

OCTOBER 2025

MELOMYS OPERATING SYSTEM

Prepared By

Daniel



About The Authors

Hi, my name is Daniel and I am a Computer Science student and I have been programming for over half a decade. I got into embedded system development three years ago when I first decided to build an operating system from scratch. Since then, I have developed numerous "semi-operating systems" projects as hobby. I don't claim to be an expert; rather, I consider myself as a random person who want to share knowledge and experience with others. I have spent countless time coding/programming, so I can say I have a decent experience to guide others.

Thanks for giving it time to read!

INTRODUCTION

The development of an operating system is arguably one of the most challenging and yet most rewarding endeavors in computing. It requires a deep understanding of low level system programming, hardware architecture and years of experience with computer. It demands dedication, patience and commitment to constantly learn and adapt, and trust me, you're going to **need** it.

My goal in writing this documentation is to provide a useful, yet simple resource for both newcomers and seasoned programmers. Unlike typical OS development guides that present large blocks of complex code with vague explanation '*most are very useful in contributing a vigilant dev*', I documented by adding a single, small feature in each part, so you can understand and experiment it yourself without getting overwhelmed. I chose this approach

because I believe every line of code is necessary and should be explained in the simplest way possible so the reader can retain more.

About This Book

This book serves as a useful resource in both building and understanding a modern operating system from the ground up, starting from the absolute minimum to the point of executing a high level code.

Who Should Read This Book?

This book is written for readers who are familiar with basic programming concepts and who have the passion and desire to understand modern computers and work in this field. It is also useful for so called “professionals” who want to read/understand a specific topics in greater depth.

Why x86_64 Assembly?

Melomys is programmed entirely in NASM Assembly for the x86_64 architecture.

I made this decision for ¹speed, This is one of the most controversial topics to talk about. Assembly is **theoretically** faster cause you have complete control over the hardware. however, despite this advantage, very few developers today truly understand

assembly level optimization well enough to ‘outperform’ modern C compilers. But in most cases speed is not a practical advantage of using assembly, at least not anymore.² Efficiency/performance, Assembly gives you full control, but that control is both a benefit and a ‘punishment’. Even with complete power over the hardware there are many opportunities to accidentally break everything, and that’s normal cause I done it, so will you . As a wise man once said, ‘with great power comes great responsibility.’

³Size of the binary, For the most part it’s smaller in size cause there is no stack protector, no hidden prologue code etc., but at the same time your limited to a specific CPU architecture. ⁴Compatibility, I have already explained the compatibility issues with assembly generated binaries, so I won’t repeat them here (and I won’t remove this either).

What's in this book?

CHAPTER 1: THE UEFI BOOTLOADER

1.1. The Bootloader

1.1.1. Definition and Fundamental Purpose

A bootloader is a small program(mostly less than 1MiB) responsible for booting the hardware and operating system. Since RAM is volatile and empty at startup the CPU can't 'know' how to run the kernel. The bootloader acts as the initial executable that locates, loads and executes the system software stored on non-volatile memory(like an SSD or USB). So bootloader is specific code that loads the kernel.

There is also something called boot manager which is an interactive interface(if you dual booted before, you likely heard of GRUB) that allows a user to choose between different operating systems.

1.1.2. The Modern Bootloader Stages

Modern systems use a multi-stage because firmware space is limited. For example in bios you can only fit 512 byte and considering that's impossible to even do anything, it's impossible)what I mean by that is if you we can look at legacy BIOS bootloader, you are restricted to a 512 byte limit. since it is almost impossible to implement meaningful logic within such a tiny size, you will need a more than one loader that does a some level of initialization for the kernel.

1. **First stage bootloader:** when you power on your pc, the CPU needs a "map" to start the system. Depending on your hardware, it follows one of two paths:
 - **Legacy BIOS :** is 16 bit standard(which means you start from 16 bit real mode). It searches for a specific "Boot sector" at the very beginning of your disk and performs a tiny, loading exactly 512 bytes of code into memory for the CPU to execute. And I personally think It's simpler to program bios bootloader than of UEFI 'and yes, I 100% do.'
 - **UEFI :** unlike legacy BIOS, it's 64 bit which the bootloader automatically start from either 32 real mode bit or 64 bit long mode, instead of doing it manually. Also It locates the .efi application within the FAT32 partition and launches it to start the boot process.
2. **Second Stage loader:** is the the main loader cause It uses the data gathered by the first stage(bootloader) to initialize the “core” system

environment like setting up paging, the memory map, graphics output, etc. before finally loading the kernel into memory.

1.1.3. UEFI

Unlike legacy BIOS, which "forces" you to fit their entire first stage into a tiny 512 byte sector, UEFI simplifies the process since it uses file system natively, directly loads(/EFI/BOOT/BOOTX64.efi) and it's native 64 bit.

1.2. Our First UEFI Bootloader

1.2.1. Do Nothing and Do It Well

Before we start with our first bootloader, it is necessary to understand the tools that make "the operating system" development possible.

Installation on Win

If your developing(wanting to develop) on windows 10 or 11, there is a bit complicated than of linux. This video will guide you through as it did to me:

- watch: [Setting up UEFI Dev on Windows](#)

Installation on Linux

- Arch linux:

```
sudo pacman -S nasm qemu-full mtools dosfstools ovmf xorriso
```

- Debian/Ubuntu:

```
sudo apt update
sudo apt install nasm qemu-system-x86 mtools dosfstools ovmf xorriso
```

Toolkit

Note: These tools are only for bootloader and we'll be using way more once we get out of low level parts

1. NASM (The Netwide Assembler): It is our primary assembler.
2. QEMU: It simulates a full x86_64 computer system in software.
3. OVMF (Open Virtual Machine Firmware): since standard QEMU defaults to legacy BIOS. By using this firmware we'll make it virtually use UEFI instead of legacy.
4. mtools and dosfstools (mkfs.vfat, mmd, mcrypt): these allow us to create and manipulate FAT32 filesystems directly from the command line
5. xorriso: tool packages our files and the boot image into a .iso file.

File Tree

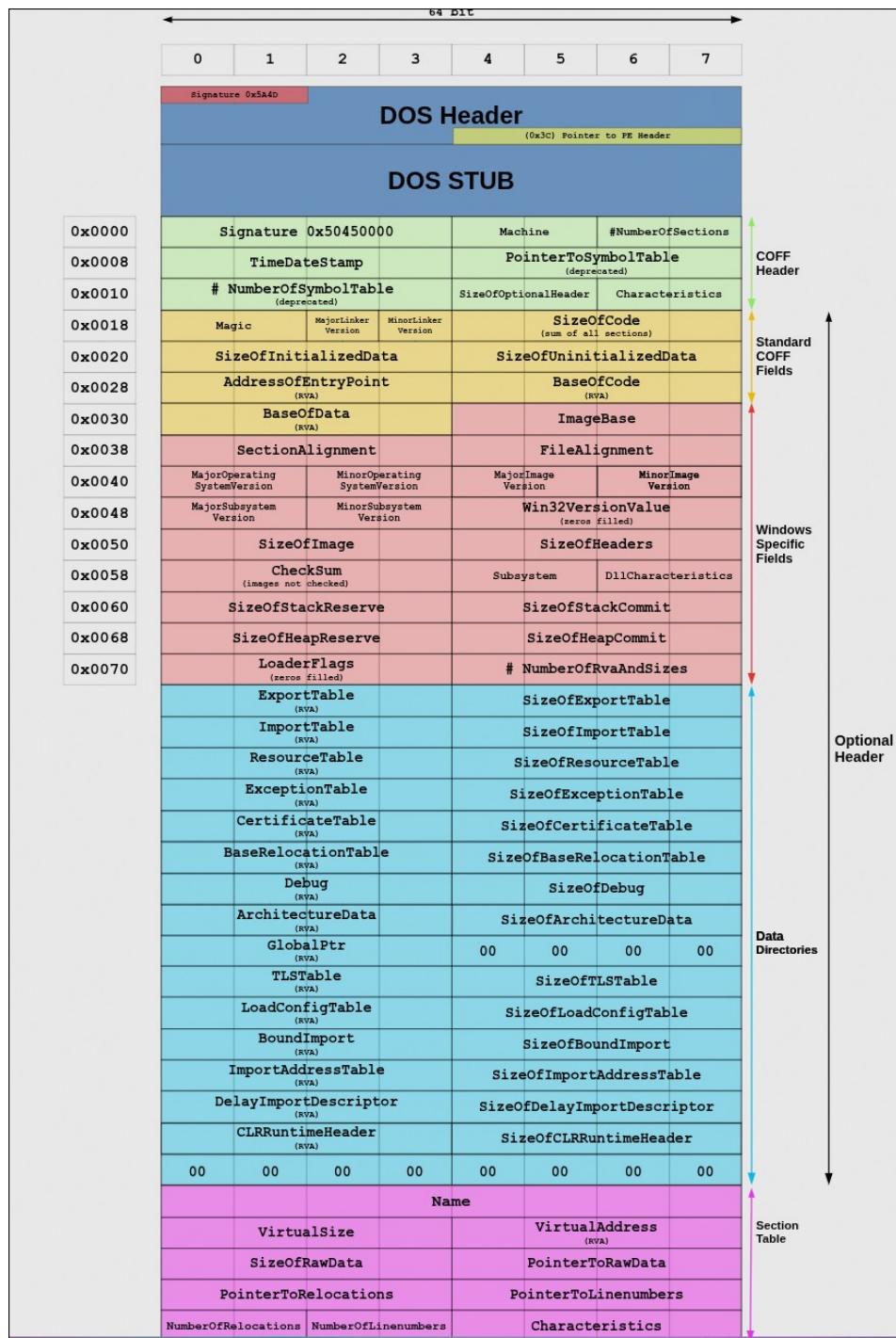
```
.
├── build
│   ├── BOOTX64.EFI
│   ├── efiboot.img
│   └── iso
│       └── MELOMYS.iso
└── iso_root
    └── EFI
        └── BOOT
            └── BOOTX64.EFI
├── Makefile
└── ovmf
    ├── OVMF_CODE.fd
    └── OVMF_VARS.fd
└── src
    └── uefi.asm

8 directories, 8 files
```

Code Explanation

If you're new, the most obvious thing to do is search. The question is what to search for. UEFI bootloaders are portable executables which means you must follow a specific structure so the firmware treats them as bootloaders rather than normal windows or app binaries. The easiest way to figure this out is honestly, to just **Google it**.

(https://en.wikipedia.org/wiki/Portable_Executable)



Before we start constructing the structure itself we first need to define the execution environment. this includes setting the architecture, addressing mode and the load address expected by the assembler.

```
bits 64          ; Assemble instructions using 64-bit mode
org 0x400000    ; Address hint for the assembler
default abs      ; Use absolute addressing by default
```

UEFI firmware loads the bootloader into memory and then transfers control to it. The `org 0x400000` directive does not force the firmware to load the bootloader at this exact address. Instead it tells the assembler in our case ‘nasm’ to assume this address when calculating labels and offsets. I used `0x400000` cause it’s commonly used because it is above low memory and matches the typical load address used by many UEFI implementations.

The next will be PE/COFF header and there no good documentation than (<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format>) but first we need to write a DOS header (yes, still in 2025)

DOS Header

```
header_start:
db 'MZ', 0x00, 0x00          ; DOS signature
times 60-($-header_start) db 0
dd pe_sig - start            ; Relative location of PE signature
times 64 db 0
```

Every PE file must start with dos header signature and `times 60-($-header_start) db 0` is just math where

- \$ = current position in the file
- \$ - header_start = how many bytes we have written so far

DOS header expects a field at offset **0x3C** (as you can see in the pic) which `dd pe_sig - start` make it have an offset to pe-header in it and finally pad it with `times 64 db 0` to align for performance

Offset (hex)	Size	Content / Label	Meaning
0x400000	2	4D 5A ("MZ")	DOS signature
0x400002	2	00 00	Padding
0x400004	56	00 ... 00	DOS stub padding (times 60-(\$-header_start) db 0)
0x40003C	4	pe_sig - start	e_lfanew → offset of PE header
0x400040	64	00 ... 00	Extra DOS stub padding (times 64 db 0)
0x400080	4	50 45 00 00 ("PE\0\0")	PE signature
0x400084	2	8664	Machine type x86_64
0x400086	2	2	Number of sections (.text + .data)
0x400088	4	1759708800	Timestamp
0x40008C	4	0	Pointer to symbol table (unused)
0x400090	4	0	Number of symbols (unused)
0x400094	2	opt_header_end - opt_header_start	Size of optional header
0x400096	2	0x222E	Characteristics flags
0x400198	Start Optional Header		
0x400108	8	opt_header_start:	PE32+ optional header begins
0x400108	8	opt_header_end:	Optional header ends

Offset (hex)	Size	Label / Content	Meaning
0x400108	8	.text\0\0\0	Section name
0x40012C	8	.data\0\0\0	Section name

```

db 'PE', 0x00, 0x00      ; PE Signature
dw 0x8664                ; x86_64 machine
dw 2                      ; Two sections: .text and .data
dd 1759708800            ; Timestamp
dd 0
dd 0
dw opt_header_end - opt_header_start
dw 0x222E                ; Characteristics

```

The `db 'PE', 0x00, 0x00` signature and tells the UEFI loader how to interpret the rest of the executable while `dw 0x8664` specifies that the target architecture is `x86_64`.

`dw 2` declares that the file contains two section headers which are the `.text` and `.data` sections defined later.

The timestamp is a conventional build time field that the loader largely ignores but still expects to exist which in our case(`1759708800`) is ‘Monday, Oct 6, 2025’.

The two zeroed fields disable the COFF symbol table which is unnecessary for a UEFI application. And the optional header size field tells the loader exactly how many bytes to read for the PE32+ optional header that follows

The characteristics flags [0x222E](#) describe the file as a 64 bit executable image with properties appropriate for a UEFI application rather than a user mode windows program.

Flag		
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image is executable
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	
IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	Image can handle virtual addresses above 2 GB, so the app's code is safe to use the high bit of a 32 bit pointer
IMAGE_FILE_DEBUG_STRIPPED	0x0200	No debug info in the binary
IMAGE_FILE_DLL	0x2000	Yeaahh, EFI applications are treated like DLLs

(<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#characteristics>)

$$0x0002 + 0x0004 + 0x0008 + 0x0020 + 0x0200 + 0x2000 = 0x222e$$

```
opt_header_start:  
dw 0x020B ; PE32+ (64-bit)  
db 0  
db 0  
dd code_sec_end - code_sec_start  
dd data_sec_end - data_sec_start  
dd 0x00  
dd code_sec_start - start  
dd code_sec_start - start  
dq 0x400000 ; Image Base  
dd 0x1000 ; Section Alignment  
dd 0x1000 ; File Alignment  
dw 0, 0, 0, 0, 0, 0  
dd 0  
dd end - start ; Full image size  
dd header_end - header_start  
dd 0  
dw 10 ; Subsystem: EFI Application  
dw 0  
dq 0x200000 ; Stack Reserve  
dq 0x1000 ; Stack Commit  
dq 0x200000 ; Heap Reserve  
dq 0x1000  
dd 0x00  
dd 0x00  
opt_header_end:
```

dw 0x020b

Magic number	PE format
0x10b	PE32
0x20b	PE32+

<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-image-only>