guide in Zephyr documentation [29]. The Zynq-7000 SoC is already supported in the OS and we can therefore start adding the files straight away.

Firstly, it is necessary to create a directory in the tree, which represents the board and make it available for west, the Zephyr build tool. As mentioned in the beginning of this chapter, the system officially supports one board with the Zynq-7000 SoC. We can thus start by copying its directory to **zephyr/boards/arm/** and renaming to **microzed**.

There are a couple of mandatory files in each board directory:

- **[board].dts** — Devicetree of the board. It usually includes devicetree of the SoC and defines interfaces and peripherals specific to the board.
- **Kconfig.board** — This file defines the Kconfig entry of our board. Listing below shows the contents for MicroZed.

Listing 5.1: Kconfig.board file contents

```
config BOARD_MICROZED
    bool "MicroZed board"
    depends on SOC_XILINX_XC7Z010 || SOC_XILINX_XC7Z020
```

The board depends on two Zynq-7000 SoC variants, on which we are going to test Zephyr and which are specified in the MicroZed TRM [4].

- **Kconfig.defconfig** — This file contains board-specific settings, which may depend on other configuration options. MicroZed defines the following options:

Listing 5.2: Kconfig.defconfig setting for MicroZed

```
if BOARD_MICROZED

config BOARD
    default "microzed"

endif # BOARD_MICROZED
```

The `BOARD` Kconfig option must be specified in this file.

- **[board]_defconfig** — These assignments are copied as is to the final Kconfig file during build:

Listing 5.3: Contents of microzed_defconfig

```
CONFIG_SOC_SERIES_XILINX_XC7ZXXX=y
CONFIG_SOC_XILINX_XC7Z020=y
CONFIG_BOARD_MICROZED=y

# Clock settings
CONFIG_ARM_ARCH_TIMER=y
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=333333333
# do not use the TTC as a kernel timer
CONFIG_XLNX_PSTTC_TIMER=n

CONFIG_PINCTRL=y
CONFIG_SERIAL=y
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
```

The file defines some essential configuration options. The general recommendations in the board porting guide suggest, among others, enabling serial output, console or pinmux (pin control) drivers. Although it also advises us to enable networking subsystems if supported, I did not include it in this primary configuration, as it considerably increases the size of the final Zephyr binary.

Other optional files usually include:

- **doc/index.rst** — A directory in a ReStructuredText file format, which describes the board and makes the page of the board in documentation.
- **doc/[board].png** — A picture of the board.
- **CMakeLists.txt** — A file for the CMake build tool that may specify additional code to the build. We do not use this file in the microzed directory.

5.3 Devicetree modification

We are going to modify the devicetree of the Digilent Zybo board in several ways. The board does not support some peripherals and subsystems that we are going to use, such as Ethernet or TTC. Other peripherals may be routed to different MIO pins of the SoC. The board also has a smaller RAM capacity and might have various clock settings.

The first task is to change some basic properties of the root node — the model name and the *compatible* property. The compatible property generally specifies the name and type of the device that the node represents. Device drivers usually parse this property to get the node they represent. The build system also uses it to find devicetree bindings at build time to check that the node specifies all required properties. The recommended value of the property is a string in the form *"vendor,device"* [30]. The modification of these two properties is shown in the listing below:

Listing 5.4: Basic devicetree properties

```
/ {
    model = "Avnet MicroZed board";
    compatible = "avnet,zynq-microzed","xlnx,zynq-microzed",
    "xlnx,zynq-7000";
}
```

The slash simply indicates that we modify properties of the root node. Secondly, we should adjust the RAM capacity. Our MicroZed board has got 1 GB DDR3 (1024 MB or 2^{30} B) memory, twice the size of memory on the Zybo board. The modified memory node is shown in the following code snippet:

Listing 5.5: Devicetree memory node

```
/ {
    sram0: memory@0 {
              compatible = "mmio-sram";
```

```
            reg = <0x0 DT_SIZE_M(1024)>;
    };
}
```

Here, `sram0:` is a *label* of the node. Labels are commonly used in devicetree to simplify references of one node to another. The property `reg` has a value of type *array* with values of type *(address, length)*. The addresses are generally **relative** to the parent node address space, in this case to the root node. The fist address in the array has to match the *unit address* `@0`. The `DT_SZIE_M` specifier is a bit shift macro defined by Zephyr.

5.3.1 UART and Pin Control

UART (Universal Asynchronous Receiver Transmitter) is a device, usually part of an SoC, which provides an interface to serial communication. The Zynq-7000 provides two, programmable UART controllers with various features, such as interrupts, parity and error detection, or variable baud rate.

The MicroZed board provides an USB-to-UART bridge, which appears as a COM port when connected to a computer. The bridge uses two MIO pins, specifically pins 48 and 49 [4], on the SoC for Tx and Rx (transmit and receive) signals. These pins correspond to **UART 1** controller on the SoC.

It is important to note, that the two UART controllers may be assigned to many different MIO pins. As a result, there is a devicetree binding rule in Zephyr requiring a **pin control node** reference in the UART node. Pin control node generally specifies pins used by a given device and some electrical characteristics of hardware **pin controllers**, such as pull-up, voltage level or slew rate [31]. High-level diagram of a pin controller is shown in figure 5.2.

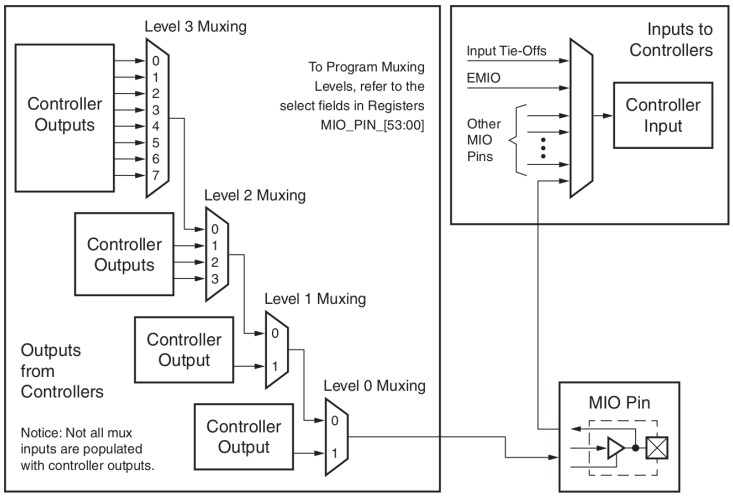


Figure 5.2: Zynq Pin Controller

As we can see, each of these controllers is able to multiplex many inputs, or serve as an input. On Zynq-7000, each pin controller has its own memory mapped register, a **SLCR** (System Level Control Register). The pin control driver parses the node

in devicetree and sets appropriate parameters in the SLCR for each pin, that needs to be configured.

The pin control node is already implemented with the correct pin assignment in the Zybo board directory. There is also an example of such node in the binding file for pin control nodes. To finish the UART devicetree node, we have to specify the **UART reference clock**. This value is used by the UART driver to set the correct divisor constant and get the correct baud rate. We can observe from U-Boot and the value of the `slcr.UART_CLK_CTRL` register that:

- The clock source is **IO PLL**.
- Frequency divisor in UART is set to **20**.

As outlined in table 5.1, frequency of IO PLL is 1000 MHz in our configuration and the UART reference clock is therefore **50 MHz**. The listing below shows the final configuration of the UART node.

Listing 5.6: UART devicetree node

```
&uart1 {
    status = "okay";
    current-speed = <115200>;
    clock-frequency = <50000000>;
    pinctrl-0 = <&pinctrl_uart1_default>;
    pinctrl-names = "default";
};
```

The expression `&uart1` is a *phandle* and references a node with label `uart1`, which is defined in the SoC devicetree. By setting value "okay" to the property `status`, we are overriding its default value "disabled". This is a very common trick, which lets us easily enable and disable various subsystems or peripherals. A node may also have more pin control nodes assigned (numbered from 0), each with different name and different configuration. For example, the node could have a "normal" and "power save" pin configuration.

5.3.2 LED and push button

LED and button nodes are already implemented in devicetree of the Zybo board. Pin numbers of these peripherals have to be changed though, as they are wired differently on MicroZed.

According to the MicroZed TRM, the LED and button are connected to pins **47** and **51** respectively [4]. However, the pin numbers in devicetree have to be **relative** to the **GPIO bank** that controls them. The GPIO subsystem in Zynq-7000 is divided into four banks, each controls a portion of the MIO or EMIO (routed to FPGA) pins. The 54 MIO pins are assigned to the first two banks — pins $[0, 31]$ to bank 1 and $[32, 53]$ to bank 2. Our peripherals are therefore connected to pins **15** and **19** of GPIO Bank 1. The modified nodes are listed below:

Listing 5.7: LED and button nodes

```
/ {
    aliases {
        led0 = &ld_mio;
        sw0 = &btn0;
    };
    leds {
        compatible = "gpio-leds";
        ld_mio: led_mio {
            gpios = <&psgpio_bank1 15 GPIO_ACTIVE_HIGH>;
            label = "LD_MIO";
        };
    };
    gpio_keys {
        compatible = "gpio-keys";
        btn0: btn0 {
            gpios = <&psgpio_bank1 19 GPIO_ACTIVE_LOW>;
            label = "BTN0";
        };
    };
};
&psgpio {
    status = "okay";
};
```

The node `psgpio_bank1` is already implemented in devicetree of our SoC, along with its parent node `psgpio`. The node `alias` assigns additional name tags to nodes, which makes it easier to reuse programs and refer to the nodes from code. For example, the alias `led0` is used in many Zephyr sample programs that use the LED. The value `"okay"` again overrides the default value `"disabled"` of `status` property in node `psgpio`.

5.3.3 Triple Timer Counters

TTC (Triple Timer Counters) is a timer module, which contains three independent, programmable timers. The Zynq-7000 SoC has two TTC modules. The timers are able to generate interrupts as well as route the output signal to a MIO or EMIO pin.

Neither the devicetree of the Zybo board or devicetree of the SoC have a node for the TTC. Although Zephyr does provide a driver for Xilinx TTC, it implements the system timer driver API [32], which offers relatively limited functionality. Additionally, Zephyr uses the ARM CPU private timer by default, even if TTC is available. I have added two nodes to the MicroZed devicetree, one for each module — **TTC 0** is going to be used by user applications and **TTC 1** is available for the optional TTC system timer. The following listing shows a node for TTC 1:

Listing 5.8: TTC 1 node

```
/ {
    soc {
```

```
        ttc0@f8001000 {
            compatible = "user,ttcps";
            status = "okay";
            reg = <0xf8001000 0x1000>;
            clock-frequency = <111111111>;
            interrupt-names = "irq_0", "irq_1", "irq_2";
            interrupts = <GIC_SPI 10 IRQ_TYPE_LEVEL IRQ_DEFAULT_PRIORITY>,
            <GIC_SPI 11 IRQ_TYPE_LEVEL IRQ_DEFAULT_PRIORITY>,
            <GIC_SPI 12 IRQ_TYPE_LEVEL IRQ_DEFAULT_PRIORITY>;
        };
    };
};
```

We are going to use an internal clock source for the timer, which is **CPU__1x** for both TTC. The property `clock-frequency` is therefore set to **111.111111** MHz, see table 5.1. According to the Zynq TRM, the TTC 0 timers correspond to interrupts **44**, **43** and **42**. However, similarly to the interrupt ID of the button and LED, the numbers have to be **relative** to the corresponding **SPI** (Shared Peripheral Interrupt) register. The 60 SPI interrupt numbers $[32, 92]$ are split between two 32-bit registers, thus the relative IDs of our timers are **10**, **11** and **12**.

The value `"user,ttcps"` in the TTC 0 node may be arbitrary, since we are making a simple driver for it. The `compatible` property in the TTC 1 node has a value `"xlnx,ttcps"`, as defined by the system driver. Once these nodes are defined in the devicetree, it is necessary to **map** the memory mapped registers to virtual address space in order to program the timer.

Listing 5.9: TTC driver initialization

```
// the matching compatible property
#define DT_DRV_COMPAT user_ttcps
DEVICE_MMIO_TOPLEVEL_STATIC(my_driver,DT_DRV_INST(0));
// virtual address of the mapped registers
#define BASE_ADDR DEVICE_MMIO_TOPLEVEL_GET(my_driver)

static int ttc_driver_init()
{
    DEVICE_MMIO_TOPLEVEL_MAP(my_driver,K_MEM_CACHE_NONE);
    // TTC initialization...
}
```

The listing above shows a part of our driver to map the registers of TTC 0 at runtime. The macro `DT_DRV_COMPAT` specifies the `compatible` property that the device or devicetree related macros use. It is commonly used throughout Zephyr, as it is more convenient than specifying the value in each subsequent macro. The set of macros `DEVICE_MMIO_TOPLEVEL` do **not** use the **Zephyr device model** [33], which offers a fully-featured but more complex interface. However, since we only need to map the registers and call the `ttc_driver_init` function, these macros are sufficient. The `BASE_ADDR` macro stores the virtual address of the TTC 0 register block.

5.3.4 Ethernet

Ethernet is a standardized family of network protocols and technologies, which operate at the physical and link layer of OSI model. The Zynq-7000 SoC provides two programmable Gigabit Ethernet Controllers (GEM). On MicroZed, the controller **GEM 0** is routed to ten MIO pins, which are wired to an external **PHY** (Physical layer device) chip. This chip connects directly to the RJ-45 port.

Zephyr supports a a wide range of Ethernet features, such as half/full duplex, MAC address filtering, 10/100/1000 Mbit link speeds or checksum offloading [34]. Drivers for the Zynq-7000 GEM controllers are implemented as well. The Zybo devicetree does not include any Ethernet configuration, although the board does support networking. The Zynq-7000 devicetree configures many, but not all required GEM properties. According to a devicetree binding for the Xilinx GEM, we must specify the following properties:

- `clock-frequency` — Frequency of the PLL supplying TX clock of the GEM. By default, this is the **IO PLL** and applies to our configuration too. The driver parses this value at run-time and adjusts two clock dividers inside GEM. The resulting frequency is configured to match the link speed that the PHY device negotiated.
- `mdc-divider` — The **MDIO** (Management data input output) interface clock divider. This interface is used to program the PHY and is accessed through a PHY maintenance register of the specified GEM. It consists of two signals — MDIO data and MDIO clock (MDC). On MicroZed, these lines are part of **RGMII** (Reduced Gigabit Media Independent Interface), which has additional 12 wires, 6 for transmit and receive signals, and connects the external PHY to the SoC. On Zynq-7000, the MDC is generated by dividing the **CPU_1x** clock and must not exceed **2.5 MHz** [28]. However, the divisor supports only values $\{8, 16, 32, 48, 64, 96, 128, 224\}$. In our case, CPU_1x runs at 111 MHz and the value **48** divides the frequency to 2.31.
- `local-mac-address` - Although this property is not required by the binding, the driver depends on a macro derived from it. If this is not defined in the devicetree, the macro is not declared and the kernel does not compile.

The snippet below shows the GEM 0 node in `microzed.dts`:

Listing 5.10: Ethernet controller node

```
&gem0 {
    status = "okay";
    clock-frequency = <1000000000>;
    mdc-divider = <XLNX_GEM_MDC_DIVIDER_48>;
    init-mdio-phy;
    local-mac-address = [ c6 8c 2b 3d 18 1e ];
};
```

The property `init-mdio-phy` is not specified either in the SoC devicetree and is set to `true` this way. It instructs the driver to not only configure the GEM, but

to also initialize the PHY using a designated driver. Zephyr includes drivers for a couple of PHY models that run alongside a Xilinx SoC. One of the supported PHY devices is Marvell Alaska 88E1512, which is used on MicroZed.

It is important to note that in the Zephyr 3.3.0 release, which is the most recent at the time of writing, PHY device initialization on MicroZed does not work. U-Boot initializes the PHY device correctly, since we are able to load the Zephyr binary over TFTP. Once Zephyr boots, the driver identifies the PHY device and starts the initialization. However, it does not respond to a polling function, which waits for the PHY to negotiate the link speed — this ultimately prevents any higher-level networking application from functioning. Furthermore, if the board is reset using the designated button, the bootloader cannot initialize the PHY anymore. Eventually, the board had to be physically disconnected from a power source and turned on again. U-Boot then configured the PHY device correctly and the whole process would start again. These issues turned out to be caused by the following two bugs:

- The reset button does not trigger a hardware reset in the PHY device. According to errata for MicroZed revision G [35], the resistor and capacitor values on the board do not reset the chip for 10 ms as required, but merely for 4.7 ms.
- The Xilinx PHY driver in Zephyr had a wrong value in a predefined macro. One of the steps when setting up the PHY device is to choose an *operating mode* — in our case, this is **RGMII to copper**. This mode is set using the MDIO interface in a register field in the PHY device, specifically bits 2:0 in register 20 of page 18 (General Control Register 1) [36]. The hardware reset value of this field is **111** for our PHY model and the value that corresponds to our required mode is **000**. Therefore, the driver simply cleared the bit field (set to the negation of the bit mask) in the register. However, instead of **111**, the bit mask was, likely by mistake, defined as **011**. This meant that the register field was set to **100**, which corresponds to mode **RGMII to SGMII**. This way, the PHY device did not negotiate the link speed and never reported it to the driver callback function. Additionally, the device was not affected by the button reset and would keep this setting after the SoC reset.

With the help of my supervisor, we created a pull request that fixes this macro [1]. As of 28. April 2023, it has been merged into the main branch of Zephyr kernel.

5.4 U-Boot and Zephyr booting

U-Boot (or Das U-Boot) is a free and open source **bootloader** (first and second stage) commonly used in embedded systems [37]. It supports many architectures, SoC and standalone development boards, MicroZed and Zynq-7000 SoC included.

Generally speaking, a processor starts executing code in an on-chip read-only memory (often called **BootROM**) once power is applied. This program probes other memory devices, such as a NAND flash or an SD card, for a **boot image**, which may be a **first** stage bootloader like U-boot.

[1] https://github.com/zephyrproject-rtos/zephyr/pull/56361

Once U-Boot starts up, it offers a **shell** over a serial connection. The bootloader comes with many predefined utilities, which are accessible through the shell. These include programs for memory access, device control, network file transfer or, most importantly, booting higher-level OS images. The process of replacing a running program by a new one is sometimes called **chain loading**. U-Boot may be also configured using persistent storage of the device and various environment variables to execute commands automatically at boot.

We cross compiled U-Boot for the MicroZed board from a **Xilinx fork** [38] of the official U-Boot GitHub repository. We used version **2022.2**, which is the most recent at the time of writing. Two of the compiled files, `u-boot.img` and `spl/boot.bin`, have to be copied to a **microSD card**, which we use as a boot device.

Once the card is inserted in the board, connect it to a computer via the MicroUSB port. Since MicroZed uses this port also for primary power delivery (and for the USB-UART converter), the board starts the boot process. Now we can open up a serial port terminal and attach to the corresponding interface (a character special file in Linux `/dev` directory). We use the open source program GTKTerm [39].

To transfer a Zephyr image (file `zephyr.bin` in an application build directory) to the board, we may use one of the following approaches:

- Copy the file to the microSD card — The image may be copied to the card alongside the bootloader files and then accessed from U-Boot. After U-Boot initializes, type the following command in its shell to load the binary to memory:

  Listing 5.11: Loading Zephyr image from SD card

  ```
  Zynq> fatload mmc 0 0x0 zephyr.bin
  ```

- Transfer the file with TFTP — U-Boot supports **TFTP** (Trivial File Transfer Protocol) to load images of operating systems to memory from a remote TFTP server. This server has to be therefore available on the network MicroZed is connected to. I have set up a simple TFTP server on my computer to share contents of directory `/srv/tftp`. After a compiled Zephyr binary is copied there, it may be loaded over LAN as:

  Listing 5.12: Loading Zephyr image over TFTP

  ```
  Zynq> tftpboot 0x0 ${serverip}:/srv/tftp/zephyr.bin
  ```

  Here, `serverip` is a predefined environment variable that I use to store the IP address of my computer. This process is clearly more flexible than the previous, as the microSD card does not have to be exchanged each time Zephyr is re-compiled.

The binary file loaded in RAM may be now executed and the system should boot. The listing below shows startup for the Hello World sample application:

Listing 5.13: Starting Zephyr from U-Boot

```
Zynq> icache off; icache flush; dcache off; dcache flush; go 0x0
```

```
## Starting application at 0x00000000 ...
*** Booting Zephyr OS zephyr-v3.3.0-1994-g177434ef845f ***
Hello World! microzed
```

The commands `icache` and `dcache` are used to disable and clear the instruction cache and data cache respectively. Zephyr needs to be started up with **caches disabled** when chain loading, otherwise it crashes during boot due to various peculiar errors. This is perhaps the most significant drawback of the current version of Zephyr - when running on 32-bit ARM processor, the system **does not re-enable the caches**, as it does not yet have a suitable driver. As a result, Zephyr performs much worse that what the Zynq-7000 SoC is capable of.

It is important to note that the Zephyr documentation does not appear to summarize any of the information regarding caches or chain loading from U-Boot. I had to contact a couple of contributors to the system (one of which ported the Zybo board) for clarification.

5.5 Performance and latency benchmarks

This section describes a couple of benchmarks to evaluate the key aspects of our Zephyr port. To compare the results, we ran Debian Jessie with a modified Linux **4.9.9** kernel on the MicroZed board and executed the same tests.

5.5.1 Ethernet bandwith

Zephyr includes a sample program **zperf** [40], which is implemented as an optional utility in the Zephyr shell. The program is compatible with **Iperf 2.0.5** [41], a cross platform application to measure the maximum bandwith and reliability of IP networks using UDP or TCP protocols. In a simple case, the program runs on two computers connected over a network, where one acts as a **server** and the other one as a **client**. The client sends data packets to the server and chooses parameters such as the protocol, server address, packets size and test duration. The server listens for packets of the specified protocol and a given port.

I have tested the bandwith between the Microzed board and my workstation computer on a LAN network. The computer and the board were connected with a UTP cable and a gigabit switch. Both UDP and TCP protocols were tested, each with the board acting as a server and as client. Table 5.3 shows the four test configurations and their results.

Protocol	MicroZed mode	Bandwith [Mbit/s]	Data transfer [MB]
UDP	Client	12.5	29
UDP	Server	10.7	26.3
TCP	Client	8.54	19.9
TCP	Server	10.3	24.7

Table 5.3: Ethernet bandwith of Zephyr on MicroZed

The duration of all benchmarks was to **20 seconds** and the datagram size was 1024 bytes, the default value. It is important to note that the second benchmark,

MicroZed acting as server and receiving UDP packets, had to be **limited to 11 Mbits/s**. Higher speeds resulted in receive buffer overruns, because the CPU could apparently not keep up with the incoming data stream. However, this caused several other errors related to the receive buffer, most likely due to a bug in the Ethernet driver. Although the shell was responsive, the network subsystem did not answer to UDP, TCP or ICMP packets anymore.

To measure the bandwith on Linux, we installed Iperf 2.0.5. The table 5.4 shows the results.

Protocol	MicroZed mode	Bandwith [Mbit/s]	Data transfer [MB]
UDP	Client	1.05	2.5
UDP	Server	1.05	2.5
TCP	Client	150	357
TCP	Server	564	1310

Table 5.4: Ethernet bandwith on MicroZed running Linux

As we can see, the bandwith over a TCP connection is significantly higher than we measured on Zephyr. On the other hand, the UDP performance turned out to be a lot worse than I expected. The bandwith appears to be limited to 1.05 MBit/s, but I could not figure out why.

5.5.2 Ethernet latency

We used the **ping** program to measure the round-trip time (RTT) of packets on the same network configuration, as outlined in the bandwith test. It was configured to send 10^5 packets in an interval of **2 ms** (more than twice as long as a typical delay). This test allows us to see if the lower levels of Zephyr network stacks efficiently handle the incoming packets and whether it adequately performs under load. Table 5.5 shows the measured latency statistics.

RTT values	Time [ms]
Min	0.472
Average	0.657
Max	1.308
Standard deviation	0.043

Table 5.5: Round-trip time statistics measured with ping

The chart 5.3 also displays a round-trip time histogram of all packets.

5.5.3 Single thread performance

As a rough indication of single-threaded performance on Zephyr, I created a small program that implements **Sieve of Eratosthenes**, a simple algorithm for finding prime numbers up to a given limit. We set this bound to 10^7. The listing below shows the function that implements this algorithm:

Listing 5.14: Implementation of Sieve of Eratosthenes

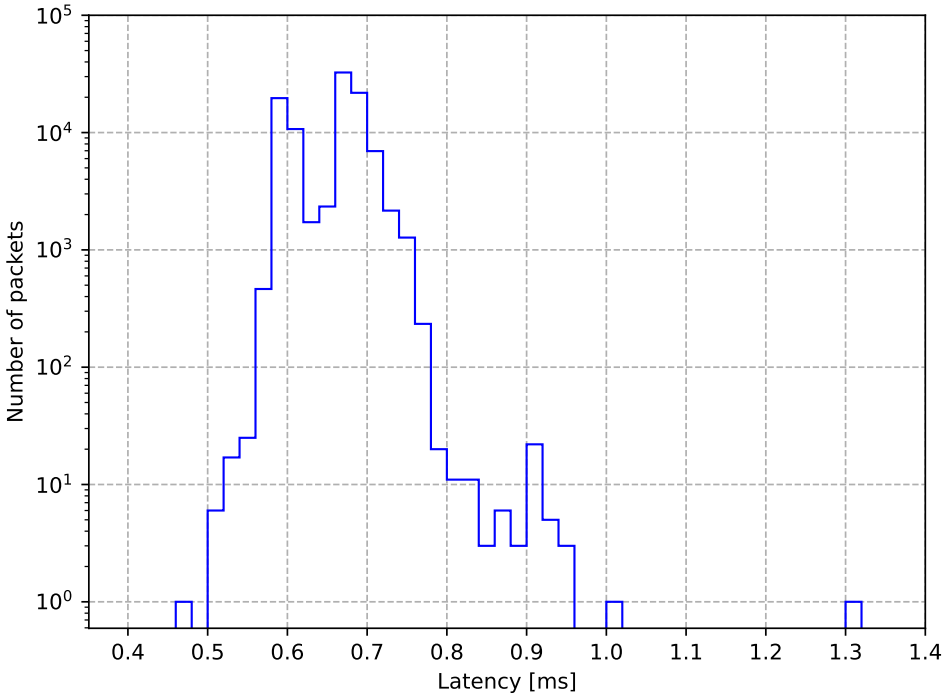


Figure 5.3: Histogram of packet latency

```
void sieve(int num) {
    memset(isPrime, 1, NUM + 1);
    for (int i = 2; i * i <= NUM; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j < NUM; j += i) {
                isPrime[j] = 0;
            }
        }
    }
}
```

The program measures the time (using hardware clock timestamps) to run this function and to count the number of prime numbers found afterwards (664579 primes from 2 to 10^7). It is important to note that the size of the Zephyr kernel address space had to be manually increased. Otherwise, the system image, containing the long, statically allocated `isPrime` array, would be larger than the default address space and it would crash during boot, see the section 3.2.3. The time to run the function `sieve()` and to count the primes was **1.436** seconds.

Similarly to the section 5.5.1, we statically compiled and ran this program with minimal changes on Linux. According to the `time` command, the program took **1.008** seconds to complete. However, this time may also include other tasks, such as loading the file to memory, than the `sieve` function and counting the primes.

Chapter 6

GDB stub porting

This chapter outlines the development of a basic GDB stub utility in Zephyr for the 32-bit ARM architecture. As mentioned in the section 4.5.2, Zephyr currently implements GDB stub only for the 32-bit x86 and Xtensa architectures.

6.1 GDB remote serial protocol

As mentioned in section 4.5.2, GDB Remote serial protocol (RSP) defines how the **host**, a computer that GDB runs on, and **target**, the device we are trying to debug, communicate. All data is exchanged in **packets**, a finite sequence of characters, which has the form `$packet-data#checksum`. The *packet data* field generally begins with one or more characters specifying the **command** and may be followed by data that the command uses. Any binary data is encoded as **two hexadecimal** characters per byte. The packet is ended with a checksum, which is the sum of all characters inside the data field modulo 256 — this byte is again encoded as two hexadecimal characters [42].

The user commands accepted by the shell on GDB host are translated to one or more RSP packets with commands for the target (from now on, *command* refers to the character inside a packet). GDB defines dozens of these commands, but only requires the target stub to implement at least the following five commands [43]:

- **g** and **G** — Commands to read and write general purpose registers. See the section 6.3.2 for details and the implementation in Zephyr.
- **m** and **M** — Read and write bytes from the memory.
- **?** — Send the code of the last exception, which caused the program to enter the stub.

The stub implementation in Zephyr also supports a couple of additional commands:

- **p** and **P** — Read and write individual registers.
- **c** — Continue execution.
- **s** — Step one machine instruction. The section 6.3.4 describes this feature for 32-bit ARM architecture.

- **z** and **Z** — Remove and insert a breakpoint of a particular type. The implementation of breakpoints is again outlined in section 6.3.3.

The protocol can use many underlying technologies for communication, such as Ethernet or serial. As mentioned in section 4.5.2, Zephyr only supports serial backend as of release 3.3.0 [44].

6.2 MZ_APO Kit

Since Zephyr does not yet support Ethernet backend for GDB stub, we have to use a dedicated UART port for the RSP. However, our standalone MicroZed board connects only the UART 1 module, which we already use for system console. We are therefore using the MZ_APO board throughout this chapter, which has an additional port available.

To enable the UART 0 port in Zephyr, we need to modify the devicetree of the MicroZed board and the pin control settings. The peripheral board contains a USB to UART converter, which is wired up through the Microheader expansion ports to MIO pins **10** and **11** for **RX** and **TX** signals respectively [5]. Thus we can duplicate the UART 1 pin control node `uart1-default` in the `microzed-pinctrl.dtsi` file and change the pin numbers accordingly. The UART node in the `microzed.dts` file needs to be copied and renamed as well, with the only modification being a different reference to the new pin control node. Additionally, it is necessary to enable **clocks** for this module. We have added the following code snippet to the initialization function of UART driver:

Listing 6.1: Enabling UART 0 clock

```
// turn on the AMBA clock for both UART controllers
reg_val = sys_read32(0xF800012C);
reg_val |= 0x300000;
sys_write32(reg_val, 0xF800012C);
// set the ref clock - turn on UART 0 reference clock
reg_val = sys_read32(0xF8000154);
reg_val |= 0x1;
sys_write32(reg_val, 0xF8000154);
```

The first three statements enable the **AMBA** (Advanced Microcontroller Bus Architecture) clock for UART 0 by modifying a peripheral clock control register (one of SLCR registers). Otherwise, any access to memory-mapped registers of UART 0 would not actually adjust them. Afterwards we turn on reference clock for the module itself. It appears that the driver expects these clocks to be set up by a bootloader, but I could not manage to configure U-Boot to enable the UART 0 port.

Since this educational shield is assembled in an enclosure of acrylic, each kit comes with a MicroZed board. However, these boards are a cheaper version with **7Z010** SoC, as opposed to the 7Z020 SoC that has been used in the previous chapter. As a result, the option `CONFIG_SOC_XILINX_XC7Z020` in the `microzed_defconfig` file needs to be replaced with `CONFIG_SOC_XILINX_XC7Z010`.

6.3 Stub functionality

This section describes the development of ARM-specific GDB stub functionality in Zephyr, as well as some issues that are yet to be resolved. We follow the main steps for GDB stub in the Zephyr architecture porting guide [45].

Zephyr already contains the core utilities of GDB stub, which are common for all architectures. The fundamental part of this module is the **loop**, which listens for packets, calls the appropriate functions and replies the host. The loop is typically entered upon generating an exception, as we want to be notified by the target in such case. An exception might be also generated by hitting a breakpoint or a watchpoint. Although the target system appears to be stopped to the host (and the thread that generated the exception is indeed halted), the system is running this loop and executes the commands. In case the exception was not fatal, the system can resume the program by simply returning from the loop.

The stub module also contains functions for reading and writing memory. These functions are called from the stub loop in response to the **m** and **M** packets. As mentioned previously, binary data is transmitted in the packets as a sequence of hexadecimal characters. The functions therefore convert this string to and from binary data. It is important to note that the stub does **not** change the **endiannes** - the GDB host sends all data in the target byte order. The Zynq-7000 supports only **little-endian** ordering for memory and instructions [28].

The following subsections describe the implementation of several architecture-dependent functions, which are called from the loop and that our stub must define.

6.3.1 Entering and exiting the loop

The most important architecture-dependent function to implement is `arch_gdb_init()`. The function has to stop the calling thread and allow the GDB stub to connect to the host. It also needs to initialize any functionality the system needs for debugging.

The function is called from a wrapper function `gdb_init()`, which is defined in the core stub module. This function is attached to one of the Zephyr runlevels, level `PRE_KERNEL_2` by default. It initializes the serial backend for GDB and, if successful, calls the `arch_gdb_init()` function.

To stop the running thread and initialize debugging, the function calls a `BKPT` instruction using inline assembly. This instruction, sometimes called a **software breakpoint**, causes the processor to generate a **software debug event**. These kinds of debug events (as opposed to **halting** debug events) may be also generated by other conditions, such as hardware breakpoints or hardware watchpoints. How the CPU reacts to debug events generally depends on the setting of **debug mode**. Briefly speaking, the CPU may ignore such event, enter a special state (debug state, which is used to halt the processor completely) or generate an exception. The instruction `BKPT` generates a **Prefetch Abort exception** regardless of the debug mode.

An exception causes the processor to stop the process, which generated the exception in the first place, and call a corresponding **exception handler**. This function, most importantly, saves the general purpose registers of this process to stack. Zephyr

defines higher level exception handlers (written in C as opposed to the primary handlers in assembly) for 32-bit ARM, which take an argument of type `stuct esf`. This is a C structure defined in Zephyr whose variables are the registers saved on the stack. See the next subsection about register details.

In the most recent version of Zephyr, these high-level handlers do nothing more than log the exception or print a core dump. In other words, each exception is **fatal** and the process does not resume. Our stub implementation uses some of these handlers to finally call a function `z_gdb_entry()` defined by the stub. This function prepares some variables before entering the loop, like specifying the **cause** for the exception (the stub loop requires such variable). Likewise, `z_gdb_entry()` restores, potentially **modified**, set of registers to `stuct esf` and returns to the handler.

After this function returns, the handler function returns a non-fatal return value. The primary, lower level handler then loads the registers from stack and returns control to the process. Entering and exiting the stub loop is outlined in the diagram 6.1.

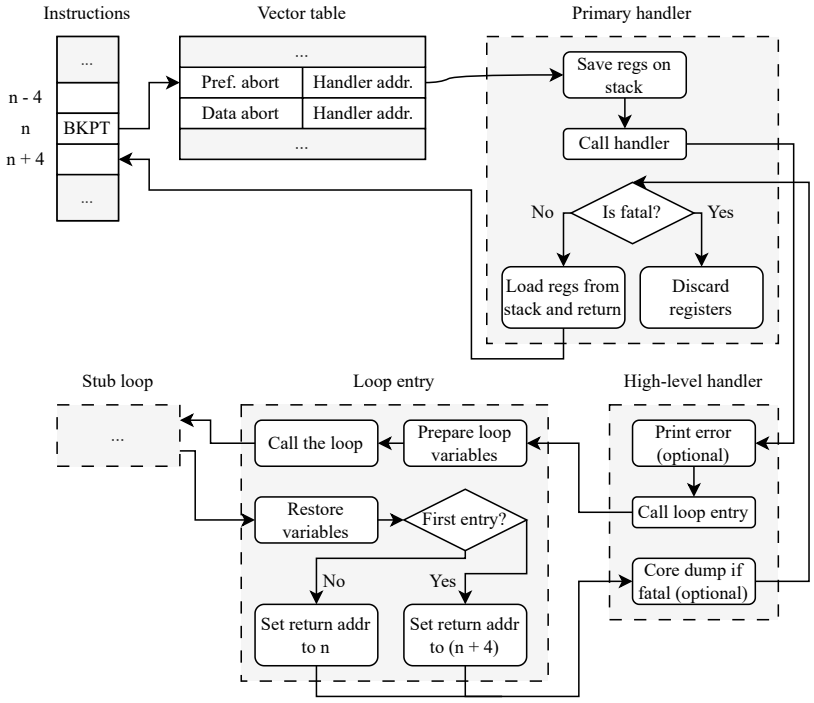


Figure 6.1: Function hierarchy of entering GDB stub

6.3.2 Register access

The stub implementation must also define four functions for register access - two functions to read and write all registers at once (**g** and **G** commands) and two for individual register modification (**p** and **P** respectively). This section does not describe our implementation of these functions, but summarizes the registers, which GDB stub uses.

The GDB host may accept or ask for up to **26** registers on 32-bit ARM. GDB defines the following array in its source code file `arm-tdep.c`:

Listing 6.2: ARM registers used in GDB

```
 static const char *const arm_register_names[] =
{"r0",  "r1",  "r2",  "r3", /*  0  1  2  3 */
 "r4",  "r5",  "r6",  "r7", /*  4  5  6  7 */
 "r8",  "r9",  "r10", "r11",    /*  8  9 10 11 */
 "r12", "sp",  "lr",  "pc", /* 12 13 14 15 */
 "f0",  "f1",  "f2",  "f3", /* 16 17 18 19 */
 "f4",  "f5",  "f6",  "f7", /* 20 21 22 23 */
 "fps", "cpsr" };       /* 24 25       */
```

Firstly, I have not been able to find the positions or sizes of some of these registers anywhere else. The official documentation refers to internal GDB functions and variables [43]. This array specifies position of each register in the **g** and **G** packets.

The registers `r0` to `r15` are referred to as **ARM core registers** [13]. These may be further subdivided into **general purpose** registers, `r0` to `r12`, and **special purpose**, which include `sp`, `lr` and `pc` (abbreviations for stack pointer, link register and program counter). The `CPSR` (Current program status register) stores various processor condition and control information. All of the registers mentioned in this paragraph are **4 bytes** long.

The remaining registers, `f0` to `fps`, used to be a part of **FPA** (Floating Point Accelerator), a coprocessor with an instruction set. It was allegedly used only in a few processors released around the year 2000 and has not been implemented since then [46]. The ARMv7 manual does not mention these registers either and GDB labels it as obsolete in other source code files. Hence, our stub implementation **ignores** these registers. Unlike the other registers, each of them, except `fps`, is **12 bytes** long.

Our stub implementation therefore handles **17** registers in total. It is important to note that the primary exception handlers in Zephyr save and restore **only 8** registers by default - `r0` to `r3`, `r12`, `lr`, `pc` and `cpsr`. The rest may be included with the Kconfig option `CONFIG_EXTRA_EXCEPTION_INFO`. However, these additional registers behave as **read only** for higher-level handlers, as they are not restored by the handler at exception return.

6.3.3 Breakpoints

The stub porting guide mentions, but does not require, the implementation of functions `arch_gdb_add_breakpoint()` and `arch_gdb_remove_breakpoint()`. These functions set and remove **hardware** breakpoints in response to **Z** and **z** commands. By default, these functions are defined in the core stub file, but only return a negative value (meaning not implemented error), as GDB uses software breakpoints by default.

Our stub does not override these functions and GDB relies solely on software breakpoints. The primary reason is the limited amount of hardware breakpoint registers - **6** registers on Cortex-A9 processors [47]. However, the code segment in

the Zephyr image is **read only** by default and software breakpoints, which inherently have to modify the code instructions, would generate a permission fault. Thus, the write permission flag has been added to the ARM MMU code that describes this segment. Nonetheless, we are using hardware breakpoint for **single stepping**, see the next section 6.3.4 for details.

If the user tries to set a breakpoint (breakpoint refers to software breakpoint in this section) at a particular instruction, GDB **replaces** this instruction with another one and saves the original instruction. In our case, GDB uses the `TRAP` instruction (encoding `0xe7ffdefe`). It is important to note, that neither the ARMv7 reference documentation or the vast majority of online manuals **do not specify** this instruction! It is perhaps an old, unsupported instruction, which is still used by GDB. The target processor thus generates an **undefined instruction** exception once it reaches the breakpoint address. As soon as the GDB loop starts and sends this exception to the host, the host orders the stub to replace `TRAP` back to the original instruction. However, the process still has to execute this instruction and the `TRAP` instruction has to be **written again**, because hitting a breakpoint does not delete it by default. Once user commands the debugger to continue, the stub **steps** a single machine instruction - the original instruction is executed, but a prefetch abort is generated on the next one. The GDB is notified and swaps the former instruction back to `TRAP`. It then continues the execution without opening a shell for the user. The result is that the instruction we set the breakpoint on has been executed, but the `TRAP` instruction is still written on its place.

The GDB consequently relies only on memory access commands in case of software breakpoints.

6.3.4 Single stepping

Single-stepping generally refers to execution of **one instruction at a time**, before halting the process (or an entire CPU) and handling the control to a debugger. Zephyr requires the stub to implement function `arch_gdb_step()`, which is called from the loop after receiving the **s** packet. It must set up the system to execute the next instruction and return control to the loop.

Firstly, the section 6.3.1 mentions that the current **debug mode** of the processor determines the implication of a debug event. By default, debug mode is disabled and all debug events, **except BKPT**, are **ignored**. Hence, the function `arch_gdb_init()` changes this mode to **Monitor, privilege level PL0 or PL1**. This is the only debug mode, which does not ignore breakpoints or causes them to enter **debug state** (normally caused by halting debug events). In this mode, hardware breakpoints cause a **Prefetch Abort**.

We have implemented single stepping as described in section 'Use of instruction address mismatch breakpoints for single-stepping' in the ARMv7 reference manual [13]. Each hardware breakpoint in ARMv7 consists of two 4-byte registers, **DBGBCR** (Debug Breakpoint Control Register) and **DBGBVR** (Debug Breakpoint Value Register). One of the fields in BCR specifies the **type** of the breakpoint, which determines what these registers actually define for debug event generation. For example, in an **unlinked instruction address match** mode, BVR holds the address of the instruction that is supposed to generate the debug event. **Linked** breakpoints

are paired with others to define more complex conditions, but unlinked breakpoints are sufficient for this application. Our stub uses **unlinked instruction address mismatch** type, which triggers an event if the address of current instruction does **not** match the address in BVR. The BVR is thus set to address of the instruction, which caused the current exception. As soon as the process returns and program counter is incremented, the breakpoint throws Prefetch Abort.

The mismatch breakpoint also has to be limited to certain processor modes or address ranges, otherwise it would halt the CPU right after enabling the breakpoint in the `arch_gdb_step()` function. By default, the BCR is configured for **PL0, Supervisor and System modes only**, which is suitable for our debugger. Zephyr normally runs in **Supervisor** mode (Privilege Level 1), but switches to **Abort** mode (also PL1) upon taking a prefetch exception. By setting the BVR to the instruction, which caused the process to enter the stub in the first place, the process returns from the stub and breaks on the next instruction.

This breakpoint configuration works as intended when stepping through the early boot phase of Zephyr. However, there are still some issues with debugging the application code, see the section 6.5 for details.

6.4 Debugging

This section briefly describes the process of remotely debugging the MicroZed board running Zephyr.

Firstly, the system must be compiled with the kernel option `CONFIG_GDBSTUB` enabled. The MicroZed board also has to be connected with both UART ports to the host computer - UART 1 used for serial output and UART 0 used solely for GDB RSP.

The boot process is the same as described in section 5.4. Unlike normal startup, the kernel halts at boot and waits for the GDB connection. As mentioned previously, the Zephyr SDK includes a fair number of GNU binutils, such as a GCC, LD, objdump and GDB. We can therefore start the debugger and load symbols from the Zephyr image:

Listing 6.3: Starting GDB

```
/home/user/zephyr-sdk-0.15.0/arm-zephyr-eabi/bin/arm-zephyr-eabi-gdb \
    build/zephyr/zephyr.elf
```

The GDB should start and offer us a shell. Before we can start debugging, it is necessary to configure the serial line. Then we may simply attach the debugger to the corresponding interface:

Listing 6.4: Connecting GDB to MicroZed over serial

```
Reading symbols from build/zephyr/zephyr.elf...
(gdb) set serial baud 115200
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
arch_gdb_init () at /home/user/zephyrproject/zephyr/arch/arm/
core/aarch32/gdbstub.c:140
```

```
140     __asm__ volatile("BKPT");
(gdb)
```

I recommend using a `.gdbinit` file with these first two commands to automate the connection process. We can see that the system stopped at this `BKPT` instruction in our stub, entered the loop and was listening for a connection from host. GDB now accepts the commands and we can start debugging, see the user documentation for any tutorials or command reference [48].

6.5 Issues

This section describes the bugs related to our GDB stub and discusses potential solutions.

6.5.1 Mismatch breakpoints interfere with interrupt handlers

We have described the use of instruction mismatch breakpoints for single stepping in section 6.3.4. It was acknowledged, that the range of addresses, where the breakpoint triggers a debug event, would be ideally limited to the addresses of the program we want to debug (or to other conditions, such as the processor mode). In our case, the BCR (Breakpoint Control Register) is programmed to allow debug events only in **PL0, Supervisor and System modes only**.

Although this restriction works for debugging the application (running in **system** mode), the breakpoint also affects some **interrupt handlers**, which run parallel to the main thread and execute in **supervisor** mode. Instead of stepping to a next instruction of the user application, the breakpoint may therefore halt an interrupt handler. The **ARM timer** handler is a primary example, since the kernel always needs an internal timer.

We have also mentioned that debugging initial sections of the Zephyr kernel, after the stub is started, works as expected. This is because the interrupts are not enabled yet - once the initial process switches to the main thread (function `z_swap_unlocked()`) and **enables interrupts**, single stepping may be obstructed by the handlers.

We have tested the debugger on a simple, single-threaded application. When stepping through the main function, the process would sometimes step to a handler of the system timer. Although Zephyr runs a **tickless** kernel by default, it still generates an interrupt every 6 to 8 seconds.

Unfortunately, the BCR does not support a restriction for system mode only. I have discussed this issue with my supervisor and we do not know about any elegant solution to this problem.

Chapter 7

Conclusion

In this work, we have successfully ported Zephyr to the MicroZed board and implemented GDB stub for the 32-bit ARM architecture. We have submitted the following three contributions (pull requests) to the Zephyr code base:

- Bug fix in the Xilinx Ethernet driver, as described at the end of chapter 5.3.4. It was accepted and merged with the main branch.
- Adding support for the MicroZed board [1]. This pull request is currently blocked, because we also modify the mapping of SLCR MMIO registers for the Zynq-7000 SoC in order to enable Ethernet. The developers are still working on a standalone clock driver for this SoC, which would allow us to use its API instead of this temporary fix.
- Extension of the GDB stub for ARM [2]. This contribution still needs needs a large amount of feedback from other developers and testing on more ARM-based boards before being merged to the kernel.

Although the Zephyr project is promising, there is still a significant room for improvement, notably on less resource-constrained devices with high performance SoCs. Courses at FEE, CTU will therefore keep using conventional operating systems on MicroZed boards for the time being.

[1] https://github.com/zephyrproject-rtos/zephyr/pull/58066
[2] https://github.com/zephyrproject-rtos/zephyr/pull/58067

Bibliography

[1] G. C. Buttazzo, *Hard Real-Time Computing Systems*. Springer New York, NY, 2011.

[2] "The FreeRTOS kernel." `https://www.freertos.org/RTOS.html`. Accessed: 17-01-2023.

[3] *Devicetree Specification*, v0.3-40-g7e1cc17 ed.

[4] Avnet, Inc., *MicroZed™ Zynq® Evaluation Kit and System on Module Hardware User Guide*, 1.7 ed., 2017.

[5] "Přípravek MicroZed APO." `https://cw.fel.cvut.cz/wiki/courses/b35apo/documentation/mz_apo/start`. Accessed: 27 Mar. 2023.

[6] "Introduction." `https://docs.zephyrproject.org/latest/introduction/index.html`. Accessed: 18-01-2023.

[7] "Threads." `https://docs.zephyrproject.org/latest/kernel/services/threads/index.html`. Accessed: 18-01-2023.

[8] "Operation without threads." `https://docs.zephyrproject.org/latest/kernel/services/threads/nothread.html`. Accessed: 18-01-2023.

[9] "Scheduling." `https://docs.zephyrproject.org/latest/kernel/services/scheduling/index.html`. Accessed: 18-01-2023.

[10] "Memory protection design." `https://docs.zephyrproject.org/latest/kernel/usermode/memory_domain.html`. Accessed: 18-01-2023.

[11] "CONFIG_KERNEL_VM_SIZE." `https://docs.zephyrproject.org/latest/kconfig.html#CONFIG_KERNEL_VM_SIZE`. Accessed: 5 Apr. 2023.

[12] "CONFIG_MMU_PAGE_SIZE." `https://docs.zephyrproject.org/latest/kconfig.html#CONFIG_MMU_PAGE_SIZE`. Accessed: 5 Apr. 2023.

[13] ARM Limited, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, 0406c.d ed., 2018.

[14] "CONFIG_ARM_MMU_NUM_L2_TABLES." https://docs.zephyrproject.org/latest/kconfig.html#CONFIG_ARM_MMU_NUM_L2_TABLES. Accessed: 5 Apr. 2023.

[15] "Threat model." https://docs.zephyrproject.org/latest/kernel/usermode/overview.html. Accessed: 18-01-2023.

[16] "Shell." https://docs.zephyrproject.org/latest/services/shell/index.html#shell. Accessed: 18-01-2023.

[17] "console.h File Reference." https://docs.zephyrproject.org/latest/doxygen/html/console_2console_8h.html#a3454f5b84d38d46a6c2bbf7fd6baa815. Accessed: 18-01-2023.

[18] "MCUmgr." https://docs.zephyrproject.org/latest/services/device_mgmt/mcumgr.html. Accessed: 24 Apr. 2023.

[19] "Supported boards." https://docs.zephyrproject.org/latest/boards/index.html. Accessed: 18-01-2023.

[20] "Configuration System (Kconfig)." https://docs.zephyrproject.org/latest/build/kconfig/index.html. Accessed: 19. May 2023.

[21] "Setting Kconfig configuration values." https://docs.zephyrproject.org/latest/build/kconfig/setting.html. Accessed: 19. May 2023.

[22] "Getting Started Guide." https://docs.zephyrproject.org/latest/develop/getting_started/index.html. Accessed: 4 Apr. 2023.

[23] "Building, Flashing and Debugging." https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html#building-west-build. Accessed: 18-01-2023.

[24] "Raspberry Pi Pico series." https://www.raspberrypi.com/products/raspberry-pi-pico/. Accessed: 18-01-2023.

[25] "HPET - OSDev Wiki." https://wiki.osdev.org/HPET. Accessed: 18 Jan. 2023.

[26] "Digilent Zybo." https://docs.zephyrproject.org/latest/boards/arm/zybo/doc/index.html. Accessed: 28 Mar. 2023.

[27] "Clock Control." https://docs.zephyrproject.org/latest/hardware/peripherals/clock_control.html. Accessed: 27 Mar. 2023.

[28] Xilinx, Inc., *Zynq-7000 SoC Technical Reference Manual*, 1.13 ed., 2021.

[29] "Board Porting Guide." https://docs.zephyrproject.org/latest/hardware/porting/board_porting.html. Accessed: 28 Mar. 2023.

[30] "Syntax and structure." https://docs.zephyrproject.org/latest/build/dts/intro-syntax-structure.html#syntax-and-structure. Accessed: 28 Mar. 2023.

[31] "Pin Control." https://docs.zephyrproject.org/latest/hardware/pinctrl/index.html. Accessed: 28 Mar. 2023.

[32] "Timer API Reference." https://docs.zephyrproject.org/latest/kernel/services/timing/timers.html#api-reference. Accessed: 30 Mar. 2023.

[33] "Device Driver Model." https://docs.zephyrproject.org/latest/kernel/drivers/index.html#system-drivers. Accessed: 31 Mar. 2023.

[34] "Ethernet." https://docs.zephyrproject.org/latest/connectivity/networking/api/ethernet.html#ethernet. Accessed: 1 Apr. 2023.

[35] Avnet, *Avnet MicroZed 7010/7020 Revision G Errata*, v1.2 ed., 2020.

[36] Marvell, *Marvell® Alaska® 88E1510/88E1518/88E1512/88E1514 Datasheet*, mv-s107146-u0 rev. g ed., 2023.

[37] "U-Boot." https://www.denx.de/project/u-boot/. Accessed: 3. May 2023.

[38] "The official Xilinx u-boot repository." https://github.com/Xilinx/u-boot-xlnx. Accessed: 3. May 2023.

[39] "GTKTerm: A GTK+ Serial Port Terminal." https://github.com/Jeija/gtkterm. Accessed: 3. May 2023.

[40] "zperf: Network Traffic Generator." https://docs.zephyrproject.org/latest/connectivity/networking/api/zperf.html#zperf. Accessed: 3 Apr. 2023.

[41] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." https://iperf.fr/. Accessed: 3 Apr. 2023.

[42] "Overview." https://sourceware.org/gdb/onlinedocs/gdb/Overview.html#Overview. Accessed: 24 Apr. 2023.

[43] "Packets." https://sourceware.org/gdb/onlinedocs/gdb/Packets.html#Packets. Accessed: 24 Apr. 2023.

[44] "GDB Stub." https://docs.zephyrproject.org/latest/services/debugging/gdbstub.html#gdb-stub. Accessed: 25 Apr. 2023.

[45] "GDB Stub." https://docs.zephyrproject.org/latest/hardware/porting/arch.html#gdb-stub. Accessed: 24 Apr. 2023.

[46] "History of ARM Linux and FPA." https://retrocomputing.stackexchange.com/questions/13400/history-of-arm-linux-and-fpa. Accessed: 4. May 2023.

[47] ARM Limited, *Cortex-A9 Technical Reference Manual*, arm ddi 0388b ed., 2008.

[48] Free Software Foundation, Inc., *Debugging with GDB: the GNU Source-Level Debugger*, 13.1.90.20230428-git ed., 2023.