3bOS: A flexible and lightweight embedded OS operated using only 3 buttons

Encarnacion, Immanuel V[†1,1,a)]   Ryohei Kobayashi[1,b)]   Kenji Kise[1,c)]

Abstract: An embedded system we developed, the MieruEMB system, is used as an educational kit for learning implementation skills and knowledge regarding embedded systems. In this paper we present 3bOS, a simple and easily customizable embedded OS, running on the MieruEMB system. 3bOS comes with a three-button interface and a built-in file explorer for FAT file systems. 3bOS is capable of running ELF executables, providing approximately 400 KB of memory for an application. It can also support basic graphics functions. This embedded OS is written in C, and just consists of around 800 lines of the code. Because of its simplicity, users can easily understand how this embedded OS runs on the MieruEMB system, and can easily modify this embedded OS if they want. We show the design, the implementation, and the features of 3bOS, and conclude that 3bOS is usable for educational purposes.

Keywords: Embedded System, FPGA, Lightweight, OS, Education

1. Introduction

Recently, the growth in popularity of embedded systems have enabled their use in several fields and applications, such as car electronics, medical products, and communication devices. These backgrounds desire engineers who have skills and knowledge about embedded systems.

We have developed an embedded system, named MieruEMB[1], as an educational kit. **Fig. 1** shows the MieruEMB system. The MieruEMB is composed of a MIPS processor on an FPGA(Xilinx Spartan XC3S500E), along with a 512 KB SRAM. The I/O is composed of three push buttons, a 128 × 128 pixel 16-bit color LCD screen, an SD card slot. This embedded system is used in experiments on computer science[2] to learn implementation skills and knowledge regarding embedded systems, ranging from soldering to software application development (**Fig. 2**).

This paper presents 3bOS, an operating system designed with simplicity and flexibility in mind. 3bOS runs on the MieruEMB system, and is operated using only three push buttons. This OS aims to be used for educational purposes; users can easily understand how this OS runs on the MieruEMB system, and they can easily implement their desired features. The operating system itself is stored in the SD card, along with some files and programs. 3bOS is capable of reading FAT-formatted SD cards. The OS has a built-in file explorer, which can be used to navigate the folder structure, as well as read and scan file contents.

Besides, the OS is capable of running ELF executables which are compiled for 32-bit MIPS. Compiling the ELF file is simple; it only needs a few included libraries for it to work properly.

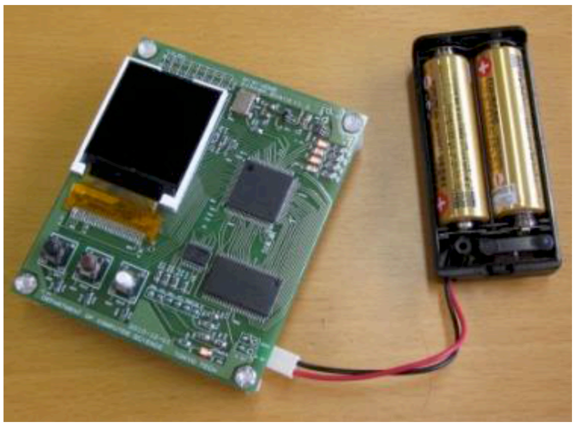


Fig. 1 MieruEMB System.

In this paper, we show the design, the implementation, and the features of 3bOS, and conclude that 3bOS is usable for educational purposes. The paper is organized as follows. Section 2 talks about the design choices made in the OS. In Section 3, we show the challenges encountered in its implementation. We discuss 3bOS in Section 4, and conclude this paper in Section 5.

2. Design

Several design choices were made in each step of 3bOS. As mentioned previously, most choices were based on simplicity and flexibility. The important design choices will be discussed in this section.

2.1 Instruction Set Architecture

While the operating system is mostly architecture-

1 Tokyo Institute of Technology, Japan
†1 Presently with University of the Philippines, Philippines
a) iman@arch.cs.titech.ac.jp
b) kobayashi@arch.cs.titech.ac.jp
c) kise@cs.titech.ac.jp

Fig. 2 Experiments on Computer Science IV EMB.

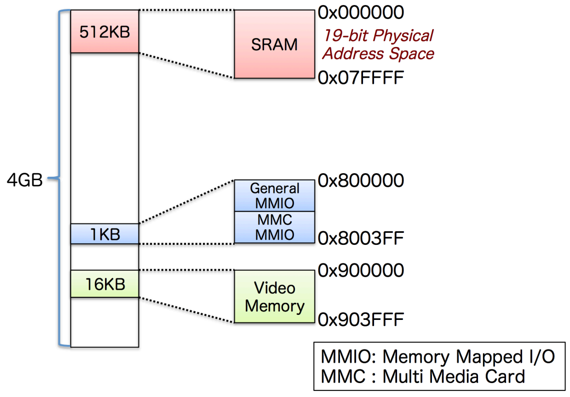


Fig. 3 32-bit Address Space.

independent, it still contains some low-level assembly code. Hence, a good target architecture should be decided on beforehand. For this purpose, 32-bit MIPS was chosen, for its simplicity and widespread use, especially for educational purposes[3]. It should be noted though that even though 3bOS is architecture-dependent, it is relatively easy to port across architectures, as most of its low-level code is kept at a minimum. Therefore, if the user wants to port it to another architecture, the user only modifies parts of codes written in assembly.

2.2 I/O

The I/O is also kept as simple as possible by using memory-mapped I/O. Physical addressing problems usually brought by memory-mapped I/O, such as buffer overflows from the physical RAM to the I/O address space, are not present. This is because addresses are 32-bit, even though the RAM is small (512 KB). **Fig. 3** shows 32-bit address space in this system. Very large address values are assigned to the memory-mapped I/O, hence a large gap between RAM addresses and I/O addresses is present. This prevents pollution of memory-mapped I/O's address space from buffer overflows.

2.3 Storage and the Operating System

The SD card serves as the primary storage device of the system. **Fig. 4** shows the SD card contents. It holds the

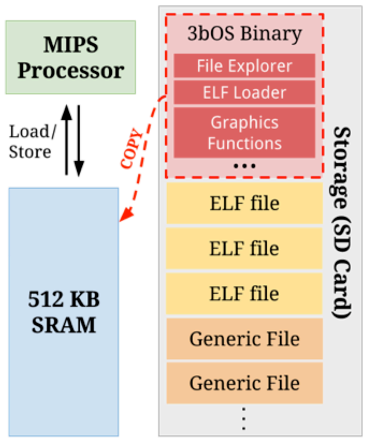


Fig. 4 The SD card contents.

OS, generic files, and ELF executables. 3bOS is a binary file which contains the memory image of the operating system. When the system is turned on, a hardware module called the program loader copies the contents of the binary file to memory. The operating system is then started by setting the program counter at the entry point. Unlike the OS, application programs are ELF executables and are much smaller in size.

Similar to many other small embedded systems, the 3bOS is programmed with a super-loop[4], where the OS has an infinite loop that executes processes and does background tasks. In the super-loop, all the tasks the processor has to do is done in one cycle of the loop, such as I/O reading and writing. Threading and modularity is basically non-existent, which introduces a lot of complexity in coding and debugging the super-loop. To reduce complexity, we made sure that the only tasks running in the main loop are the most necessary ones, which in this case would be the button interaction.

2.4 File system and File explorer

The FAT file system is chosen for its simplicity and its popularity on many platforms. Hence, it is suitable to choose the FAT file system as an educational purpose.

A file explorer is included with the operating system; it is not implemented as an ELF executable. This way, the operating system can be executed and be usable even as a standalone, and would not depend on any external programs or libraries. As an educational tool, it would be more user-friendly if it could work right out of the box. **Fig. 5** shows the file explorer screen flow. The file explorer is able to navigate directories, as well as open files. File contents can be scanned after opening, and is useful for debugging executables and the OS itself.

It is again important to note that, for simplicity, the OS is only driven by user interaction. During its idle time, the OS does nothing except for managing the button I/O. In other words, no background computations such as animations or background processes are made.

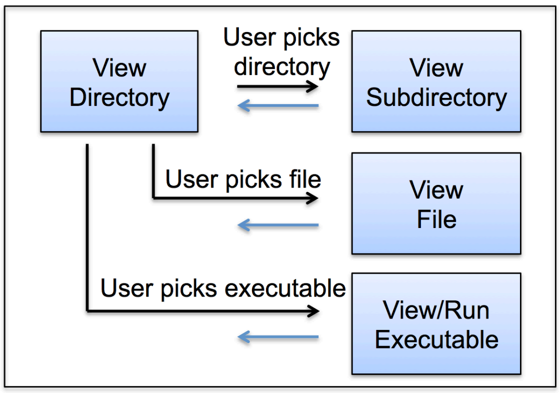


Fig. 5 Basic file explorer screen flow. The blue arrows indicate that you can go back from the current screen to the previous one.

2.5 Executables

The standard format of linux executables, ELF (Executable and Linkable Format), is one of the most popular forms of programs today. ELF files are capable of a lot of features such as shared libraries and core dumps, but only the most basic features are supported in 3bOS. For instance, the OS only supports static instead of dynamic linking. Dynamic linking could also be implemented without spending a lot of resources, but it would add much complexity to the OS. Therefore, we did not think that dynamic linking is suitable for basic OS knowledge, and did not include it in 3bOS.

3. Implementation

3.1 Compiling the OS

The operating system is mostly coded in C, then compiled with gcc for MIPS[5]. Before the main code is run, a startup script is first executed, which contained the assembly instructions for resetting the registers (**Fig. 6**). These files are compiled and linked together using a custom linker script (**Fig. 7**). The OS is then loaded into a memory image, which would then be copied into memory once the system is turned on. **Fig. 8** shows the memory layout for 3bOS. The program entry point is fixed, and the stack for both the OS and the application programs are the same.

It is also worth noting that most common C libraries such as stdio or stdlib are not used. Instead, only necessary functions are implemented, such as a printf-like function. **Fig. 9** shows character outputs using the printf-like function. But, unlike the standard printf, the implemented printf has fewer patterns and do not have compile-time type checking.

3.2 Executing Programs

Whenever the user wants to execute an ELF file in the 3bOS, a *pre-execution menu* is first displayed, confirming if the user indeed wants to run the program, or if the user wants to open it as a text file instead (**Fig. 10**). The "back", "run", and "read" in pre-execution menu correspond to the left, center, and right buttons on the MieruEMB system respectively. 3bOS displays 3 actions that correspond to 3 buttons dynamically.

```
      .text
      .globl   _start
      .ent     _start
_start:
      .set     noreorder
      .set     noat

      nop
      move     $1, $0
      move     $2, $0
      move     $3, $0
      move     $4, $0
      move     $5, $0
      move     $6, $0
      move     $7, $0
      move     $8, $0
      move     $9, $0
      move     $10, $0
      move     $11, $0
      move     $12, $0
      move     $13, $0
      move     $14, $0
      move     $15, $0
      move     $16, $0
      move     $17, $0
      move     $18, $0
      move     $19, $0
      move     $20, $0
      move     $21, $0
      move     $22, $0
      move     $23, $0
      move     $24, $0
      move     $25, $0
      move     $26, $0
      move     $27, $0
      move     $28, $0
      move     $29, $0
      move     $30, $0
      move     $31, $0
      li $sp, 0x7f000
      j        main          # jump to the main
      nop

      .end _start
```

Fig. 6 The Assembly Instructions for Resetting the Registers.

```

ENTRY(_start)

SECTIONS
{
  .startup 0x0000 : { startup.o(.text) }
  . = 0x0200;

  .init   : { KEEP (*(.init)) } = 0
  .plt    : { *(.plt) }
  .text   : { *(.text .stub .text.*
                .gnu.linkonce.t.*)
              KEEP (*(.text)) } = 0
  .fini   : { KEEP (*(.fini)) } = 0
  .rodata : { *(.rodata .rodata.*
                .gnu.linkonce.r.*) }
  .tdata          : { *(.tdata .tdata.*
                .gnu.linkonce.td.*) }
  .tbss : { *(.tbss .tbss.*
                .gnu.linkonce.tb.*)
              *(.tcommon) }
  .ctors  : { start_ctors = .;
              KEEP (*(SORT(.ctors.*)))
              KEEP (*(.ctors))
                 end_ctors = .; }
  .dtors  : { start_dtors = .;
              KEEP (*(SORT(.dtors.*)))
              KEEP (*(.dtors))
              end_dtors = .; }
  .data   : { *(.data .data.*
                .gnu.linkonce.d.*)
              SORT(CONSTRUCTORS) }
  .got.plt : { *(.got.plt) }
  . = .;
  _gp = ALIGN(16) + 0x7ff0;
  .got     : { *(.got) }
  .bss     : { *(.dynbss)
               *(.bss .bss.*
                 .gnu.linkonce.b.*)
               *(COMMON) }
}
```

Fig. 7 A Custom Linker Script.

The user can run the ELF file by pressing the center button, or can open it as a text file by pressing the right button. The state saving, program loading, and program counter

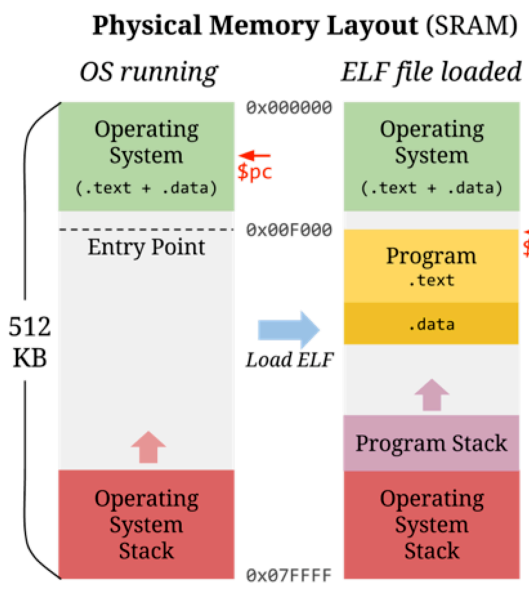


Fig. 8 Physical Memory Layout.

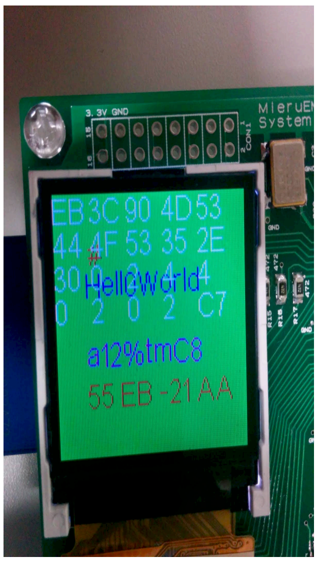

```
EB 3C 90 4D 53
44 4F 53 35 2E
30 HelloWorld
0  2  0  2  C7
a12%tmC8
55 EB -21 AA
```

Fig. 9 Character outputs using the Printf-like Function.

jump happens in this screen. After the executable exits, the OS returns to this screen.

To implement this feature, it is necessary to accurately save and restore both processor and memory state. The memory to be saved is composed of two parts: the stack, and the video memory. Since these memory chunks are too large, saving them somewhere (such as the SD card) would introduce a lot of overhead in the code. As a fix, the stack could be left as it is, and the new program could just build over the existing stack, leaving the older variables intact. This comes with the limitation of a smaller available memory space for programs.

The problem in the implementation of this feature is the video memory, since it would be overwritten when the new program runs. When the pre-execution menu opens, it draws the UI on the screen, but that UI should also be restored after the program exits. A possible workaround

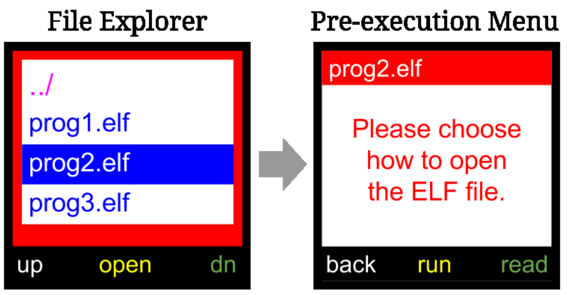


Fig. 10 A rendering of the 3bOS File explorer and the Pre-execution menu UI.

would be to save the state before the pre-execution menu UI is drawn. In other words, when the executable exits, the state is restored, and the OS acts again as if the pre-execution menu just opened.

However, in this case it is inevitable that there will be some instructions between state saving and the actual changing of the program counter (**Fig. 11**). Doing this will most likely change the current stack after the state has been saved. This will result in an imperfect state restoration, since the stack in the restored state is different.

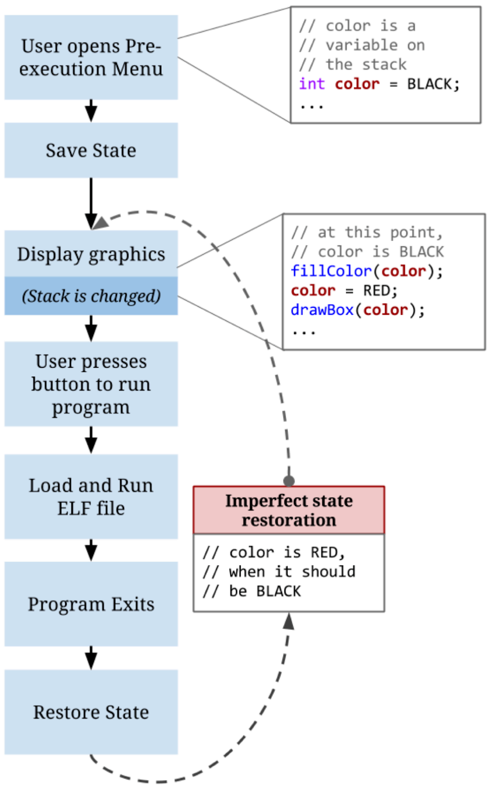
The solution is shown in **Fig. 12**, where the state saving is done after user interaction, instead of before it. The pre-execution menu UI is first drawn, and when the user presses the "Run" button, the state is saved and the program is executed. When the program exits, the state is restored and the program counter is restored to the instruction right after the user presses "Run". We can now put a flag called "done" here, so we can check whether or not the program has been run. If "done == true", then the program jump is skipped, and the instruction goes on. The important point here is that the flag done should be a global variable, so that it is not stored in the stack, so that the stack remains unchanged when the program is called. After that, the redraw functions are called to display the UI again.

3.3 Loading ELF Executables into Memory

Fig. 13 shows the loading process of an ELF file. The ELF file should be interpreted and its sections should be copied into specific locations in memory. Since only static linking is used, no shared external libraries were needed to be loaded, keeping the code simple. A fixed, predefined memory address that is set from the linker script, which defines where the program should be loaded in memory. Since the entry point is fixed, a single jump instruction would be enough to go to the program's instructions. In saving the current state, all the register values are saved to memory using assembly code. The program counter is then moved to the entry point of the program.

3.4 Program Exit Procedure

Since the operating system does not use threads, exiting to the OS can only be done by the program. All programs should run an exit procedure, in which the register states and the program counter would be restored (**Fig. 14**). This



```
// color is a
// variable on
// the stack
int color = BLACK;
...
```

```
// at this point,
// color is BLACK
fillColor(color);
color = RED;
drawBox(color);
...
```

```
// color is RED,
// when it should
// be BLACK
```

Fig. 11 Execution Procedure (wrong behavior).

exit procedure is implemented in assembly, and can be executed by programs by including a simple OS library which is added at compile-time. Exiting can be done in code by just using a simple function call.

3.5 Debugging the OS

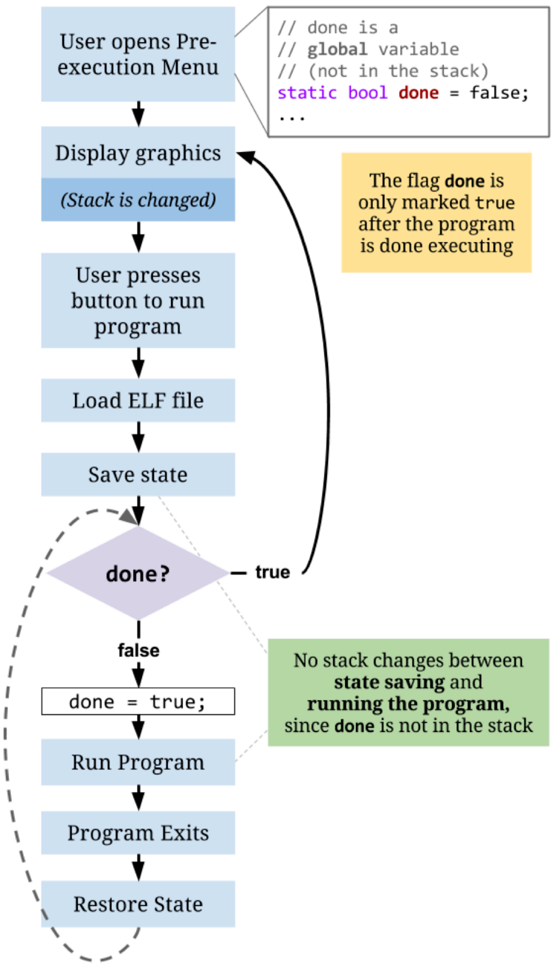
While creating basic components for the OS such as the drivers and program execution, some painstaking debugging is done. Since there is no simple way to communicate with the system, the only way to debug on the system side is by printing logs on the screen. Most of the time, these logs are used to inspect memory contents. Debugging SD card reading/writing, for instance, also required reading the SD card with a hex editor on a PC (**Fig. 15**) to verify correctness. As it is not easy to locate the correct data addresses in the hex editor, some techniques such as inserting unique constants are done. Unique constants are inserted in several parts of the code, to act as markers. This will make it easier to find certain instructions adjacent to those markers when debugging binaries and machine code using hex editors.

4. Discussion

Fig. 16 shows the 3bOS file explorer running on the MieruEMB system. Users can select the ELF files that they want to run.

We show 3bOS's important points, and conclude that 3bOS is usable for educational purposes.

- Simplicity and Flexibility
  - The finished code consists of just around 800 lines total. In terms of the number of lines of code, 3bOS is about one-eighth the size of FreeRTOS[6]. The small number of lines of code, especially considering most operating systems, shows 3bOS' simplicity. Hence, users can easily understand how this embedded OS runs on the MieruEMB system, and can easily modify this em-


User opens Pre-execution Menu
Display graphics
(Stack is changed)
User presses button to run program
Load ELF file
Save state
done?
true
false
Run Program
Program Exits
Restore State
The flag done is only marked true after the program is done executing
No stack changes between state saving and running the program, since done is not in the stack

```
// done is a
// global variable
// (not in the stack)
static bool done = false;
...
```

```
done = true;
```

Fig. 12 Execution Procedure (fixed).

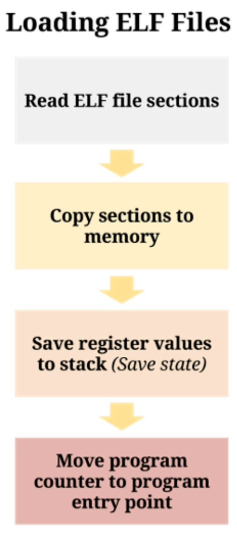

Loading ELF Files
Read ELF file sections
Copy sections to memory
Save register values to stack (Save state)
Move program counter to program entry point

Fig. 13 The Loading Process of an ELF File.

```

void os_exit() {
  asm("li␣$t9,␣0x7ff00;");
  asm("addi␣$t9,␣$t9,␣-32;");
  asm("lw␣$9,␣0($t9);");
  asm("lw␣$8,␣4($t9);");
  asm("lw␣$7,␣8($t9);");
  asm("lw␣$6,␣12($t9);");
  asm("lw␣$5,␣16($t9);");
  asm("lw␣$4,␣20($t9);");
  asm("lw␣$3,␣24($t9);");
  asm("lw␣$2,␣28($t9);");
  asm("addi␣$t9,␣$t9,␣-32;");
  asm("lw␣$17,␣0($t9);");
  asm("lw␣$16,␣4($t9);");
  asm("lw␣$15,␣8($t9);");
  asm("lw␣$14,␣12($t9);");
  asm("lw␣$13,␣16($t9);");
  asm("lw␣$12,␣20($t9);");
  asm("lw␣$11,␣24($t9);");
  asm("lw␣$10,␣28($t9);");
  asm("addi␣$t9,␣$t9,␣-32;");
  asm("lw␣$24,␣4($t9);");
  asm("lw␣$23,␣8($t9);");
  asm("lw␣$22,␣12($t9);");
  asm("lw␣$21,␣16($t9);");
  asm("lw␣$20,␣20($t9);");
  asm("lw␣$19,␣24($t9);");
  asm("lw␣$18,␣28($t9);");
  asm("addi␣$t9,␣$t9,␣-28;");
  asm("lw␣$31,␣4($t9);");
  asm("lw␣$30,␣8($t9);");
  asm("lw␣$29,␣12($t9);");
  asm("lw␣$28,␣16($t9);");
  asm("lw␣$27,␣20($t9);");
  asm("lw␣$26,␣24($t9);");
  asm("addi␣$t9,␣$t9,␣28;");
  asm("lw␣$25,␣0($t9);");
  asm("nop;");
  asm("jr␣$ra;");
}
```

Fig. 14 The Program Exit Procedure.

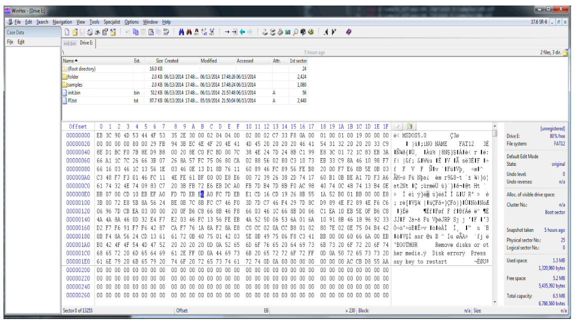


Fig. 15 Debugging the SD Card.

bedded OS as they desire.

- Low Memory Usage
  - The maximum memory usage of the operating system itself is very small. This leaves around 400 KB of usable space for running programs, which is around 80% of the total size of the main memory. The amount of available memory for programs is also relatively decent, taking into account that most programs meant to be run in this OS are non-intensive and perform only simple tasks.

The 3bOS is simple and flexible. Hence, it is suitable to use this embedded OS as an educational material. Its low memory usage also enables users to develop richer programs and features.

5. Conclusion

We have introduced 3bOS, a simple and customizable embedded OS operated using only 3 buttons. The design choices made to achieve simplicity and flexibility were explained. Important details of the system implementation was discussed. Finally, we showed some metrics highlighting 3bOS' size and simplicity.

Since this embedded OS is simple, users who want to learn OS programming can easily understand it. Its flexibility makes it simple for users to modify, add, or remove features. Therefore, we conclude that 3bOS is usable for educational purposes.

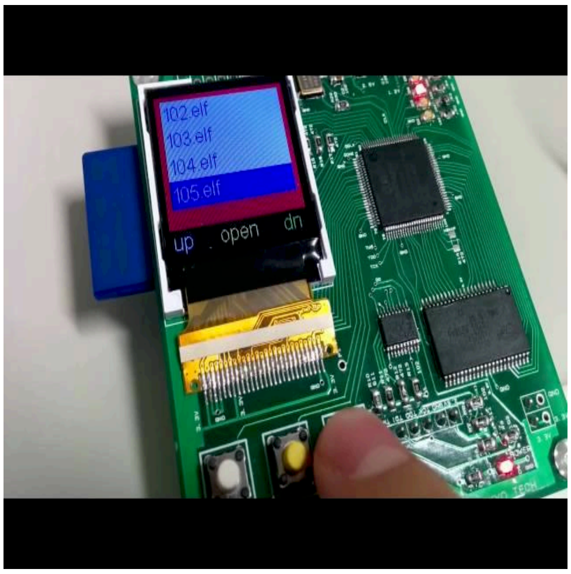


Fig. 16 OS with file explorer on screen.

References

[1] KISE, K.: Designing and Using The Simple Computer Systems and Embedded Systems, *ESS2013: EMBEDDED SYSTEM SYMPOSIUM 2013*, Vol. 2013, p. 1 (2013).
[2] Experiments on Computer Science IV EMB: http://www.arch.cs.titech.ac.jp/lecture/emb/.
[3] Patterson, D. A. and Hennessy, J. L.: *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition (2013).
[4] Glistvain, R. and Aboelaze, M.: Romantiki OS; A single stack multitasking operating system for resource limited embedded devices, *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pp. 1–8 (2010).
[5] Buildroot: making embedded Linux easy: http://buildroot.uclibc.org.
[6] FreeRTOS: http://www.freertos.org/.