

Hangman Game in x86 Assembly

Mraz, Ryan M
Lane Department of Computer Science
West Virginia University
Morgantown, West Virginia
rmm0049@mix.wvu.edu

Kudlak, Kryzstof J
Lane Department of Computer Science
West Virginia University
Morgantown, West Virginia
kjk0027@mix.wvu.edu

Griffith, Drew A
Lane Department of Computer Science
West Virginia University
Morgantown, West Virginia
dag0032@mix.wvu.edu

Labowski, Kollin K
Lane Department of Computer Science
West Virginia University
Morgantown, West Virginia
kkl0009@mix.wvu.edu

Abstract—*The main task described in this proposal is to create a game of hangman using assembly language and MS DOS. The project is to be completed through several regular group meetings and frequent communication between group members. It will allow the player of the game to attempt to guess a word/phrase selected by the program from a database of words/phrases. This is to be done by guessing letters one at a time, with a penalty imposed for letters that are guessed incorrectly. If the player is able to guess enough letters to complete the word/phrase without guessing too many incorrect letters, they will win the game and have the option to play again. The game will be complete with various graphics to make it more aesthetically appealing.*

I. OVERVIEW

For this project, our group set out to program a simple hangman game in x86 assembly language for MS DOS. This goal was based on the sole requirement that we were to “make a project” that “demonstrate[s] a level of complexity and utilize[s] DOS interrupts covered in lab.” The game features ASCII-based graphics, rudimentary sound effects, responsive reactions to user input, difficulty modes, and a wordlist consisting of 100 words.

Upon startup, the player is greeted with a startup screen and followed by a main menu screen. The player then has the option to start a new game along with choosing a mode of difficulty of easy, medium, or hard. The mode of difficulty changes the player’s number of chances to guess at the mystery word. During gameplay, the player has a view of all letters of the alphabet along with a drawing of the hangman based on how many lives they have left. Words are chosen at random from a file and the blanks of the word are shown to the player. Letters disappear when they have been used up and the player is greeted with a message when they have guessed correctly, or they need to try again. Upon failure to guess the correct word, the word is displayed to the player with uppercase letters with which the player guessed correctly and lowercase letters of those that they did not manage to guess. If the player guesses all of the letters to the word, they are greeted with a success message. Either way, the player then has the option to return to the main menu and start a new game or end the application.

II. DESIGN CONSTRAINTS AND ASSUMPTIONS

A. File System Structure

The structure of the file system was chosen so that there was one main file with all other files being included inside of this file in order to organize things better. In assembly however, when a file is included, it is as if the

code from that file was directly pasted at that point when it was included. This meant that it was necessary to pay attention to when things were referenced so as to not cause an error by referencing something that was not defined yet. The other important files that were initially setup were the functions file, which housed important reusable functions, the startup file which helped display the startup sequence, the main menu file which helped handle the main menu option screens, and finally the game file which housed the game loop. The functions file contained functions such as `clear_screen`, `move_cursor`, `wait`, `keyboard_draw`, and `hangman_draw`. These operations were made into reusable functions because of the frequency in which they were needed within the game. The `wait` function allowed for any type of delay that was needed in the game which was useful when causing a delay in the startup sequence. The `clear_screen` function is used throughout the game anytime, things needed to be redrawn to the screen or it switched to a different screen, such as back to the main menu. The `move_cursor` function made it easy to set registers CX and DX with coordinates to which the cursor needed to be moved to and made the call to `move_cursor`. This was useful when positioning where text or ASCII graphics needed to be displayed. The `keyboard_draw` function helped redraw the updated keyboard of the alphabet during the game and the `hangman_draw` helped update the hangman ASCII graphic to match how many lives the player had left. There were some other common tasks such as printing a string or printing an entire header text that could have been made into functions in order to reduce the amount of redundant code and streamlined things better. This was realized later into the process.

A design flaw that was not realized as the game was being developed, was not considering how far the program had to jump to a subroutine from its current position. The problem arose of jumping out of range because the subroutine that it was being called to, or a jump instruction, was outside of the offset. This resulted in moving certain subroutines around or making the code shorter by making it more efficient. By doing this, it resolved some of those issues with jumping outside of the range, but there were some instances in which it could not be avoided.

III. THEORETICAL EXPLANATIONS

A. Interrupts in DOS

DOS interrupts are special functions provided for use in a DOS environment that allow programmers to perform various functions. Perhaps the most versatile and widely used of the DOS interrupts is DOS interrupt 21h. This interrupt provides the programmer with the ability to read information from the user or a file and write that information to the console. The interrupt has many different options, several of which were used while programming the hangman game. The programmer may choose which option within an interrupt they would like to use typically by setting the value in the AH register to a value corresponding with the desired option. Option 01h could be used to read in input characters from the user for the hangman game, and option 02h could be used to display that character to the screen. Option 09h, which prints out a full string of characters until it reached a terminal dollar sign (\$) character, was used to display the word every time the console was reloaded. In order to read in different strings from a file each time the game is run, options 40h, 42h, 3Eh and 3Fh would be used to open the file, read from it or write to it, and close the file. Another useful interrupt is interrupt 10h, which allows for video settings to be changed and other related tasks to be completed. Option 00h of this interrupt allows the video mode to be set, and options 01h and 02h allow the position of the cursor to be changed. These options would be helpful for ensuring that graphics that are printed to the screen appear in the desired location. Another interrupt useful to the hangman project would be interrupt 15h, particularly option 86h, which causes the console to wait. This would be particularly useful for ensuring that sound effects are able to be played out in their entirety and are not cut off. One more useful interrupt was interrupt 1Ah, an interrupt which deals with setting and getting the current system time. This was important because the current system time is commonly used when creating pseudo-random algorithms, which would be useful for this project to ensure that words appear to be chosen at a random rate. Information gathered about these interrupts comes from a full manual from Laval University [1].

B. Graphics in DOS

Graphics in DOS are commonly generated using DOS interrupt 21h as previously discussed. For the hangman game, most graphics could be generated using ASCII characters, which was previously stated to be printed out by using options 02h (for a single character) or 09h (for a full string). One way to print out a full image which was created using ASCII characters would be to place each line of the image into a string, then use interrupt 21h to print out the image one line at a time, using a loop. This could be used to print out the actual image of the hangman, and it would allow the player to determine how many more guesses they will have before they lose the game. Additionally, interrupt 10h would also need to be used to set the video mode to graphical mode. This would allow characters that are printed to the screen to appear as they are intended to, and in the right place. Setting the cursor using the options discussed previously for interrupt 10h would be important in ensuring that the layout of the screen is clean and not grouped together. The cursor position is set and changed using

options 01h and 02h of interrupt 10h as previously stated, specifically by setting the specific (x,y) coordinate of the position that the graphics are desired to appear. In addition to ensuring the main game is visually appealing, using this interrupt would also allow the menu to appear clean and easy to navigate.

C. Persistent Data

In order for a game of hangman to be played, there must be a word that is to be guessed. In this implementation of the game, it was decided to use a word bank that could be pulled from at random to select a word to be guessed. For an effective implementation of this word bank, a persistent data structure needed to be used. In particular, a random-access TXT file was selected for use called "words.txt."

1	6170	706c	6524	0020	2020	2020	2020	2020
2	6261	6e61	6e61	2400	2020	2020	2020	2020
3	6772	6170	6524	0020	2020	2020	2020	2020
4	636f	6d70	7574	6572	2400	2020	2020	2020
5	7363	6965	6e63	6524	0020	2020	2020	2020
6	656e	6769	6e65	6572	696e	6724	0020	2020
7	6772	6f75	7024	0020	2020	2020	2020	2020
8	7072	6f6a	6563	7424	0020	2020	2020	2020

Figure 1. Raw data inside words.txt

At present, this file stores 100 words that are 16 bytes wide. Each word is effectively an array of ASCII characters ending with a "\$" which is used as a special sentinel to signal the end of a word. Any remaining space in the 16-byte word is character "20h," the SPACE character.

In the program code for this project, "rw.asm" is the file that handles any reads from or writes to files. This program uses functions 3Dh, 3Eh, 3Fh, 40h, and 42h in the interrupt 21h library of functions. This interrupt library makes use of file handles. When a file is opened using function 3Dh of interrupt 21h, a file handle is returned. The file handle is simply a number which addresses an open file, and it is required when using the interrupts to read, write, and close the file.

A random-access file, as opposed to a sequential access file, is a file whose data can be accessed at any point in the same amount of time without needing to consume any data ahead of it. Random access files effectively work like arrays in higher level programming languages. All data contained in a random-access file has an address (analogous to an index) and can be accessed in the same amount of time as all other data in the file. The alternative to random access files are sequential access files which can only read data in sequential order.

In order to use the words.txt file as a random-access file, the file pointer was taken advantage of. A file pointer is an address which points to a location in a file. When interpreting data from a file, the file pointer is used as a reference for where to start reading or writing data. In the case of this program, the pointer is repositioned using interrupt 42h to point to the address of the beginning of the word to read when performing a read operation. For performing write operations, the file pointer was simply

moved to the end of the file. The file pointer determines where data will be written and where it will be read from.

IV. DESIGN CRITERIA, SAMPLE CALCULATIONS, AND SIMULATIONS

A. User Interface

The program provides a very simple interface for the player to interact with. The initial screen shown to the player is a brief title screen depicted in Figure 2 below. The hangman graphic is an example of a fully drawn hangman used in the game.

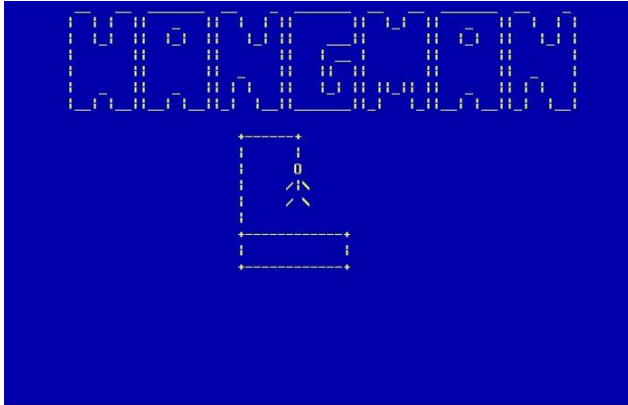


Figure 2. Initial title screen

```
_startup_screen:
;backup registers
push ax
push bx
push cx
push dx

;sets screen to blue color
push dx
mov ah, 06h
xor al, al
xor cx, cx
mov dx, 184fh
mov bh, 1eh
int 10h
pop dx

;display startup header
mov X_COR, (COLS - header_len) / 2
mov Y_COR, 0

mov ah, 09h
lea dx, header

print_header:
call move_cursor
int 21h
inc Y_COR
add dx, header_len
cmp Y_COR, header_height
jne print_header

; play startup sound
call startup_sound

; waits 5 seconds
call wait1
call wait1
call wait1
call wait1
call wait1

;return registers
pop dx
pop cx
pop bx
pop ax

ret
```

Figure 3: Initial title screen code

There are multiple steps that are taken to create the blue title screen that are contained in startup.asm (Figure 3). First, the code sets the background color to blue using the corresponding int 10h interrupt.

Then the “Hangman” large text graphic is printed line by line from the corresponding string array with the use of a loop with the print_header label.

Finally, both a sound is played as well as 5 calls to the wait1 procedure that waits for 1 second each.

This screen only persists for a few seconds, then transitions to the main menu screen, where the player can choose to either input 1 to start a new game, or 2 to exit the program. This screen is also shown in Figure 4 below.



Figure 4. Main menu screen

```
;changes color of screen
push dx
mov ah, 06h
xor al, al
xor cx, cx
mov dx, 184fh
mov bh, 0eh ; yellow text on black
int 10h
pop dx

; print main menu header
mov X_COR, (COLS - menu_len) / 2
mov Y_COR, 0

mov ah, 09h
lea dx, menu_header ; "lea" load effective address

print_menu_header: ; draw loop start
call move_cursor ; set the cursor to point (X_COR, Y_COR)
int 21h ; interrupt 21 function 09, prints the string pointed to by DS:DX (the first line of the main menu header)
inc Y_COR ; Increment the y coordinate (drop a line)
add dx, menu_len ; change where dx is pointing (we need to progress to the next string to print)
cmp Y_COR, menu_height ; check and see if we have reached the end
jne print_menu_header ; if we aren't to the end yet, jump back to the start of the loop

; print menu options
mov Y_COR, (ROWS / 2) ; this appears to set the y coordinate to the middle of the screen on the y axis
call move_cursor ; move the cursor there
lea dx, option1 ; load the effective address of option 1
int 21h ; output option 1

inc Y_COR ; basically the same shit except we drop 2 lines below & output option 2
call move_cursor
lea dx, option2
int 21h

; wait for user input
option_select:
mov ah, 07h ; character input no echo
int 21h
cmp al, '1'
je option_exit
cmp al, '2'
je option_game
jmp option_select

option_exit:
;return registers
pop dx
pop cx
pop bx
pop ax
call exit
```

Figure 5. Main menu screen code

The steps taken to display the screen in Figure 4 are very similar to displaying the initial startup screen seen in Figure 3. The background and header are virtually identical to that seen in Figure 3, with a few parameter changes. The menu options are displayed in the same way as

well, but without the need to loop since they are a single line each.

Where this section differs is that the program stalls for the user to give input before continuing with execution.

If the player chooses the option to quit the game, the program simply exits. However, if the player chooses to start a new game, they are then prompted to choose 1 of the 3 provided difficulty settings.



Figure 6. Difficulty selection screen

```
option_game: this is the secondary screen with difficulty options
call clear_screen
mov X_COR, (COLS - menu_len) / 2
mov Y_COR, 0

mov ah, 09h
lea dx, menu_header

print_menu_header2: ; outputs all the secondary menu junk basically the same way it's done everywhere above
call move_cursor
int 21h
inc Y_COR
add dx, menu_len
cmp Y_COR, menu_height
jnc print_menu_header2

mov Y_COR, (ROWS / 2)
call move_cursor
lea dx, diff1
int 21h

inc Y_COR
inc Y_COR
call move_cursor
lea dx, diff2
int 21h

inc Y_COR
inc Y_COR
call move_cursor
lea dx, diff3
int 21h

diff_select: ; read user input for difficulty
mov ah, 0Ah ; character input no echo
int 21h
cmp al, '1'
je easy_game
cmp al, '2'
je med_game
cmp al, '3'
je hard_game
jmp diff_select

; Need a way to store the difficulty selected into the DIFF variable
; I think the problem is that "equ" effectively makes DIFF a constant (line 17 main)
; maybe try initializing it the same way you did with X_COR and Y_COR
easy_game:
; mov bl, al
; mov DIFF, bl
mov CURR_LIVES, 10
jmp end_menu

med_game:
; mov bl, al
; mov DIFF, bl
mov CURR_LIVES, 7
jmp end_menu

hard_game:
; mov bl, al
; mov DIFF, bl
mov CURR_LIVES, 5
jmp end_menu

end_menu:
iretd
pop dx
pop cx
pop bx
pop ax
call main_game
ret
```

Figure 7. Difficulty selection screen code

Both the header, and difficulty options are displayed using the same operations as before. Where this section differs is the logic that occurs when selecting a difficulty. The CURR_LIVES variable is initialized with a different number depending on which option is chosen.

Regardless of which difficulty option is chosen by the player, the game will transition to the primary game state shown in Figure 8.

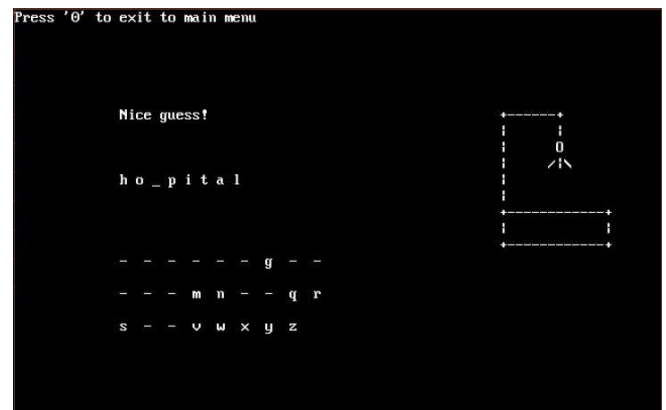


Figure 8. Active game screen

This screen outputs graphics using the same functions as before, but with added logical components.

```
keyboard_draw:
mov X_COR, COLS/6
mov Y_COR, ROWS/2 + 3
call move_cursor

mov ah, 02h
mov cx, 0
mov dl, 'a'
kb_draw_loop:
mov al, dl
mov dl, ' '
CALL check_if_used
cmp ah, 11h
je omg_it_got_used

mov dl, al
mov ax, 0200h
int 21h ; output the letter
jmp cont

omg_it_got_used:
mov dl, al
mov ax, 0200h
push dx
mov dx, '-'
int 21h
pop dx

cont:
inc dl
inc cx

; append spacing
push dx
mov dl, 32
int 21h
int 21h
pop dx

; drop a line every 9 letters
cmp cx, 9
jnz skip
mov cx, 0
add Y_COR, 2
mov X_COR, COLS/6
call move_cursor
skip:

; increment to next letter
cmp dl, 'z' + 1
jne kb_draw_loop

RET
```

Figure 9. Keyboard draw code

The code in Figure 9 iterates from a-z and outputs either a dash character or a letter based on if the letter has been used or not. It also has a counting variable that counts to 9, then drops the line down and resets the column to the leftmost point.

```

hangman_draw:
    ; draw a hangman based on a counter variable CURR_LIVES
    mov ax, 0ah

    ; check CURR_LIVES and go to the matching label
    cmp CURR_LIVES, 0      ; 0 lives
    je draw_hangman_0
    cmp CURR_LIVES, 1      ; 1 life
    je draw_hangman_1
    cmp CURR_LIVES, 2      ; 2 lives
    je draw_hangman_2
    cmp CURR_LIVES, 3      ; 3 lives
    je draw_hangman_3
    cmp CURR_LIVES, 4      ; 4 lives
    je draw_hangman_4
    cmp CURR_LIVES, 5      ; 5 lives
    je draw_hangman_5
    cmp CURR_LIVES, 6      ; 6 lives
    je draw_hangman_6
    cmp CURR_LIVES, 7      ; 7 lives
    je draw_hangman_7
    cmp CURR_LIVES, 8      ; 8 lives
    je draw_hangman_8
    cmp CURR_LIVES, 9      ; 9 lives
    je draw_hangman_9
    cmp CURR_LIVES, 10     ; 10 lives
    je draw_hangman_10
    jmp draw_hangman_0     ; else go to 0 lives as default

    ; loads the matching effective address, 'Yes' - load effective address
draw_hangman_0:
    lea dx, HANGMAN_LIVES_00
    jmp end_lives_cond
draw_hangman_1:
    lea dx, HANGMAN_LIVES_01
    jmp end_lives_cond
draw_hangman_2:
    lea dx, HANGMAN_LIVES_02
    jmp end_lives_cond
draw_hangman_3:
    lea dx, HANGMAN_LIVES_03
    jmp end_lives_cond
draw_hangman_4:
    lea dx, HANGMAN_LIVES_04
    jmp end_lives_cond
draw_hangman_5:
    lea dx, HANGMAN_LIVES_05
    jmp end_lives_cond
draw_hangman_6:
    lea dx, HANGMAN_LIVES_06
    jmp end_lives_cond
draw_hangman_7:
    lea dx, HANGMAN_LIVES_07
    jmp end_lives_cond
draw_hangman_8:
    lea dx, HANGMAN_LIVES_08
    jmp end_lives_cond
draw_hangman_9:
    lea dx, HANGMAN_LIVES_09
    jmp end_lives_cond
draw_hangman_10:
    lea dx, HANGMAN_LIVES_10
    jmp end_lives_cond

end_lives_cond:
    mov X_CUR, 34COLS/4
    mov Y_CUR, 8ROWS/4

print_msg:
    ; draw loop start
    call move_cursor
    int 21h
    ; interrupt 21, function 09, prints the string pointed to by DS:00 (the first line of the main
new_line:
    ; increased the y coordinate (drop a line)
    add dx, HANGMAN_WIDTH ; change where dx is pointing (we need to progress to the next string to print)
    cmp Y_CUR, HANGMAN_HEIGHT + 10000 ; check and see if we have reached the end
    je print_msg ; if we aren't to the end yet, jump back to the start of the loop
    ret

```

Figure 10. Hangman graphic code

Another added logical component to the main game screen seen in Figure 8, is the hangman picture on the right side of the screen. The first block of cmp-je combos simply jumps to a corresponding draw hangman_# label that handles the loading of the address into the dx register. Once the correct resource is loaded into memory, the code that prints the string array is virtually the same as in previous examples.

This interface includes many graphical components to convey to the player what the state of the current game is.

The progress toward completing the word is shown as the list of characters in the left-middle of the screen. Characters shown by an underscore character (“_”) are letters yet to be guessed by the player. If there are any other characters in place, then they have been successfully guessed and are being shown where they appear in the word.

The graphic below the word uses inverse of that logic, where a dash (“-”) indicates a letter that has already been guessed, and any other character has not yet been guessed by the player.

The hangman graphic on the right of the screen displays a visual representation of the amount of lives the player has left. When a player guesses incorrectly, their number of lives goes down, and the graphic gets updated accordingly.

Above the word is a comment made in response to the move of the player. If a player’s guess is present in the word but not yet unveiled, the comment is “Nice

guess!”. If the player guesses a character that is not present in the word, the comment is “Nope, try again!”. If the player repeats a guessed character at any time during the game, the comment is “You already guessed that...”. When the player wins a game, the comment is “Congrats, you win!”. Finally, if the player loses a game, the comment displayed is “You lose! The word was: ”, followed by the actual word such that capital letters in the string represent correctly guessed letters, and all unguessed characters are lower case.

B. Parsing Keyboard Inputs

An important functionality the program had to support was the ability to read character inputs from the user, and then change the state of the game based on this input. Whenever a user inputs a character to the game (read using DOS interrupt 21h), the character will be read by the program to determine whether it can be found in the current hidden word. If the input letter is part of the secret word, then every instance of the letter within the word will be filled in and a message to the screen will be displayed to show that the guess was correct. In the case that all letters in the word have been filled in, display a win message to the screen and allow the player to return to the menu. If the input letter instead is not a word, the player will lose a life, and an additional part of the hangman picture will be output to the screen. In the case that losing a life causes the player to have zero remaining lives, then a message will be displayed to the screen to tell the player they have lost, and they will be able to return to the menu. See the flowchart below in Figure 11 to see the decision-making process for parsing character inputs.

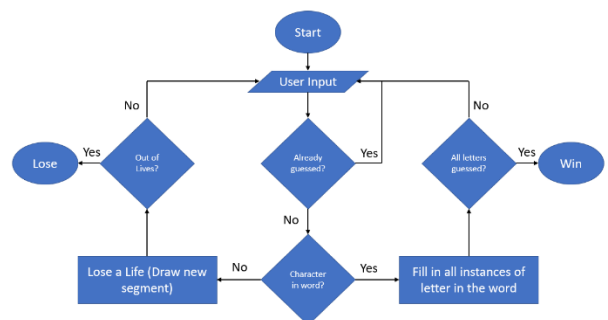


Figure 11. Parsing Character Inputs for the Main Game

As can be seen in the flowchart, the first main conditional following the user input was the conditional to determine whether the input letter has been guessed already. This is important because it will prevent the player from being penalized for guessing a previously guessed letter unintentionally. This was essentially done by allocating 4 bytes worth of space and dedicating it to keeping track of which letters have already been guessed. This was done by assigning 26 of the bits within the 4-byte space to correspond to the 26 characters of the alphabet. Every time the user guesses a letter, the bit corresponding to that particular letter is checked against the database of letters using AND logic

by checking if the solution of the database AND the bit for the input letter is nonzero, indicating the letters presence in the database. If the letter is indeed present in the database, then the appropriate action is to output a message notifying the user of this, then wait for a new user input as shown in the flowchart. If the letter is not in the database, this means the letter was not guessed yet, and so it will be parsed as a regular input by continuing through the flow of the algorithm. Additionally, in the case it has not yet been guessed, the letter must be added to the database. This was accomplished by using OR logic to set the database to itself OR the bit for the current letter. Performing this simple function would let the program keep track of the most recently guessed letter for later guesses. See Figure 12 below to see how the AND command was used to check for the presence of a letter within the file, and Figure 13 to see how the database of the letters was updated.

```

end_check:

    call two_to_n
    and cl, bl
    cmp cl, 00h
    je not_used
    jmp used

```

Figure 12. Using the AND function to check whether the input letter is in the database of guessed characters

```

used_zero:
    or [si], bl
    jmp end_of_method

used_one:
    or [si + 01h], bl
    jmp end_of_method

used_two:
    or [si + 02h], bl
    jmp end_of_method

used_three:
    or [si + 03h], bl
    jmp end_of_method

```

Figure 13. Using the OR function to update the database in the event that an input character has not already been guessed by the player

Assuming that an input character was not already guessed, the next step was to determine whether the input character could be found within the secret word.

This was done by looping through the secret word from the beginning of the word until the dollar sign (\$) which indicated the end of the word. For each character within the word, it would determine whether that character was equal to the character that was input by the user, and if it were, it would replace that lowercase letter with its uppercase counterpart. This was done as a way of keeping track of characters that were already guessed while retaining the meaning of the original word. This would also be an important mechanic for determining when the game has been won. See Figure 14 below to see how the loop was implemented in assembly code for the project.

```

check_in_word: ; INPUT PARAMETERS => (character input -> al)

    push bx
    push cx
    push dx

    mov di, offset readWord

    mov ah, 00h

loop_in_word:

    mov cl, [di]          ; Read in next character of the word
    cmp cl, "$"           ; Exit the loop if terminal character
    je exit_word_loop

    cmp cl, al            ; If character is same as the input
    je loop_positive
    jmp increment_di

loop_positive:

    mov cl, [di]
    sub cl, 20h           ; Change the lowercase letter to uppercase
    mov [di], cl
    mov ah, 11h           ; Move 11h to AH, used as an output to let
                        ; the program know an instance has been found

increment_di:

    inc di

    jmp loop_in_word

exit_word_loop:

    pop dx
    pop cx
    pop bx

    ret

```

Figure 14. The subroutine used to check whether the input character is part of the word, and to set any instances of the character within the word to their uppercase equivalents

After checking whether the letter appears in the secret word, the next thing to do is to determine whether the user has won the game in the event that the letter was found in the word, or whether they have lost the game if the letter was not found. Checking whether the user has won the game was made much simpler with the uppercase characters. Checking if the user has won was as simple as looping through the word and checking if each character was uppercase. If all characters were uppercase, then the user has guessed all letters and has won the game. An appropriate message could then be displayed to the output. See Figure 15 below to see how the program determines when the win condition was met.

```

check_if_win:
    push bx
    push cx
    push dx

    mov di, offset readWord
    mov ah, 00h

check_loop:
    mov cl, [di] ; Read in the next character of the word
    cmp cl, "$" ; Exit on the terminal character
    je check_win

    cmp cl, "z" ; Check if the letter is lowercase
    ja exit_check_loop ; If it is, exit the loop

    inc di

    jmp check_loop

check_win:
    mov ah, 11h ; Win condition is set

exit_check_loop:
    pop dx
    pop cx
    pop bx

    ret

```

Figure 15. The subroutine to determine whether the player has won the game

In the case that the input letter was not found within the word, a variable used to keep track of how many lives the player has will be decremented. If this causes the player to have 0 lives, the user has lost the game and an appropriate message will be displayed to the screen.

C. Random Access File

As previously mentioned, a random-access file was used to store the list of words to be used in the game. This file was maintained and accessed using read/write methods contained in rw.asm.

The write function has four major steps. First, function 3Dh of interrupt 21h is used with a parameter to open the file in read mode. For this to work, the address of the file's name is stored in memory at DS:DX. After the interrupt routine, register AX stores the handle for the opened file. This handle is then transferred to register BX because the rest of the functions require it to be there. Next, the file pointer is moved to the end of the file using function 42h of interrupt 21h with the parameter "02h" which positions the file pointer at the end of the file. After this, DS:DX is loaded to contain the address of the word to write to the file, and CX is loaded to contain the number of bytes to be written. Interrupt 21h function 40h is called, and the word is written to the file. Finally, interrupt 21h function 3Eh is used to close the file.

```

; open file and prepare for write
mov ax, 3D01h ; open file as u
int 21h
mov bx, ax ; move the file han

; move file pointer to write at
mov ax, 4202h ; moves the file
mov cx, 0000h ; cx:dx is the o
mov dx, 0000h
int 21h

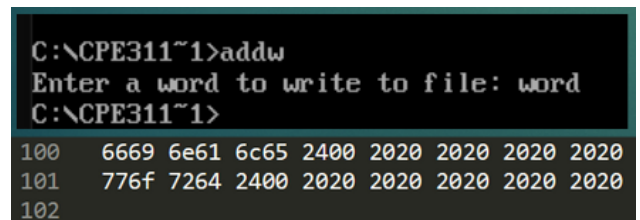
; write to the f-ing file final
mov ax, 4000h ; write mode
mov cx, 16 ; number of bytes t
mov dx, offset writeWord ; it
int 21h

; close the file
mov ax, 3E00h
int 21h

```

Figure 16. Write Function

The write function is used in a program called "addw.asm." This program is the sole way that words are added to words.txt. The program presents the user with a prompt asking them to enter a word, then when the user presses the tab key, the word is added to the file.



100	6669	6e61	6c65	2400	2020	2020	2020	2020	2020
101	776f	7264	2400	2020	2020	2020	2020	2020	2020
102									

Figure 17. addw.asm functionality

The read function is a bit more complex than the write function. First, the read function opens the file into read mode using interrupt 21, function 3Dh, with the parameter 00h. The file handle is then stored in register BX. When the function is called, it is expected that register DX will contain the index of the word to be read from the file. On return, the variable "readWord" will store the word read in from the file. Because DX contains the address of the word to read and because each word is 16 bytes wide, DX needs to be multiplied by 16 or 2^4 . This can be done easily by shifting the bits of DX left 4 times because each time a left shift operation occurs, the number is multiplied by 2. With DX pointing to the appropriate address in the file, the file pointer is moved to that position using interrupt 21h function 42h. Next, 16 bytes are read from the file and stored in the readWord variable using interrupt 21h function 3Fh. Finally, the file is closed with interrupt 21h, function 3Eh.

```

; open file and prepare for r
mov ax, 3D00h ; open file a
int 21h
mov bx, ax ; move the file

pop cx ; cx contains the se
pop dx ; dx contains the in
push dx
push cx

; ***NOTE*** come back later
; depending how many words we
shl dx, 1 ; multiply dx by
shl dx, 1 ; for some reason
shl dx, 1
shl dx, 1

; move file pointer to read t
mov ax, 4200h ; moves the t
mov cx, 0000h ; cx:dx is th
int 21h

; read from the file
mov ax, 3F00h ; read mode
mov cx, 16 ; number of byte
mov dx, offset readword ;
int 21h

; close the file
mov ax, 3E00h
int 21h

```

Figure 18. The read function

V. FUTURE WORK

There is a lot that could be done both to optimize the produced assembly code and to expand on this project in the future.

As for optimization, there are plenty of areas in the code base where things are poorly optimized and written. One particular example is the “addw” program

that was written to add words to the wordlist.txt file. This method is very finicky and unintuitive to use. Pressing any key other than a letter will cause it to mess up, and one has to re-execute the program for each word that is added rather than being able to add multiple words at a time. Additionally, the size of the wordlist is hard coded into the program code. Ideally, the size of the wordlist would be stored in some sort of metadata file and updated any time the wordlist grows. This size could then be read in by code in production and used to track the number of words contained in the list.

Another area of inefficiency belongs in the “charin” library within the project. This library takes advantage of two custom built functions: two_to_n and simple_mod. These functions are intended to perform mathematical operations on the contents of certain registers. Both functions, however, effectively hard code the solutions to any possible case rather than solving them mathematically leading to the inability to produce robust code and an abuse of space. Other potential optimizations exist in addition to what has already been mentioned; these are just the two most egregious examples.

The game is left wide open for plenty of expansion. In particular, more words could be added to the wordlist, the graphics could be overhauled, the sound quality could be improved, high scores and leaderboards could be implemented, longer words and phrases could be permitted, and data about players could be stored. There is plenty more that could be done than just those few ideas, but for now the project is adequate to satisfy the requirements set for it in the beginning of the course.

REFERENCES

- [1] “DOS Interrupts.” Laval University, Quebec City, Canada.

TEAMMATE EVALUATION

Team Member	<i>Contributions</i>	<i>Grade</i>
Drew Griffith		
Kollin Labowski		
Ryan Mraz		
Kryzstof Kudlak		