**MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA**

**FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

Operating Systems

***Laboratory work 3: Floppy disk input/output operations***

Elaborated:

st.gr. FAF-212 Cristian Brinza

Cara Alexandr

Bogdan Zlatozcean

Verified: Rostislav Calin

Chis, ina˘u, 2023

# Task:

Toți studenții de sine stătător vor forma echipe a câte 3 membri din cadrul grupei academice. Componența grupelor va fi aprobată de profesor, inclusiv echipele de 2 sau 4 membri.

Echipele trebuie formate operativ, în limita termenului de timp anunțat de profesor la ora de curs.

Obiectivele lucrării date de laborator sunt focusate pe însușirea abilităților de lucru cu metodele de acces la floppy disk, în special citirea și scrierea datelor, dar nu se limitează doar la acestea. Procedurile date se extind si asupra altor medii de stocare permanentă a datelor precum sunt discurile HDD.

Volumul total la dischetei de 1474560 bytes va fi structural în 96 de blocuri logice a câte 30 de sectoare fiecare. Fiecare student va avea alocat un bloc individual de 15360 bytes, pentru înscrierea datelor sale după cum urmează.

Distribuția blocurilor este reprezentată în fișierul inclus (”Floppy space distribution.xlsx”).

1. În primul și în ultimul sector al blocului fiecărui student (pe dischetă), trebuie înscrisă informația textuală după următorul format (fără ghilimele): ”@@@FAF-21\* Prenume NUME###”.

Acest șir de text trebuie dublat de 10 ori fără caractere adiționale de delimitare.

Exemple:

@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###

@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###

2. Creați un program în limbajul de asamblare care va avea următoarele funcții:

(KEYBOARD ==> FLOPPY) : Citirea de la tastatură a unui string cu lungimea maximală de 256 caracetre (corectarea cu backspace trebuie să funcționeze) și scrierea acestui șir pe dischetă de "N" ori, începând cu adresa {Head, Track, Sector}. Unde "N" poate primi valori în intervalul (1-30000). După detectarea tastei ENTER, dacă lungimea șirului este mai mare ca 0 (zero), trebuie afișată o linie goală și apoi șirul recent introdus. Variabilele "N", "Head", "Track" și "Sector" trebuie citite vizibil de la tastatură. După finisarea operației de scriere pe dischetă, trebuie de afișat la ecran codul de eroare.

**(FLOPPY ==> RAM) : Citirea de pe dischetă a "N" sectoare începând cu adresa {Head, Track, Sector} și transferul acestor date în memoria RAM începând cu adresa {XXXX:YYYY}. După finisarea operației de citire de pe dischetă, trebuie de afișat la ecran codul de eraore. După codul de eroare, trebuie de afișat la ecran tot volumul de date aflat la adresa {XXXX:YYYY} care a fost citit de pe dischetă. Dacă volumul de date afișat este mai mare decât o pagină video atunci este necesar de implementat paginație prin apăsarea tastei ”SPACE”. Variabilele "N", "Head", "Track" și "Sector" precum și adresa {XXXX:YYYY}, la fel trebuie citite de la tastatură.**

(RAM ==> FLOPPY) : Scrierea pe dischetă începând cu adresa {Head, Track, Sector} a unui volum de "Q" bytes, din memoria RAM începând de la adreasa {XXXX:YYYY}. Blocul de date de "Q" bytes trebuie afișat la ecran, iar după finisarea operației de scriere pe dischetă, trebuie de afișat la ecran codul de eroare.

3. După executarea unei funcții de mai sus, programul trebuie să fie gata la executarea următoarei funcții (oricare funcție din cele 3 descrise mai sus).

4. Codul compilat este preferabil să nu depășească 512 bytes. În caz contrar, este necesar de implementat ocolirea acestei restricții și în final trebuie creată imaginea bootabilă a dischetei care să funcționeze în VirtualBox.

5. Fiecare echipă trebuie să perfecteze un raport structurat, inclusiv cu concluzie, similar cu lucrările anterioare. Raportul trebuie să includă codul-sursă complet și funcțional, precum și procedura completă (testată) de compliare și de producere a imaginii bootabile, care să funcționeze corect în VirtualBox. Dacă descrierea procedurii de creare a imaginii bootabile va lipsi din raport sau dacă procedura nu va fi completă sau nu va funcționa (nu va putea fi reprodusă), întreaga echipă riscă să primească nota 5(CINCI)!! Din acest considerent este necesar ca fiecare membru a echipei să testeze procedura. În raport este necesar să specificați inclusiv versiunea compilatorului folosit și a tuturor altor componente sau dependențe care au fost folosite în procesul de construcție (build) a imaginii bootabile.

# Execution:

This assembly code is designed for a boot sector program, typically used in the initial stages of a computer's boot process. The program is written for the x86 architecture and is intended to be loaded at memory address 0x7C00, a standard location for boot sector code. Let's break it down into its components:

**Data Section:** The program begins with a data section where various strings and constants are defined. These are primarily used for user prompts and messages.

**BSS Section:** This section reserves space for buffers and variables, indicating preparation for data storage and manipulation.

**Text Section:** The main code of the program, containing logic for initializing, handling input/output, and clearing screen and memory buffers.

**Detailed Analysis**

Initialization and Memory Reset: The program starts by initializing various buffers and setting registers to zero. This is crucial for preparing the system for the operations to follow.

Screen Clearing: Utilizing BIOS interrupt 0x10, the program clears the screen, setting up a clean state for user interaction.

String and Buffer Handling: There are operations for copying strings into buffers, determining the end of strings, and other string manipulations.

Input and Output Handling: The code includes routines to handle user input commands, display error codes, and reset the program to its start point after confirmation.

Buffer Operations: The program contains routines for printing buffer contents and clearing buffers, indicating its role in handling dynamic data.

**Key Takeaways:**

Low-Level System Interaction: The program directly interacts with system hardware, a characteristic of assembly language programming. This is evident in the use of BIOS interrupts for screen operations and keyboard inputs.

Memory Management: The program meticulously manages memory, as seen in the allocation and clearing of buffers. This highlights the importance of memory management in system-level programming.

User Interface Elements: Despite being a low-level program, it includes elements of user interaction, such as screen clearing and displaying prompts, indicating a bridge between the system and the user.

**Data Section**

The data section is used to define various strings and constants, which are utilized throughout the program for user interaction and internal logic.

function1\_str db "1. Keyboard to Floppy", 0

; This line defines a string '1. Keyboard to Floppy' followed by a null character, indicating the end of the string.

**BSS Section**

This section reserves space for data storage, crucial for dynamic data handling within the program.

string resb 256

; Reserves 256 bytes for a buffer named 'string', to be used for storing user input or other data.

Text Section

The text section is the core of the program, where the actual operations and logic are implemented.

**Initialization and Memory Reset**

At the start, the program initializes various buffers and sets registers to zero, preparing the system for operations.

mov si, storage\_buffer

mov di, storage\_buffer + 512

call clear\_buffer

; These lines set the SI and DI registers to specific memory locations and then call the 'clear\_buffer' routine to initialize the buffer.

**Screen Clearing**

The program uses BIOS interrupts for clearing the screen, setting up a clean interface for user interaction.

mov ah, 0x06

mov al, 0

int 0x10

; Sets up the registers for the BIOS interrupt 0x10 to clear the screen.

**String and Buffer Handling**

Operations for copying strings to buffers and determining string lengths are demonstrated.

mov si, string

call find\_end

; Moves the address of 'string' into SI and calls 'find\_end' to determine the string's length.

**Input and Output Handling**

The program contains routines to handle user input, display messages, and manage the program flow.

input\_command:

mov ah, 03h

int 10h

; This routine uses the BIOS interrupt to read user input from the keyboard.

**Buffer Operations**

Routines for printing buffer contents and clearing buffers are included, showcasing dynamic data handling.

print\_buff:

mov cx, 0

; Sets up a loop to print the contents of a buffer until a null character is encountered.

**Bootloader Program Analysis**

Program Structure

The bootloader program is compact and focuses on loading the operating system into memory. It is structured as follows:

Initialization: Sets up the environment and prepares the system for reading from the disk.

Reading from Disk: Uses BIOS interrupts to read the operating system from disk into memory.

Jump to Loaded OS: Transfers control to the loaded operating system.

Detailed Analysis

Initialization

The bootloader begins by setting up the required register values for disk operations.

org 7c00h

; Sets the origin point, indicating that the bootloader will be loaded at memory address 7C00h.

mov ah, 00

int 13h

; Resets the disk system using interrupt 13h.

**Reading from Disk**

The program reads a specific sector from the disk into memory. This is typically where the operating system kernel is located.

mov ax, 0000h

mov es, ax

mov bx, 7e00h

; Sets up segment and offset addresses to where the OS will be loaded.

mov ah, 02h

mov al, 4

int 13h

; Performs the read operation from the disk using BIOS interrupt 13h.

**Jump to Loaded OS**

After loading the OS into memory, the bootloader transfers control to the operating system.

jmp 0000h:7e00h

; Jumps to the memory address where the OS is loaded, effectively transferring control.

**Boot Signature**

The program ends with a boot signature, which is a requirement for any bootable device.

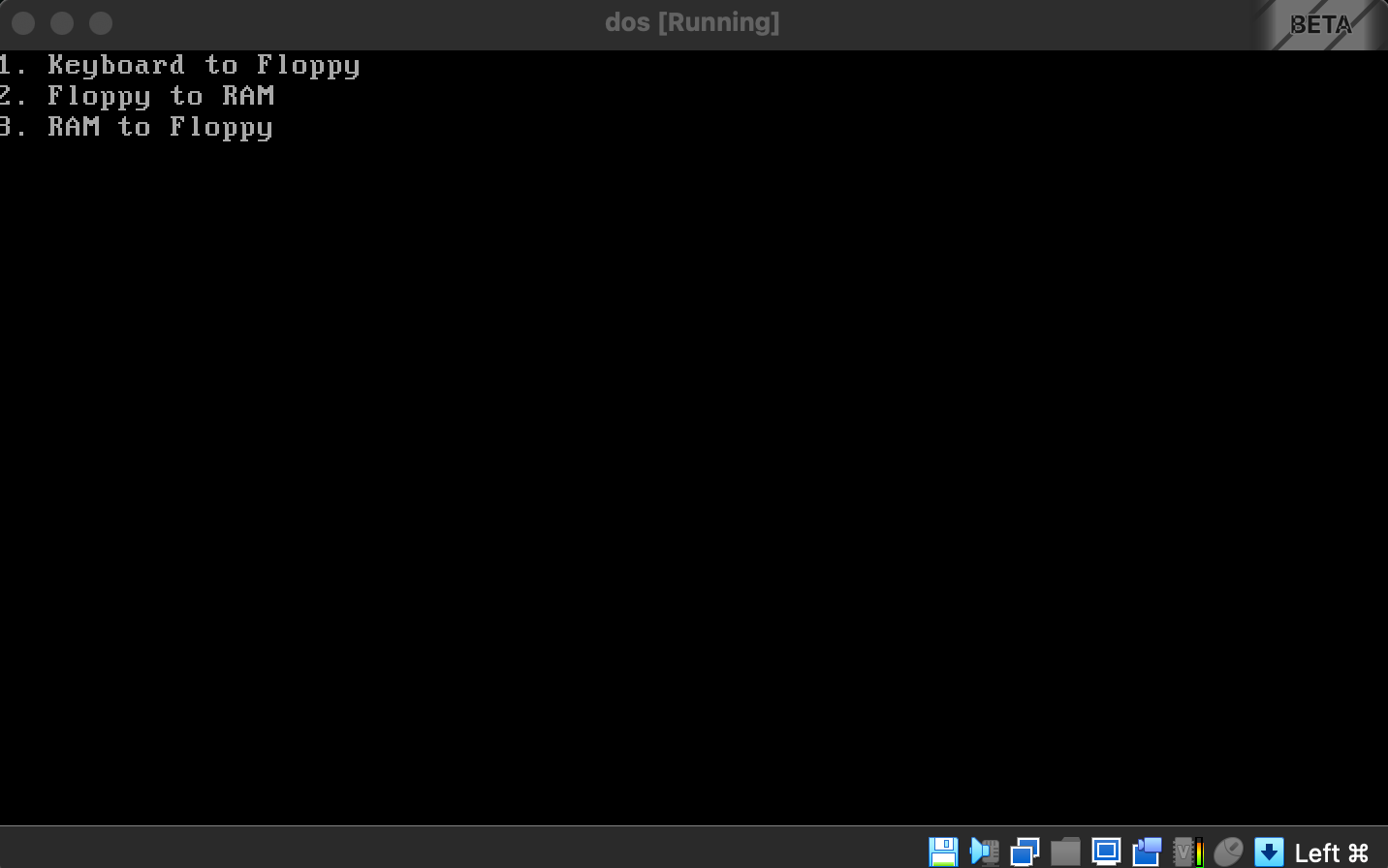
times 510-($-$$) db 0

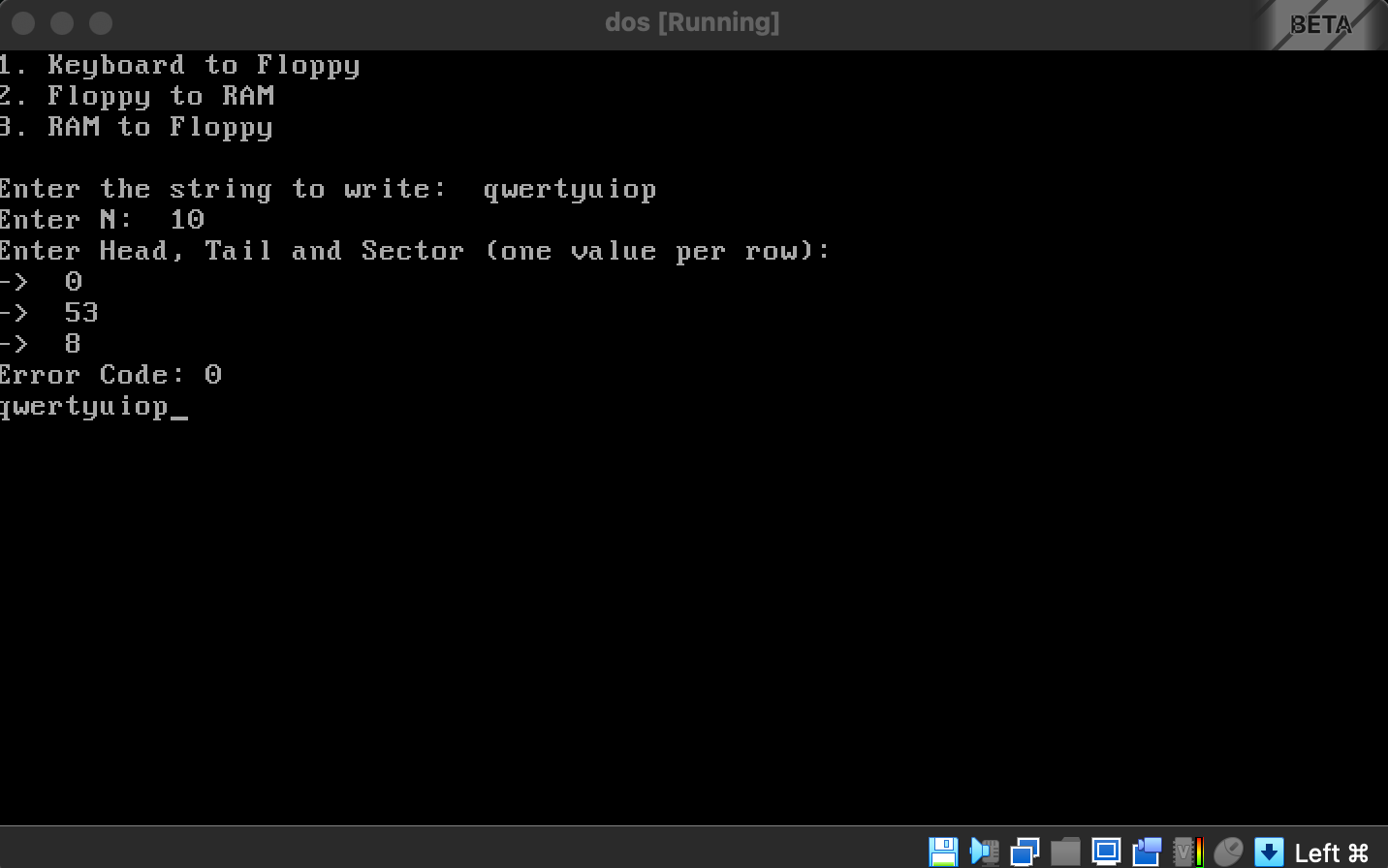
dw 0AA55h

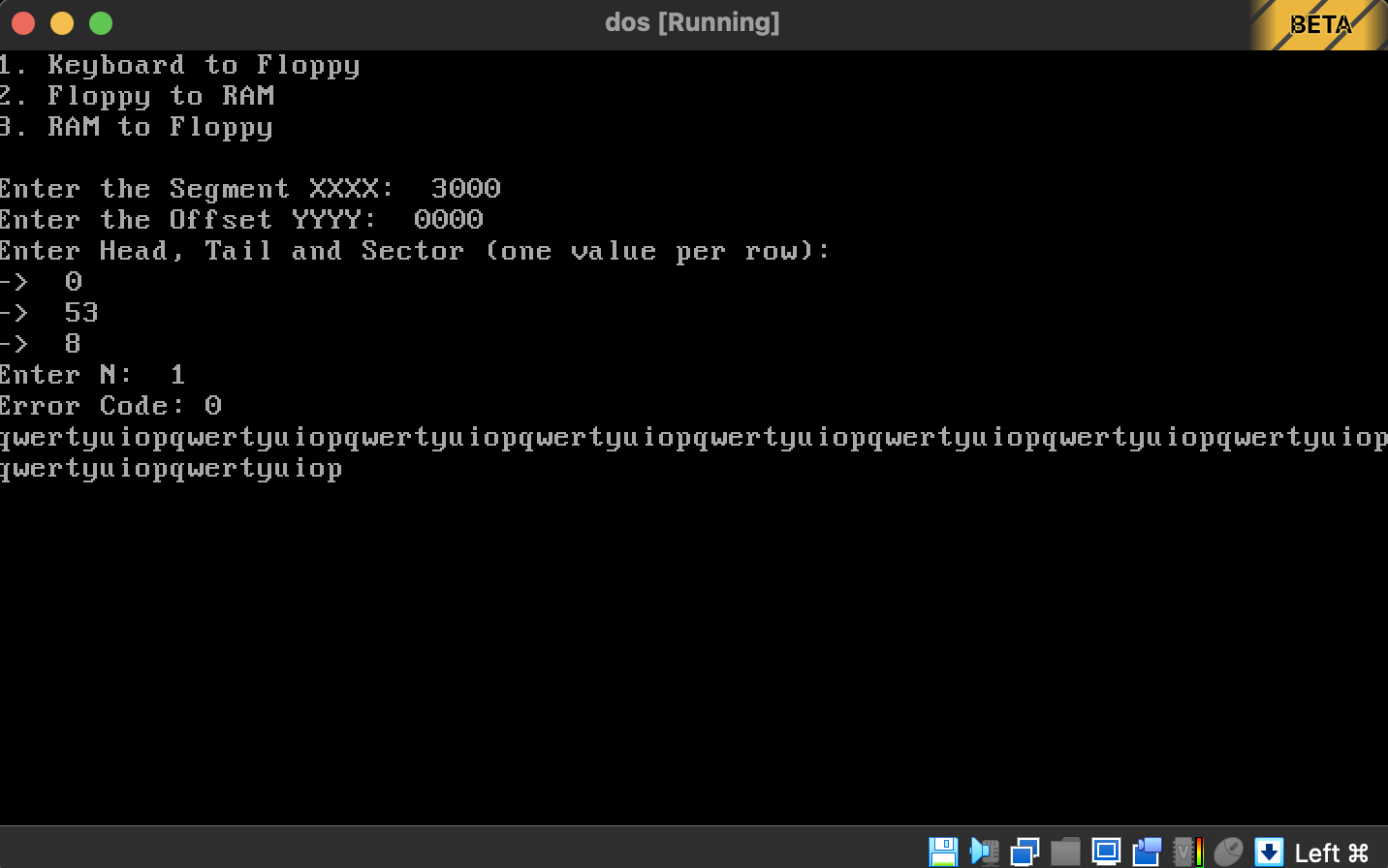
; Fills the remaining bytes to make a 512-byte sector and adds the boot signature 0xAA55.

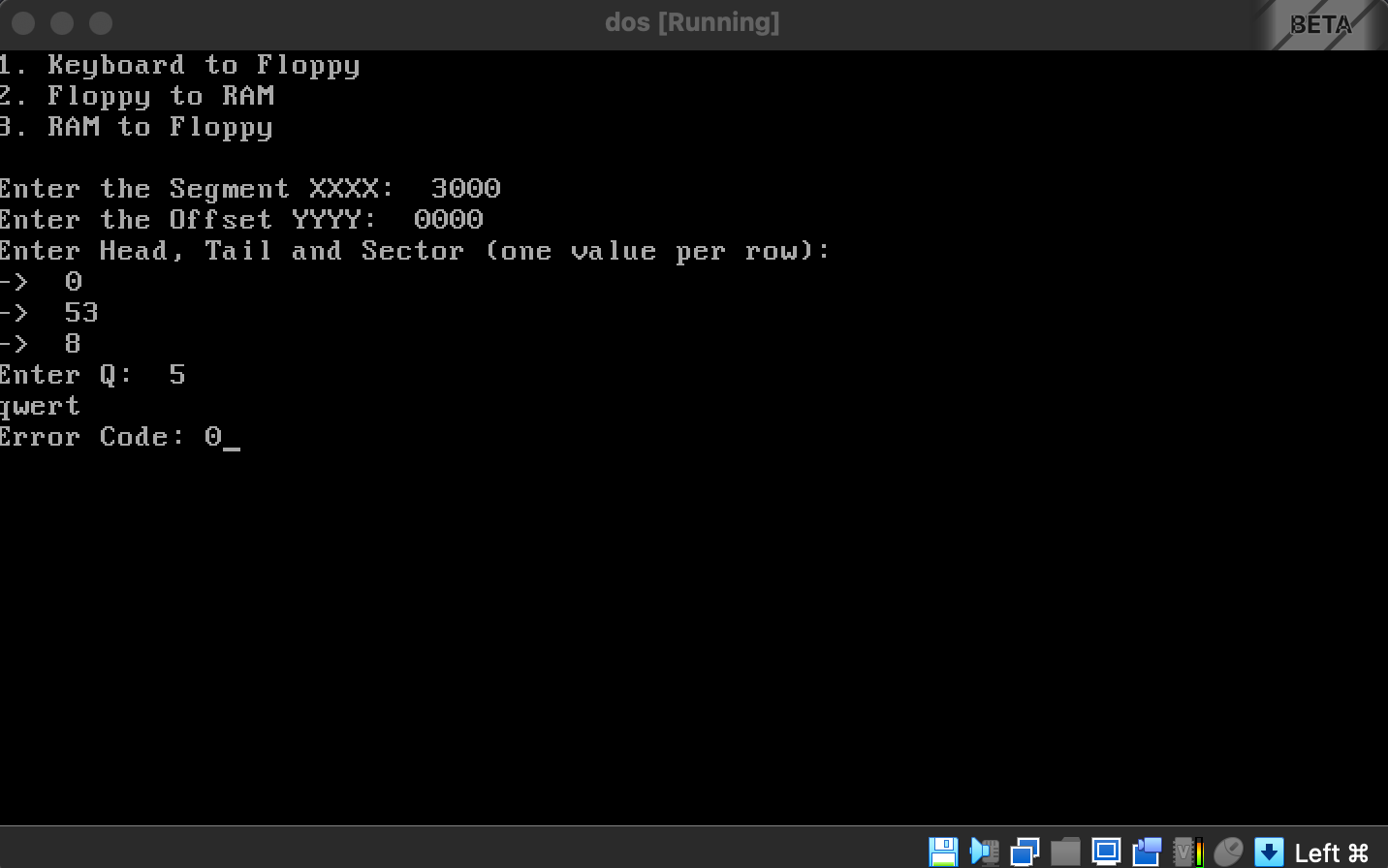
**Integration with Previous Analysis**

The bootloader complements the previously analyzed program by demonstrating a critical aspect of system programming: the initial loading phase of an operating system. This program, while concise, is crucial as it sets the stage for the operating system to take over. It demonstrates direct manipulation of hardware, BIOS-level programming, and low-level memory management, similar to the previously examined program.









**Conclusion:**

Through this laboratory work, we have gained a basic understanding of assembly language programming and the process of creating a bootable floppy image. In summary, the program is a representation of the foundational elements of system programming at the assembly level. It underlines the importance of understanding hardware-level interactions and memory management in software development. The skills and techniques demonstrated in this program are essential for any system programmer or anyone interested in the inner workings of computer systems.We have successfully created assembly code that is a basic boot sector program for an x86 system, designed to execute right after a computer starts. Upon execution, it displays a menu for the user to choose from three tasks: writing a string to disk, reading from disk, or writing prepared RAM data to disk. The code handles user input, performs the selected disk operations using BIOS interrupts, and includes routines for displaying messages and reading strings from the keyboard.

**References:**

1. "Assembly Language for x86 Processors" by Kip R. Irvine
2. VirtualBox User Manual
3. NASM Documentation