Integrating a Virtual Machine on a system-on-a-chip

Boro Sitnikovski[1] and Biljana Stojcevska[2]

Abstract – **Manufactured systems on a chip (SoC) with the current technologies cannot be modified without making hardware changes, which makes their alteration challenging. In this paper, we present the concept of implementing a minimal virtual machine (VM) on an SoC, as a possible solution to this problem. Our concept is performant and allows for dynamic program updates of the SoC without the need to reprogram the ROM, whenever the requirements change. The only prerequisite is for the system to be equipped with network capabilities.**

Keywords – **Virtual machines, CHIP-8, Arduino ESP32, C/C++, instruction set.**

I. Introduction

Flashing specific software to ROM is a common practice when it comes to SoCs. But as the nature of the chip requirements is ever-changing, it would be practical if there is a way to alter the functionality of the system without having to flash it. In this paper, we will present a solution to this challenge by using an Arduino device (ESP WROOM32) as a proof-of-concept, together with a modified version of the CHIP-8 VM.

II. Implementation

A. Arduino

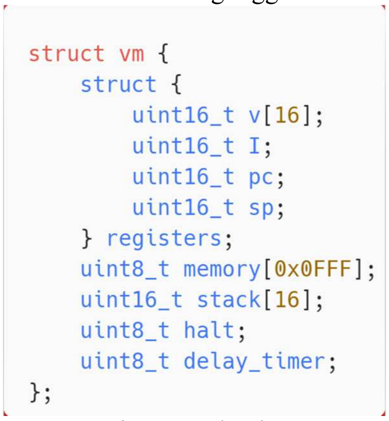
Arduino is an open-source platform based on software and hardware. It allows boards to be programmed such that they read inputs (temperature sensor, light sensor, etc.), and write outputs (turn on an LED, write to a screen, etc.) [1] in a standardized way.

Manufactured by Espressif Systems [2], ESP32 is a series of low-cost, low-power SoC microcontrollers with integrated Wi-Fi and Bluetooth, which can be programmed using the Arduino IDE.

B. Modified CHIP-8

A VM represents an emulation of a computer system that can run operating systems and software and perform all the functions of the physical computer system. The CHIP-8 VM was designed by Joseph Weisbecker in the 1970s [3], mainly intended to be used for game programming. What makes CHIP-8 interesting is Turing completeness, and as such, can compute everything any computer system can. Our choice was to use CHIP-8 for our proof of concept because it provides all needed functionality and yet it is simple to implement [4].

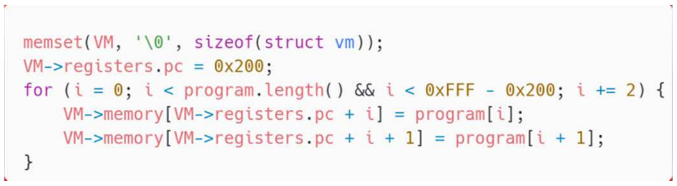
Though for this project, we use a modified version of CHIP-8, where the gaming-related instructions are removed, and instead, instructions for controlling IO pins are added. Another alteration is that the registers will be 16-bit, rather than 8-bit, to allow for calculating bigger numbers.



```c
struct vm {
    struct {
        uint16_t v[16];
        uint16_t I;
        uint16_t pc;
        uint16_t sp;
    } registers;
    uint8_t memory[0x0FFF];
    uint16_t stack[16];
    uint8_t halt;
    uint8_t delay_timer;
};
```

Fig. 1. VM header

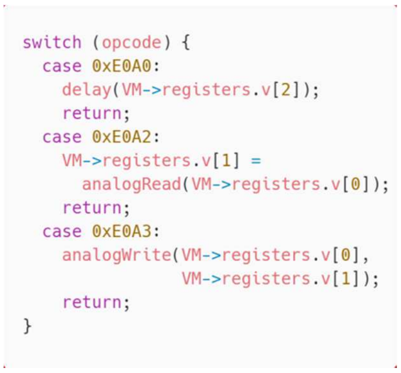
The function in Fig. 2 is used to initialize the default values and the program for the machine (the header in Fig. 1), given a string of bytes.



```c
memset(VM, '\0', sizeof(struct vm));
VM->registers.pc = 0x200;
for (i = 0; i < program.length() && i < 0xFFF - 0x200; i += 2) {
    VM->memory[VM->registers.pc + i] = program[i];
    VM->memory[VM->registers.pc + i + 1] = program[i + 1];
}
```

Fig. 2. The `vm_init` function

The heart of the VM lies in the evaluation of opcodes, `eval_opcode` (Fig. 3).



```c
switch (opcode) {
  case 0xE0A0:
    delay(VM->registers.v[2]);
    return;
  case 0xE0A2:
    VM->registers.v[1] =
      analogRead(VM->registers.v[0]);
    return;
  case 0xE0A3:
    analogWrite(VM->registers.v[0],
                VM->registers.v[1]);
    return;
}
```

Fig. 3. The `vm_eval` function

We remove the unnecessary instructions, and extend the instruction set within it with instructions to communicate with IO ports, as per Table 1:

[1]Boro Sitnikovski works as a Senior Software Engineer at Automattic. E-mail: buritomath@gmail.com

[2]Biljana Stojceska is with the University of Tourism and Management, Faculty of Informatics, Skopje, North Macedonia. E-mail: b.stojcevska@utms.edu.mk

TABLE I
MODIFIED CHIP-8 INSTRUCTION SET

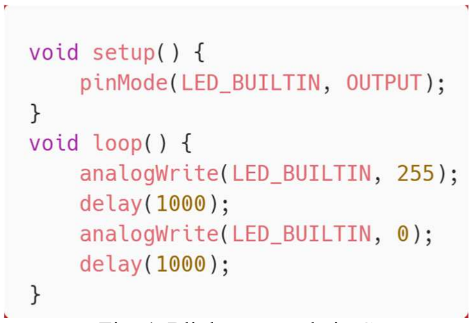
Instruction	Purpose	Delta
00E0	Gaming	Removed
DXYN	Gaming	Removed
EX9E	Gaming	Removed
EXA1	Gaming	Removed
FX0A	Gaming	Removed
FX18	Gaming	Removed
FX29	Gaming	Removed
E0A0	Delay	Added
EXA1	Set pin mode	Added
E0A2	Analog read	Added
E0A3	Analog write	Added

The instruction set can be further extended with other instructions that map to the Arduino API, as necessary.

III. EXAMPLES

A. Blinker

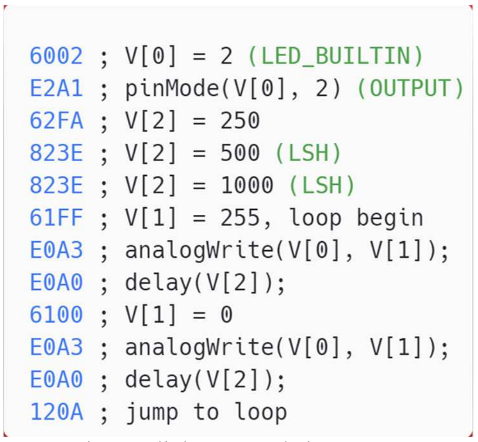
In Fig. 4 we show the Blinker example code from the Arduino's official documentation [5], adjusted to use `analogWrite`. The code will repeatedly turn on and off an LED, with a 1-second delay between each operation.



```
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
    analogWrite(LED_BUILTIN, 255);
    delay(1000);
    analogWrite(LED_BUILTIN, 0);
    delay(1000);
}
```

Fig. 4. Blinker example in C

The same code for Blinker can be represented as follows in the modified CHIP-8 instruction set, as shown in Fig. 5.



```
6002 ; V[0] = 2 (LED_BUILTIN)
E2A1 ; pinMode(V[0], 2) (OUTPUT)
62FA ; V[2] = 250
823E ; V[2] = 500 (LSH)
823E ; V[2] = 1000 (LSH)
61FF ; V[1] = 255, loop begin
E0A3 ; analogWrite(V[0], V[1]);
E0A0 ; delay(V[2]);
6100 ; V[1] = 0
E0A3 ; analogWrite(V[0], V[1]);
E0A0 ; delay(V[2]);
120A ; jump to loop
```

Fig. 5. Blinker example in CHIP-8

In terms of space, the code above takes only 24 bytes as well as keeping the values of the registers, stack pointer, etc. which is light overhead. In terms of runtime performance, the overhead is the switch statement when determining which opcode to evaluate, which is also light overhead.

B. Serving programs from HTTP

The program can be fed through any input, most commonly through HTTP [6] as it allows more flexibility if we assume the existence of a web application that allows users to easily generate VM programs through a UI, as shown in Fig 6.

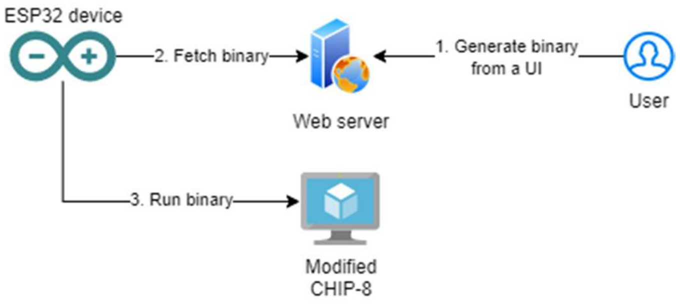


Fig. 6. Architecture

This can be especially useful for projects like the Internet of Things, where devices have the potential to be re-programmed periodically by the users.

IV. CONCLUSION

In this paper, we showed an alternative way of representing programs in embedded devices without needing to physically flash the device whenever there is a need for reprogramming. There are other solutions to address the reprogramming problem [7], however, they have a bigger performance impact on the device.

A variant of the virtual machine was used in a formal verification project to ensure that the programs running on the SoC are correct [3].

REFERENCES

[1] Arduino. [Online]. Available: https://www.arduino.cc/ (Accessed: Feb. 2022)
[2] Espressif Systems. [Online]. Available: https://www.espressif.com/ (Accessed: Feb. 2022)
[3] B. Sitnikovski. "Formal verification of Instruction Sets in Virtual Machines", Master thesis, 2019.
[4] B. Sitnikovski, "CHIP-8 implementation in C". [Online]. Available: https://github.com/bor0/chip-8 (Accessed: Aug. 2014)
[5] Arduino, "Blink". [Online]. Available: https://www.arduino.cc/en/Tutorial/BuiltInExamples/Blink/ (Accessed: Feb. 2022)
[6] B. Sitnikovski, "evm-esp32". [Online]. Available: https://github.com/bor0/evm-esp32 (Accessed: Feb. 2022)
[7] MicroPython. [Online]. Available: https://micropython.org/ (Accessed: May 2022)