FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master's Thesis

ARMv6 Processor Emulator for Raspberry Pi Environment Emulation

Jakub Šilhavý

PILSEN, CZECH REPUBLIC

2023

FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master's Thesis

ARMv6 Processor Emulator for Raspberry Pi Environment Emulation

Bc. Jakub Šilhavý

Thesis advisor
Ing. Martin Úbl

PILSEN, CZECH REPUBLIC 2023

Citation in the bibliography/reference list:
ŠILHAVÝ, Jakub. *ARMv6 Processor Emulator for Raspberry Pi Environment Emulation*. Pilsen, Czech Republic, 2023. Master's Thesis. University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering. Thesis advisor Ing. Martin Úbl.

Místo této strany bude přední strana zadání vaší kvalifikační práce.

Místo této strany bude zadní strana zadání vaší kvalifikační práce.

Declaration

I hereby declare that this Master's Thesis is completely my own work and that I used only the cited sources, literature, and other resources. This thesis has not been used to obtain another or the same academic degree.

I acknowledge that my thesis is subject to the rights and obligations arising from Act No. 121/2000 Coll., the Copyright Act as amended, in particular the fact that the University of West Bohemia has the right to conclude a licence agreement for the use of this thesis as a school work pursuant to Section 60(1) of the Copyright Act.

V Plzni, on 10 September 2023

. . . . . .

Jakub Šilhavý

Abstract

<TODO English>

Abstrakt

<TODO Czech>

Keywords

`ARMv6` • Processor • Emulator • Raspberry Pi

Contents

1 Design of a Raspberry Pi Zero Emulator

When designing a complex software system, it is important to take into consideration deciding factors such as the intended usage environment, interaction methods, system dependencies, preferred technologies, or different components constructing the final application. Addressing these questions early on enhances the likelihood of its successful completion as well as its long-term maintainability.

This chapter outlines the key design decisions made within the implementation process of the **ZeroMate emulator**, which is the chosen name for the project [1].

1.1 Input

The emulator necessitates a single input file in the `ELF` format, simplifying the booting process. In this process, the *stage 1 bootloader*, residing in `ROM`, reads the contents of the `SD` card and then initiates and delegates control to the `GPU`, which subsequently resets the `CPU` and loads the kernel into `RAM`.

This file is further referred to as the **kernel** since the emulator was designed within the context of operating systems development. Nevertheless, the input file can fundamentally represent any application intended for execution on Raspberry Pi Zero. Figure 1.2 illustrates the general process of building an `ELF` file, which contains all essential data and information required for code emulation.

1.1.1 Executable and Linkage Format

`ELF` stands for *Executable and Linkage Format* [1], and it is one of the most commonly used formats for executable files, especially on Unix-like systems. There are a number of other representations used in embedded development. For instance, the *Motorola S-Record format*, or `SREC` for short, is often used for programming non-volatile types of memory, such as flash or `EEPROM`.

[1] It combines the word *Zero*, as in Raspberry Pi Zero, and *Mate*, which in this case is used as a synonym for a friend or „buddy“.

In terms of this project, the key advantage of ELF over SREC is that ELF is **used for both linkage and execution**. Therefore, if the kernel is compiled with debug symbols turned on [2], the symbol table stored in the final ELF file can be used during the parsing process, which is discussed in section 1.3.2, to provide the user with function names as they were used in the source code, which should improve the readability of the final disassembly. SREC, on the other hand, is **only used for execution**. Therefore, it can be viewed as highly compressed as it comprises only the necessary information for uploading firmware onto a microcontroller 1.1, which is commonly referred to as flashing.

It can be concluded that ELF provides more information that can be useful when reconstructing the original source code. Hence, it is used as the supported format for the input files. It is worth mentioning that this choice does not have as much impact on the core functionality as it does on visual aspects, which is discussed more in detail in section 1.5.

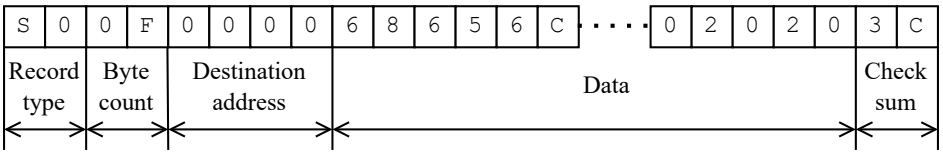

Figure 1.1: Single SREC record (16-bit addressing)

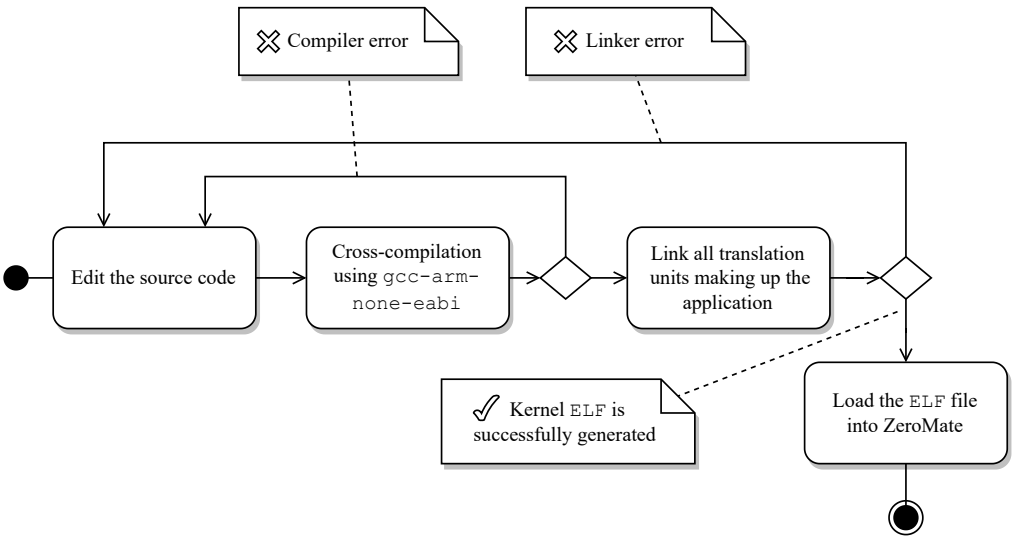

Figure 1.2: Process of building an ELF file (input for the emulator)[3]

[2] In the case of the `gcc-arm-none-eabi` compiler, the `-g -O0` flags should be used.

[3] Cross-compiling is a process where the source code targets a different platform than the one it is compiled on.

1.2 User's Interaction

As shown in the deployment diagram in figure 1.4, the emulator was **designed to run as a native desktop application on Windows, Linux, and MacOS operating systems**. It places a strong emphasis on visualization, serving as a debugging tool to assist with troubleshooting embedded applications targeting Raspberry Pi Zero.

The primary interaction with the system, from the user's perspective, is visualized in figure 1.3, where the user is provided with an interface that allows them to load an input file as well as to control the state of the emulation.

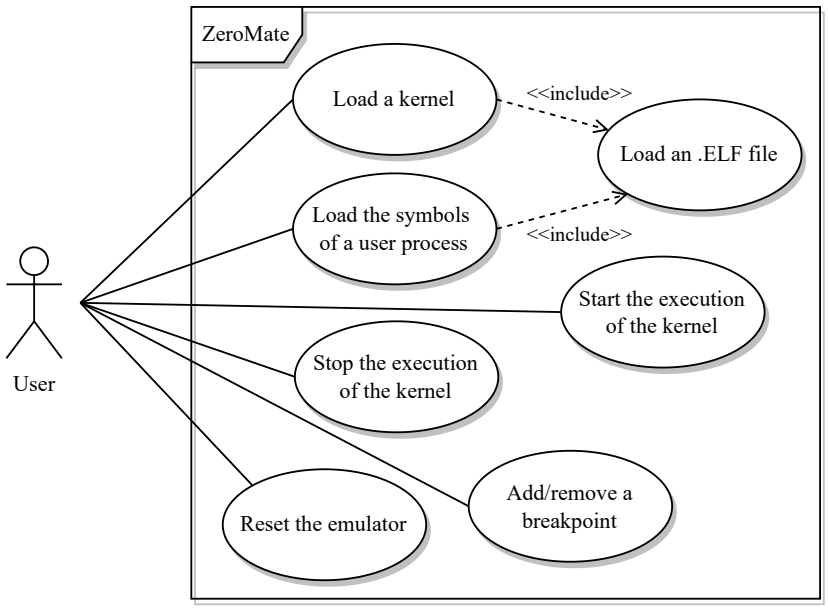

Figure 1.3: Primary use-cases of the ZeroMate emulator

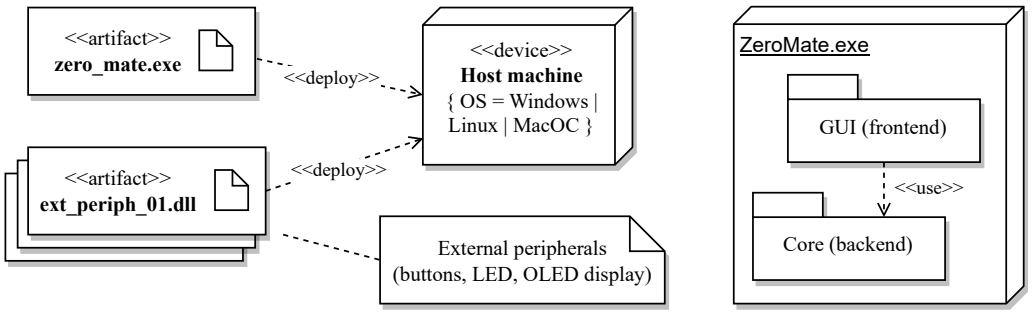

Figure 1.4: Deployment diagram of the ZeroMate emulator

The main application is designed as a **two-tier architecture**. In this arrangement, the top layer, which is the `GUI`, serves the dual purpose of visualizing data and acting as the primary user interface. The following chapters delve into the architectural structure of the core of the emulator.

1.3 Core Components

There are a number of different components working alongside to achieve a thorough emulation of a given kernel. Among these components, the `ARM1176JZF_S` component, which represents the `CPU` itself, may arguably stand out as the most complex one due to its encapsulation of various sub-components, including the `CPU` context, `ALU`, `MMU`, `ISA` decoder, and more. The role of every component will be examined further in the following sections.

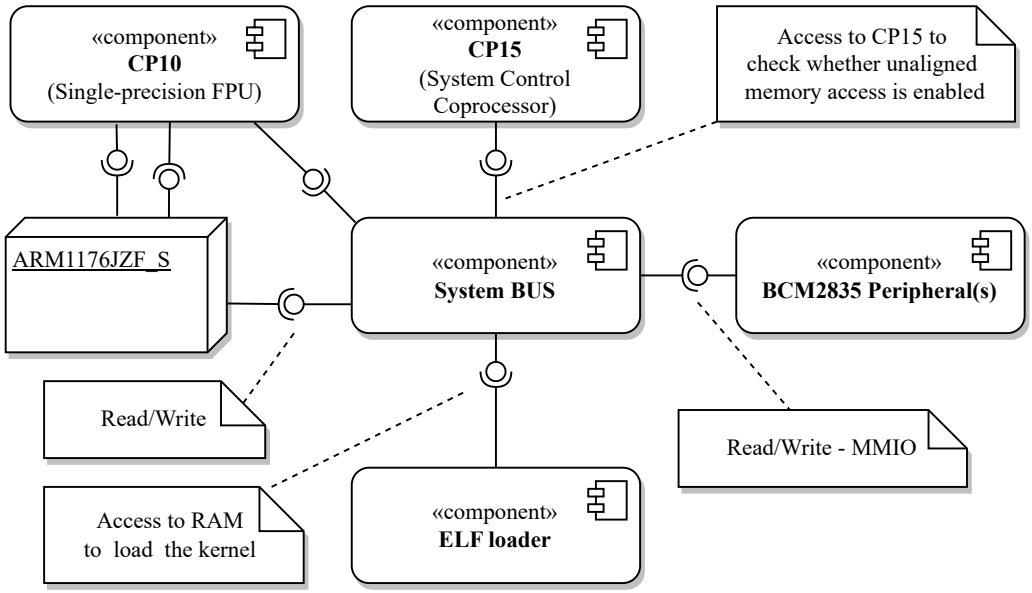

Figure 1.5: Core components of the ZeroMate emulator

Figure 1.5 illustrates the fundamental interactions among the core components. It can be observed that the majority of the components communicate with one another via the system bus [4]. For example, when the `CPU` executes a load/store instruction, it propagates the target address to the system bus, and the system bus then forwards the request to the corresponding peripheral associated with that address.

[4]What ZeroMate denotes as the system bus is typically regarded as the primary `CPU` bus.

1.3.1 System Bus

As mentioned previously, the system bus serves as an **intermediate unified interface** for accessing the `RAM` or any of the `BCM2835` memory-mapped peripherals [2]. Each peripheral that is meant to be mapped into the address space must implement the same interface, so the bus can forward the read/write request independently of the peripheral's implementation (see section 1.3.3). All actions associated with the request itself are then handled internally within the target peripheral.

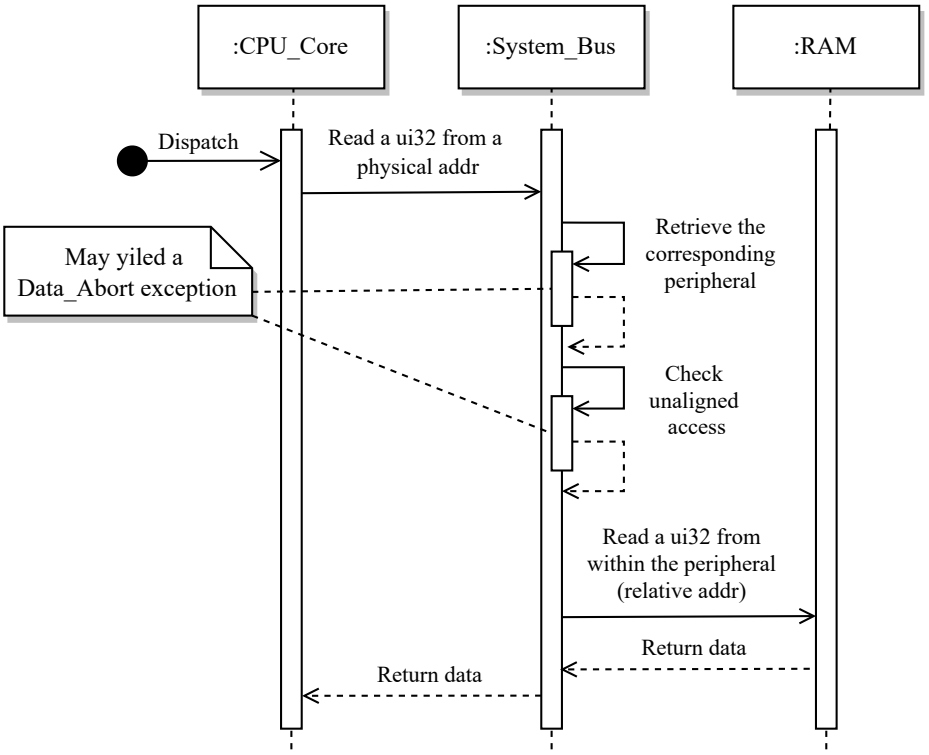

Figure 1.6: Example of a read/write data request issued by the CPU

As shown in figure 1.6, there are two internal steps the system bus carries out before proceeding with the request. First, the bus needs to determine what peripheral should the request be forwarded to. Secondly, it checks whether unaligned memory access is taking place or not.

> In reality, the main system bus does not manage peripherals the same way it does in ZeroMate. It only serves as a medium for connecting different types of memory-mapped devices. Nevertheless, from an architectural point of view, it is a reasonable place for implementing common validity checks as it plays the role of a single point of access to all memory-mapped peripherals.

Additionally, the system bus ensures that the peripheral receives an **address relative to its location in the address space** [5]. In other words, it does not have any knowledge about its location on the bus, which is desired, as it decreases coupling and increases cohesion between the two components [3].

Source code 1.1: System bus interface for I/O operations

```
class CBus final {
public:
    template<typename Type>
    void Write(std::uint32_t addr, Type value);

    template<typename Type>
    [[nodiscard]] Type Read(std::uint32_t addr);
};
```

It can be argued that permitting the reading or writing of a general data type may diverge from real hardware specifications, as the system bus is typically of a fixed size, e.g. 32 bits. This simplification was made for convenience reasons when accessing predefined data structures in the `RAM`, such as the page table(s).

1.3.1.1 Managing Peripherals

The system bus component maintains a collection of references to all memory-mapped input-output devices, further referred to as `MMIOs`. Whenever a peripheral needs to be attached to the bus, it is inserted into the appropriate position within the collection. This ensures that the entire collection remains sorted in ascending order based on the starting addresses of the peripherals. This property enables the use of a binary search algorithm, resulting in faster lookup times, particularly in $O(log_2 n)$ time complexity [4], which is crucial for improving the overall speed of the emulation.

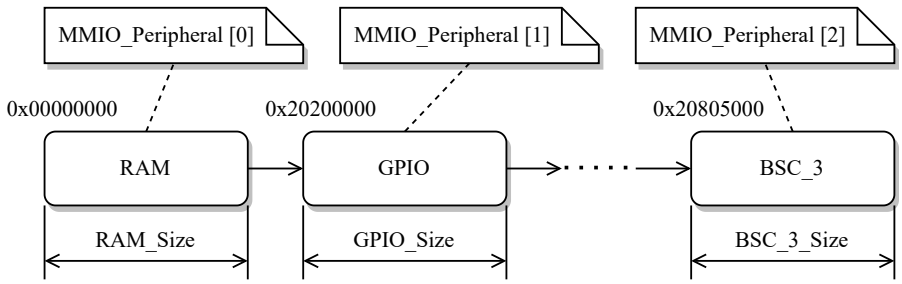

Figure 1.7: Collection of memory-mapped peripherals

[5] The relative address is calculated as the address contained in the `R/W` request issued by the `CPU` minus the address of the peripheral on the system bus.

According to statistics, on average, **load-store instructions account for more than 50% of all instructions in an x86 application** [5]. Although this research applies to a different architecture, it is reasonable to conclude that optimizing peripheral access efficiency might be crucial for emulation speed.

When connecting a peripheral, the bus must also ensure that there is no overlap between two peripherals and that they all fit within the address space, which, on a `32-bit` architecture, spans out to `4GB`.

1.3.1.2 Unaligned Memory Access

Unaligned memory access occurs when the **CPU attempts to read or write data from an address that is not divisible by the word size**. For example, reading 4 bytes from address `0x00000011` triggers unaligned access as the address is not `word`-aligned [6]. Nevertheless, this behavior can optionally be disabled, for example, for compatibility reasons, in the system control coprocessor `CP15` using the following sequence of instructions.

Source code 1.2: Enabling unaligned access in `CP15`

```
1 mrc p15, #0, r0, c1, c0, #0   ;@ Copy ctrl reg of CP15 to R0
2 orr r0,  #0x400000            ;@ Set bit 22 in R0
3 mcr p15, #0, r0, c1, c0, #0   ;@ Update CP15
```

[6] A `word` is a fixed-size number of bits that the `CPU` can process as a single unit. In the case of `ARM1176JZF_S`, one `word` equivocates to 4 bytes.

1.3.2 Executable and Linkage Format File Loader

The main objective of this module is to **parse an input ELF** file and copy the `.text` section, `word` by `word`, into `RAM`, as specified by the linker script. Additionally, it performs code disassembly, which is a process of reconstructing the source code from machine code. This allows the user to observe individual instructions in a more user-friendly way as they are executed.

For visualization purposes, the component also features the capability to parse an `ELF` file without copying its data into memory, which can be useful for viewing user processes that are compiled separately from the kernel itself. During this process, the **ELF loader also demangles all symbols** found in the input file [7], thereby presenting the user with function names that have not undergone modification by the compiler for its internal purposes.

Source code 1.3: Example of symbol demangling

```
1 Demangle("_ZNSt6vectorIiSaIiEE9push_backERKi") =
2 "std::vector<int,std::allocator<int>>::push_back(int␣const&)"
```

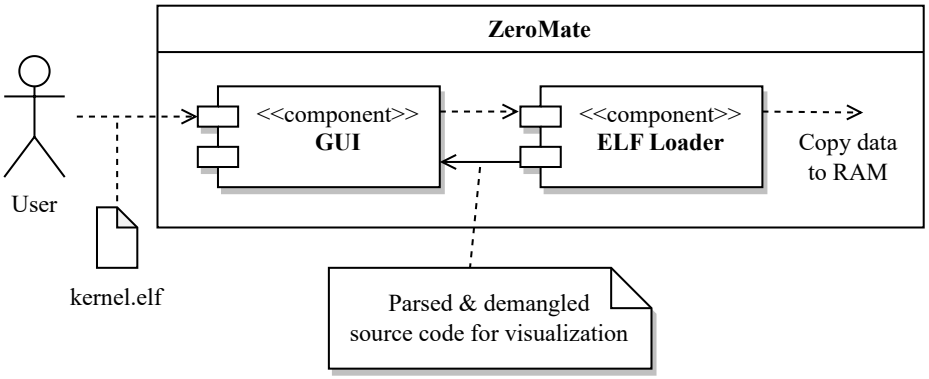

Figure 1.8: Loading an input `ELF` file (kernel)

[7] Demangling is a process of transforming `C/C++` `ABI` identifiers (like `RTTI` symbols) into the original `C/C++` source [6].

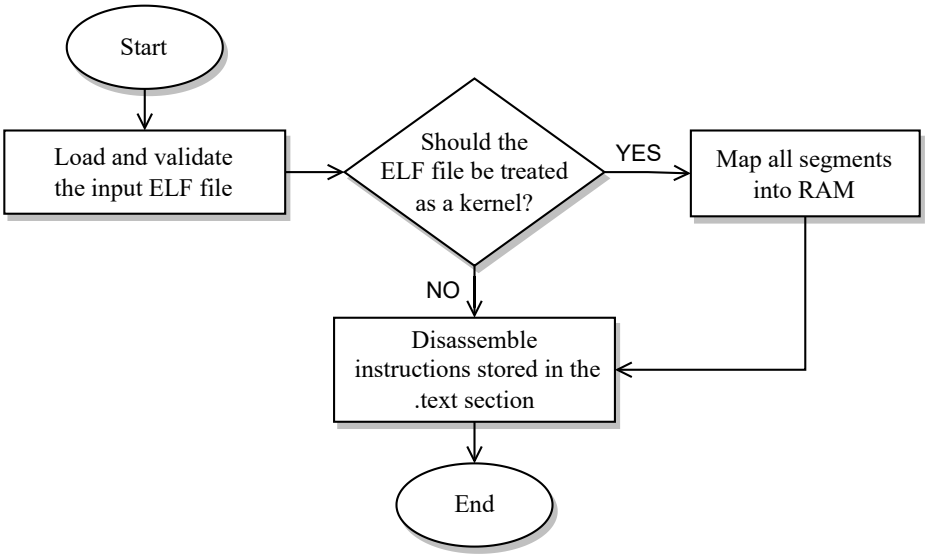

Figure 1.9: Internal logic of the ELF Loader component

It is important to emphasize that ZeroMate does not perform the tasks of parsing an ELF file and demangling symbols all by itself. Instead, it utilizes two external open-source libraries, *ELFIO* [7] and *Demumble* [8], to accomplish these functions.

1.3.3 BCM2835 Peripherals

ZeroMate distinguishes between two types of peripherals; those directly integrated with the microcontroller, such as RAM, ARM timer, or the interrupt controller, and those referred to as external peripherals, which are externally connected to another internal peripheral, the GPIO pins, forming a cascading set of connected peripherals. Examples of external peripherals may include buttons, switches, LEDs, displays, or keyboards. The following sections focus on the internal peripherals of Raspberry Pi Zero.

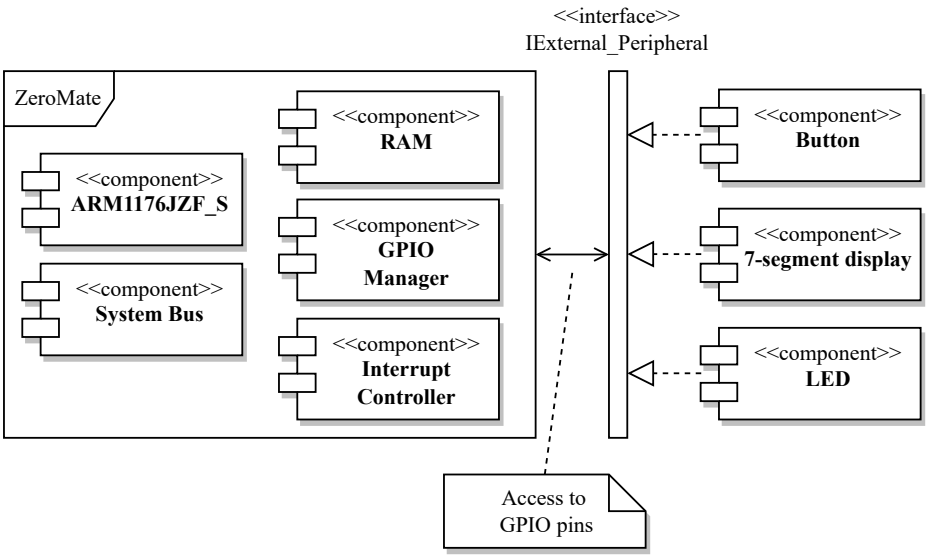

Figure 1.10: Internal vs External peripherals

In section 1.3.1, it is explained that the system bus manages a collection of references to all peripherals that are mapped into the address space. Using a general interface, the bus does not need to be concerned about how each peripheral functions internally. It simply forwards a `R/W` request initiated by the `CPU` to the corresponding peripheral.

Every `BCM2835` peripheral encapsulates a set of registers, whose functions are detailed in the manual [2] [8]. By reading from or writing to these registers, the internal state of the peripheral can be modified, which is specific for each peripheral.

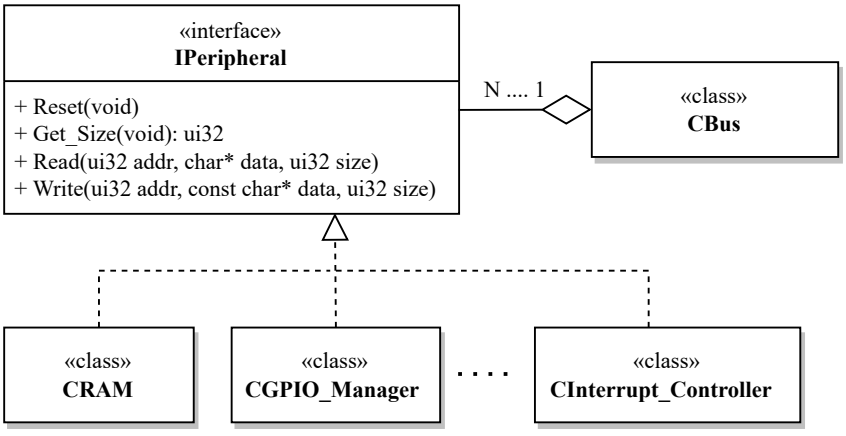

Figure 1.11: Hierarchy of internal peripherals

[8] The `BCM2835` manual is known to contain several typographical errors. As a result, the community surrounding it published a list of these errors along with their respective corrections [9].

The `Get_Size()` method is used primarily for detecting bus collisions when mapping peripherals into the address space, which was mentioned previously in section 1.3.1.1.

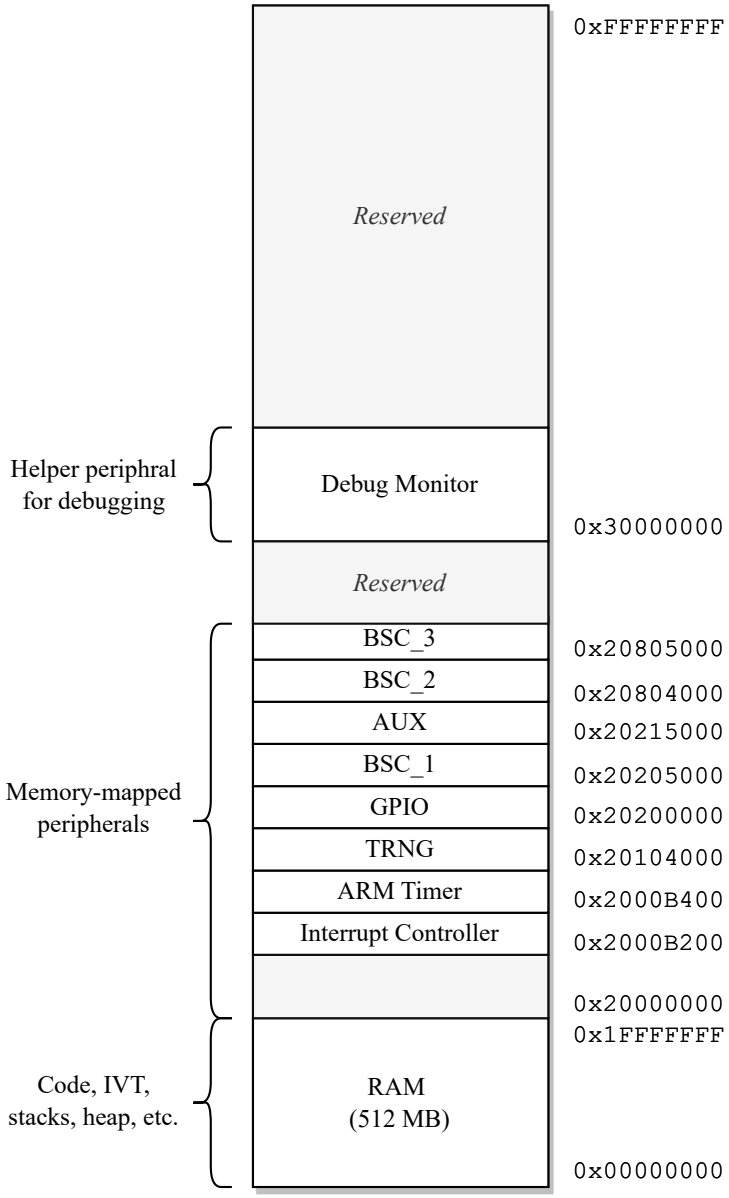

Figure 1.12: `BCM2835` physical memory layout emulated by ZeroMate

It can be noticed that **ZeroMate does not account for all `BCM2835` peripherals** since emulating every single one in its entirety would pose a significant complexity. Consequently, ZeroMate focuses emulation efforts on the most frequently utilized peripherals, such as the `ARM` timer, `GPIO`, `IC`, and others.

Nonetheless, the system's overall design is structured to allow for a smooth integration of additional peripherals in the future if needed. The following sections explain the fundamental emulation principles of each of the peripherals listed in figure 1.12.

1.3.3.1 Memory-mapped Registers

As mentioned previously, each `BCM2835` peripheral encapsulates a set of registers, through which the user can interact with the peripheral. From an implementation perspective, these registers can be represented as a fixed-size array of `32-bit` integers, which are addressed relative to the peripheral's base address.

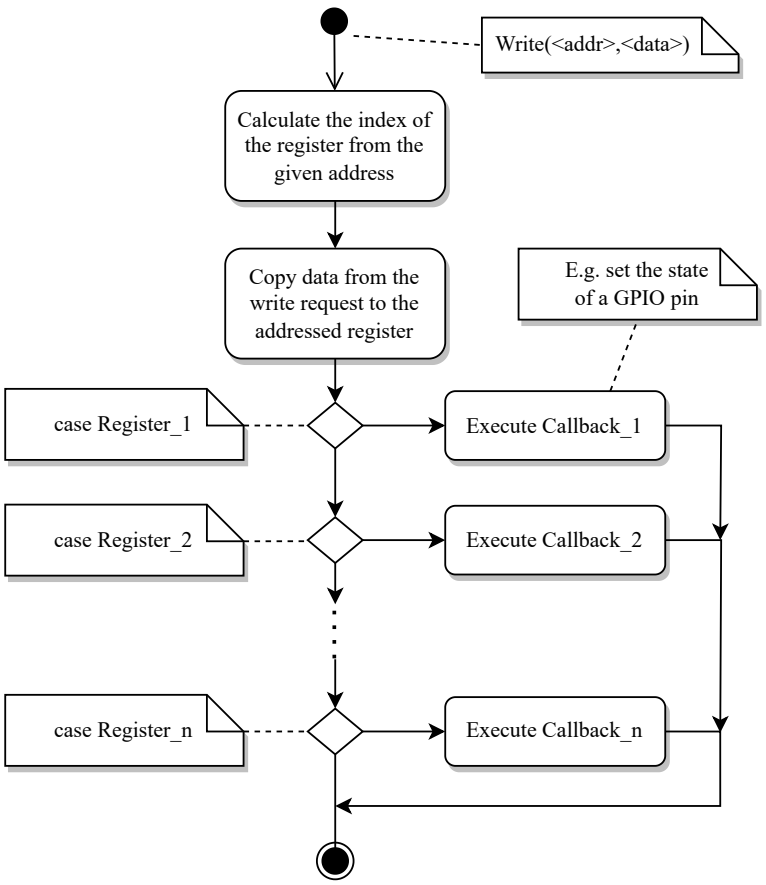

Figure 1.13: Writing to a peripheral's register[9]

Whenever a read/write request is sent through the bus, the peripheral identifies the addressed register, and the execution is then dispatched to the correspond-

[9] Reading works in a similar way. Optionally, there might be some prior actions taken before the value of the register is copied from the peripheral, such as inserting a random number into the `data` register when utilizing `TRGN` 1.3.3.5.

ing callback function, which carries out the necessary actions associated with that specific register. Generally, this approach can be applied to the vast majority of memory-mapped peripherals.

1.3.3.2 System Clock Listener

Optionally, each peripheral can implement the `ISystem_Clock_Listener` interface, which allows it to register with the CPU as a system clock listener. Whenever an instruction is executed, the CPU notifies all of its system clock listeners of how many CPU cycles it took to execute the instruction, allowing them to update themselves accordingly. Examples of such listeners may include the ARM Timer 1.3.3.6 or the AUX 1.3.3.9 and BSC 1.3.3.10 peripherals, which encapsulate time-based hardware communication functions.

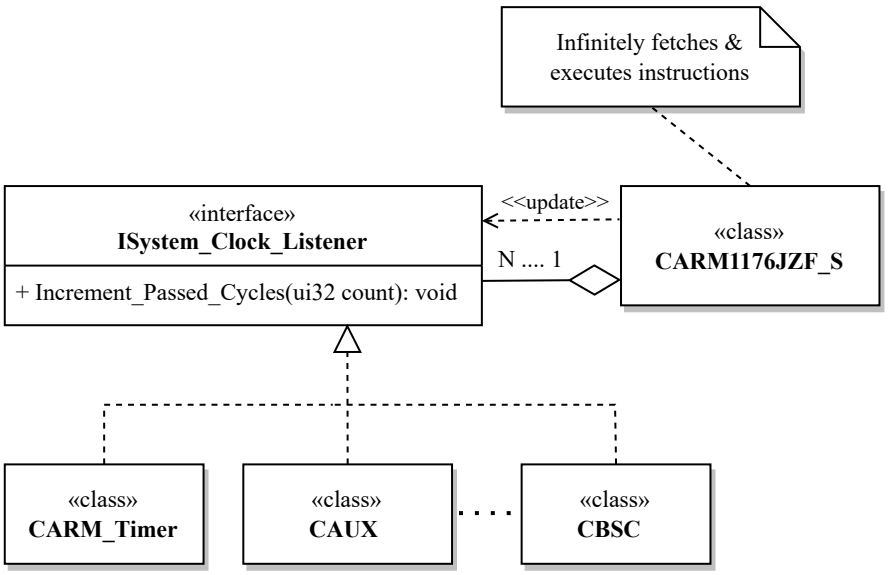

Figure 1.14: `ISystem_Clock_Listener` interface

It is important to mention that **updating a system clock listener is, from the emulated CPU's perspective, a blocking operation**. Therefore, the peripheral's callback function should avoid any unnecessary actions that might further prevent the CPU from executing the next instruction. Alternatively, updating system clock listeners could be performed asynchronously within a separate thread. However, this approach would introduce additional concurrency-related challenges that would require thorough consideration.

1.3.3.3 Random Access Memory

From an oversimplified perspective, a computer consists of two essential components; the CPU and memory. One key parameter used to classify various types of memory is their ability to retain data even after the power supply is shut down. Raspberry Pi Zero is equipped with an SD card slot, which houses non-volatile [10] memory for storing the kernel image. However, from ZeroMate's point of view, this type of memory is implicitly provided by the host machine.

The board is also featured with 512MB of RAM, which functions as volatile memory for executing the kernel code. It accommodates runtime-critical sections such as the stacks [11], heap, page tables, or the interrupt vector table, often referred to as the IVT.

The implementation of RAM is straightforward since it can be represented as an array of bytes as shown in the figure 1.15 below. However, the downside of this approach is that it immediately takes up 512 MB of the host's RAM, which may become an issue on older computers with limited resources.

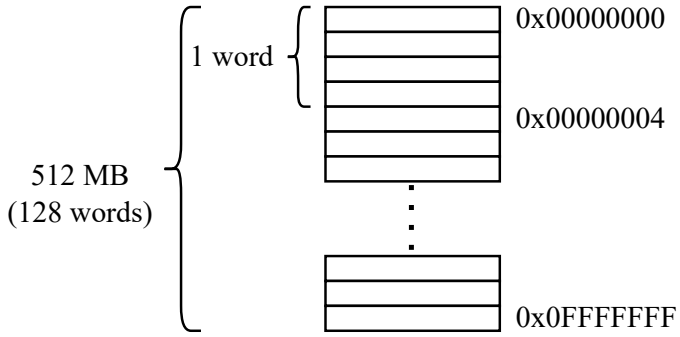

Figure 1.15: RAM implementation as a continuous piece of memory

A more effective approach would involve dynamically allocating fragmented pieces of memory as they are being addressed by the CPU. However, the author would argue that such an implementation would be algorithmically more complex, which could lead to distracting errors when implementing memory-related instruction, especially in the early stages of development. As a result, it was classified as a *nice-to-have* feature that would be worth addressing in the future once the emulator has been thoroughly QA-tested.

[10] Non-volatile memory is capable of persisting data even after the supply voltage is turned off.
[11] ARM1176JZF_S uses a different set of registers for each CPU mode.

1.3.3.4 Debug Monitor

The debug monitor plays the role of a memory-mapped output device for displaying `8-bit` character-based information.

> The component is not included in Raspberry Pi Zero itself; its presence serves solely for debugging purposes during the development of ZeroMate.

The ZeroMate project also includes a basic driver of the peripheral that the user can seamlessly integrate into their build system. This allows them to use **„print-like" functions** they might be familiar with from high-level programming languages, which may result in easier troubleshooting and resolving errors.

Source code 1.4: Demonstration of the use of the debug monitor

```
#include "monitor.h"

int main() {
    bool flag = false;
    unsigned int my_var = 155;

    sMonitor << "Hello␣World\n";
    sMonitor << "myVar␣=␣" << my_var << '\n';
    sMonitor << "flag␣=␣" << flag << '\n';

    return 0;
}
```

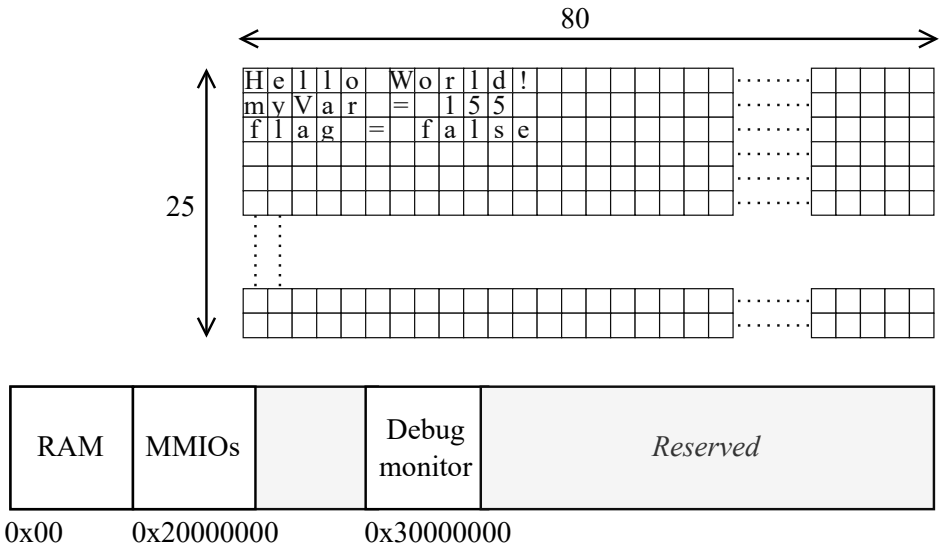

Figure 1.16: Memory-mapped debug monitor

To attain the same capabilities in practice, the user would need to utilize a form of serial communication, such as `UART`, through which they could transmit characters to an external device [12]. This is commonly achieved by running software like *PuTTY* [10] on the user's computer.

As shown in figure 1.16, the debug monitor is mapped to an unoccupied address `0x30000000`. It is structured as a flat memory region, which is managed by the driver the user code interacts with. The size of the monitor was chosen to be `80x25 8-bit` characters [13].

1.3.3.5 True Random Number Generator

The `TRGN` peripheral is an integrated `32-bit` hardware random number generator. Although it is not documented in the official `BCM2835` manual [2], its existence can be confirmed, for instance, by examining the implementation in the GNU/Linux kernel [11].

For simplification purposes, ZeroMate primarily focuses on providing random numbers while omitting more advanced features such as configuring the generator's speed, generating interrupts, or the warm-up count. The warm-up count refers to the process of generating and immediately discarding a set of random numbers before the initialization is completed [14].

From the user's code perspective, the process of **retrieving a random number consists of two steps**, which are displayed in figure 1.17.

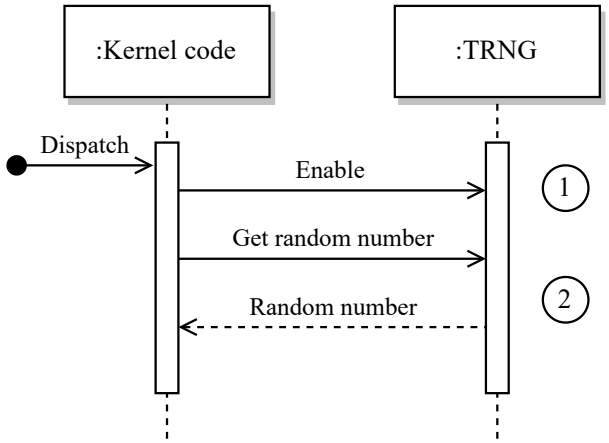

Figure 1.17: Reading random numbers from the `TRNG` peripheral

[12] While there are alternative methods to achieve the same functionality, this approach is among the most common ones.

[13] From an implementation point of view, it could vary in size as long as it does not overlap with other memory regions.

[14] The initial values are „less random".

Enabling TRNG.

The TRNG peripheral is enabled by setting bit 0 of the **control register** to 1. If implemented, this action would also trigger the processing of „warming up“, which was mentioned previously.

Reading random numbers.

First, the user should check the availability of random numbers in the TRNG's queue by examining the most significant 8 bits of the **status register**. If this number is 0, they should wait until the generator accumulates a sufficient amount of entropy to generate a random number. When data is ready, reading from the **data register** will retrieve a random number from the queue.

> ZeroMate can almost instantly generate a random number using a pseudo-random number generator. As a result, when reading the most significant 8 bits of the status register, the user will consistently receive the value 1, meaning they can read random numbers without delay.

Utilizing a pseudo-random number generator, such as an *LCG* [12] or *Mersenne-Twister*, **can greatly improve the performance of the emulation**. Depending on the implementation, accessing a true random number generator via the host operating system may have a detrimental impact on overall speed, as it may continually gather entropy from user inputs, like key presses or cursor movements. This can potentially lead to a blocking operation if there is currently insufficient entropy available.

1.3.3.6 ARM Timer

One of the most frequently used functions of the ARM timer is to periodically trigger interrupts, whether it is for toggling an LED, generating a PWM signal, or switching the current CPU context, which is an integral part of any preemptive OS scheduler.

As shown in figure 1.18, there are two data/control paths through which the timer can be interacted with. The first path, when the timer is treated as a memory-mapped peripheral, serves the purpose of reading from and writing to its internal registers in order to configure its desired functionality. This may involve steps such as setting up the prescaler, enabling interrupts, or defining the initial threshold value. The other path is used implicitly by the CPU to notify the peripheral about how many CPU cycles it took to execute the last instruction. The ARM timer then leverages the prescaler to divide the input frequency, as the main CPU frequency may not always be suitable for the given task.

As far as ZeroMate is concerned, all time-related functionalities, such as the ARM timer, UART, or I²C, are for synchronization purposes, inherently derived from the CPU's clock.

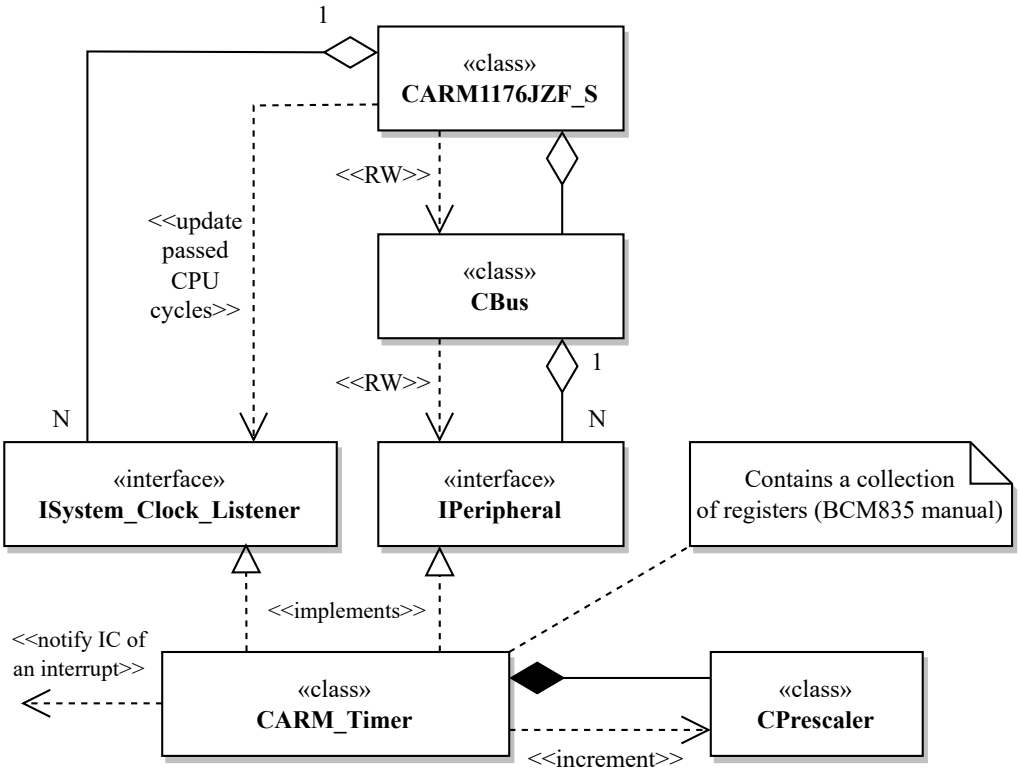

Figure 1.18: Context of the ARM timer component

As stated previously, the purpose of the prescaler is to divide the CPU's frequency by a factor of 1, 16, or 256, which ultimately affects the timer's period - how rapidly the **value register** counts down to zero. Additionally, the timer's period can be adjusted by modifying the value in the **load register**, which serves to re-initialize the value register whenever it reaches zero. If enabled, with each such event, the timer will trigger an interrupt. This concept is visualized in figure 1.19.

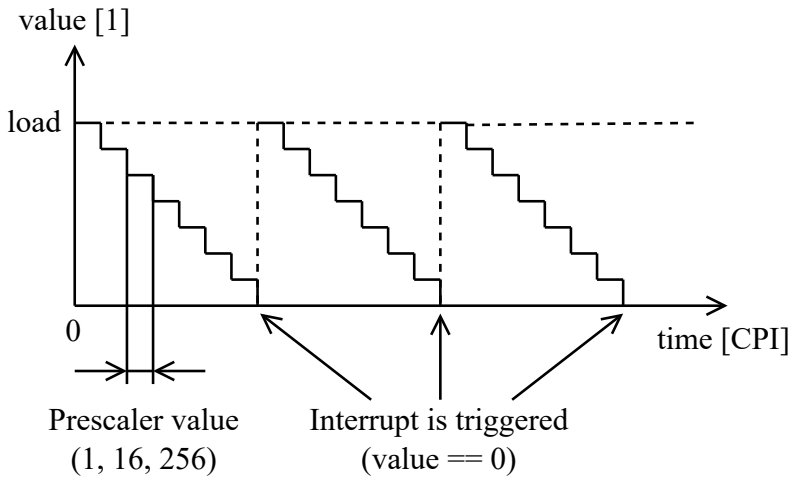

Figure 1.19: Content of the value register of the ARM timer over time

1.3.3.7 General Purpose I/O

The GPIO manager is a peripheral that manages the **54 general-purpose input-output pins** (GPIO) that are available to the programmer. These GPIO pins also function as a connecting interface for all external devices, which is shown in figure 1.10. Each pin is represented as a separate class encapsulating its current state, which consists of the pin's function, a list of enabled interrupts, an indication of any pending interrupts, and its current state.

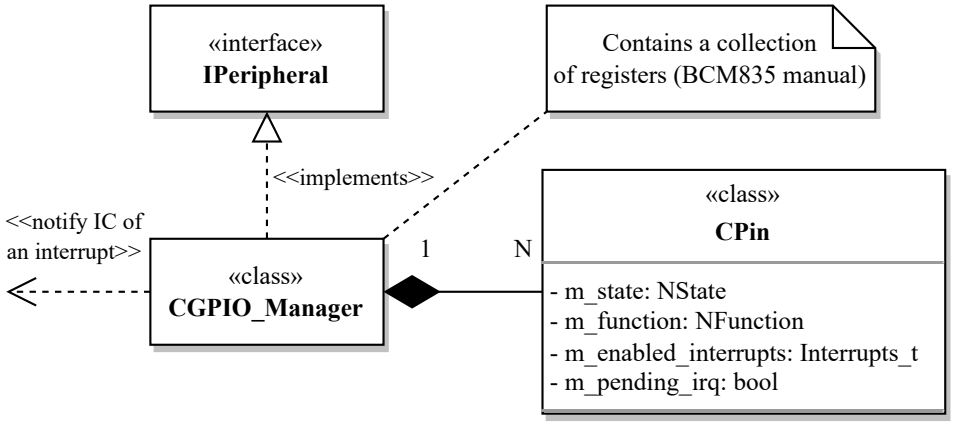

Figure 1.20: Structure of the GPIO manager[15]

All GPIO pin functions, as well as interrupt types, can be found explained in the BCM2835 manual [2].

[15]The CPin class also exposes a set of functions that allow the caller to access its private members.

The pin's function restricts the way the pin can be interacted with. For instance, when the user tries to read from an output pin, they will be prompted with a warning message indicating that their action is inconsistent with the current pin configuration.

> ZeroMate does not provide support for analog pins. Therefore, all `GPIO` pins mentioned in this document are regarded as digital pins, with only two possible states, `HIGH` and `LOW`.

Emulation of latches.

Some of the `GPIO` registers work, on a hardware level, as latches, which may not be as intuitive from a software emulation point of view. An example of this principle would be the `GPSETx` and `GPCLRx` registers, which respectively set an output pin to a logical one and zero. These registers are stateless, meaning they do not retain their previous state internally. One approach to emulate such behavior is to reset the register to its default value after a write operation has been performed.

Source code 1.5: `SW` emulation of a `HW` latch register

```
1 void Set_Pin_High(std::uint32_t& gpset0)
2 {
3     // Iterate over all bits of gpset0.
4     // If gpset0[i]==1, then set the
5     // corresponding pin to HIGH.
6
7     gpset0 = 0; // SW latch emulation
8 }
```

Detecting Interrupts.

Whenever the state of a pin changes, a series of checks is performed to determine whether an interrupt has occurred. One of the most commonly used types of interrupts is triggered by a change in the logical value of a specific pin, either transitioning from `HIGH` to `LOW` or vice versa. The types of interrupt to be detected for each pin can be specified in the corresponding registers. When an interrupt is detected, it is reported to the interrupt controller, which can then initiate further actions, which is captured in figure 1.20.

1.3.3.8 Interrupt Controller

The interrupt controller serves as the primary interface for managing all peripherals that can generate interrupts. Through the interrupt controller, users can enable or disable various interrupt sources, including GPIO pins, the ARM timer, and the UART peripheral. **When an interrupt is signaled to the interrupt controller, it checks whether the source is enabled; otherwise, the event is disregarded**.

From the CPU's point of view, after an instruction is executed, it checks with the interrupt controller to ascertain the existence of any pending interrupts. If the interrupt controller has a record of a pending interrupt, and global interrupts are enabled, the CPU proceeds to throw an IRQ exception.

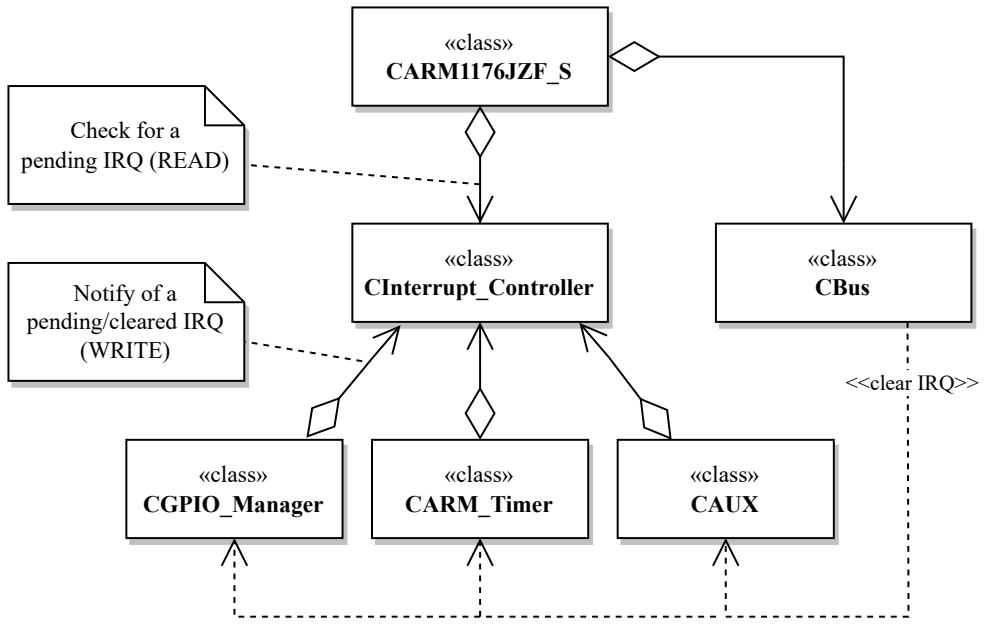

Figure 1.21: Context of the interrupt controller

In terms of design, the interrupt controller encapsulates an associative storage that pairs each IRQ source [16] with its associated metadata, indicating whether it is enabled and if there is a pending interrupt. From the CPU's perspective, this storage is read-only, as its only purpose is to check for any pending interrupts. The contents of this storage are modified by the peripherals themselves, either when they generate an interrupt or clear a pending interrupt.

The ARM1176JZF_S processor also features so-called fast interrupts, or FIQ for short. However, ZeroMate does not offer support for it as it primarily focuses on fundamental principles rather than more advanced features.

[16] You can find a comprehensive list of all available IRQ sources in the BCM2835 datasheet [2].

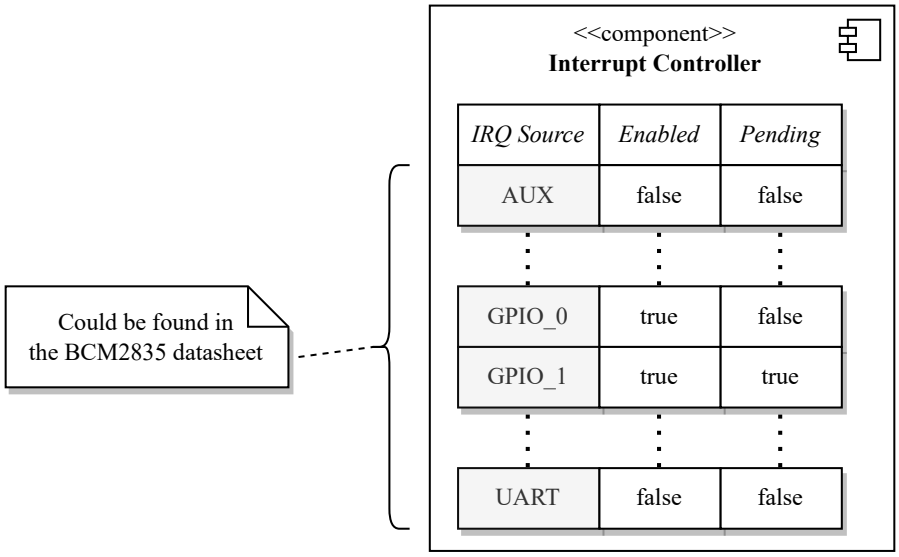

Figure 1.22: Encapsulated information about `IRQ` sources

It is worth mentioning that the interrupt controller of the `BCM2835` microcontroller distinguishes between two types of interrupt sources: Basic `IRQ` and `IRQ`, both of which are listed in the `BCM2835` datasheet [2]. Nevertheless, from a design perspective, the underlying principles remain the same.

1.3.3.9 Auxiliaries

The auxiliary peripheral comprises three distinct peripherals: `Mini_UART`, `SPI_0`, and `SPI_1`. Among these, only `Mini_UART` is currently supported by ZeroMate.

There are two primary registers shared among all auxiliary peripherals: the `enable` register, responsible for activating the respective peripheral, and the `IRQ` register, which signals pending interrupts. The remaining registers are specific to each peripheral, as depicted in the figure below.

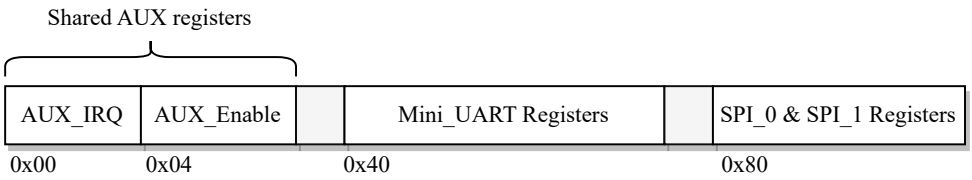

Figure 1.23: Registers of the `AUX` peripheral

Whenever a read/write request is received, using the technique described in section 1.3.3.1, the `AUX` class can efficiently redirect the execution to the relevant auxiliary peripheral, which will subsequently handle the request internally.

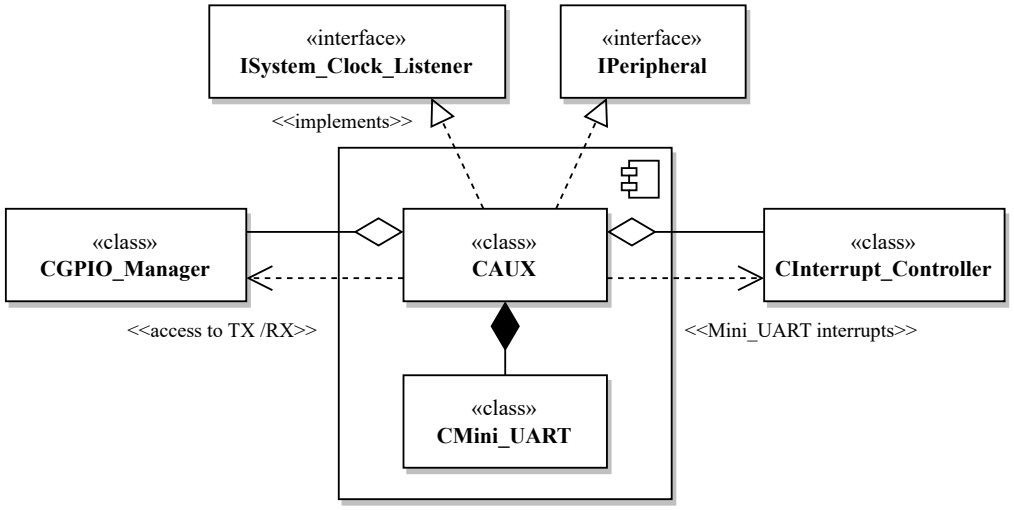

Figure 1.24: Structure of the AUX peripheral

Figure 1.24 shows the internal structure and dependencies of the AUX peripheral. It can be noticed that it implements the ISystem_Clock_Listener interface 1.3.3.2, which allows it to synchronize with the rest of the system.

Mini_UART.

UART, short Universal Asynchronous Receiver-Transmitter, inherently operates as asynchronous communication, which means there is no explicit synchronization between the two devices. Since these devices may have different clock speeds, they must adjust their frequencies to establish a common speed known as the baudrate, which expresses how many bits can be received/transmitted per second.

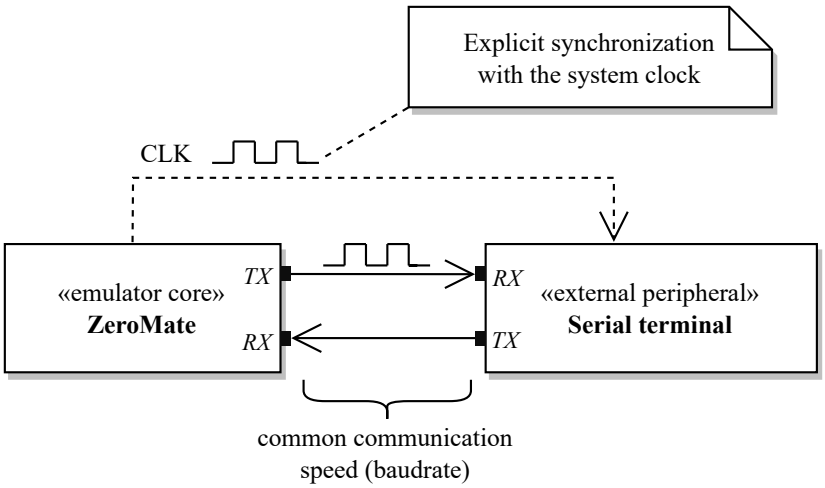

Figure 1.25: UART communication with an external peripheral

The BCM2835 microcontroller does not incorporate a full version of UART. Instead, it supports a simplified version known as Mini_UART, which omits some of the extended features. As a result, the user is only able to modify the baudrate and the number of data bits transferred within a single frame, which can be either seven or eight. The remaining parameters, such as parity and the number of stop bits, are fixed according to the datasheet [2].

In ZeroMate, all external peripherals are provided read-only access to the system clock, enabling them to synchronize themselves if required. This approach contradicts the fundamental principles of asynchronous communication since it introduces a form of synchronization. However, this design decision was made to enhance the emulation's reliability while still enabling users to utilize Mini_UART as if they were interacting with real hardware.

The implementation of Mini_UART communication can be accomplished through a state machine driven by a pre-divided system clock as shown in figure 1.27. When a predefined number of CPU cycles have passed, the state machine updates itself to transmit and receive the next bit of the current data frame, which can be seen in figure 1.26 [17].

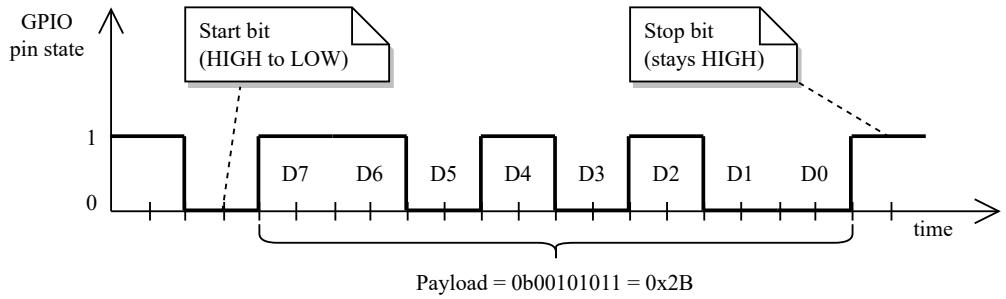

Figure 1.26: Example of a Mini_UART data frame with 8 bits of data

Receiving data can be implemented in a similar manner. Instead of setting the value of the TX pin, the state machine reads the value of the RX, which has been set previously by the external peripheral.

[17] Transmitting a single bit can be done by setting the corresponding TX pin to the desired value. Further details on what GPIO pins are designated for Mini_UART can be found in the datasheet [2].

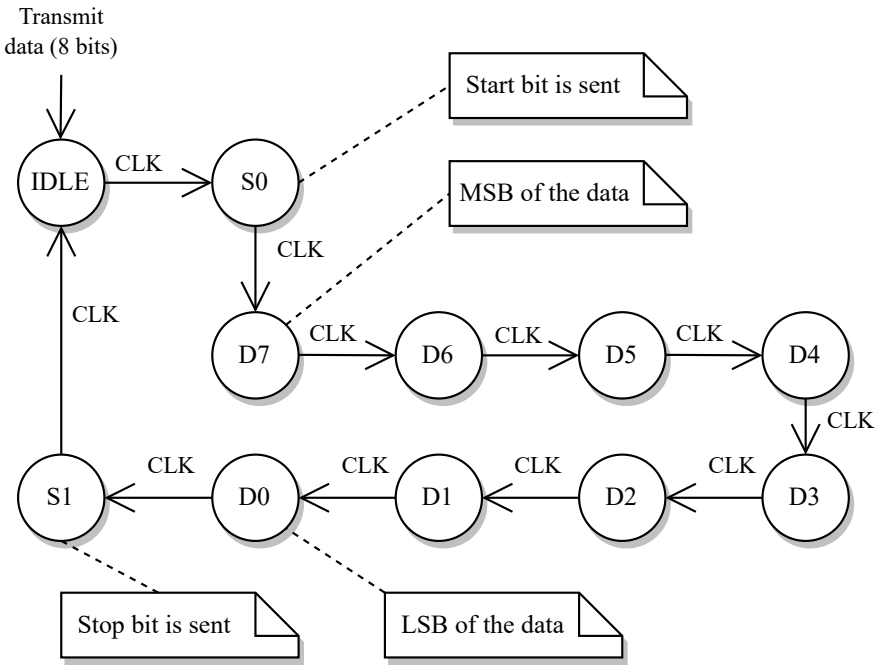

Figure 1.27: `Mini_UART` state machine (transmitting 8 bits)

1.3.3.10 Broadcom Serial Controller

The Broadcom serial controller, often abbreviated as `BSC`, is a peripheral device that allows the user to communicate with external devices, typically sensors, using the I^2C protocol, which is a synchronous serial communication standard. I^2C utilizes two `GPIO` pins: `SDA` for data transmission and `SCL` for synchronization. Raspberry Pi Zero is equipped with three of these devices that can be found mapped to their respective addresses in figure 1.12.

From a design point of view, the emulation of an I^2C bus is nearly identical to `UART`, with the primary distinctions being the frame structure and the synchronous transmission of individual bits using an additional `GPIO` pin instead of the emulated `CPU`'s clock.

While there are various configurations of the I^2C communication protocol [a], ZeroMate exclusively supports only one, which is presented in figure 1.28. As stated previously, ZeroMate emphasizes the emulation of fundamental principles, rather than attempting to implement every conceivable configuration, which would only add unnecessary complexity without delivering any added value.

[a] They vary in voltage levels for representing logical 1 and 0, as well as in how they define the start and stop bits using the `SDA` and `SCL` signals.

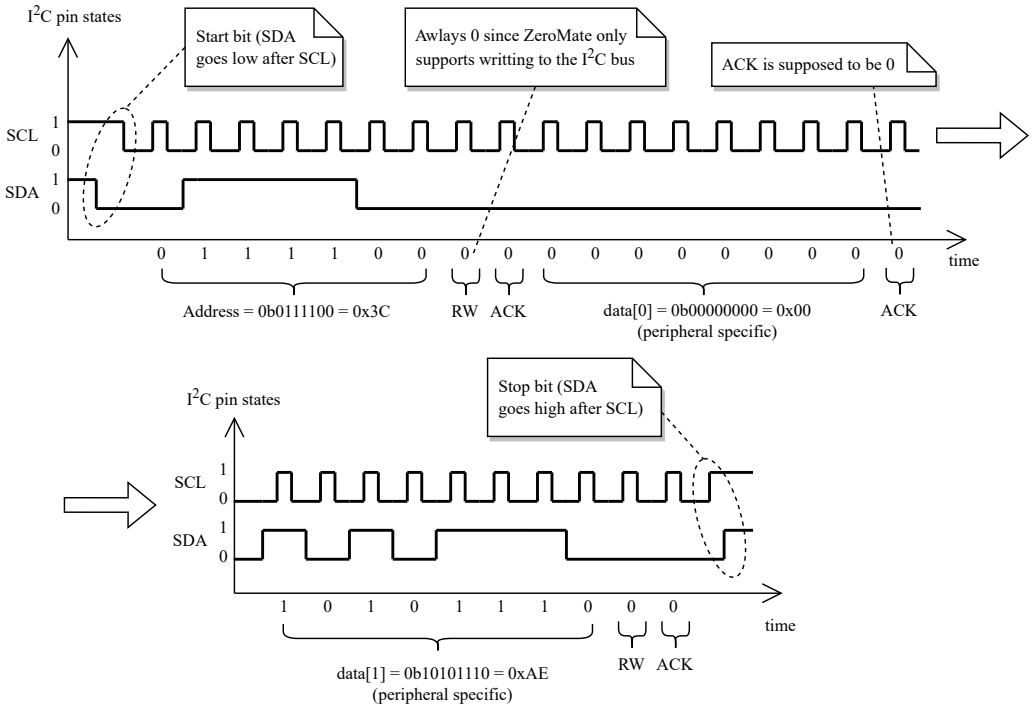

Figure 1.28: Structure of an I²C frame

The ZeroMate emulator comes with several standalone external peripherals that employ the same communication interface. Users can connect these peripherals to the system based on their preferences using a configuration file, which is further discussed in section 1.4.

1.3.4 ARM1176JZF_S

All the previously mentioned peripherals are not self-sufficient in operation; they require external control. This is where `ARM1176JZF_S`, which serves as the central processing unit in the `BCM2835` microcontroller, comes into play, as it executes individual `32-bit` `ARM` instructions that may utilize these peripherals in various ways.

In terms of architecture, the `ARM1176JZF_S` component is divided into several tightly integrated building blocks, as shown in figure 1.29, to maintain a structured and organized design. The subsequent sections detail how each of these subperipherals contributes to the overall emulation of the `ARM1176JZF_S` processor.

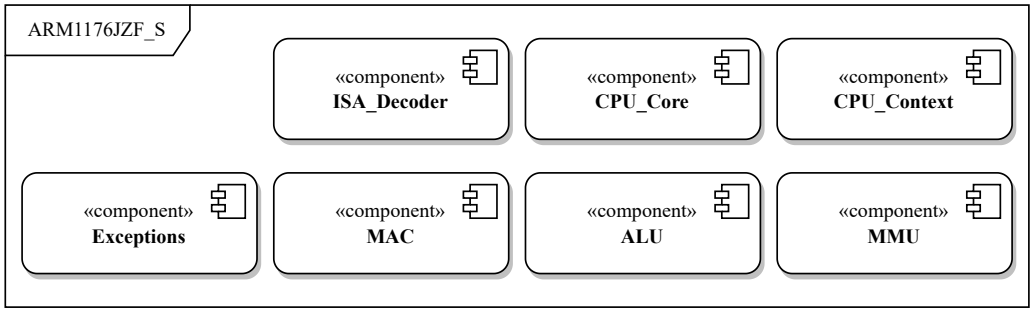

Figure 1.29: Internal components of the `ARM1176JZF_S` processor

1.3.4.1 Central Processing Unit Context

The CPU context functions as an encapsulation of the current state of the CPU, including details such as the current contents of the registers and the active CPU mode. It is designed to provide an interface for accessing registers, enabling transitions between different modes, and offering other utility functions that abstract the underlying low-level logic, such as reading the state of individual bits of the control register.

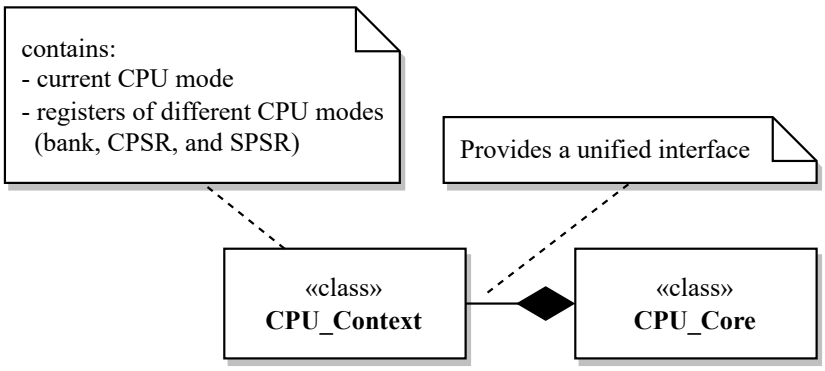

Figure 1.30: Relationship between the CPU context and the CPU core

Bank Registers.

One of the presented challenges involves implementing so-called *bank* registers, where each CPU mode possesses its distinct subset of registers that are automatically loaded when the mode changes. Additional information regarding this topic is available in section A2.3 of the official ARM Architecture Reference Manual [13].

As illustrated in listing 1.6, one approach to address this challenge involves creating a lookup table for all bank registers within each CPU mode, effectively

replacing the ones used in the User/System mode, which are otherwise used by default.

Source code 1.6: Retrieving a CPU register

```
uint32_t& Get_Register(uint32_t idx, NCPU_Mode mode)
{
    // Check if a banked register should be returned
    if (m_banked_regs.at(mode).contains(idx))
        return m_banked_regs[mode][idx];

    // A non-banked register is being addressed
    return m_banked_regs[NCPU_Mode::System][idx];
}
```

Control Registers.

The current program state register and the saved program state registers, commonly known as the CPSR and SPSR registers, are implemented in a similar way, with each CPU mode having its designated pair of these registers.

1.3.4.2 Instruction Set Architecture Decoder

The primary role of the instruction set architecture decoder, abbreviated as the ISA decoder, is to analyze a 32-bit value and ascertain the type of the ARM instruction it represents, so it could be treated and decoded accordingly. This task must be executed with the utmost efficiency, as it is repetitively performed for each instruction the CPU executes.

In the case of the emulated CPU, the ISA decoder functions as a „black box“ offering a single-function interface, as demonstrated in listing 1.7 below.

Source code 1.7: Interface of the ISA decoder

```
// Returns the type of a given 32-bit ARM instruction
[[nodiscard]] static CInstruction::NType
Get_Instruction_Type(uint32_t instruction) noexcept;
```

The typical and sole use-case of this interface is further illustrated in the sequence diagram shown in Figure 1.31. In this diagram, the CPU fetches the next instruction from RAM and employs the ISA decoder to determine its type, enabling it to proceed with the execution.

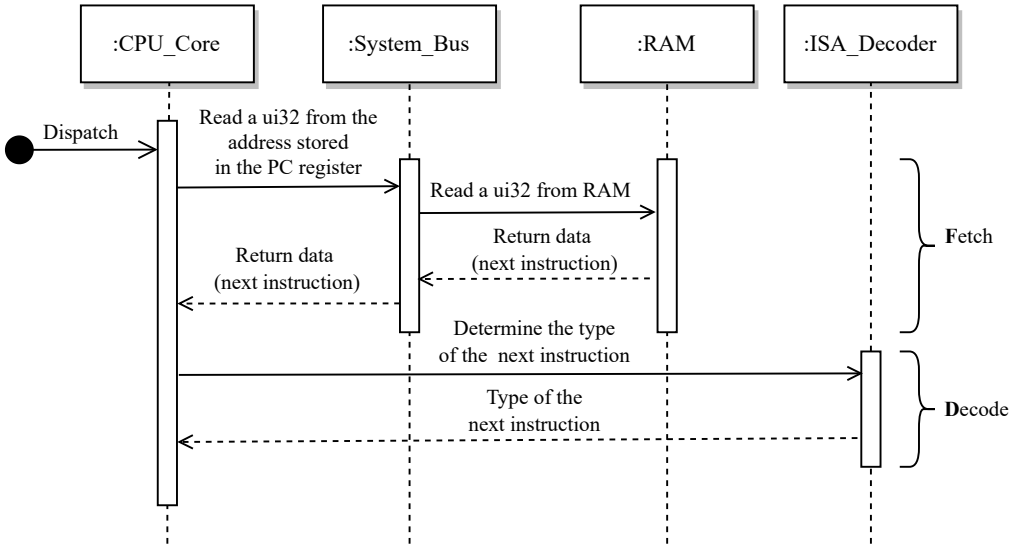

Figure 1.31: Use of the `ISA` decoder by the `CPU`

Internally, the `ISA` decoder maintains a look-up table of instructions masks that are sequentially applied to the given `32-bit` value until the operation's result matches the expected value associated with the current mask. Each mask serves the purpose of zeroing out the variable bits specific to the instruction, leaving only the known bits in place, the expected value, which is then used to unambiguously determine the type of instruction.

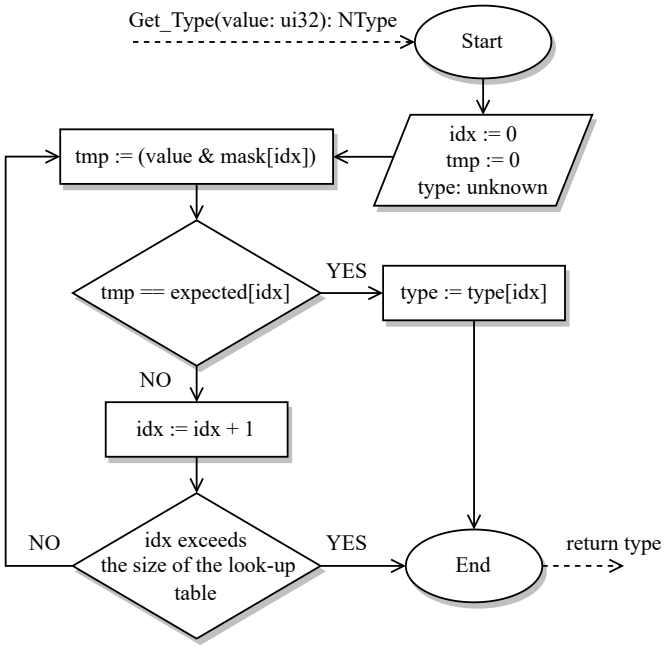

Figure 1.32: Algorithm for decoding `ARM` instructions

Figure 1.32 shows the algorithm for decoding ARM instructions, which, in the worst-case scenario, operates with time complexity of $O(n)$, where n represents the size of the look-up table. **To ensure unambiguous decoding, it is essential to test the bit masks in a specific order, starting with the most restrictive one and proceeding to the least restrictive one, based on the number of bits set within each mask**. Otherwise, there might be a risk of incorrectly identifying the instruction type, inevitably leading to unexpected behavior.

Once an instruction is classified, the `32-bit` value can be encapsulated within its corresponding class representation, providing an interface to access the specific fields relevant to that instruction. The contents of the look-up table can be constructed using resources such as the B2 Appendix authored by Andrew Sloss and Chris Wright, which provides ARM instruction encodings [14].

1.3.4.3 Exceptions

Exceptions can originate from various components within the ZeroMate emulator. These exceptions may arise from factors such as the absence of an addressed page, unaligned memory access, or the execution of a privileged instruction in a non-privileged CPU mode. ZeroMate handles ARM CPU exceptions as run-time errors on the host machine. As a result, the emulated CPU core must be prepared for the possibility that the currently executing instruction may abruptly trigger an exception that must be properly handled. Figure 1.33 shows the hierarchy of CPU exceptions, where each class may include additional information pertaining to the exception, such as a descriptive message, the address at which it occurred, or the address of the vector associated with that specific exception.

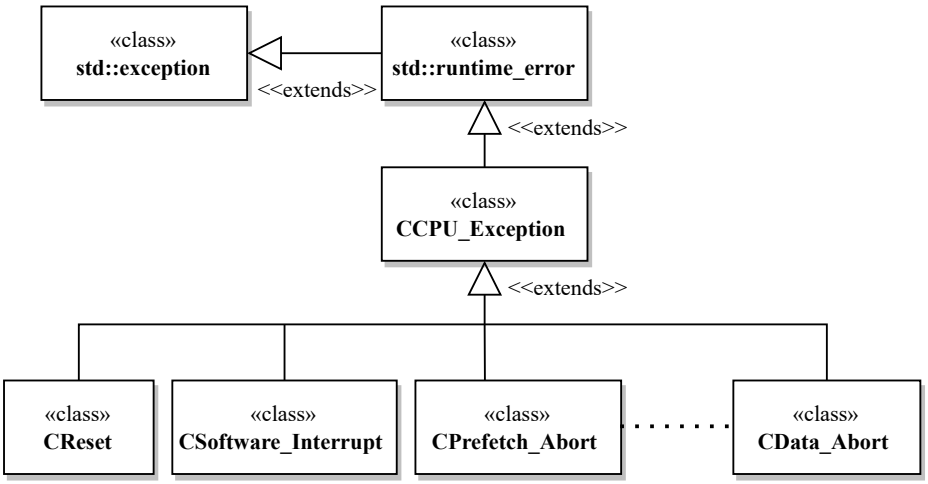

Figure 1.33: Hierarchy of the CPU exceptions

1.3.4.4 Central Processing Unit Core

In terms of implementation, the CPU is composed of multiple private member functions that are invoked based on the type of instruction currently being executed. These functions can be seen as microprograms, as they handle the specific operations associated with the current instruction.

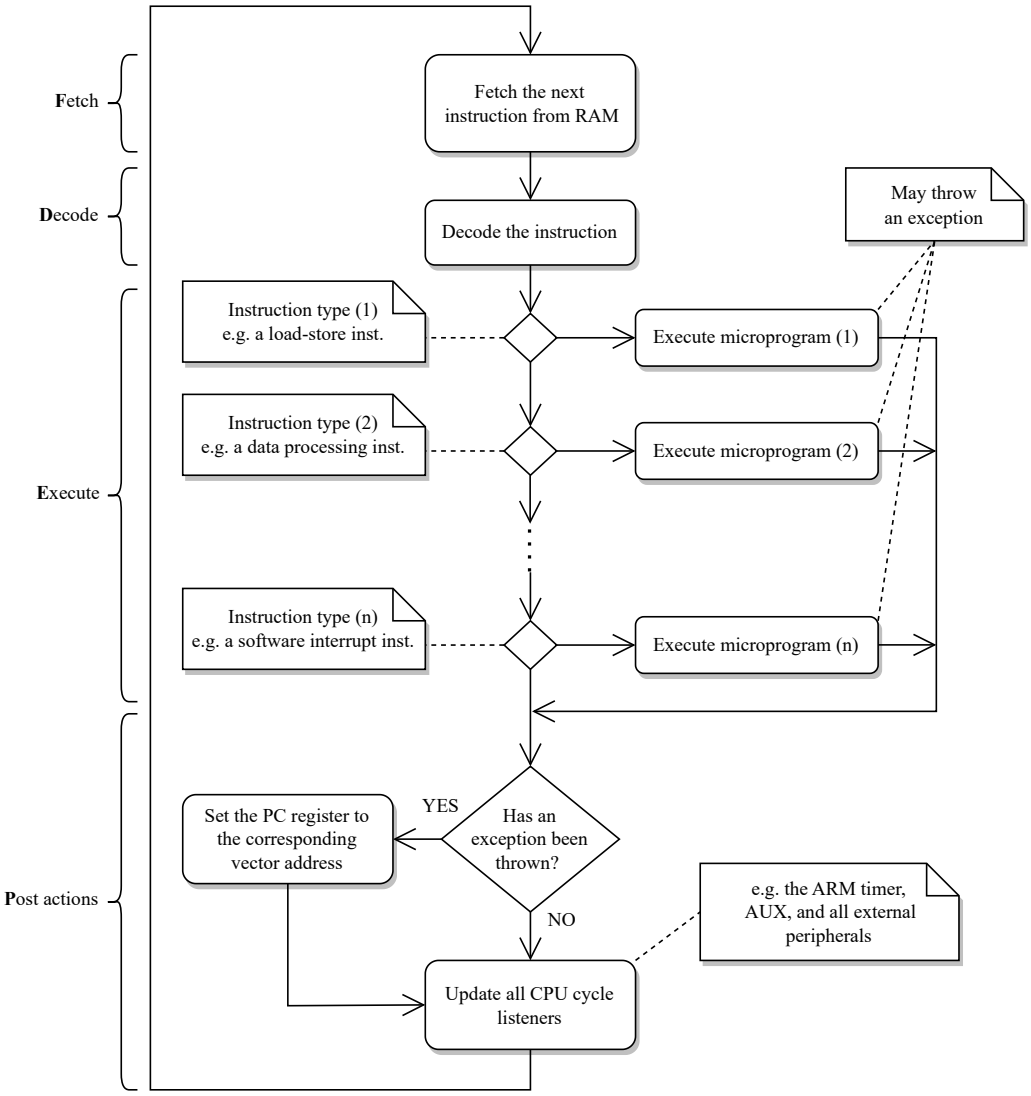

Figure 1.34: Execution loop of the emulated CPU

As shown in figure 1.34, the microprograms are called within an infinite loop known as the execution loop, which fetches the next instruction from RAM, decodes it using the ISA decoder, and directs the execution to the appropriate microprogram responsible for carrying out the necessary steps to execute the instruction. These

steps may include tasks such as reading data from the stack, modifying register contents, utilizing the arithmetic-logic unit, and more.

> It is worth noting that a real system is significantly more complex than how ZeroMate implements it. However, the author would contend that any form of emulation involves trade-offs that result in the omission of certain details.

Catching Exceptions.

As described in section 1.3.4.3, the CPU must handle any exceptions that may arise during the execution of the current instruction. When an exception occurs, the CPU switches to the corresponding mode, saves the return address in the link register, as if it were invoking a function call, and sets the address of the next instruction to the value stored at the corresponding offset in the interrupt vector table.

Updating CPU clock listeners.

As mentioned in section 1.3.3.2, before moving on to the next instruction, the CPU notifies all peripherals that are subscribed as system clock listeners about how many CPU cycles it took to execute the last instruction.

> While it is true that each instruction may require a varying number of CPU cycles for execution, in the current implementation, ZeroMate empirically averages this number to 8, which provides a potential area for future improvement. Users should be aware that the emulation speed does not match the speed of real hardware, and they may need to adjust their timings accordingly.

1.3.4.5 Memory Management Unit

1.3.5 Coprocessor 15

1.3.6 Coprocessor 10

1.4 External Peripherals

1.5 User Interface

1.6 Logging System

1.7 Technologies

Bibliography

1. COMMITTEE, TIS. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Linux Foundation. Available also from: `https://refspecs.linuxfoundation.org/elf/elf.pdf`.
2. BROADCOM. *BCM2835 ARM Peripherals*. Broadcom Corporation, 2012. Available also from: `https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf`.
3. COPLIEN, James O. *Multi-Paradigm Design for C++*. Taschen, 1998. ISBN 978-0201824674.
4. BALOGUN, Ghaniyyat Bolanle. *A Comparative Analysis of the Efficiencies of Binary and Linear Search Algorithms*. Department of Computer Science, University of Ilorin, Ilorin, Nigeria., 2020. Available also from: `https://afrjcict.net/wp-content/uploads/2020/03/Vol13No1Mar20pap3journalformatpagenumb.pdf`.
5. KANKOWSKI, Peter. *x86 Machine Code Statistics*. 2008. Available also from: `https://www.strchr.com/x86_machine_code_statistics`.
6. *The GNU C++ Library - Chapter 28. Demangling*. The GNU Compiler Collection. Available also from: `https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html`.
7. LAMIKHOV-CENTER, Serge. *ELFIO - ELF (Executable and Linkable Format) reader and producer implemented as a header-only C++ library*. Available also from: `https://elfio.sourceforge.net`.
8. WEBER, Nico. *Demumble - A better c++filt and a better undname.exe, in one binary*. Available also from: `https://github.com/nico/demumble`.
9. *BCM2835 datasheet errata*. Embedded Linux Wiki. Available also from: `https://elinux.org/BCM2835_datasheet_errata`.
10. TATHAM, Simon. *PuTTY - free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator*. Available also from: `https://www.putty.org`.

11. TORVALDS, Linus. *Linux kernel*. Available also from: https://github.com/torvalds/linux/blob/master/drivers/char/hw_random/bcm2835-rng.c.

12. BHATTACHARJEE, Kamalika; DAS, Sukanta. *A search for good pseudo-random number generators: Survey and empirical studies*. 2022. Available also from: https://www.sciencedirect.com/science/article/pii/S1574013722000144.

13. *ARM Architecture Reference Manual*. ARM. Available also from: https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a?token=.

14. SLOSS, Andrew; WRIGHT, Chris. *ARM and Thumb Instruction Encodings (B2 APPENDIX)*. ARM. Available also from: https://gab.wallawalla.edu/~curt.nelson/cptr380/textbook/advanced%20material/Appendix_B2.pdf.

List of Abbreviations

RAM - Random Access Memory
ELF - Executable Linkage Format
GPU - Graphics Processing Unit
SREC - Motorola S-record (file format)
GUI - Graphical User Interface
EEPROM - Electrically Erasable Programmable Read-only Memory
CPU - Central Processing Unit
ALU - Arithmetic-Logic Unit
MAC - MAC Unit (performs multiply-accumulate operations)
MMU - Memory Management Unit
ISA - Instruction Set Architecture
MMIO - Memory-mapped Input-Output device
I/O - Input-Output
R/W - Read-Write
ABI - Application Binary Interface
RTTI - Run-Time Type Information
GPIO - General Purpose Input/Output
LED - Light-emitting diode
SD - Secure Digital
IVT - Interrupt Vector table
QA - Quality Assurance
UART - Universal Asynchronous Receiver/Transmitter
TRNG - True Random Number Generator
LCG - Linear Congruential Generator
OS - Operating System
I^2C - Inter-Integrated Circuit
CPI - Cycles Per Instruction
IC - Interrupt Controller
PWM - Pulse Width Modulation
IRQ - Interrupt Request
FIQ - Fast Interrupt Request

`AUX` - Auxiliary
`MSB` - Most Significant Bit
`LSB` - Least Significant Bit
`TX` - Transmit
`RX` - Receive
`BSC` - Broadcom Serial Controller
`CPSR` - Current Program Status Register
`SPSR` - Saved Program Status Register

List of Figures

List of Tables

List of Listings