Episode 3.08 – Intro to ASCII Character Encoding

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. Our topics involve a bit of figuring, so it might help to keep a pencil and paper handy.

Let's say you crack open memory and find the 32-bit hexadecimal value 0x9768523C. What does this seemingly random pattern of ones and zeros represent? Well, it could be an integer stored in unsigned binary. That would mean it represents 2,540,196,412. In twos complement, 0x9768523C would represent -1,754,770,884. How about packed BCD? That would be -9,768,523. Could this be single precision IEEE-754 representation? If that was the case, 0x9768523C would represent something close to $-7.5067037 \times 10^{-25}$. Heck, this binary value could be the reading from an analog sensor, in which case we have no idea what it represents until we know the analog range it maps to.

Is this all that we can represent in binary? Absolutely not! For example, noticed that not one of these conversions could represent letters. They were all about the numbers. There must be a way to represent letters in binary so that we can store things such as documents, or messages, or contact information, or…well… anything involving language.

This is where character encoding comes in. Remember in Episode 2.4, we discussed how binary coded decimal, or BCD, was a method for encoding the ten digits of the decimal numbering system to different patterns of bits. We determined that since there were ten unique decimal digits, we would need at least four bits. Three bits wouldn't be enough since there are only eight different patterns of ones and zeros in three bits. When it came to assigning the bit patterns, we just mapped each decimal digit to the corresponding four-bit pattern with a matching unsigned decimal value. We could have, however, made the mapping any way that we wished.

Character encoding is simply an expansion of what we did with BCD to letters, punctuation, and other symbols. We begin by determining how many different letters and symbols we wish to represent. From this list, we can determine how many bits we will need in order to generate enough patterns of ones and zeros so that each letter and symbol can be mapped to a unique pattern.

For example, if we want to create an encoding scheme to represent the ten decimal digits, the twenty-six letters of the modern English alphabet (ignoring the difference between uppercase and lowercase), the fourteen common punctuation marks of the English language, the space, the tab, and the carriage return (we can talk about line feeds later), we're up to fifty-three different symbols. If we assign a different pattern of ones and zeros for each of these fifty-three symbols, we will need at least six bits. Five bits is only capable of thirty-two unique patterns, so it won't be enough. If we also want to distinguish uppercase letters from lowercase, then we will need to find twenty-six additional unique patterns bringing the total to seventy-nine patterns. Since this goes beyond 64 different patterns available from six bits, we need to move to seven bits.

Before we go any further, let's look at one of the granddaddies of character encoding. ASCII, or the American Standard Code for Information Interchange, is a seven-bit character encoding designed in the early 1960's to exchange data between processing elements, communication systems, and peripherals.[1] The original specification included twenty-five punctuation marks and mathematical symbols, a space, the ten decimal digits, the twenty-six letters of the alphabet (uppercase only), one character each to represent the left arrow and the up arrow, and thirty-six unprintable control codes defining things like delete, escape, message identifiers, acknowledge, and an audible bell.
For the most part, the symbols were organized so that the two most significant bits of the seven bit binary pattern identified the general type of symbol represented. If the two most significant bits were 00, the character represented was a control character. If the two most significant bits were 01, the character represented was most likely a punctuation mark, mathematical symbol, or a decimal digit. All uppercase letters had an ASCII code starting with a binary 10. All but four of the patterns starting with 11 were left unassigned. Later versions of ASCII used these unassigned patterns for the lowercase letters.

As eight-bit memory locations became more common, variations of the ASCII character set began appearing that allowed for 256 characters. Extended ASCII used the additional 128 patterns to represent symbols ranging from graphical elements to symbols from other languages such as the German ligature ß (pronounced "eszett"), which had an extended ASCII value of 0xE1, and the Spanish diacritical mark of a tilde over an n creating ñ (pronounced "eh-nyeh"), which had an extended ASCII value of 0xA4. There were also familiar symbols such as the British pound, £, 0x9C, the copyright symbol, ©, 0xB8, and the single character versions of one-half, ½, and one quarter, ¼, 0xAB and 0xAC respectively.

Anyone who worked with ASCII for a while began to memorize some of the patterns and learn some of its tricks. For example, 0x20 was the code for a space, decimal numbers began at 0x30, capital letters began at 0x41, and lowercase letters began at 0x61. To convert between a single digit integer in binary or BCD, we simply added 0x30. To take any letter and force it to be a capital letter, just clear the second most significant bit. To take any letter and force it to be lowercase, set that second bit to one. In a later episode, we will present simple operations you can use to set and clear individual bits like this.
Let's say we crack open memory again to find the hexadecimal value 0x486F773F. If we know that these thirty-two bits represent four characters encoded using ASCII, all we need to do is find how each byte maps to a character or symbol. The Internet is riddled with ASCII tables we can use to determine this mapping. Just search on the phrase, "ASCII table," and millions of results will appear. From one of these tables, you'll see that 0x48 maps to an uppercase 'H', 0x6F maps to a lowercase 'o', 0x77 maps to a lowercase 'w', and 0x3F maps to a question mark. The four bytes we just found represent the string, "How?"

ASCII was not developed for today's world of apps aimed at an international audience or smart phones displaying elements far beyond a simple alphabet. In its seven-bit form, ASCII is hopelessly limited to unaccented English letters. In our next episode, we will present encoding schemes for our modern international world. We will even throw in some of those weird little digital images called emojis. It turns out, however, that going back to seven-bit ASCII with its leading zero in the eighth bit position will be a great place to start. For transcripts, links, or other podcast notes, please check us out at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until the next episode remember that while the scope of what makes a computer is immense, it's all just ones and zeros.

References:

1 – https://worldpowersystems.com/ARCHIVE/codes/X3.4-1963/index.html