INTRODUCTION: ASSEMBLY LANGUAGE X86 TOPICS

Basic Concepts: Applications of assembly language, basic concepts, machine language, and data representation.

x86 Processor Architecture: Basic microcomputer design, instruction execution cycle, x86 processor architecture, Intel64 architecture, x86 memory management, components of a microcomputer, and the input–output system.

Assembly Language Fundamentals: Introduction to assembly language, linking and debugging, and defining constants and variables.

Data Transfers, Addressing, and Arithmetic: Simple data transfer and arithmetic instructions, assemble-link-execute cycle, operators, directives, expressions, JMP and LOOP instructions, and indirect addressing.

Procedures: Linking to an external library, description of the book's link library, stack operations, defining and using procedures, flowcharts, and top-down structured design.

Conditional Processing: Boolean and comparison instructions, conditional jumps and loops, high- level logic structures, and finite-state machines.

Integer Arithmetic: Shift and rotate instructions with useful applications, multiplication and division, extended addition and subtraction, and ASCII and packed decimal arithmetic.

Advanced Procedures: Stack parameters, local variables, advanced PROC and INVOKE directives, and 2/31 recursion.

Strings and Arrays: String primitives, manipulating arrays of characters and integers, two-dimensional arrays, sorting, and searching.

Structures and Macros: Structures, macros, conditional assembly directives, and defining repeat blocks.

MS-Windows Programming: Protected mode memory management concepts, using the Microsoft-Windows API to display text and colors, and dynamic memory allocation.

Floating-Point Processing and Instruction Encoding: Floating-point binary representation and floating-point arithmetic. Learning to program the IA-32 floating-point unit. Under- standing the encoding of IA-32 machine instructions.

High-Level Language Interface: Parameter passing conventions, inline assembly code, and linking assembly language modules to C and C++ programs.

16-Bit MS-DOS Programming: Memory organization, interrupts, function calls, and standard MS- DOS file I/O services.

Disk Fundamentals: Disk storage systems, sectors, clusters, directories, file allocation tables, handling MS-DOS error codes, and drive and directory manipulation.

BIOS-Level Programming: Keyboard input, video text, graphics, and mouse programming.

Expert MS-DOS Programming: Custom-designed segments, runtime program structure, and Interrupt handling. Hardware control using I/O ports.

ASCII Control Characters

These are special non-printable characters triggered by pressing keys like Ctrl + C or Ctrl + G. Instead of displaying text, they send commands — like moving the cursor, making a beep, or telling a printer to start a new page. They're often used for low-level control in screen output or serial communication.

🎓 Explained Like a Pro:

Think of ASCII control characters as the behind-the-scenes crew in a theater — they don't appear on stage (screen), but they cue the lights, move props, and tell actors where to go. In assembly or old-school terminal systems, they manage things like:

- Line breaks (LF = Line Feed)
- Bell sound (BEL)
- Cursor position resets (CR = Carriage Return)

👦 Explained Like You're 15:

Imagine hitting Ctrl + something and your computer doesn't write a letter — instead, it does something weird like beep or go to a new line. That's an ASCII control character doing its thing. It's like giving your keyboard secret signals to control what's going on under the hood.

🧠 ASCII Control Characters: Mnemonics & Labels

🗣️ What's really going on?

Each ASCII control character has:

- A **mnemonic** (short name like BEL, CR, or ESC)
- A **hex code** (like 07h, 0Dh, 1Bh)
- And a **purpose** (e.g., beep the speaker, move to a new line, etc.)

These aren't magic — they're just **keyboard signals** that programmers use to control how text gets handled by the screen, printer, or communication channel.

Find the **ASCII.html** OR **ASCII.png** attached for the full table.

🤖 Why do mnemonics matter?

Think of mnemonics like emoji names. Instead of memorizing what 1Bh does, you just use its nickname: ESC. That way, your code is readable and your brain doesn't fry.

```
mov ah, 0Ah         ; function to read string
mov dx, buffer
int 21h             ; when user presses ESC (1Bh), we can detect it easily
```

🧠 Real-World Analogy:

Mnemonics = street names. Hex codes = GPS coordinates.
Both point to the same spot, but one is easier for humans.

❗ Ctrl-Hyphen Weird Case:

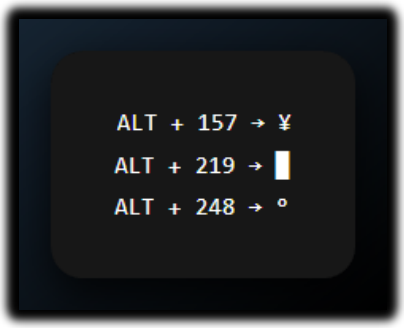
Yup, the combo Ctrl + - gives ASCII 1Fh. It's one of those lesser-known control characters and **it *is* valid.** You won't see it often, but it's real and in the official ASCII spec.

🔤 ALT Key Combinations

🎹 What they do:

Holding down the ALT key and pressing a number or letter generates a **scan code** or **extended ASCII character**. These are mostly used to:

- Create **keyboard shortcuts**
- Enter **special symbols** (like ©, ■, ü, etc.)
- Work with **non-English characters** or old-school UIs



🛠️ Why they mattered (and still do):

In DOS and early Windows programming, these were used in:

- Text-based UI drawing (ALT+219 for block chars)
- Input systems that handled ALT sequences
- Custom shortcut triggers in software

Today they're less used in modern GUI apps but **still work in BIOS-level or console programming.**

Find the **AltKeyCombinations.html** OR **AltKeyCombinations.png** for the full table.

🧱 Keyboard Scan Codes vs ASCII vs ALT-Key Combos

(Know what your keyboard is really saying behind the scenes.)

1. Keyboard Scan Codes – Low-Level Hardware Signals

When you press a key (say A), your keyboard doesn't send "A" to the computer — it sends a **scan code**, a raw hardware signal that says:

"Hey, row 2, column 1 key was pressed!"

- These are **hexadecimal values** like 1Eh for A
- They're used by the **keyboard controller**, **BIOS**, or **OS** to figure out *which* key was hit
- Think of scan codes as **keyboard positions**, not characters

🛠️ Example:

```
mov ah, 00h
int 16h      ; wait for key press, get scan code in AH
```

Find the **KeyboardScanCodes.html** OR **KeyboardScanCodes.png** for the full table.

🧩 Why This Matters (for ASM, C, or hacking):

- You'll need **scan codes** for raw key detection (int 16h)
- You'll use **ASCII** when processing strings or displaying characters
- **ALT combos** come in handy for custom keyboard input or UI design (old school terminal-style)

TLDR Keyboard Cheatsheet

ASCII Control Characters	Keyboard signals that do stuff (like trigger actions or formatting), but don't actually show up as visible characters on your screen.
Mnemonics	Human-readable labels or short codes (like ESC, BEL, ETX) that make it easier for programmers to remember and use specific control characters instead of just their numbers.
Hex Codes	The actual numerical values (e.g., 1Bh, 07h) that represent each ASCII character, whether it's a control character or a visible one. This is how computers truly understand them.
Ctrl Key Combos	Specific keyboard combinations (like Ctrl+C, Ctrl+G) that often trigger those ASCII control characters. They're like sending a direct command to the system or an application.
ALT Key Combos	Keyboard combinations (like Alt+F, Alt+Tab) that generate extended characters (not always part of standard ASCII) or special scan codes. These are often used for custom shortcuts, menu navigation, or entering unique symbols.