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

Design of a Raspberry Pi Zero Emulator — 1

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 selected name of the project [1].

1.1 Input

The emulator requires a single input file in the `ELF` format. 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 `ELF` File

`ELF` stands for *Linkage Executable 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`.

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

[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“.

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

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 1.4.

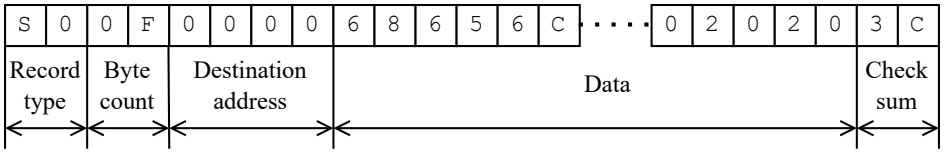

Figure 1.1: Single SREC record (16-bit addressing)

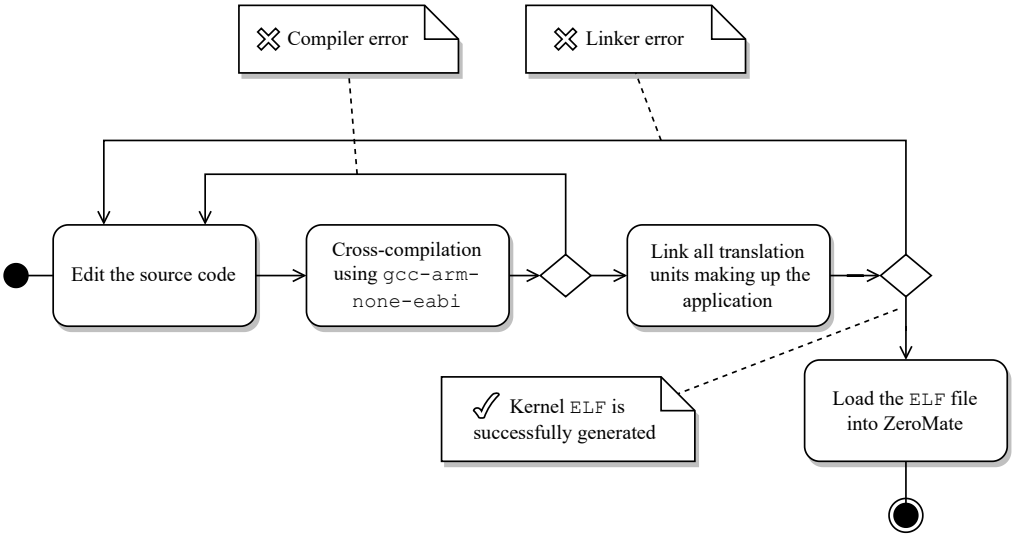

Figure 1.2: Process of building an ELF file (input for the emulator)[3]

[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 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.

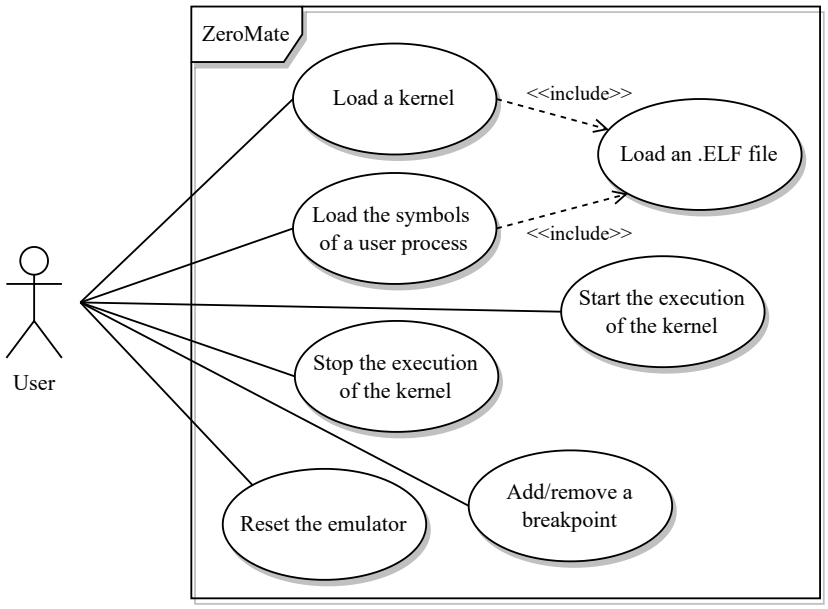

Figure 1.3: Primary use-cases of the ZeroMate emulator

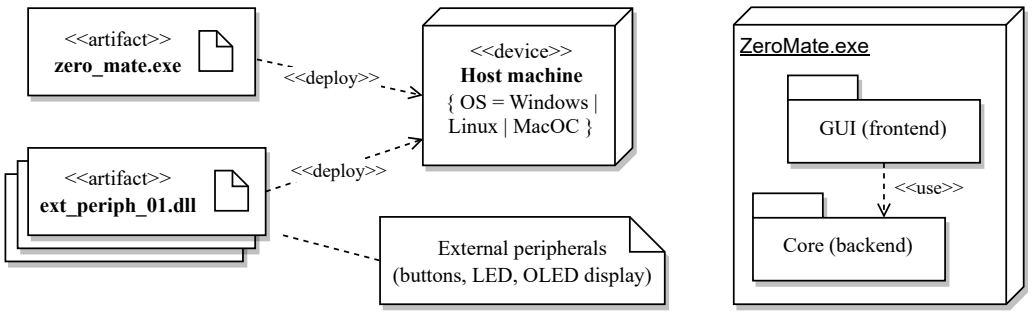

Figure 1.4: Deployment diagram of the ZeroMate emulator

The main application is designed as a **two-tier architecture**. In this arrangement, the top layer, which is the `GUI` 1.4, 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.

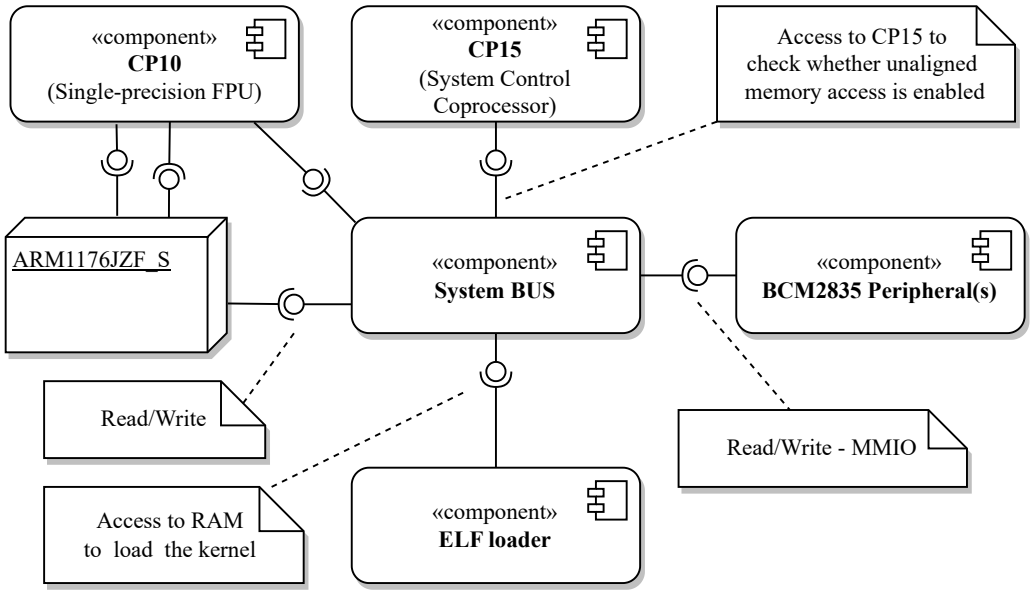

Figure 1.5: Core components of the ZeroMate emulator

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.

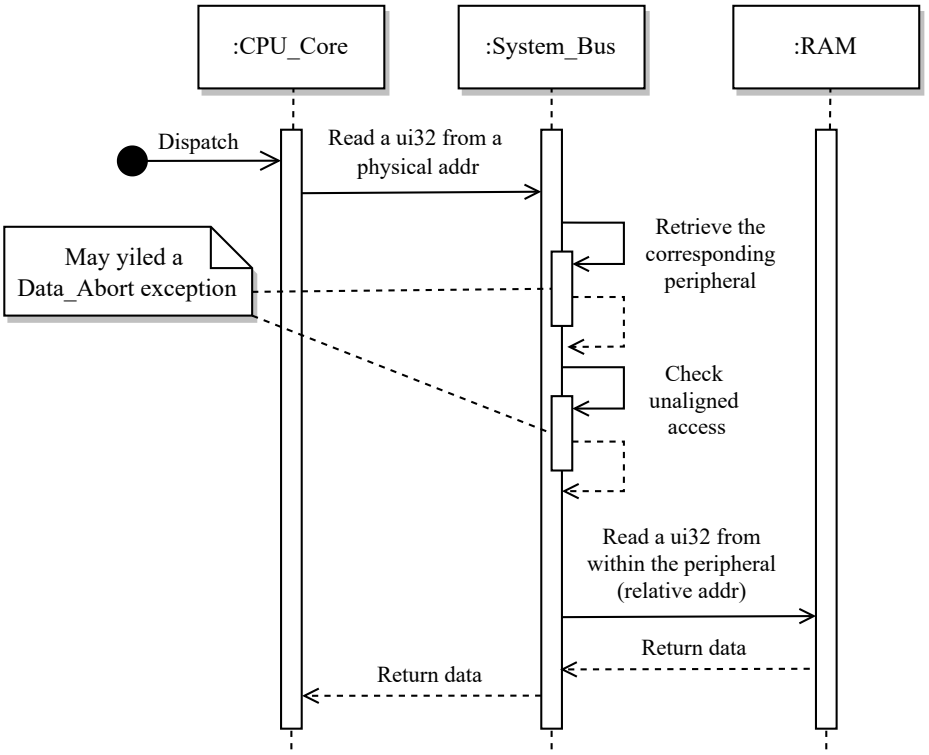

Figure 1.6: Example of a read/write data request issued by the CPU

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.

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 [3], which is crucial for improving the overall speed of the emulation.

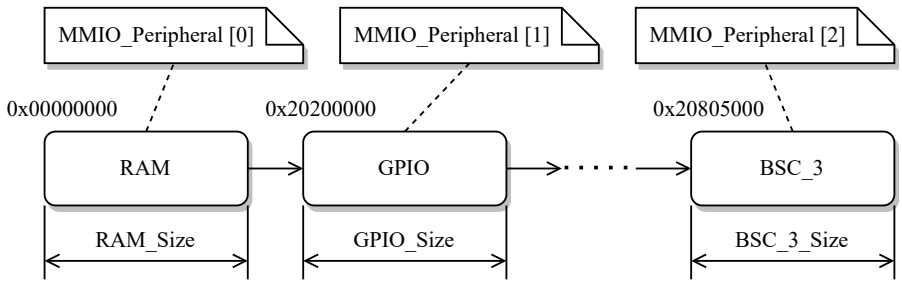

Figure 1.7: Collection of memory-mapped peripherals

[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.

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 size of the data**. 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
```

1.3.2 `ELF` 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&)"
```

[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.

[7] Demangling is a process of transforming `C/C++` `ABI` identifiers (like `RTTI` symbols) into the original `C/C++` source [4].

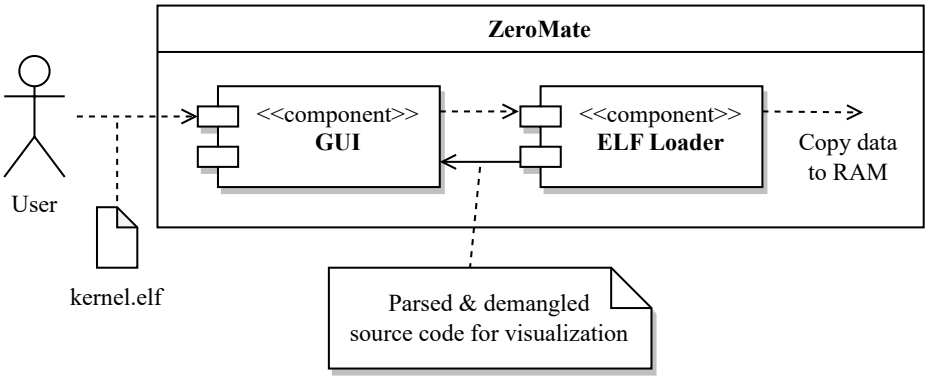

Figure 1.8: Loading an input ELF file (kernel)

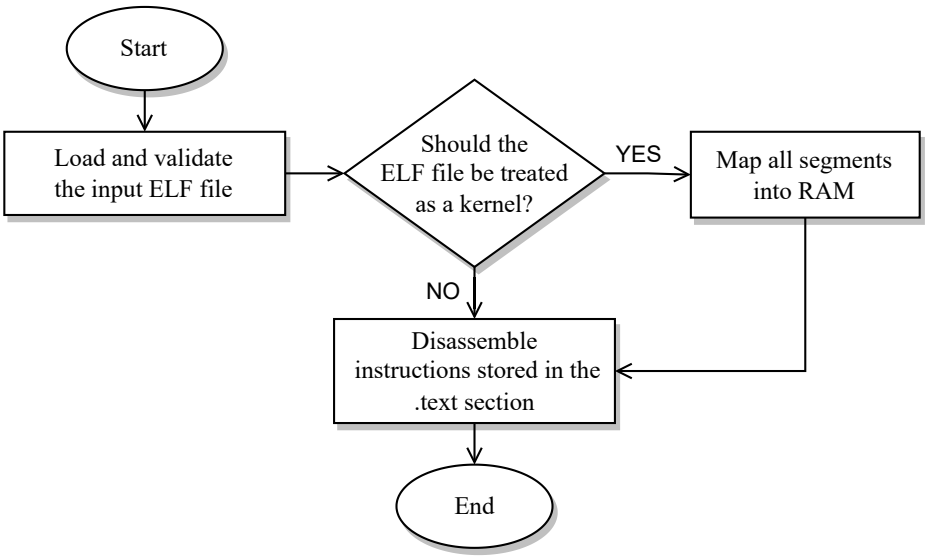

Figure 1.9: Internal logic of the ELF Loader component

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 libraries, *ELFIO* [5] and *Demumble* [6], to accomplish these functions.

1.3.3 Peripherals

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

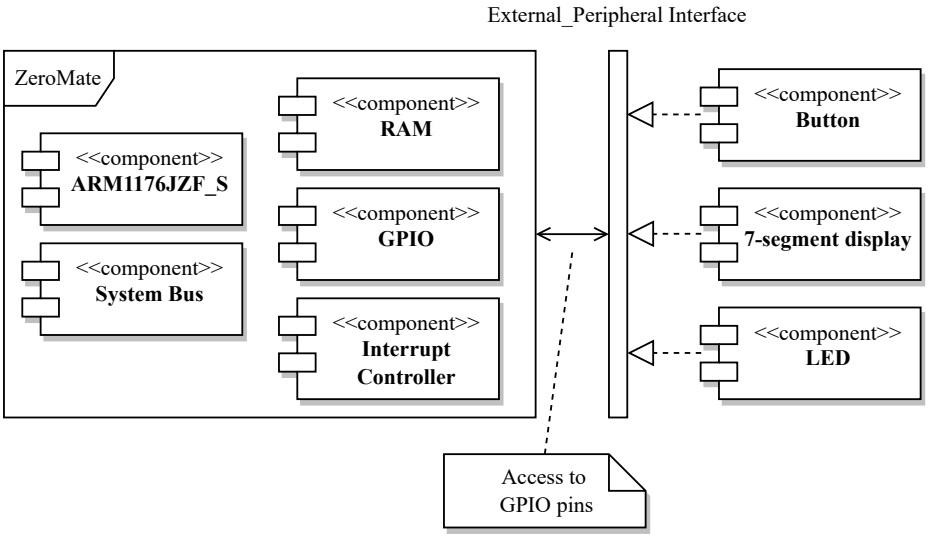

Figure 1.10: Internal vs External peripherals

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.

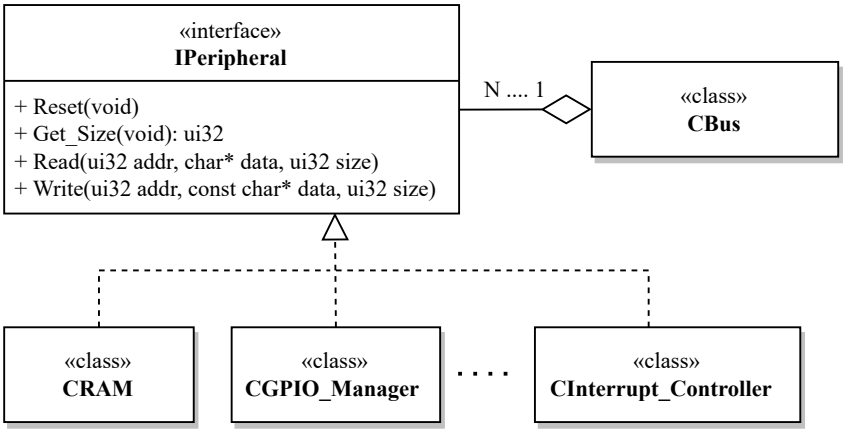

Figure 1.11: Hierarchy of internal peripherals

[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 [7].

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

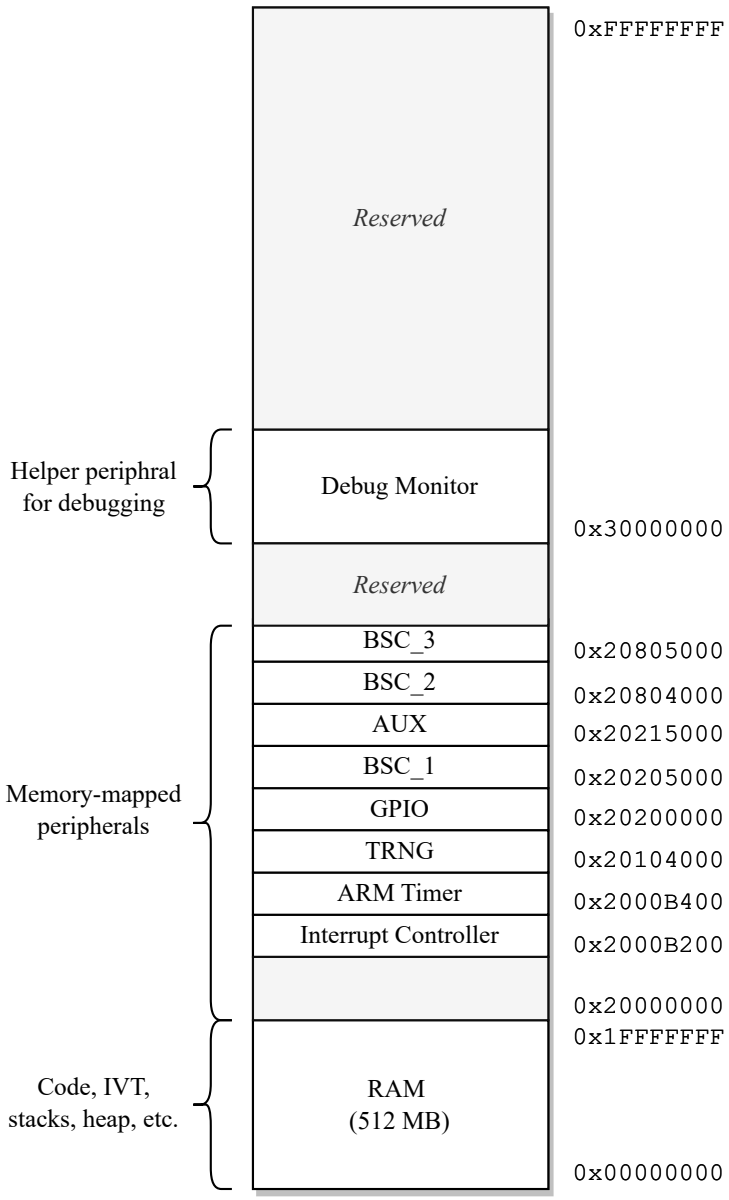

Figure 1.12: `BCM2835` physical memory layout emulated by ZeroMate

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 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 1.3.3.5 or the AUX 1.3.3.8 and BSC 1.3.3.9 peripherals, which encapsulate time-based hardware communication functions.

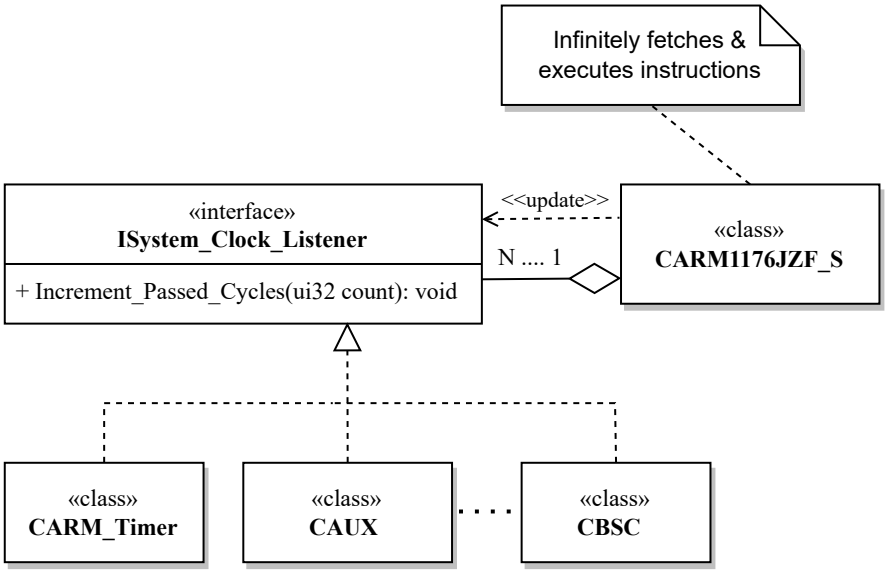

Figure 1.13: `ISystem_Clock_Listener` interface

It is important to mention that **updating a system clock listener is, from the 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 using a separate thread. However, this approach would introduce additional concurrency challenges that would require thorough consideration.

1.3.3.2 RAM

From a simplified 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. The Raspberry Pi Zero board is equipped with an SD card, which serves as non-volatile [9] memory for storing the kernel image. For emulation purposes, 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 [10], 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 1.14 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.

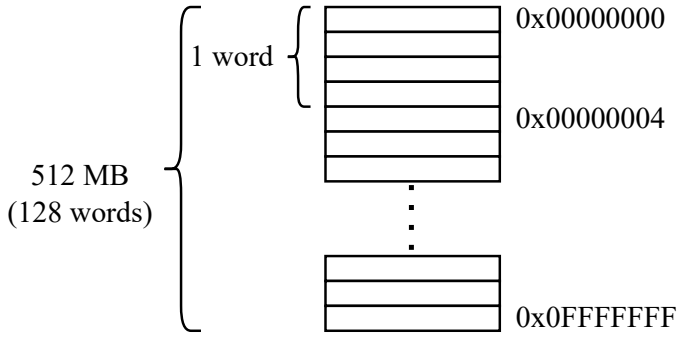

Figure 1.14: RAM implementation as a continuous piece of memory

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.

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

1.3.3.3 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.

ZeroMate also comes with a simple driver of the peripheral that the user can effortlessly integrate into their build system. This allows them to use „print-like“ functions they might be used to 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;
}
```

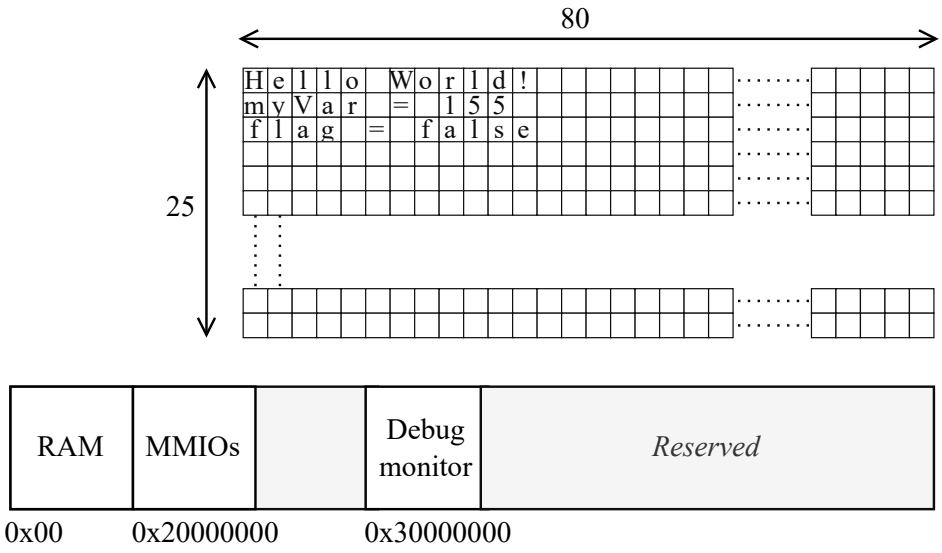

Figure 1.15: Memory-mapped debug monitor

To attain these same capabilities in practice, the user would need to utilize a form of serial communication, such as `UART`, through which they can transmit characters to an external device [11]. This is commonly achieved by running software like *PuTTY* [8] on the user's computer.

As shown in figure 1.15, the debug monitor is mapped to an unoccupied address `0x30000000`. It is structured as a flat memory layout, which is managed by the driver the user code interacts with. The size of the monitor was chosen to be `80x25 8-bit` characters [12].

1.3.3.4 TRNG

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 Linux kernel [9].

For simplification purposes, ZeroMate primarily focuses on providing random numbers while omitting 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 [13].

From the user's code perspective, the process of **retrieving a random number consists of two steps**, which are displayed in figure 1.16.

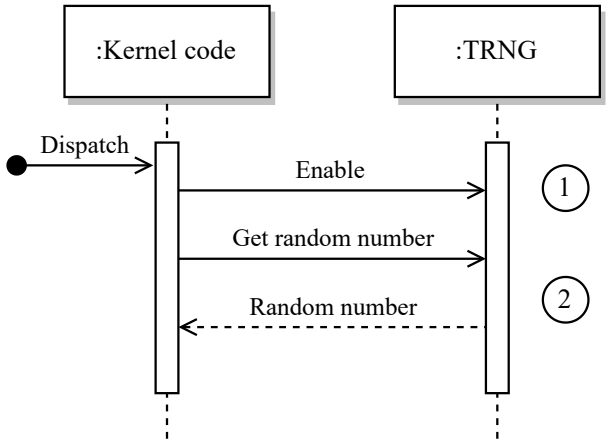

Figure 1.16: Reading random numbers from the `TRNG` peripheral

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

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

[13] 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 earlier.

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.

> However, 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* [10] or *Mersenne-Twister*, **can greatly improve the performance of the emulation**. Depending on the implementation, accessing a true random number generator via the host machine 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.

1.3.3.5 ARM Timer

The ARM timer is commonly used to periodically trigger interrupts, whether it is for toggling an LED or switching the current CPU context, which is an integral part of any preemptive OS scheduler. The configuration of the peripheral is done via its registers that are listed in the BCM2835 manual [2].

As shown in figure 1.17, 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 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.

In ZeroMate, all time-related functionalities, such as the ARM timer, UART, or I^2C, are for synchronization purposes, inherently derived from the CPU's clock.

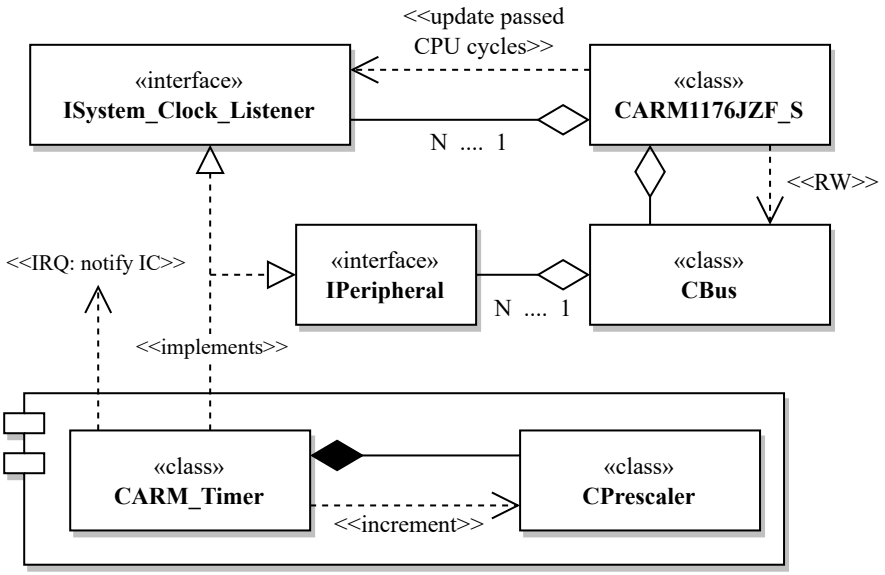

Figure 1.17: Context of the ARM timer peripheral

As mentioned earlier, 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 when it reaches zero. If enabled, with each such event, the timer will trigger an interrupt. This concept is visualized in figure 1.18.

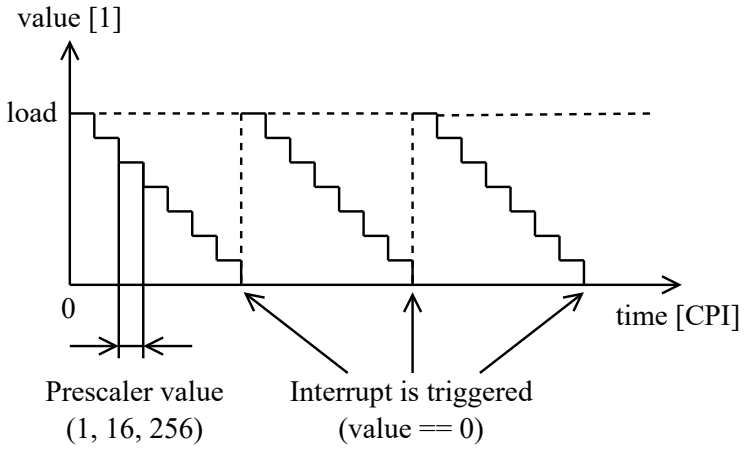

Figure 1.18: Content of the value register of the ARM timer over time

1.3.3.6 GPIO Manager

1.3.3.7 Interrupt Controller

1.3.3.8 AUX

1.3.3.9 BSC

1.3.4 ARM1176JZF_S

1.4 User Interface

1.5 External Peripherals

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. 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`.
4. *The GNU C++ Library - Chapter 28. Demangling*. The GNU Compiler Collection. Available also from: `https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html`.
5. 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`.
6. WEBER, Nico. *Demumble - A better c++filt and a better undname.exe, in one binary*. Available also from: `https://github.com/nico/demumble`.
7. *BCM2835 datasheet errata*. Embedded Linux Wiki. Available also from: `https://elinux.org/BCM2835_datasheet_errata`.
8. 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`.
9. TORVALDS, Linus. *Linux kernel*. Available also from: `https://github.com/torvalds/linux/blob/master/drivers/char/hw_random/bcm2835-rng.c`.
10. 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`.

List of Abbreviations

RAM - Random Access Memory
ELF - Executable Linkage Format
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
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

List of Figures

List of Tables

List of Listings