Final Report

OSDev Group 18

consisting of

Christopher Sanden - GitHub@chsanden
Teodor Salvesen - GitHub@Teodorsa

presenting

GitHub Pull Request
https://github.com/uiaict/2026-ikt218-osdev/pull/39

GitHub repository
https://github.com/chsanden/IKT218-AdvOpsys

IKT218

Advanced Operatingsystems

Faculty of technology and science

April 2026

Contents

List of Listings

1 Introduction

This report documents both the boot-process background behind operating-system startup and the development of a small 32-bit operating-system kernel for the `Advanced Operatingsystems` project. It starts by explaining the firmware and bootloader stages that happen before a kernel runs, then follows the project implementation from a minimal bootable kernel into a more interactive environment with processor setup, interrupts, memory management, timing, sound, keyboard input, and simple applications.

The background section covers the general boot sequence: Power-On Self-Test, firmware handoff, BIOS and UEFI differences, bootloader responsibilities, early i386 memory layout, modern operating-system boot chains, and the role of virtualization. This gives context for why the project can rely on a bootloader, why the kernel starts from a constrained early machine state, and why QEMU is useful for testing this kind of low-level work.

The implementation itself is intentionally low-level. Most of the kernel interacts directly with x86 processor structures and hardware interfaces, including the Global Descriptor Table, Interrupt Descriptor Table, Programmable Interrupt Controller, Programmable Interval Timer, VGA text output, keyboard controller, and PC speaker. Because the kernel runs in a freestanding environment, basic facilities that are normally provided by an operating system or standard library also had to be implemented inside the project.

After the boot background, the report follows the same order as the implementation work. The first implementation part covers the early boot path and processor setup, with particular focus on the Global Descriptor Table and the need to reload segment registers after installing it. The second part describes interrupt handling, including CPU exceptions, hardware IRQs, PIC remapping, and keyboard input. The third part introduces memory management, paging, PIT-based timing, sleep functions, and PC speaker music playback. The final part shows how these individual subsystems were combined into a small menu-driven application environment with a music player, an interactive Snake game and a piano application.

The overall goal was not to build a complete general-purpose operating system, but to demonstrate how core operating-system mechanisms fit together. Each stage adds a specific capability, and later stages reuse earlier ones: the PIT depends on interrupt handling, the music player depends on PIT timing and PC speaker output, Snake depends on memory allocation, keyboard input, terminal drawing, timing, and sound feedback, and Piano app depends on keyboard input, terminal drawing, PIT timing and memory allocation. By the end of the project, the kernel has moved from static startup output to an event-driven system that can run simple interactive applications.

2 Boot Process Background

This section gives background context for the boot process before the kernel-specific implementation described in the rest of the report. It covers the firmware stages, bootloader responsibilities, memory layout, and the effect of virtualization on booting.

2.1 Power-On Self-Test

When a computer is powered on or reset, the processor begins execution from a firmware-defined reset vector. At this point, the operating system is not running yet. The first responsibility belongs to the system firmware, either Basic Input/Output system (BIOS) on older x86 systems or Unified Extensive Firmware Interface (UEFI) on newer systems [5, 28].

One of the earliest firmware tasks is the Power-On Self-Test, usually shortened to POST. POST checks that the basic hardware needed to continue booting is present and responding [3].

POST might include testing and verifying some of these components:

- Processors
- Storage
- Memory
- Keyboard (Keyboard not found. Press f1 to continue.)

Note: The exact list depends on the system [26].

POST is important because later boot stages assume that the basic machine state is usable. If a required device fails or memory cannot be initialized, the firmware may stop the boot process and report the failure through screen output, status LEDs, or beep codes [26].

The interaction between POST and hardware is therefore direct and low-level. Firmware initializes chipset state, configures memory controllers, discovers attached devices, and prepares enough of the platform that a bootloader can be found and executed. This is different from normal operating-system hardware management, because POST happens before the OS has loaded drivers or enabled its own abstractions.

2.2 Boot Sequence After POST

After a successful POST, the firmware chooses a boot device according to its configured boot order. On BIOS systems, this usually means reading the first sector of a bootable disk into memory and jumping to it. On UEFI systems, the firmware instead loads an EFI application from a filesystem on the EFI System Partition [3, 25].

The high-level sequence after POST is:

1. firmware completes hardware initialization,
2. firmware selects a boot device,
3. the first bootloader stage or EFI boot application is loaded,
4. the bootloader prepares the environment expected by the kernel,
5. the kernel image is loaded into memory,
6. control is transferred to the kernel entry point.

In a BIOS boot flow, the first loaded code is small because the initial boot sector is limited to 512 bytes. This usually forces the bootloader to use multiple stages. The first stage is responsible for loading a larger second stage, and the second stage can then parse filesystems, load the kernel, and prepare boot information[3, 13].

In a UEFI boot flow, the firmware provides more services before the operating system starts. A UEFI bootloader can use firmware-provided filesystem and device services, and it is loaded as a normal executable rather than as raw code from a fixed disk sector. This makes UEFI bootloaders more flexible, but the kernel must still eventually take control and stop depending on firmware runtime assumptions [25].

2.3 Bootloaders

A bootloader is the bridge between firmware and the operating-system kernel. Its purpose is not only to start the kernel, but also to place the machine into a state the kernel understands [5, 7].

Common bootloader responsibilities include:

- locating and loading the kernel image,
- loading additional modules or initrd files,
- obtaining a memory map from the firmware,
- selecting or setting a graphics mode,
- preparing boot information structures,
- switching CPU mode when required,
- transferring control to the kernel entry point.

Different bootloaders provide different levels of support. GRUB is a general-purpose bootloader with filesystem support, configuration files, multiboot support, and broad hardware compatibility. Limine is a modern boot protocol and bootloader often used in hobby OS development because it provides a clear boot protocol and supports both BIOS and UEFI systems[8, 13, 29]. A manually

implemented bootloader gives full control over the boot path, but requires more work and exposes the project to more early-stage hardware and filesystem details.

The choice of bootloader depends on the goals of the operating-system project. For a kernel-focused project, using an existing bootloader is usually the most practical choice because it avoids spending most of the work on disk loading, filesystem parsing, firmware differences, and CPU mode transitions. For a project specifically about bootstrapping, implementing a bootloader manually can be useful because it exposes the details normally hidden by existing tools [13].

Manual bootloader implementation is challenging because the environment is very limited. In a BIOS first-stage bootloader, code size is constrained, there is no standard library, only a small amount of state is initialized, and disk access must use firmware interrupts or direct hardware access. The bootloader must also be careful about where it places itself, the kernel, stacks, tables, and temporary buffers in memory.

2.4 Memory Layout in the Boot Process

In a traditional i386 BIOS boot process, early memory layout is constrained by legacy conventions. The first MiB of memory contains several important regions, including the interrupt vector table, BIOS data area, conventional memory, video memory, and firmware regions[18].

The bootloader and kernel must avoid overwriting memory that is already used by firmware, hardware mappings, or the bootloader itself. In small BIOS examples, a common simple kernel load address is `0x10000`. This address is above the lowest BIOS data structures and gives the kernel more room than the original boot sector area, while still being within conventional memory that is easy to access in early boot stages. In this project, the linker script places the kernel at `1 MiB`, which is also a common protected-mode kernel load address and keeps the kernel away from the lowest legacy BIOS regions [18].

Choosing a fixed early kernel load address has practical implications:

- the bootloader needs to copy or load the kernel to a known address,
- the kernel linker script must match the address where the kernel expects to run,
- early stacks and temporary buffers must be placed so they do not overlap the kernel,
- later memory management must identify which regions are already occupied.

Once the kernel has control, it can replace this simple early memory model with its own memory management. In this project, that later stage is represented by kernel heap initialization, page-aligned allocation, and paging setup.

2.5 Boot Process in Modern Operating Systems

Modern operating systems use more complex boot processes than small teaching kernels, but the same basic idea remains: firmware initializes the platform, a bootloader or boot manager loads the kernel, and the kernel takes control of the machine.

Linux systems commonly use firmware to start a bootloader such as GRUB or systemd-boot. The bootloader loads the kernel and an initramfs, passes a command line and boot information, and then transfers control to the Linux kernel. Windows systems use Windows Boot Manager, which

loads the Windows OS loader before the kernel and core system components are started. macOS uses Apple's boot chain, which is tightly integrated with Apple hardware, APFS, Secure Boot policies, and system volume verification[1, 2, 9, 11].

Modern boot processes have changed because hardware and security requirements have changed. UEFI replaced many BIOS conventions, disks moved from MBR partitioning toward GPT, and Secure Boot introduced signature verification into the boot chain. Storage devices are also more complex than older BIOS-era disks, and modern systems often need early support for NVMe, encryption, graphics initialization, and platform security features.

The result is that modern booting is both more capable and more controlled. Firmware and bootloaders provide richer services, but the operating system must also participate in a stricter trust chain and handle more platform variation.

2.6 Virtual Machines and Booting

Booting inside a virtual machine follows the same conceptual stages as booting on physical hardware, but the hardware being initialized is virtual. The guest operating system still sees firmware, CPU state, memory, storage devices, timers, and interrupt controllers. However, those devices are provided or emulated by the hypervisor[24].

The main difference is that a virtual machine does not start from real motherboard hardware. Instead, the hypervisor creates a virtual hardware environment and then starts the guest firmware inside it. The guest firmware performs a boot sequence that looks normal from inside the VM, but many device operations are handled by the hypervisor on the host [10].

The hypervisor has several roles in this process:

- allocating guest memory,
- exposing virtual CPUs,
- providing virtual disks and network devices,
- emulating or virtualizing interrupt controllers and timers,
- presenting BIOS or UEFI firmware to the guest,
- handling privileged operations that cannot run directly on the host CPU.

Virtualized booting has advantages for operating-system development. It is faster to test than rebooting physical hardware, the virtual machine can be reset easily, debugging is easier with tools such as QEMU and GDB, and hardware behaviour is more reproducible. This is why the kernel in this project was tested through QEMU rather than directly on physical hardware.

There are also limitations. Emulated hardware may not behave exactly like real hardware, and some devices are simplified compared to physical machines. For example, PC speaker output in QEMU can be limited, which affected the clarity of fast note changes in the music player. Even so, virtualization is highly useful for kernel development because it provides a controlled environment for testing boot, interrupts, memory management, and device interaction[10, 24].

3 Boot Process and Processor Setup

The first stage of the project established a more complete early boot path for the kernel. Before this work, the kernel entered C code, initialized the VGA terminal, printed `Hello, World!`, and then remained in an infinite halt loop. The terminal output in this stage used direct VGA text-mode writes rather than BIOS text services, which is the normal approach once the kernel is running in protected mode [20]. The visible output remained the same after this stage, but the processor setup became more explicit and reliable.

The main change was the addition of a minimal Global Descriptor Table for the i386 kernel. Even though the kernel uses a flat memory model, protected mode still requires valid segment descriptors. The GDT therefore gives the processor valid code and data segment definitions before the kernel continues into higher-level initialization[14].

3.1 Kernel Entry Flow

After the GDT work, the early kernel flow became:

1. the bootloader transfers control to the kernel entry point,
2. the kernel begins execution in C,
3. the GDT is initialized and loaded,
4. segment registers are reloaded,
5. the VGA terminal is initialized,
6. `Hello, World!` is written to the screen,
7. the kernel enters an infinite halt loop.

This keeps processor setup before terminal setup. Later stages build on this ordering by adding interrupts, memory management, timing, and applications after the low-level CPU state has been initialized.

3.2 GDT Layout

The implemented GDT contains three descriptors:

- a null descriptor,
- a kernel code segment descriptor,
- a kernel data segment descriptor.

The code and data segments both use base address `0x00000000`, a 4 GiB address span, 4 KiB granularity, and ring 0 privilege level. The code segment uses selector `0x08`, while the data segment uses selector `0x10`. This matches the flat protected-mode layout commonly used in small i386 kernels [6, 14].

```
void GdtInitialize(void) {
    gdtDescriptor.size = sizeof(gdtEntries) - 1;
    gdtDescriptor.offset = (uint32_t)&gdtEntries;

    GdtSetEntry(0, 0, 0, 0, 0);
    GdtSetEntry(1, 0, 0x000FFFFF, 0x9A, 0xCF);
    GdtSetEntry(2, 0, 0x000FFFFF, 0x92, 0xCF);

    GdtFlush((uint32_t)&gdtDescriptor);
}
```

Listing 1: Minimal GDT initialization with null, kernel code, and kernel data descriptors.

3.3 Descriptor Construction

The GDT setup is split between C and assembly. The C code builds the descriptor table and prepares the GDT descriptor that contains the table size and address. The helper that creates each descriptor splits the base and limit into the fields expected by the processor [6, 14].

```
static void GdtSetEntry(
    uint32_t index,
    uint32_t base,
    uint32_t limit,
    uint8_t access,
    uint8_t granularity
) {
    gdtEntries[index].base_low = (uint16_t)(base & 0xFFFF);
    gdtEntries[index].base_middle = (uint8_t)((base >> 16) & 0xFF);
    gdtEntries[index].base_high = (uint8_t)((base >> 24) & 0xFF);

    gdtEntries[index].limit_low = (uint16_t)(limit & 0xFFFF);
    gdtEntries[index].granularity = (uint8_t)((limit >> 16) & 0x0F);
    gdtEntries[index].granularity |= (uint8_t)(granularity & 0xF0);
    gdtEntries[index].access = access;
}
```

Listing 2: Building an x86 GDT descriptor from base, limit, access, and granularity fields.

3.4 Reloading Segment Registers

Loading the GDT with `lgdt` is not enough by itself. The CPU caches segment descriptor information in the segment registers, so the kernel must reload those registers after installing the new table. The assembly routine performs the architecture-specific part: it loads the GDT, performs a far jump to reload `cs`, and then reloads the data segment registers[14].

```
GdtFlush:
    mov eax, [esp + 4]
    lgdt [eax]

    jmp GDT_CODE_SELECTOR:.reload_cs

.reload_cs:
    mov ax, GDT_DATA_SELECTOR
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    ret
```

Listing 3: Loading the GDT and refreshing the cached code and data segment registers.

4 Build and Verification

The build configuration was updated so that both the C GDT implementation and the assembly reload routine are compiled and linked into the kernel binary. New build artifacts were generated, including an updated kernel binary and bootable ISO image.

4.1 Static Binary Inspection

The generated kernel binary was inspected to confirm that the expected GDT setup sequence was present. The inspection showed a call to the GDT load routine, a far jump to selector `0x08`, and data segment reloads using selector `0x10`.

4.2 Runtime Debugging

The GDT setup was also verified in QEMU using `gdb-multiarch`. Execution was stopped inside the GDT initialization and reload path. The GDT descriptor limit was `0x17`, which matches three 8-byte entries minus one. After the far jump, `cs` contained `0x08`; after the reload instructions, `ds`, `es`, `fs`, `gs`, and `ss` contained `0x10`.

5 State After Processor Setup

At the end of this stage, the kernel:

- builds inside the development container,
- includes a minimal i386 GDT implementation,
- loads the GDT during startup,
- reloads the code and data segment registers correctly,
- still initializes the VGA terminal,
- still prints `Hello, World!` after boot.

This stage did not add new user-facing behaviour. Its purpose was to make the early processor state explicit so later kernel subsystems could be added on top of a controlled protected-mode setup.

Later stages changed the visible startup flow by initializing the terminal before later setup code prints diagnostics for interrupt, memory, and paging work. The important dependency from this stage remains that the GDT is loaded before the kernel relies on later interrupt and application behaviour.

6 Interrupt Handling

The next stage added interrupt support to the kernel. This required an Interrupt Descriptor Table so the processor has defined entry points for CPU exceptions and hardware interrupts. The implementation follows the same model used by the OSDev Wiki material, the `os-tutorial` project, and the lecturer-provided assignment files [4, 16, 27].

Interrupt support changed the kernel from a purely linear boot program into a system that can react to events while it is running. CPU exceptions are handled through ISR stubs, hardware interrupts are handled through IRQ stubs, and both paths eventually reach C handlers with a saved register state.

6.1 IDT Descriptor Setup

First, we define what an entry and the idt register should look like.

```
struct IdtEntry{
        uint16_t    interrupt_low;
        uint16_t    kernel_cs;
        uint8_t     reserved;
        uint8_t     attributes;
        uint16_t    interrupt_high;
} __attribute__((packed));

struct Idtr {
    uint16_t    limit;
    uint32_t    base;
} __attribute__((packed));
```

Listing 4: Defining a struct for IDTEntries and one for the IDT register [16].

Each IDT entry stores the handler address, the kernel code segment selector, and the descriptor attributes used by the processor. The handler address is split into low and high parts because this is the layout expected by the 32-bit x86 IDT format.

```
void IdtSetDescriptor(uint8_t vector, uint32_t interrupt, uint8_t flags) {
    struct IdtEntry* descriptor = &idt[vector];

    descriptor->interrupt_low  = (uint32_t)interrupt & 0xFFFF;
    descriptor->kernel_cs      = GDT_CODE_SELECTOR;
    descriptor->attributes     = flags;
    descriptor->interrupt_high = (uint32_t)interrupt >> 16;
    descriptor->reserved       = 0;
}
```

Listing 5: Constructing an IDT gate for a specific interrupt vector [16].

6.2 ISR and IRQ assembly code

The assembly code below uses stubs to act as a bridge between the interrupt mechanism and interrupt handlers that will show up later in the report.

First, we have the stubs. There are 5 of them defined in the assembly code, but they can be split into 2 groups. We have the isr stubs and the irq stubs. What both of these categories are will be defined more clearly later, but for now isr's can be thought of as CPU exceptions and irq's as hardware interrupts. These are arranged into 2 tables that are set to global so that they can be accessed from anywhere [15].

As isr-common-stub and irq-common-stub are exact copies of each other apart from which C function they call we will not be explaining both. All three macros and tables will get a brief explanation, but for further reading follow sources in captions.

```
isr_common_stub:
    ; 1. Save CPU state
    pusha ; Pushes general purpose registers onto stack
        mov ax, ds ; Lower 16-bits of eax = ds.
        push eax ; save the data segment descriptor
        mov ax, 0x10 ; loads data segment descriptor into kernel
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
    push esp ; pushes stack pointer as arg to function

    call IsrHandler

    ; 2. Restore CPU state
    add esp, 4 ; removes arg pushed to function
        pop eax ; restores the data segment descriptor
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        popa ; restores general purpose registers
        add esp, 8 ; removes interrupt num and error num from stack
        iret ; returns from interrupt
```

Listing 6: Constructing common stub for isr in assembly code [4].

```
%macro isr_no_err_stub 1
isr_stub_%+%1:
    push dword 0 ; pushes dummy error
    push dword %1 ; pushes interrupt vector number
    jmp isr_common_stub
%endmacro

%macro isr_err_stub 1
isr_stub_%+%1:
    push dword %1 ; pushes interrupt vector number
    jmp isr_common_stub
%endmacro

%macro irq_stub 1
irq_stub_%+%1:
    push dword 0 ; pushes dummy error
    push dword (32 + %1) ; pushes interrupt vector number + 32 to account for isr's
    jmp irq_common_stub
%endmacro
```

Listing 7: Constructing macros for isr and irq stubs in assembly code [16].

```
global isr_stub_table
isr_stub_table:
%assign i 0 ; creates a variable and sets it to 0
%rep    32
    dd isr_stub_%+i ; adds one address to the table
%assign i i+1 ; increments variable
%endrep

global irq_stub_table
irq_stub_table:
%assign i 0 ; creates a variable and sets it to 0
%rep    16
    dd irq_stub_%+i ; adds one address to the table
%assign i i+1 ; increments variable
%endrep
```

Listing 8: Constructing tables for isr and irq stubs. [16].

6.3 IDT Initialization

During initialization, the kernel fills vectors `0` through `31` with CPU exception stubs and vectors `32` through `47` with hardware IRQ stubs. This means the implementation provides ISR coverage for all 32 standard CPU exception vectors, which is more complete than only defining the minimum three handlers required for basic testing. The PIC is then remapped before the IDT is loaded with `lidt` and interrupts are enabled with `sti` [16].

```
void IdtInitialize(void) {
    idtr.base = (uint32_t)&idt[0];
    idtr.limit = (uint16_t)sizeof(struct IdtEntry) * IDT_ENTRIES - 1;

    for (uint8_t iNum = 0; iNum < 32; iNum++) {
        IdtSetDescriptor(iNum, (uint32_t)isr_stub_table[iNum], 0x8E);
    }

    for (uint8_t iNum = 32; iNum < 48; iNum++) {
        IdtSetDescriptor(iNum, (uint32_t)irq_stub_table[iNum - 32], 0x8E);
    }

    PicRemap();
    __asm__ volatile ("lidt %0" : : "m"(idtr));
    __asm__ volatile ("sti");
}
```

Listing 9: Initializing the IDT with CPU exception and hardware IRQ entries [16].

6.4 PIC remapping and configuration

When an IRQ interrupt is over (EOI) a command signifying this has to be sent to the PIC. However, what where it is sent depends on if the IRQ came from Master PIC or Slave PIC. If it came from Master it is enough to only send the command to Master, but if it came from Slave it has to be sent to both Master and Slave [12].

```
void PicSendEoi(uint8_t irqNum) {
    if (irqNum >= 8) {
        OutPortByte(PIC2_COMMAND, PIC_EOI); // Slave
    }
    OutPortByte(PIC1_COMMAND, PIC_EOI); // Master
}
```

Listing 10: Sends an end of interrupt (EOI) to the PIC after handling an IRQ [12]

Hardware interrupt support was added for IRQ0 through IRQ15. These interrupts come from devices through the Programmable Interrupt Controller. The following function remaps the PIC so that hardware IRQs begin at vector `32` instead of overlapping with the CPU exception vectors.

```
void PicRemap(void) {
    OutPortByte(PIC1_COMMAND, 0x11); // sends init command to Master
    OutPortByte(PIC2_COMMAND, 0x11); // sends init command to Slave

    OutPortByte(PIC1_DATA, 0x20); // sets Master vector offset to 32
    OutPortByte(PIC2_DATA, 0x28); // sets Slave vector offset to 40

    OutPortByte(PIC1_DATA, 0x04); // Tells Master that Slave is available on IRQ2
    OutPortByte(PIC2_DATA, 0x02); // Tells Slave that it is connected to Master's IRQ2

    OutPortByte(PIC1_DATA, 0x01); // Set Master to use 8086 mode
    OutPortByte(PIC2_DATA, 0x01); // set Slave to use 8086 mode

    OutPortByte(PIC1_DATA, 0x00); // Unmask all Master IRQs
    OutPortByte(PIC2_DATA, 0x00); // Unmask all Slave IRQs
}
```

Listing 11: Remapping the PIC so IRQs use interrupt vectors `32` through `47` [12]

7 CPU Exceptions

Basic Interrupt Service Routines were implemented for CPU exceptions. The low-level stubs save the CPU state and pass a `Registers` structure into the shared C handler. This lets the kernel inspect which exception occurred and print useful debugging information to the terminal. Instead of each ISR containing its own separate print logic, the individual stubs share the same C handler, which prints the interrupt number and message for the triggered exception.

```
void IsrHandler(struct Registers* registers) {
    TerminalWriteString("\n=== INTERRUPT RECEIVED ===\n");

    TerminalWriteString("Interrupt Number: ");
    TerminalWriteUInt(registers->int_no);
    TerminalWriteString("\n");

    TerminalWriteString("Message: ");
    if (registers->int_no < 32) {
        TerminalWriteString(isrMessages[registers->int_no]);
    } else {
        TerminalWriteString("Unknown Interrupt");
    }

    for (;;) {
        __asm__ volatile("cli; hlt");
    }
}
```

Listing 12: Shared CPU exception handler using the saved interrupt number [4].

The handler stops the kernel after reporting the exception. This is appropriate for the current stage because exception recovery has not yet been implemented, and halting prevents the kernel from continuing after an undefined or unsafe processor state.

8 Hardware Interrupts

The IRQ handler dispatches to a registered C handler when one exists. After the interrupt has been handled, the kernel sends an end-of-interrupt (EOI) signal to the PIC. This tells the controller that the current interrupt has been processed and that new interrupts may be delivered.

```
void IrqHandler(struct Registers* registers) {
    uint8_t irqNum = (uint8_t)(registers->int_no - 32);

    if (interruptHandlers[registers->int_no] != 0) {
        interruptHandlers[registers->int_no](registers);
    } else if (irqNum != 0) {
        TerminalWriteString("IRQ triggered: ");
        TerminalWriteUInt(irqNum);
        TerminalWriteString("\n");
    }

    PicSendEoi(irqNum);
}
```

Listing 13: Dispatching hardware interrupts and acknowledging them with the PIC.

9 Keyboard Input

Keyboard support was implemented using IRQ1. The keyboard handler reads the scancode from port `0x60`, stores it in a small buffer, ignores key-release events, and translates key-press scancodes into ASCII through a lookup table [22].

The assignment describes the keyboard task as printing translated ASCII characters to the screen. In this project, the keyboard logic was extended slightly beyond a direct logger: the handler stores the latest translated ASCII character, and higher-level terminal or application code consumes it. This allows the same keyboard path to support the application menu, Snake input and piano input instead of only echoing characters immediately from the interrupt handler.

```
void KeyboardHandler(struct Registers* registers) {
    (void) registers; // cast to void to silence warning

    uint8_t scancode = InPortByte(KEYBOARD_DATA_PORT);

    if (index < KEYBOARD_BUFFER_SIZE) {
        keyboardBuffer[index] = scancode;
        index++;
    }

    if (scancode & 0x80) {
        return;
    }

    if (scancode < 128) {
        char ascii = scancodeToAscii[scancode];

        if (ascii != 0) {
            TerminalPutChar(ascii);
            lastKeyPressed = ascii;
        }
    }
}
```

Listing 14: Keyboard IRQ handler reading scancodes and storing the latest ASCII key.

The most recent translated key can then be consumed by higher-level code through `GetLastKeyPressed()`. This becomes important in later stages where the application menu and Snake game need keyboard input without directly handling raw scancodes.

Note: With the current implementation of the buffer there is a limit to how many keyboard presses can be performed during the OS's run time. This could be fixed with some sort of circular queue, but it was not implemented as it was not deemed necessary for the assignment.

10 State After Interrupt Work

At the end of this stage, the kernel:

- initializes and loads an Interrupt Descriptor Table,
- handles CPU exceptions through ISR support,
- supports hardware IRQs from IRQ0 to IRQ15,
- remaps the PIC so hardware interrupts do not overlap with CPU exceptions,
- dispatches hardware interrupts to registered handlers,
- sends end-of-interrupt signals after IRQ handling,
- handles keyboard input through IRQ1,
- reads keyboard scancodes from port `0x60`,
- translates key presses into ASCII characters,
- writes key presses to terminal,
- stores the latest key for later application-level input.

This stage moved the kernel from static startup output to event-driven execution. The later PIT, music, menu, Snake and Piano work all depend on this interrupt layer.

11 Memory Management

The next stage added basic memory management to the kernel. This work introduced kernel heap initialization, paging, dynamic allocation, page-aligned allocation, and memory debugging output. Together these changes allow later kernel components to request memory dynamically instead of relying only on static storage[17, 27].

11.1 Kernel Heap Initialization

The heap is initialized from the linker-provided end of the kernel image. The kernel stores this address in `last_alloc`, then reserves a normal heap region below the page heap area. A separate page heap is placed below `0x400000`, with one descriptor byte per page-aligned allocation [27].

```
void InitKernelMemory(uint32_t* kernel_end) {
    uint32_t kernelEndAddr = (uint32_t)kernel_end;

    last_alloc = kernelEndAddr + 0x1000;
    heap_begin = last_alloc;
    pheap_end = 0x400000;
    pheap_begin = pheap_end - (MAX_PAGE_ALIGNED_ALLOCS * 4096);
    heap_end = pheap_begin;
    memset((char*)heap_begin, 0, heap_end - heap_begin);
    pheap_desc = (uint8_t *)malloc(MAX_PAGE_ALIGNED_ALLOCS);
}
```

Listing 15: Kernel heap initialization using the linker-provided kernel end symbol.

This creates two allocation areas:

- a byte-addressed heap for normal `malloc()` allocations,
- a page heap for 4 KiB page-aligned allocations through `pmalloc()`.

11.2 Paging

Paging was implemented using a page directory at `0x400000` and page tables starting at `0x404000`. The setup identity-maps the first 4 MiB of memory and the 4 MiB region beginning at `0x400000`. This keeps virtual addresses equal to physical addresses for the early kernel environment while still enabling the processor's paging mechanism[23, 27].

The paging setup writes the page directory address to `cr3` and sets the paging bit in `cr0`. After that point, memory references are translated through the page tables [6, 23].

```
void paging_enable() {
    asm volatile("mov %%eax, %%cr3": :"a"(page_dir_loc));
    asm volatile("mov %cr0, %eax");
    asm volatile("orl $0x80000000, %eax");
    asm volatile("mov %eax, %cr0");
}
```

Listing 16: Enabling paging by loading `cr3` and setting the paging bit in `cr0`.

11.3 Dynamic Allocation

A simple heap allocator was implemented for `malloc()` and `free()`. Each allocation is preceded by a small metadata header containing the allocation status and size. New allocations advance `last_alloc`, while freed blocks are marked as unused and may be reused by later allocations [17].

The allocator scans existing blocks before creating a new one. If it finds an unused block large enough for the request, that block is reactivated and returned. This was verified by freeing one allocation, requesting a smaller block, and observing that the allocator reused the same address.

```
if(a->size >= size) {
    a->status = 1;
    memset(mem + sizeof(alloc_t), 0, size);
    memory_used += a->size + sizeof(alloc_t) + 4;
    return (char*)(mem + sizeof(alloc_t));
}
```

Listing 17: Reusing a freed heap block when it is large enough for a new allocation.

11.4 Memory Debugging

The kernel also gained memory debugging output. `PrintMemoryLayout()` prints the amount of memory used, the amount free, the heap size, and the start and end addresses of both the normal heap and page heap. `TerminalWriteHex()` was added so addresses can be printed in hexadecimal instead of being confused with decimal values [27].

12 Programmable Interval Timer

The Programmable Interval Timer was then added so the kernel can measure time and schedule delays. The PIT is configured on channel 0 at `1000 Hz`, so one timer tick corresponds approximately to one millisecond [21].

12.1 PIT Initialization

The PIT driver defines the command port, channel 0 port, base frequency, target frequency, divider, and tick conversion in `pit.h`. During initialization, the kernel registers an IRQ0 handler, programs the PIT command byte, and writes the divisor as low byte followed by high byte[21, 27].

```
void PitInitialize(void){
    uint16_t divisor = DIVIDER;

    RegisterInterruptHandler(IRQ0, PitIrqHandler);

    OutPortByte(PIT_CMD_PORT, 0x36);
    OutPortByte(PIT_CHANNEL0_PORT, (uint8_t)(divisor & 0xFF));
    OutPortByte(PIT_CHANNEL0_PORT, (uint8_t)((divisor >> 8) & 0xFF));
}
```

Listing 18: PIT channel 0 initialization and IRQ0 handler registration.

The IRQ0 handler increments a `volatile` tick counter. Keeping the counter inside `pit.c` reduces the chance of accidental writes from unrelated kernel code [21, 27].

12.2 Sleep Functions

Two sleep functions were implemented. `SleepBusy()` repeatedly checks the tick counter until enough time has passed. This is simple but consumes CPU time for the whole delay.

`SleepInterrupt()` uses `sti; hlt` inside the loop. This enables interrupts and halts the CPU until the next interrupt occurs. The loop then rechecks the current tick and halts again if the requested delay has not elapsed [6, 27].

```
void SleepInterrupt(uint32_t ticks_to_wait){
    uint32_t start_tick = GetCurrentTick();
    uint32_t end_tick = start_tick + ticks_to_wait;

    while(GetCurrentTick() < end_tick){
        __asm__ volatile ("sti; hlt");
    }
}
```

Listing 19: Interrupt-based sleeping using the PIT tick counter and `hlt`.

12.3 Verification

Several integration issues were fixed during PIT work. Incorrect inline assembly syntax was corrected, terminal output functions replaced unsuitable `printf()` calls, and the missing `TerminalWriteHex()` declaration was added to `terminal.h`.

The build process also showed that rebuilding only `uiaos-kernel` updates `kernel.bin`, but does not update the bootable ISO image. Rebuilding `uiaos-create-image` was necessary before QEMU booted the newest kernel. Runtime testing then showed paging output, heap output, allocation output, and repeated sleep-test messages for both busy-wait and interrupt-based sleeping.

13 PC Speaker Music Player

Assignment 5 used the PIT and PC speaker to implement a simple music player. The PC speaker is controlled through I/O port `0x61`, while PIT channel 2 generates the square wave used for audible notes[19].

13.1 Sound Generation

The speaker is enabled by setting bits 0 and 1 on port `0x61`. To play a note, the kernel calculates a divisor from the PIT base frequency and the requested note frequency, programs PIT channel 2, and then enables the speaker.

```
void PlaySound(uint32_t frequency) {
    if (!frequency) return;

    uint32_t divisor = PIT_BASE_FREQ / frequency;

    OutPortByte(PIT_CMD_PORT, 0xB6);
    OutPortByte(PIT_CHANNEL2_PORT, divisor & 0xFF);
    OutPortByte(PIT_CHANNEL2_PORT, (divisor >> 8) & 0xFF);

    EnableSpeaker();
}
```

Listing 20: Programming PIT channel 2 to generate a PC speaker tone.

13.2 Song Representation and Playback

Songs are represented as arrays of notes. Each note stores a frequency and duration in milliseconds. A rest is represented by frequency `R`, which is defined as `0`. Playback therefore consists of selecting a frequency, sleeping for the note duration, and then stopping the speaker.

The implementation uses `SleepInterrupt()` for note timing. This replaced busy-wait delays because the interrupt-based delay gave better timing and avoided wasting CPU cycles while each note was playing.

```
if (currentNote.frequency == R) {
    StopSound();
    SleepInterrupt(currentNote.duration);
} else {
    PlaySound(currentNote.frequency);
    SleepInterrupt(currentNote.duration);
    StopSound();
}
```

Listing 21: Song playback using rests, PC speaker tones, and interrupt-based timing.

14 Application Framework, Snake, and Piano

The final stage integrated the individual OS components into a simple application framework. Instead of booting into a single test routine, the kernel now presents a terminal menu where the user can choose between the music player, Snake, and a piano application.

14.1 Menu-Based Control Flow

The kernel initialization path now sets up the terminal, GDT, IDT, PIT, keyboard interrupt handler, kernel memory, and paging. After that initialization, the kernel repeatedly asks for an application number and dispatches to one of the available programs. This ties the higher-level menu flow back to the keyboard, PIT, and memory-management infrastructure implemented earlier [17, 21, 22].

```
while (1) {
    TerminalWriteString("Enter application number:\n");
    TerminalWriteString("0. Play Music\n");
    TerminalWriteString("1. Play Snake\n");
    TerminalWriteString("2. Play Piano\n");
    char input = TerminalGetChar();

    switch (input) {
        case '0':
            TerminalClear();
            PlayMusic();
            TerminalClear();
            break;
        case '1':
            TerminalClear();
            PlayGame();
            TerminalClear();
            break;
        case '2':
            TerminalClear();
            PlayPiano();
            TerminalClear();
            break;
        default:
            TerminalWriteString("\nInvalid application number.\n");
            SleepInterrupt(1000);
            TerminalClear();
            break;
    }
}
```

Listing 22: Application menu dispatch in the kernel main loop.

This demonstrates that the terminal, keyboard, interrupt, timer, memory, and application code

can cooperate through a single kernel control flow.

14.2 Snake Game State

Snake uses a dynamically allocated game state. The game state contains the board, snake, food, score, and pseudo-random state. Creating and destroying the game therefore exercises the kernel's `malloc()` and `free()` implementation in a more realistic setting than isolated allocation tests.

```
struct GameState* CreateGame(void) {
    struct GameState* game = (struct GameState*)malloc(sizeof(struct GameState));
    if (!game) return 0;

    game->snake = (struct Snake*)malloc(sizeof(struct Snake));
    if (!game->snake) {
        free(game);
        return 0;
    }

    game->food = (struct Food*)malloc(sizeof(struct Food));
    if (!game->food) {
        free(game->snake);
        free(game);
        return 0;
    }

    game->score = 0;
    game->rngState = GetCurrentTick();
    return game;
}
```

Listing 23: Snake creates its game objects using the kernel heap allocator.

14.3 Input, Timing, and Feedback

The Snake loop reads the last key pressed by the keyboard handler, updates the snake direction, moves the snake, checks collisions, redraws the board, and then sleeps using the PIT. The game uses `GAME_SPEED_MS` to control pacing.

Sound is also integrated into the game. Eating food, dying, and winning each trigger short PC speaker effects. This connects the application layer back to the PIT and PC speaker support implemented earlier.

```
input = GetLastKeyPressed();
HandleInput(game, input);
tail = MoveSnake(game->snake);

collisionType = CheckCollision(game->snake, game->food);
if (collisionType == FOOD) {
    game->score++;
    PlayFoodSound();
    AddSegment(game->snake, tail.x, tail.y);
    SpawnFood(game);
}

DrawBoard(game);
SleepInterrupt(GAME_SPEED_MS);
```

Listing 24: Main Snake loop combining keyboard input, game state, sound, drawing, and PIT timing.

14.4 Piano Application State

The piano application adds a second interactive program that uses the same core kernel services in a different way. Instead of a continuously advancing game loop, the piano keeps a dynamically allocated application state with recording flags, the current note, timing information, and a song library. This makes the application a direct integration point between the heap allocator, keyboard input, PIT timing, and PC speaker output.

```
struct PianoAppState* CreatePiano(void) {
    struct PianoAppState* piano =
        (struct PianoAppState*)malloc(sizeof(struct PianoAppState));
    if (!piano) return 0;

    piano->songLibrary =
        (struct SongLibrary*)malloc(sizeof(struct SongLibrary));
    if (!piano->songLibrary) {
        free(piano);
        return 0;
    }

    piano->songLibrary->songs =
        (struct Song*)malloc(sizeof(struct Song) * MAX_SONG_COUNT);
    if (!piano->songLibrary->songs) {
        free(piano->songLibrary);
        free(piano);
        return 0;
    }
}
```

Listing 25: The piano allocates persistent application state and storage for recorded songs.

14.5 Key Mapping and Sound Generation

The piano maps keyboard characters to musical notes. White and black keys are laid out in the terminal UI, and pressing a mapped key programs PIT channel 2 with the matching frequency before enabling the PC speaker. This reuses the same low-level mechanism as the earlier music player, but now the sound generation is driven directly by live keyboard input rather than a predefined song array [19, 21].

```
case 'z':
    PianoPlaySound(C4);
    piano->activeFrequency = C4;
    break;

case 's':
    PianoPlaySound(Cs4);
    piano->activeFrequency = Cs4;
    break;

case 'x':
    PianoPlaySound(D4);
    piano->activeFrequency = D4;
    break;
```

Listing 26: Keyboard input is translated into note frequencies for live piano playback.

14.6 Recording and Playback

The piano also adds a simple song-recording feature. When recording is enabled, played notes are stored in a song buffer together with their durations. The code also tracks the PIT tick count between notes, so pauses longer than a small threshold are stored as rests. During playback, those notes and rests are replayed through the PC speaker using `SleepInterrupt()` for timing [19, 21].

```
if (piano->recording && piano->lastNoteEndTick != 0) {
    uint32_t now = GetCurrentTick();
    uint32_t restDuration = now - piano->lastNoteEndTick;

    if (restDuration > 20) {
        RecordNote(piano, R, restDuration);
    }
}

PianoHandleInput(piano);

if (piano->recording) {
    RecordNote(piano, piano->activeFrequency, PIANO_NOTE_DURATION);
}
```

Listing 27: The piano stores both played notes and longer gaps so recorded songs preserve rhythm.

14.7 Interactive Piano Loop

The runtime structure of the piano differs from Snake. Snake updates on every timer-based iteration, while the piano mostly waits for keyboard input, reacts to note and control keys, redraws the terminal UI, and uses short PIT sleeps to avoid a tight polling loop when no key has been pressed. This makes the piano a useful example of an event-driven terminal application built from the same kernel mechanisms as the rest of the project [21, 22].

```
while (1) {
    char input = GetLastKeyPressed();

    if (!input) {
        SleepInterrupt(1);
        continue;
    }

    if (input == 'q') {
        PianoStopSound();
        break;
    }

    if (input == 'r' || input == 'p') {
        piano->lastInput = input;
        PianoHandleInput(piano);
        TerminalClear();
        DrawPianoUi(piano);
        continue;
    }
}
```

Listing 28: The piano loop combines keyboard polling, terminal redraws, and PIT-backed waiting.

15 Final State of the Project

At the end of the documented work, the kernel contains:

- early GDT setup and segment register reloading,
- IDT, ISR, IRQ, PIC, and keyboard interrupt support,
- kernel heap initialization and simple dynamic allocation,
- identity-mapped paging for the early kernel address space,
- PIT-based timer ticks and sleep functions,
- PC speaker sound generation through PIT channel 2,
- a menu-driven application flow,
- a music player application,
- an interactive Snake game using memory allocation, keyboard input, timing, terminal drawing, and sound feedback,
- an interactive piano application with live note input, song recording, and playback.

The project therefore moved from a minimal booting kernel that printed static text into a small interactive operating-system environment with hardware interrupts, timing, memory management, terminal input, sound, and application-level control flow. The final piano addition is especially useful as an integration example because it combines dynamic allocation, live keyboard input, PIT-based timing, terminal rendering, and PC speaker output inside one self-contained application.

16 Problems and Challenges

Several issues appeared during the project because the kernel runs in a freestanding environment where there is little separation between build configuration, CPU state, hardware programming, and application behaviour. Most problems were not isolated to one source file; they came from interactions between the boot image, low-level initialization, interrupts, timing, and terminal output.

16.1 Processor State and Verification

The GDT implementation required careful verification because a mistake in this stage can prevent the kernel from continuing at all. One important detail was that loading the GDT with `lgdt` does not automatically update the cached segment registers. The implementation therefore needed a far jump to reload `cs`, followed by explicit reloads of `ds`, `es`, `fs`, `gs`, and `ss`.

This was verified both statically and at runtime. Static inspection confirmed that the kernel binary contained the expected GDT load sequence, far jump, and data segment reloads. Runtime debugging in QEMU with `gdb-multiarch` confirmed that `cs` became `0x08` and that the data segment registers became `0x10`. This was necessary because the code could compile correctly while still failing if the processor did not actually enter the expected segment state.

16.2 Build and Boot Image Mismatch

A major integration problem appeared during the PIT work. Rebuilding only the kernel target updated `kernel.bin`, but did not update the bootable ISO image. As a result, QEMU could still boot an older image even after the source code and kernel binary had changed.

This made the runtime output misleading. The newer `kernel.bin` contained PIT and memory-management strings, while the older `kernel.iso` still showed the earlier `Hello, World!` output. The issue was resolved by rebuilding the bootable image target so the ISO contained the updated kernel.

This problem showed that kernel development requires verifying not only the source code, but also the exact artifact being booted. When the build system produces multiple outputs, it is possible to test the wrong one without noticing immediately.

16.3 Freestanding C and Header Consistency

Because the kernel does not run with a normal hosted C environment, ordinary library assumptions do not always apply. During the PIT integration, `printf()` calls had to be removed from test code and replaced with the kernel's terminal output functions. The terminal interface also needed to expose `TerminalWriteHex()` through the header because the function was implemented in `terminal.c` but not declared for other modules.

Inline assembly syntax also caused a build issue. An incorrect form of `__asm__volatile` had to be corrected to `__asm__ volatile`. This was a small syntax problem, but in low-level code it blocked the PIT sleep implementation because `sti; hlt` was part of the interrupt-based delay loop.

16.4 Interrupt Dependencies

Several later features depended on interrupt handling being correct. The PIT relies on IRQ0, keyboard input relies on IRQ1, and both the music player and Snake depend on timer and keyboard behaviour. This meant that interrupt setup was not only an isolated assignment task; it became part of the foundation for the rest of the project.

The PIC also had to be remapped so hardware IRQs did not overlap with CPU exception vectors. Without this separation, hardware events and processor exceptions would be harder to distinguish, and debugging later features would become much less reliable.

16.5 Terminal Output Limitations

The terminal output implementation was sufficient for early debugging and application display, but repeated output exposed a limitation. During the PIT sleep test, once the VGA text buffer wrapped back to the top of the screen, new text overwrote old text directly. This made long-running debug output harder to read.

The later application framework reduced this issue by clearing the terminal before drawing menus and game frames. However, the underlying limitation remains: the terminal would benefit from proper scrolling or a cleaner screen-management model for continuous logs.

16.6 Subsystem Integration

The final Snake application exposed the main integration challenge of the project. Snake uses dynamic memory allocation for game state, keyboard interrupts for input, PIT timing for movement speed, terminal output for drawing, and PC speaker output for feedback. A problem in any of these lower-level systems can appear as an application-level bug.

The Piano implementation also exposed this integration challenge. It uses dynamic memory allocation for piano state which contains a small library implementation, keyboard interrupts for input, PIT for note durations, terminal output for drawing, and PC speaker output for playing actual notes.

This made the final stage useful as more than a demonstration. It acted as an integration test for the kernel. The menu, Snake game and Piano application showed that the separate subsystems could cooperate through a single control flow, but they also made clear that kernel features must be built in a stable order: processor setup first, then interrupts, then memory and timing, then application behaviour.

16.7 Sound issue

One final issue worth mentioning concerns audio functionality. We are confident that our kernel is configured correctly with respect to sound. However, when Teodor runs the project using the qemu-start.sh script, the audio does not function properly. In contrast, Christopher is able to run the same project without any sound issues. The primary difference between their setups is that Teodor is using Windows, while Christopher is using Linux.

The issue on Teodor's system consistently follows this pattern:

- Sound initially works as expected
- Audio quality gradually degrades
- Increasing delays occur between sounds
- Audio becomes fragmented into short, intermittent bursts
- Sound eventually stops completely

According to Teodor's experience, this behavior occurs regardless of which command is used to run the project, and it affects all applications within the operating system. Based on this, we suspect that the issue may be related either to sound handling on Windows or to a problem specific to Teodor's setup, despite our careful adherence to the provided setup guides.

17 Conclusion

The project started with a minimal bootable kernel and developed it into a small interactive operating-system environment. The final kernel includes early processor setup, interrupt handling, memory management, PIT-based timing, PC speaker sound, keyboard input, terminal output, a menu system, a music player, and a Snake game.

The first major step was to make the processor setup explicit by adding a minimal Global Descriptor Table and reloading the segment registers correctly. This provided a controlled protected-mode foundation for later work. Interrupt handling then extended the kernel from linear startup code into an event-driven system, with CPU exceptions, hardware IRQs, PIC remapping, and keyboard input.

Memory management and paging added another important layer. The kernel gained heap initialization, dynamic allocation, page-aligned allocation, memory utility functions, and identity-mapped paging. These features made it possible for later code to allocate state dynamically rather than relying only on static data.

The PIT then provided a timing source through IRQ0. This supported both busy-wait and interrupt-based sleeping, with the interrupt-based version becoming important for later features. The same timing infrastructure was reused by the music player and Snake game. The PC speaker work showed how PIT channel 2 could be used for sound generation.

The final application framework tied the lower-level pieces together. The menu system used terminal output and keyboard input to choose between applications. The music player used PIT timing and PC speaker output. Snake combined memory allocation, keyboard input, PIT-based pacing, terminal drawing, and sound effects. And Piano combined terminal drawing, memory allocation, keyboard input, PIT-based pacing and sound. This made the final application stage a practical test of whether the kernel subsystems could work together.

Overall, the project demonstrates how small operating-system mechanisms build on each other. GDT setup makes the early CPU state reliable, interrupts make asynchronous hardware events usable, memory management allows dynamic state, timing enables delays and pacing, and device output makes applications interactive. The result is not a complete general-purpose operating system, but it is a working kernel environment that shows the relationship between low-level hardware control and higher-level program behaviour.

18 Work Distribution

The project work was divided primarily by assignment stage. Teodor was mainly responsible for assignments 2, 4, and 5, and shared assignment 6. Christopher was mainly responsible for assignments 1 and 3, and also shared assignment 6. If only the implementation assignments are considered, this gives an approximate 60/40 split in Teodor's favour.

The report writing was distributed somewhat differently. Christopher wrote approximately 60% of the report text, which balanced much of the earlier implementation difference. Taken together, the implementation work and the report writing therefore gave a contribution split that was close to 50/50 overall, which we consider a reasonable balance for a group project of this type.

We also worked deliberately to maximize learning from the parts of the project that each of us had not implemented directly. After each work session, we wrote summary notes describing what had been done, what design choices had been made, and what problems had appeared. This made it possible for the other person to read the notes afterward and then write that part into the report with a clear understanding of both the implementation and the reasoning behind it.

In practice, this meant that when Teodor implemented an assignment task, Christopher would write the corresponding report section, and the same principle also applied in the opposite direction. This gave both of us a reason to study and explain the other person's work instead of only documenting our own. In that way, the report became part of the learning process rather than a duplicate writing pass over the same task.

References

[1] Apple. *Boot process for a Mac with Apple silicon*. https://support.apple.com/guide/security/secac71d5623/web. Accessed 2026-04-20.

[2] Apple. *Boot process for an Intel-based Mac*. https://support.apple.com/guide/security/sec5d0fab7c6/web. Accessed 2026-04-20.

[3] Gustavo Duarte. *How Computers Boot Up*. https://manybutfinite.com/post/how-computers-boot-up/. Accessed: 2026-04-28. 2011.

[4] Carlos Fenollosa. *os-tutorial*. https://github.com/cfenollosa/os-tutorial/tree/master. Accessed 2026-04-16.

[5] Erik Helin and Adam Renberg. *The Little Book About OS Development*. https://littleosbook.github.io/. Accessed: 2026-04-28. 2015.

[6] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed 2026-04-28.

[7] IONOS. *What is a bootloader?* https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/. Accessed: 2026-04-28. Nov. 2022.

[8] Limine Bootloader Project. *Limine Boot Protocol*. https://github.com/limine-bootloader/limine-protocol. Accessed 2026-04-20.

[9] Linux Kernel Documentation. *The Linux/x86 Boot Protocol*. https://docs.kernel.org/arch/x86/boot.html. Accessed 2026-04-20.

[10] Logeshwaran N. *Virtual Machine Boot Process Explained – Easy Guide*. https://logeshwrites.hashnode.dev/virtual-machine-boot-process-explained-easy-guide. Accessed: 2026-04-28. Oct. 2024.

[11] Microsoft. *Configure and edit boot options in Windows for driver development*. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-options-in-windows. Accessed 2026-04-20.

[12] OSDev Wiki. *8259 PIC*. https://wiki.osdev.org/8259_PIC. Accessed 2026-04-20.

[13] OSDev Wiki. *Boot Sequence*. https://wiki.osdev.org/Boot_Sequence. Accessed 2026-04-20.

[14] OSDev Wiki. *GDT Tutorial*. https://wiki.osdev.org/GDT_Tutorial. Accessed 2026-04-20.

[15] OSDev Wiki. *Interrupts*. https://wiki.osdev.org/Interrupts. Accessed: 2026-04-28. Mar. 2026.

[16] OSDev Wiki. *Interrupts Tutorial*. https://wiki.osdev.org/Interrupts_Tutorial. Accessed 2026-04-16.

[17] OSDev Wiki. *Memory Allocation*. https://wiki.osdev.org/Memory_Allocation. Accessed 2026-04-20.

[18] OSDev Wiki. *Memory Map (x86)*. https://wiki.osdev.org/Memory_Map_(x86). Accessed 2026-04-20.

[19] OSDev Wiki. *PC Speaker*. https://wiki.osdev.org/PC_Speaker. Accessed 2026-04-20.

[20] OSDev Wiki. *Printing To Screen*. https://wiki.osdev.org/Printing_To_Screen. Accessed 2026-04-28.

[21] OSDev Wiki. *Programmable Interval Timer*. https://wiki.osdev.org/Programmable_Interval_Timer. Accessed 2026-04-20.

[22] OSDev Wiki. *PS/2 Keyboard*. https://wiki.osdev.org/PS/2_Keyboard. Accessed 2026-04-29.

[23] OSDev Wiki. *Setting Up Paging*. https://wiki.osdev.org/Setting_Up_Paging. Accessed 2026-04-20.

[24] QEMU Project. *i440fx PC (pc-i440fx, pc)*. https://qemu.readthedocs.io/en/v8.1.5/system/i386/pc.html. Accessed 2026-04-20.

[25] Raiden. *The Windows Operating System Boot Process in UEFI Mode*. https://cyberraiden.wordpress.com/2025/07/24/the-windows-operating-system-boot-process-in-uefi-mode/. Accessed: 2026-04-28. July 2025.

[26] Robert Sheldon. *POST (Power-On Self-Test)*. https://www.techtarget.com/whatis/definition/POST-Power-On-Self-Test. Accessed: 2026-04-28. Aug. 2022.

[27] Turgay Celik. *assignment_files.zip*. Source code and assignment material provided by course lecturer.

[28] Wikipedia. *Power-on self-test*. https://en.wikipedia.org/wiki/Power-on_self-test. Accessed: 2026-04-28. Apr. 2026.

[29] Wikipedia contributors. *Limine (bootloader)*. https://en.wikipedia.org/wiki/Limine_(bootloader). Accessed: 2026-04-28. Apr. 2026.