Luke Cutter

Professor Hall

CSI-370-01/02

10 December 2024

MASM Morse Code Translator and Telegraphy

I, Introduction:

Beginning this project, I settled on creating an NES game using 6502 Assembly. However, my hopes of accomplishing this were dashed over multiple attempts over weeks to find a working assembler, and finding a good program to create the tile maps and character sprites. I found NESASM, however, it was quite outdated and would not run on my Windows 11 computer because it was made for 32-bit Windows. I found all of the required tools such as Tiler, FCEUX, and even a C-based NES program maker called NES Starter Kit, but could not get them to work in tandem which signaled the need for a shift in my project. To fulfill the final project requirements, I needed to create a new idea that was technically challenging and similar in scope. After careful consideration, I decided to shift to creating a Morse Code translator and telegrapher using the control and speed offered by MASM.

II, What is a Morse Code Translator and Telegrapher?

Morse Code is a system of communication devised by American inventor Samuel F.B. Morse in the 1830s as a means of sending messages over long distances (Britannica). This invention revolutionized communication and quickly, telegraph lines were built alongside rail lines to help the rail industry run more efficiently (Shelburne Museum). The original Morse Code has been adapted and improved many times, eventually settling on a variant called

International Morse Code. International Morse Code differs from the original iteration because it "uses dashes of constant length rather than the variable lengths used in the original Morse Code" (Brittanica). The International Morse Code was incredibly influential in World War II, the Korean War, and the Vietnam War (Brittanica). Until the 1990s, the shipping industry used the International Morse Code to maintain sea safety (Brittanica). At the turn of the century, International Morse Code's usefulness was waning and most countries dropped their requirements for decoding Morse Code to obtain an amateur radio license (Brittanica). Now, on the internet, there are many Morse Code translators where users can input their text and have it translated into Morse Code or vice versa. These translators create the tones associated with the dots and dashes like the telegraph. One such translator is MorseCode.World, which was influential in the testing of the following MASM Morse Code Translator and Telegrapher.

III, Why Was the Morse Code Translator and Telegrapher Chosen?

I chose to work on a Morse Code translator and telegrapher because I have always had an underlying interest in Morse Code. In high school, my robotics team WAR-4533 would demonstrate how to use old and obsolete technologies and have us interact with them. One such technology was Morse Code. I got to go hands-on and use a simple telegraph with another student and send messages back and forth while listening to the tones. This inspired the pivot from the original NES game idea to the current project. It seemed like an interesting challenge and one that I could implement well. On top of this, learning through Stack Overflow that I could use the Windows API Beep function, made me extra excited to make this project as it would have multiple layers and be extra useful as a telegraphy tool.

IV, How The Program Works:

The program is built off the International Morse Code standard table with the letters A-Z, numbers 0-9, and symbols . , ? ! / : @ = and space all being mapped to their Morse Code representations in a simple lookup table in the DATA section of the code. The output buffer is aligned to a 16-byte boundary (ALIGN 16) for the best memory access and is given 1024 bytes of space to ensure there is enough room for conversion. These symbol definitions were found on Moratech, a resource for Morse Code translation.

The Morse Code Translator and Telegrapher use the Windows API call to properly handle program exiting, file creation, file writing and closing, and tone creation through the beep function. For file operations, GENERIC_WRITE is set to 40000000h to enable write-only access to the output file. Due to this, it is initialized by reserving 40 bytes of shadow space for these API calls. In the Windows API constants, I specified CREATE_ALWAYS EQU 2 so that the program always creates a new file and overwrites the old file. The FILE_ATTRIBUTE_NORMAL is set to 80h. This program takes an input string (message) and processes the characters (in process_loop). It sends them all to uppercase through subtraction of 32 unless the character is above z in the ASCII alphabet. The characters are passed to check_char and through jump if equal comparisons it takes the numbers, letters, and symbols and sends their data to the character handlers. The character handlers load the address of the Morse Code for the character and pass it to process_morse. Process_morse gets the character from the pattern, checks if it is properly null-terminated, and copies it to the output buffer position. It then checks if the character is a dot or a dash. If it is a dot the registers are preserved and the Beep function is called from the Windows API and plays an 700 hz for 110 ms tone followed by a 110 ms pause. For dashes, it plays at the same frequency but following the International Morse Code standard,

it plays for three times as long, 330 ms. This is followed by another 110 ms pause after each dash because the pause between elements is essential in keeping the timing of the Morse Code consistent and clear. After playing the tones, the registers are restored and move on to the next Morse pattern.

Once a null-terminator is found in the pattern it indicates the end of the character's code and the program jumps back to the process_loop and handles the next character in the input string. When process_loop finds the null terminator of the input string, it jumps to write_file.

The write_file section uses the WriteFile API call from Windows and writes the contents of the output buffer to the file specified in the outFile. It then passes the file handle, buffer address, the total number of bytes to write, and a variable to store the number of bytes written. After completing the writing, the file handle is closed using the CloseHandle API call.

Finally, the program gets a handle to the standard output using the GetStdHandle API and exits with code 0 from the xor rcx, rcx call. Throughout the program, registers are used as efficiently as possible with RSI serving as the source string pointer, RDI as the destination buffer index, and RBX holding the address of the current Morse pattern. Unknown characters are ignored efficiently by being skipped in the check_char section and spaces are converted into "/" for proper Morse code word separation standards.

V, Challenges and Solutions:

The biggest challenge for me was the actual processing logic for manipulating characters and shifting them to their uppercase representations. Also, using the Beep API functionality was a major challenge that took a lot of looking through documentation to resolve weird issues I had with shadow space not being enough to support the calls. I got many errors involving "ADDR32

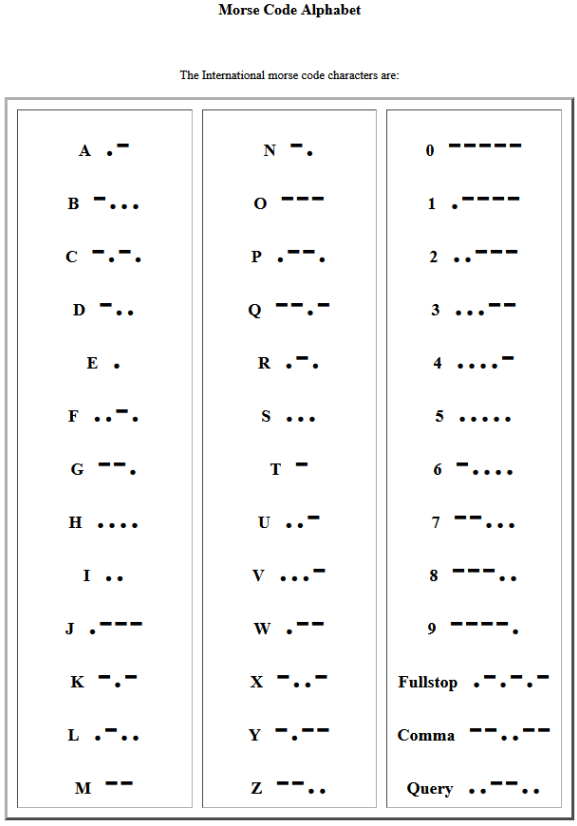
relocation invalid without /LARGEADDRESSAWARE:NO (VS)" which had to do with large addresses not being denied in the settings of the Linker in Visual Studio. Once I set Enable Large Addresses in Properties >> Linker >> System >> to NO (/LARGEADDRESSAWARE: NO), the issues were resolved.

Another major challenge was managing the proper timing between the Morse code elements. The International Morse Code standard requires specific timing between elements and this requires careful consideration of timing ratios through the Beep API function. Another issue was buffer management so the program would not overflow the 1024-byte output buffer. I had to carefully track the buffer position through the RDI register and ensure all the writes stayed within the bounds.

Register preservation around the Windows API calls proved very challenging. Each API call required proper shadow space calls and preservation of register values using push and pop. If the registers were not preserved, the program would lose track of its position in the input string and output buffer leading to corrupted Morse Code outputs.

Lastly, the lookup table was straightforward however, careful attention to detail was essential because each definition had to have the precise amount of dots and dashes and a space following the definition. A null-terminating was also needed at the end of the string to ensure the characters were properly separated in the output.

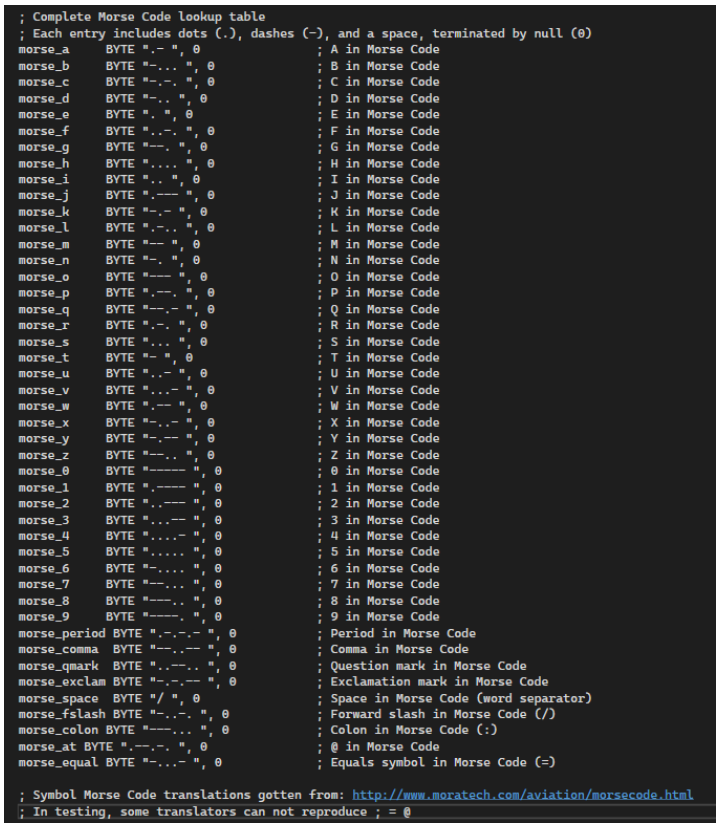
VI, Visualizations:



Morse Code Alphabet

The International morse code characters are:

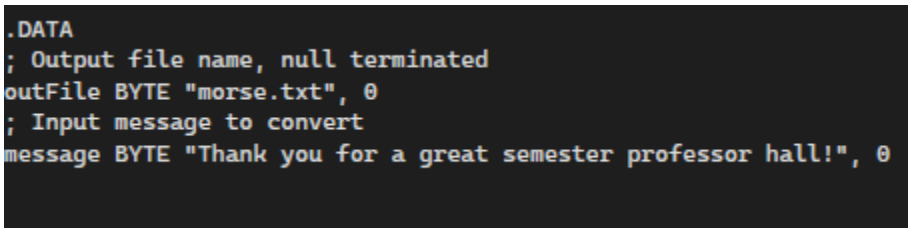
A .-	N -.	0 -----
B -...	O ---	1 .----
C -.-.	P .--.	2 ..---
D -..	Q --.-	3 ...--
E .	R .-.	4 ....-
F ..-.	S ...	5 .....
G --.	T -	6 -....
H	U ..-	7 --...
I ..	V ...-	8 ---..
J .---	W .--	9 ----.
K -.-	X -..-	Fullstop .-.-.-
L .-..	Y -.--	Comma --..--
M --	Z --..	Query ..--..



```
; Complete Morse Code lookup table
; Each entry includes dots (.), dashes (-), and a space, terminated by null (0)
morse_a     BYTE ".- ", 0               ; A in Morse Code
morse_b     BYTE "-... ", 0             ; B in Morse Code
morse_c     BYTE "-.-. ", 0             ; C in Morse Code
morse_d     BYTE "-.. ", 0              ; D in Morse Code
morse_e     BYTE ". ", 0                ; E in Morse Code
morse_f     BYTE "..-. ", 0             ; F in Morse Code
morse_g     BYTE "--. ", 0              ; G in Morse Code
morse_h     BYTE ".... ", 0             ; H in Morse Code
morse_i     BYTE ".. ", 0               ; I in Morse Code
morse_j     BYTE ".--- ", 0             ; J in Morse Code
morse_k     BYTE "-.- ", 0              ; K in Morse Code
morse_l     BYTE ".-.. ", 0             ; L in Morse Code
morse_m     BYTE "-- ", 0               ; M in Morse Code
morse_n     BYTE "-. ", 0               ; N in Morse Code
morse_o     BYTE "--- ", 0              ; O in Morse Code
morse_p     BYTE ".--. ", 0             ; P in Morse Code
morse_q     BYTE "--.- ", 0             ; Q in Morse Code
morse_r     BYTE ".-. ", 0              ; R in Morse Code
morse_s     BYTE "... ", 0              ; S in Morse Code
morse_t     BYTE "- ", 0                ; T in Morse Code
morse_u     BYTE "..- ", 0              ; U in Morse Code
morse_v     BYTE "...- ", 0             ; V in Morse Code
morse_w     BYTE ".-- ", 0              ; W in Morse Code
morse_x     BYTE "-..- ", 0             ; X in Morse Code
morse_y     BYTE "-.-- ", 0             ; Y in Morse Code
morse_z     BYTE "--.. ", 0             ; Z in Morse Code
morse_0     BYTE "----- ", 0            ; 0 in Morse Code
morse_1     BYTE ".---- ", 0            ; 1 in Morse Code
morse_2     BYTE "..--- ", 0            ; 2 in Morse Code
morse_3     BYTE "...-- ", 0            ; 3 in Morse Code
morse_4     BYTE "....- ", 0            ; 4 in Morse Code
morse_5     BYTE "..... ", 0            ; 5 in Morse Code
morse_6     BYTE "-.... ", 0            ; 6 in Morse Code
morse_7     BYTE "--... ", 0            ; 7 in Morse Code
morse_8     BYTE "---.. ", 0            ; 8 in Morse Code
morse_9     BYTE "----. ", 0            ; 9 in Morse Code
morse_period BYTE ".-.-.- ", 0          ; Period in Morse Code
morse_comma  BYTE "--..-- ", 0          ; Comma in Morse Code
morse_qmark  BYTE "..--.. ", 0          ; Question mark in Morse Code
morse_exclam BYTE "-.-.-- ", 0          ; Exclamation mark in Morse Code
morse_space  BYTE "/ ", 0               ; Space in Morse Code (word separator)
morse_fslash BYTE "-..-. ", 0           ; Forward slash in Morse Code (/)
morse_colon BYTE "---... ", 0           ; Colon in Morse Code (:)
morse_at BYTE ".--.-. ", 0              ; @ in Morse Code
morse_equal BYTE "-...- ", 0            ; Equals symbol in Morse Code (=)

; Symbol Morse Code translations gotten from: http://www.moratech.com/aviation/morsecode.html
; In testing, some translators can not reproduce ; = @
```

Figure I: Implementation of Morse Code Lookup Table Versus MoraTech Morse Code Alphabet



```
.DATA
; Output file name, null terminated
outFile BYTE "morse.txt", 0
; Input message to convert
message BYTE "Thank you for a great semester professor hall!", 0
```

Figure II: Message Input to be Translated Into Morse Code

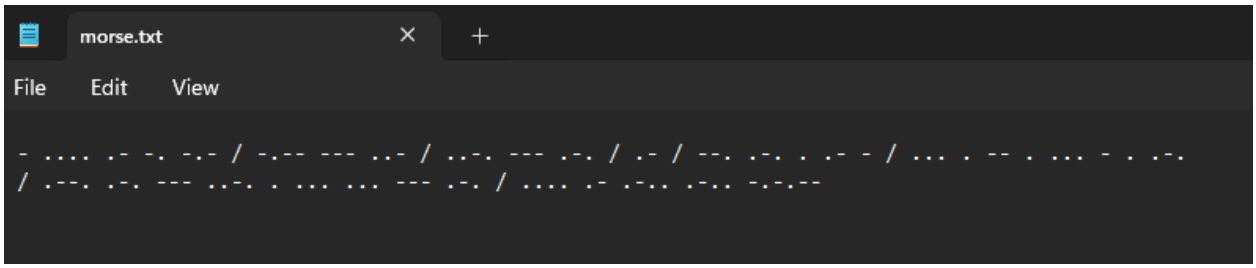


Figure III: Morse Code Output

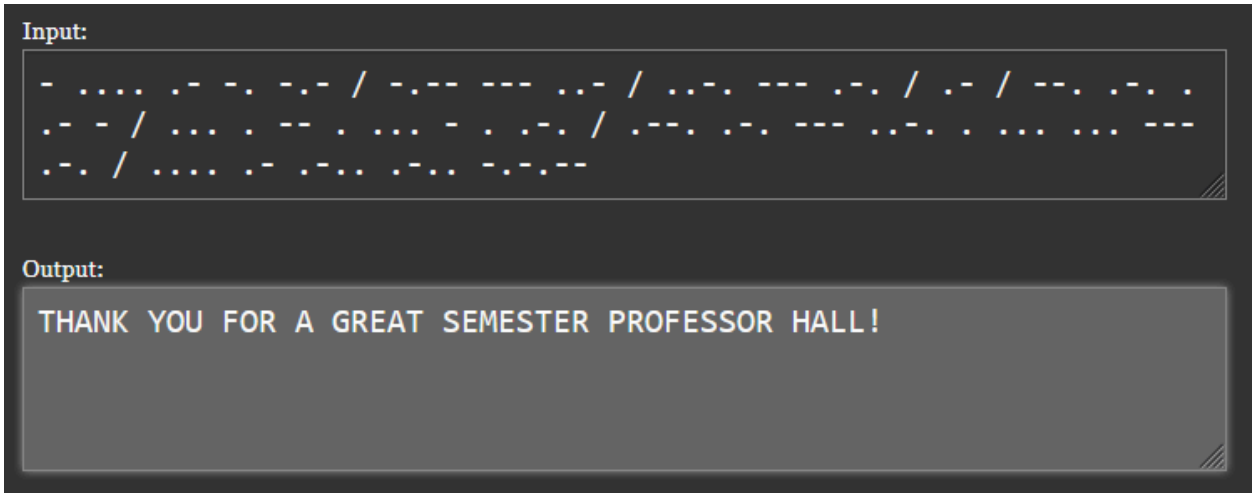


Figure IV: Morse Code Translated Back Into Text on MorseWorld.Code

https://youtube.com/shorts/08wty-zQTI4?feature=share

Figure V: Video of the Code Running with Tones Playing and Morse Output

Works Cited

"Board Index." *ADDR32 Relocation Invalid without /LARGEADDRESSAWARE:NO (VS)*, f.osdev.org/viewtopic.php?t=33500. Accessed 10 Dec. 2024.

GrantMeStrength. "Windows API Index - Win32 Apps." *Win32 Apps | Microsoft Learn*, learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list. Accessed 10 Dec. 2024.

KpuonyerKpuonyer4711 silver badge66 bronze badges, and MichaelMichael 58.4k1111 gold badges8484 silver badges128128 bronze badges. "Windows API Beep Function in NASM Assembly." *Stack Overflow*, 1 Jan. 1959, stackoverflow.com/questions/20569984/windows-api-beep-function-in-nasm-assembly.

"Morse Code Translator." *International Morse Code*, www.moratech.com/aviation/morsecode.html. Accessed 10 Dec. 2024.

"Morse Code Translator." *Morse Code Translator | Morse Code World*, morsecode.world/international/translator.html. Accessed 10 Dec. 2024.

"Morse Code." *Encyclopædia Britannica*, Encyclopædia Britannica, inc., 22 Oct. 2024, www.britannica.com/topic/Morse-Code.

Sending Messages Using Morse Code,

shelburnemuseum.org/wp-content/uploads/2020/05/travel-1-learning.pdf. Accessed 10 Dec. 2024.