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:

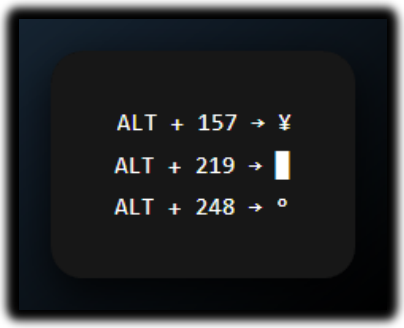
Yup, the combo Ctrl + - gives ASCII 1Fh. It's one of those lesser-known control characters and **it *is* valid.** You won't see it often, but it's real and in the official ASCII spec.

🔤 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

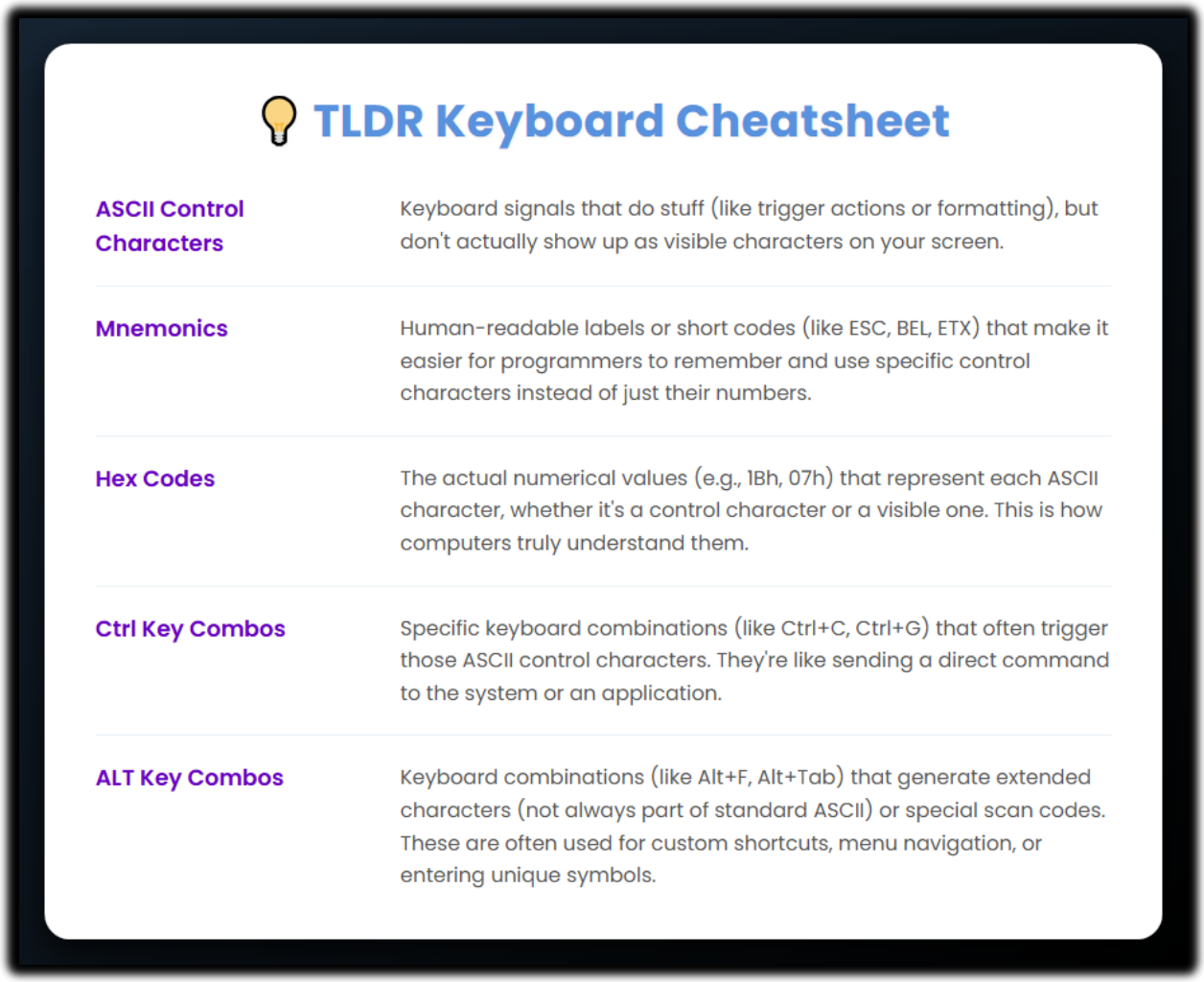
🛠️ Example:

```
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)



🧠 How Programming Languages Understand ASCII

💡 What's ASCII and Why Does It Matter?

ASCII stands for **American Standard Code for Information Interchange**. It's one of the earliest and most fundamental **character encodings** in computer history. It creates a simple mapping between:

- **Characters** (like A, ! , or 3)
- And **numeric values** (like 65, 33, or 51)

This mapping allows computers — which only understand **binary** (1s and 0s) — to **store, process, and display text** in a meaningful way.

📌 Character Encoding = Translation System

Computers don't know what an "A" is. They only know numbers.

That's where **character encoding** comes in. It assigns each character a unique numeric value. When a key is pressed on your keyboard, the hardware generates a **scan code**. The operating system or low-level firmware then translates that into an **ASCII value** — and from there, the software understands what letter or symbol was typed.

- Press A → Computer sees scan code → Translates to ASCII 65 → Stores 01000001 in memory (the binary version of 65)

This system makes it possible for:

- Programming languages to represent and manipulate text
- Operating systems to display letters on screen
- Applications to store readable characters in files

🧪 A Closer Look at ASCII

ASCII uses **7 bits** to represent each character. That gives us 2^7 = 128 unique combinations.

- Values **0–31** → Control characters
- Values **32–126** → Printable characters
- Value **127** → The DEL (Delete) character

So, ASCII defines **128 characters total**, broken into:

🔧 1. Control Characters (0–31, and 127)

These are not visible on the screen but affect how text is handled or transmitted. Examples:

- **LF (10):** Line Feed — move cursor down
- **CR (13):** Carriage Return — move to start of line
- **TAB (9):** Horizontal tab
- **BEL (7):** Bell — triggers a beep
- **DEL (127):** Delete — originally used to erase tape

These were crucial in early computing for text formatting, terminal communication, and hardware control.

🔤 2. Printable Characters (32–126)

These include:

- **Letters**: A–Z, a–z
- **Digits**: 0–9
- **Punctuation**: ., ,, ?, ;, etc.
- **Symbols**: @, #, $, %, etc.
  These are what you actually see when typing or printing.

🧮 Binary and Hex — Just Different Views

- ASCII values are usually written in **decimal** (e.g., 65 for A)
- But under the hood, they're stored as **binary** (01000001)
- And programmers often use **hexadecimal** for readability (41h)

Hex is just a more compact way to represent binary. It's like shorthand for the machine world.

🌐 Unicode: The Global Language of Computers

ASCII is limited to 128 characters — which is fine for English and basic symbols, but not for other languages, emojis, or global scripts.

That's where **Unicode** comes in — it extends the character set to support:

- Characters from **every language**
- **Emoji**
- **Math symbols**
- **Rare scripts**
  Unicode includes ASCII as its first 128 values — so ASCII is technically a **subset of Unicode**.

🤔 What is Unicode?

Unicode is the modern character encoding standard designed to represent **text from every language, script, and symbol set on Earth — plus emojis, math symbols, currency, and more**. It's the universal answer to ASCII's limitations.

Where ASCII stops at 128 characters, Unicode:

- Supports **144,000+ characters (and growing)**
- Covers scripts from **over 150 writing systems**
- Encodes **emoji**, **symbols**, and **non-Latin characters**
- Is used in everything from websites to mobile apps to operating systems

---

📦 Fixed vs Variable-Length Encoding

🔹 ASCII: Fixed-Length

- Uses **7 bits** per character
- Always exactly **1 byte** per character (in modern storage)
- Can only represent 128 characters → English-only

🔹 Unicode: Variable-Length

Unicode supports **multiple encodings** depending on the use case (See the UTF.html):

ENCODING	DESCRIPTION	BYTES PER CHAR	COMMON USE
UTF-8	The most popular and flexible encoding. It's backward compatible with ASCII, meaning standard English text uses just 1 byte, but it can expand to 4 bytes for complex characters from other languages. It's like a smart backpack that gets bigger only when you need more space.	1 to 4 bytes	Web pages, Linux systems, APIs, and pretty much everywhere on the internet. It's the global standard for text.
UTF-16	Used by systems like Windows and Java. It's designed to handle multilingual text efficiently, typically using 2 bytes for most common characters, but can extend to 4 bytes for less common ones. Think of it as a medium-sized suitcase, good for most international trips.	2 or 4 bytes	Windows applications, Java programming environments, and some internal system processes.
UTF-32	A fixed-length encoding where every character, no matter how simple or complex, always uses 4 bytes. This makes it very simple for computers to process (no variable length to figure out!), but it's also very space-heavy. It's like bringing a massive, empty suitcase for every single item you pack.	4 bytes per char	Rarely used for storage or transmission due to its size. More commonly found in internal processing tools or specific scientific applications where speed of character access outweighs storage efficiency.

⚠️ **UTF-8** is the king. It's compact, fast, compatible, and everywhere — from HTML files to JSON to APIs.

💬 ASCII vs Unicode: Key Differences

FEATURE	ASCII	UNICODE
Max Characters	128	1.1 million+ (over 144k defined)
Language Support	English only (basic Latin alphabet, numbers, basic symbols)	All global languages, scripts, symbols, and emoji. It's truly universal.
Bits Used	7 bits (usually stored in 1 byte)	8, 16, or 32 bits (varies depending on the specific Unicode encoding like UTF-8, UTF-16, UTF-32)
Encoding Type	Fixed-length (each character takes the same amount of space)	Variable-length (characters can take different amounts of space, like in UTF-8) or fixed-length (like in UTF-32)
Popularity	Legacy systems, embedded devices, very basic text files. Still foundational, but rarely used alone for modern text.	The default standard in all modern operating systems, web browsers, programming languages, and applications. It's the backbone of global digital communication.

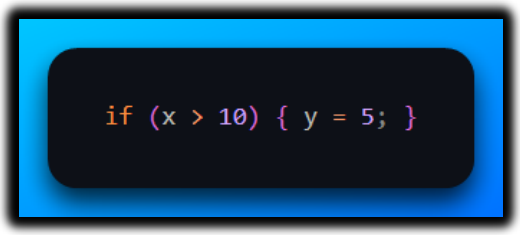
🧩 From Characters to Machine Code: Lexical Analysis and Beyond

🧠 Step 1: Tokenization (Lexical Analysis)

When you write code, the first thing a compiler or interpreter does is **break the code into small pieces** called **tokens**. This phase is called **lexical analysis**.

Think of it like turning a full sentence into individual words so the computer can "read" and "understand" what you're saying.

Example code:



This gets broken down into tokens like:

- if → keyword
- (→ punctuation
- x → identifier
- > → operator
- 10 → constant
- { → block opener
- y → identifier
- = → operator
- 5 → constant
- ;, } → punctuation

🔎 Analogy: It's like chopping a sentence into "The", "quick", "brown", "fox"… so it's easier to parse.

⚙️ Step 2: Compilation or Interpretation

Once your source code has gone through the lexical analysis (tokenization) phase, the next big step is for the programming language to actually turn those tokens into something the computer can run.

This process depends entirely on whether the language is compiled or interpreted.

LANGUAGE TYPE	WHAT HAPPENS AFTER TOKENIZATION
Compiled Languages	The tokens are transformed into machine code (or sometimes an intermediate assembly-like code). This machine code is a low-level, binary representation of your program that your computer's CPU can execute directly. Think of it like translating a book into a specific machine language that only one type of robot understands. This translation happens before you run the program.
Interpreted Languages	Instead of creating a separate machine code file, the tokens are executed one-by-one as instructions by an interpreter program. It's like having a live translator who reads each sentence of your book and immediately tells the robot what to do, without writing down the whole translated book first. This translation happens as you run the program.

At the end of the day, whether through compilation or interpretation, the goal is to get your high-level code down to machine code.

This machine code is the raw, binary instruction set that your computer's Central Processing Unit (CPU) understands and executes directly.

It's the ultimate language of the hardware, telling the CPU exactly what operations to perform, step by binary step.

💡 Where ASCII/Unicode Come In

Remember: source code starts as **text**, and that text is encoded using **ASCII or Unicode**. So before any of this tokenization or compilation can happen, the compiler reads the **binary values** of each character in the source file — based on the encoding.

For example:

- **'i'** → 01101001 (ASCII: 105)
- **'f'** → 01100110 (ASCII: 102)

Every keyword, symbol, and variable starts as **a sequence of bytes**, which the compiler uses to identify tokens.

🧠 TLDR: Code to CPU

1. **You write code** in a text editor
2. It's stored using **ASCII or Unicode encoding**
3. The **compiler reads** this text as bytes
4. It **tokenizes** the characters into meaningful parts (keywords, identifiers, etc.)
5. These tokens are **converted to machine instructions**
6. The **CPU executes** those binary instructions

Boom. That's the journey — from letters to logic to machine.

✅ Closing Thoughts

You now understand:

- How characters (like A, +, =, etc.) are mapped to binary using ASCII/Unicode
- How programming languages tokenize source code using those binary character values
- How those tokens eventually become machine code that the CPU can execute

It's not every little compiler detail, but you've got the big picture.
Enough to write code *and* understand what's happening under the hood.

🧠 1. Machine Code Structure — Opcodes & Operands

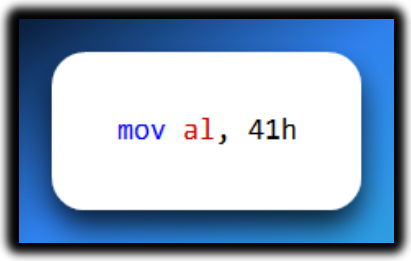
🔧 What is machine code?

It's the actual **binary instructions** executed by the CPU. Every line of assembly or C code eventually becomes machine code under the hood.

Each instruction typically has:

- An **Opcode**: the command (e.g. MOV, ADD, JMP)
- One or more **Operands**: the data it acts on (e.g. registers, memory addresses, immediate values)

🧱 Example Breakdown

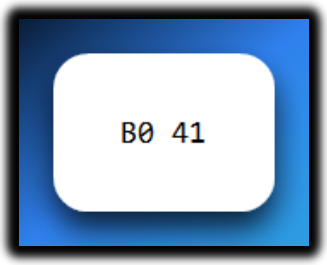


This means: "Move the value 41h (which is ASCII 'A') into the AL register."

Let's break that into real machine code (x86):

- Opcode for MOV AL, imm8 = B0
- Operand = 41

➞ Final Machine Code:



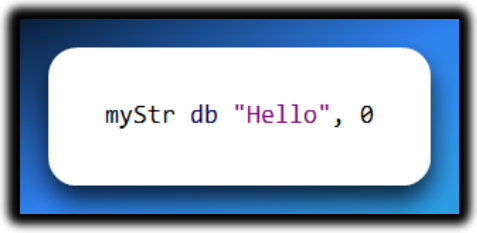
The CPU reads this as:

"Instruction B0 means: move the next byte into AL. The next byte is 41h — which is 'A'. Okay, done."

🔤 2. Assembly and String Handling

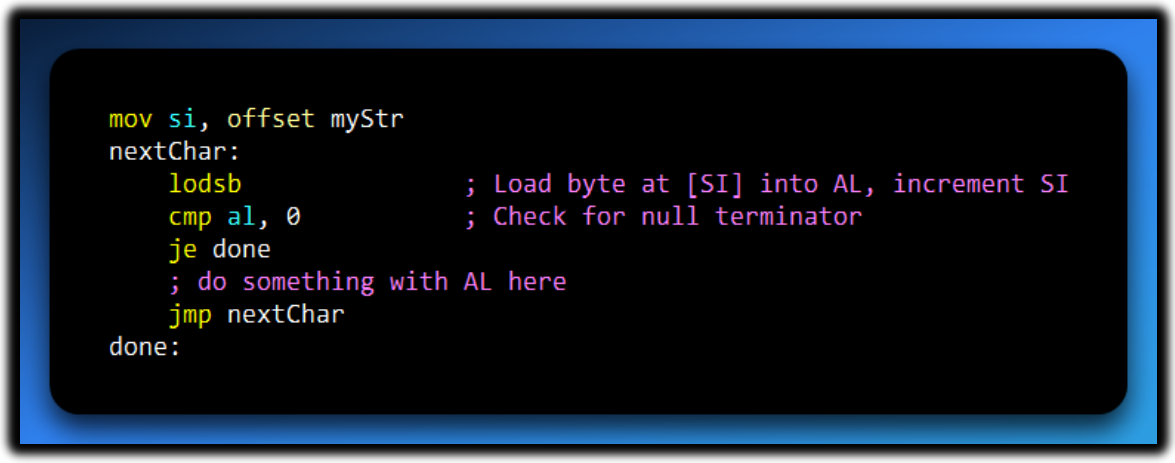
In Assembly, strings are just **sequences of bytes in memory** — nothing fancy. You handle them manually, byte by byte.

💾 Defining a string:



- db means "define byte"
- The 0 at the end is the null terminator (like in C)

💉 Reading character by character:



This is like your bare-metal for (char c : str) loop — just much closer to the metal.

🧠 3. Memory and ASCII/Unicode Storage

Let's say you store the word "Hi" in memory.

ASCII:

```
msg db 'H', 'i', 0
```

Memory layout (byte by byte):

```
Address:    Value:
0x1000      48h   ; 'H'
0x1001      69h   ; 'i'
0x1002      00h   ; null terminator
```

The CPU doesn't *see* characters — it just sees:

```
01001000 01101001 00000000
```

Unicode (UTF-16):

```
msg dw 0x0048, 0x0069, 0x0000  ; Same "Hi" but in UTF-16
```

Now every character is 2 bytes:

```
Address:    Value:
0x1000      48h
0x1001      00h
0x1002      69h
0x1003      00h
0x1004      00h
0x1005      00h
```

That's why **UTF-8 is more compact** — it uses just as many bytes as needed.

🎯 Key Takeaways:

✨ Key Assembly Concepts: Quick Reference

CONCEPT	MEANING
Machine Code	The final, raw binary form of your instructions that the CPU directly understands and executes. It's the computer's native language.
Opcode	The "verb" or action part of an assembly instruction (e.g., MOV for move, ADD for add, JMP for jump). It tells the CPU *what* to do.
Operand	The "subject" or data that the opcode acts upon (e.g., a register like AX, a memory address like [BX], or a constant number). It tells the CPU *what* to do it with.
Assembly Strings	In assembly, strings are just arrays (sequences) of bytes. You have to manually process them character by character, unlike high-level languages where strings are often built-in types.
Memory View	When you look at memory in assembly, you're seeing raw bytes. Characters (like 'A', '!', or '😊') are stored as their numerical ASCII or Unicode values. It's like looking at the raw ingredients before they're cooked into a meal.