Data Encapsulation in Plain Text

stgiga

Introduction

Over the years, there have been MANY approaches to encoding binary data as text, such as the well-known Base64 (as inefficient as it is), as well as more-efficient methods such as Base96 (all 94 visible ASCII characters, plus space and tab), and less-efficient methods such as Base36 (numbers and either lowercase OR uppercase letters, but not both). However, out of all the methods mentioned earlier, even the methods more efficient than Base64 fall short of being the most efficient way to store binary data as text, because you can only go up to a base of 96 without having to deal with the OTHER ASCII tab, as well as the competing newline standards across OS choices. Even using ALL of those (So, 94 visible ASCII characters, space, HT, VT, CR, LF, and RS, which will get you to a base of 100, for better or for worse), assuming ASCII is what we are using here. HOWEVER, if we break out of ASCII and set our sights on Unicode, MUCH more efficiency is possible, which results in a LOT of options, which this paper will touch on.

It must be said that this paper is written by a nonbinary software developer (who uses They/Them pronouns & certain neopronouns, including Unicode ones) whose code, BWTC32Key (available at http://b3k.sourceforge.io), is the topic of discussion here. This COULD count as a conflict of interest, though it IS a program which is FOSS, down to not costing any money. No entity was ever paid to write this paper.ꞡ

Using Unicode Well

Unicode has a LOT of characters (even 💣) compared to ASCII, so the amount of data we can store in a single character is significant. However, the key to doing this right is how big the resulting string would be in memory, and not entirely how many bits fit in one character. Using a base of 65,536 would require using characters that are outside the Basic Multilingual Plane (Plane 0), which, in addition to creating compatibility problems on older OS choices, are also up to 4 bytes in size, even though Unicode as a whole is ONLY 21 bits. Yes, UTF-32 has 11 wasted bits at the time of writing, 8 of which are the first byte, which is technically reserved for further expansion later on, though getting THAT to play nicely with UTF-16 would require additional surrogates in a higher plane (theoretically, a great place for that would be in a chunk of Plane 14, the Supplementary Special Purpose Plane). UTF-16 may have its quirks, but what it has that other Unicode Transformation Formats do NOT have is being a perfect vessel for data storage.ड

In UTF-16, ALL Plane 0 characters take up exactly 16 bits (two bytes), including Plane 0's CJKV (Chinese, Japanese, Korean, and Vietnamese characters) characters, which take up three bytes in UTF-8, much to the chagrin of those who use said characters. On that subject, there are a LOT of said characters in Plane 0. There are TWO Plane 0 blocks of Han characters (what Westerners would consider to be "Chinese characters") which have multiple thousands of characters in them (as well as another block of these characters that has 472 of them), which, in combination with the 11,172 characters of the Plane 0 Unicode block "Hangul Syllables" (all modern Korean Hangul syllable blocks, used FAR more often than their components, known as Hangul Jamo. If you go to a Korean website, the vast majority of the characters you will see are from the “Hangul Syllables” Unicode block), will put you well over a pool of 2^15 (32,768). If an approach similar to Base64 but with this pool of characters is used, you can store 15 bits of data per character, and since in UTF-16 these characters take up EXACTLY 16 bits, you store 15 bits of data per 16-bit character, allowing for an efficiency of 15/16ths (93.75%), with clean code to boot. Going any higher into numbers like 2^15.25 (which can be hit if you use the entirety of the three Plane 0 “CJK Unified Ideographs” blocks and the “Hangul Syllables” block) would NOT be clean code at all, so stopping at a base of 32,768 (henceforth called Base32768) is a GOOD thing for anyone trying to implement it.

With regards to implementations of Base32768 (My program, BWTC32Key, uses a Base32768 implementation by “@kusano_k” on Github, who used it in their JavaScript program “Tinygma”, and it uses the ranges of U+3400-U+4CFF [Which is the first 6,400 characters of the 6,592-character Unicode block: “CJK Unified Ideographs Extension A”, so it COULD fit in the Plane 0 Private Use Area if backporting to Unicode 1 with its lack of that block], U+4E00-U+9EFF [The first 20,736 characters of the 20,992-character “CJK Unified Ideographs” Unicode block, which had 20,902 characters in it in Unicode 1, so a backport to that version would ALSO be safe], and U+AC00-U+C200 [The first 5,633 characters of the 11,172-character “Hangul Syllables” Unicode block, which didn’t exist in Unicode 1. Unicode 1 had 6,656 Hangul characters across 3 blocks located where “CJK Unified Ideographs Extension A” and “Yijing Hexagram Symbols” {Divination symbols used in ancient China} reside today. Since only 5,633 Hangul Syllables are used by the program, they can be shoved into the old location. Also, since the 5,633rd character in that range is WELL before the position where U+4D00 and U+4D01 are in modern Unicode, the program’s use of those values as part of the header and as the trailer/terminator

CAN be safely backported too. Also, the U+FEFF “Byte Order Mark” used as part of the header in BWTC32Key to make its “.B3K” files decode properly in text editors existed in Unicode 1 as a “Zero-Width Non-Breaking Space”, that will ALSO backport safely, allowing the program to be made to work in Unicode 1 if one were to want to.] The 5,633rd character [U+C200] is used the same way the “Equals Sign” is used in Base64, but only ONE is needed. As mentioned earlier, U+4D00 and U+4D01 are used as part of the header, and the footer/terminator respectively, with the first half of the header being U+FEFF. So, the alphabet is 32,772 characters, but the amount beyond 32,768 don’t serve as data characters with the exception of U+C200 if it is needed to make data ending in a certain bit decode properly. This is actually the SIMPLEST way to do Base32768 🤓 [Yes, I’m using emoji and other Unicode characters in this document, made possible by my mod to GNU Unifont, known as “UnifontEX”, available at http://stgiga.github.io/UnifontEX whose focus is on high compatibility].) having its high efficiency is good, but one can always do better, and what better way to achieve that than heavy compression?

Compression

Compression is ANOTHER thing in which there are MANY ways to do it. There’s lossy compression like in JPEG, HEIC, AVIF, WEBP, and such (ALL of which one should NEVER use, because data loss is a BAD thing. PNG is an infinitely-better format❣), and then there’s lossless compression (such as PNG, FLAC, LZW, LZ77, LZ78, Huffman, GIF, ZIP, 7-Zip, RAR, SIT, Bzip, Bzip2, and such.) That said, not all lossless compression is created equal. Considering that the goal is to efficiently fit data into characters, a format with a large header, a high probability of expansion on some files, or weak compression is not a good idea. One marker of a good compression algorithm is it beating out DEFLATE (ZIP, PNG, and Gzip use this). Bzip2 does this, but Bzip1 and LZMA, LZMA2, and LZHAM do it better. However, LZMA and LZMA2 (most likely LZHAM too) can eat up a LARGE amount of RAM and expand. Bzip1 in all its discontinued glory has these problems too, though the expansion is more serious but the memory overuse is more excessive. Bzip1 uses an Arithmetic Coder (slow, eats memory, but effective.), while Bzip2 uses a Huffman Coder (weaker, used in DEFLATE, is implemented in a way that creates further expansion than either Bzip archiver can do due to their shoddy RLE1 step at the beginning of compression.) But there IS a middle ground. The Range Coder is VERY minimally (0.01%) weaker than an Arithmetic Coder, but MUCH faster, and in fact, it compresses faster than EITHER Bzip archiver. Thus, it works fine. Also, getting rid of the first RLE (RLE1) in the Bzip archivers (which can expand a file by 1.25x rather than compress it, and was a historical mistake by the creator of the Bzip archivers because of not being exposed to the paper on how to make a Burrows-Wheeler Transform INHERENTLY resistant to slowdowns from highly-redundant input data) prevents any possibility of expansion THAT way, and NOT using what Bzip2 does for compression saves us from its Huffman Coder implementation that uses Unary as part of it in some cases that can result in an ADDITIONAL 1.015x expansion on top of any earlier RLE1-induced 1.25x expansion had one happened. So, not great. Also, Bzip2 has a MUCH more bloated header than Bzip1 (down to using BCD of Pi and the Square Root of Pi as certain sections). While this DOES help when dealing with blocks, since we are encoding it to text, that use does not apply here. ड

So, there IS a good method of heavy compression involved that won't break the memory or header size budget, lose detail, or such (the algorithm used is the JavaScript implementation of “BWTC”, by “cscott” on Github that is from

“CompressJS”, but BWTC32Key uses the now-gone “compressjs-flattened” by “eladkarako”, formerly on Github. Evidently BWTC32Key preserves THAT.), but there’s still more one can do to TRULY shine at making a binary-to-text encoding. Optional encryption perhaps?

Encryption

Encryption is another way of increasing entropy in data beyond compression, but of course, if you are to combine those, you want to have the encryption come AFTER the compression to not destroy the compression. Also, encryption, by design, produces what seems like random bits, caring not for whether something is printable text or not, so you would want to do your binary-to-text encoding AFTER the encryption. Oh, and this does not eliminate the need to do the compression prior to the encryption. One would still want to do that. With regards to doing binary-to-text encoding on encryption in a situation where one would want to fit large amounts of data in a character, having a long padding trailer is BAD, but one would also not want to use something like the ancient DES (Data Encryption Standard) method with horrible security. Something such as AES256-CTR (Counter, is secure but does not require or use any padding, making it GREAT for this use. Better still, it’s not bloated.) would work nicely. (The implementation used in BWTC32Key is “aes-js” by “ricmoo” on Github, and the keys are derived from passwords using a JavaScript function that isn’t an example of “rolling your own crypto” done by BWTC32Key’s developer.) 𓂀

So, the pipeline would be to run data through an efficient and non-bloated compression algorithm, then AES256-CTR, and then Base32768’s very efficient, yet safe binary-to-text encoding, to fit data securely and efficiently (and doubly greatly so) into text, as best as possible. This is what BWTC32Key does, and it does so in JavaScript, & in a way that isn’t bloated or non-libre, making it shine♥