# N3608: Variable length prefixed length strings


| | |
|---|---|
| Document #: | N3608 |
| Date: | 2025-06-30 |
| Project: | Programming Language C |
| Reply-to: | Niall Douglas <s_sourceforge@nedprod.com> |


Null terminated byte strings (NTBS) have been the traditional way of implementing text strings in C almost since its inception. NTBSs have the upside of minimum possible storage overhead, but they cost more processor time to use, are unnecessarily heavy on CPU memory caches, and are a major source of bugs and security vulnerabilities. Given how RAM capacities have exponentially exploded in recent decades, it is probably time for C to standardise a length prefixed byte string.

WG14 has been working on this for many years now in one form or another. Most recently, it has been around [N3306] *`strb_t`: A standard string buffer type*, however that is actually orthogonal to length prefixed arrays as it concerns dynamic memory allocated string buffers. Rather, the most recent paper I am aware of specifically dealing with length prefixed variably sized arrays is [N3210] *A string type for C* in 2024. That proposed:

```
struct {
  size_t size;
  char8_t data[/* size */];
};
```

... with the `char8_t` array being required to be zero terminated for backwards compatibility.

WG14 discussion for that paper felt that spending a `size_t` overhead per 'modern string' was too much, and it was wondered if a more compact length prefix could be designed. It was suggested we use the same technique as what UTF-8 uses to create a variable length prefix to minimise overhead. However others were concerned that this would confuse UTF-8 parsers, and that could open security concerns.

There **is** encoding space left unused by UTF-8 however. Would this be sufficient for our needs? Let's find out!

## Contents

# 1 Quick recap of UTF-8 encoding

UTF-8 uses the top bits of each octet to describe a variable length encoding of a UTF codepoint. The rules are simple:

1. If the top bit is zero, the remaining seven bits are codepoints 0-127.
2. If the top three bits are `110xxyyy`, there will be a second continuation octet of the form `10yyzzzz`.
3. If the top four bits are `1110wwww`, there will be two continuation octets of the form `10xxxxyy`, `10yyzzzz`.
4. If the top five bits are `11110uvv`, there will be three continuation octets of the form `10vvwwww`, `10xxxxyy`, `10yyzzzz`.

As continuation bytes always have the top bits set of `10`, you can always find the beginning of the current UTF-8 sequence from any pointer or index into an array by scanning backwards by up to three bytes. This is known as *self-synchronisation* and it is a useful property.

The current Unicode standard defines seventeen *planes* i.e. codepoints between zero and `0x10ffff` which is a subset of twenty-one bits. Any value higher than this is therefore not possible in the current Unicode standard. This, in turn, means that leading bytes from `0xf8 (11111000)` upwards are invalid (they would indicate four continuation bytes), but also the leading byte values `111101vv` (i.e. where `u` is one and `vv` is one or higher) because that would mean Unicode plane 17 or higher is indicated. Later on we shall discuss the possibility that Unicode may use more than seventeen planes in the future.

Bringing all the above together, you will see that these octet values can never appear in a legal UTF-8 sequence:

There are two illegal values at:

- `0xc0 (11000000)`
- `0xc1 (11000001)`

There are three illegal values at:

- `0xf5 (11110101)`

- 0xf6 (11110110)
- 0xf7 (11110111)

And finally, there are eight illegal values at:

- 0xf8 (11111000)
- 0xf9 (11111001)
- 0xfa (11111010)
- 0xfb (11111011)
- 0xfc (11111100)
- 0xfd (11111110)
- 0xfe (11111101)
- 0xff (11111111)

The proposal is that we build a variable length prefix encoding which is 'UTF-8 aware' in the sense that a correct UTF-8 parser will never parse our variable length prefix as a valid UTF-8 sequence.

## 2 The proposed variable length prefix encoding

It is proposed that the layout in memory for a variable length array of `uint8_t` would be:

```
struct varuint8_t
{
  uint8_t length[/* 1, 2, 4 or 8 */];
  uint8_t data[/* decoded length */];
};
```

`length` would always be one of one, two, four or eight as some processors have SIMD optimisations for UTF-8 processing which could be reused here[1].

`data` would be *optionally* zero terminated. This is because `varuint8_t` is a variable length array of `uint8_t` which can *also* be treated as a variable length array of `char8_t`, as we shall see later.

`varuint8_t` is not the nicest name for the type, however all the nicer ones turn up as being widely used in a github search for identifiers. `varuint8_t` appears to not be used by any open source codebase.

Something perhaps not immediately obvious is that this encoding is *endian independent*. It has identical memory layout on all platforms and therefore has the same advantages in this area as NTBSs for serialisation/deserialisation etc. Also, like in UTF-8, this encoding is similarly *self-synchronising* – a pointer to anywhere within the prefix header can be deterministically rewound to the beginning of the prefix header.

[1] https://lemire.me/blog/2020/10/20/ridiculously-fast-unicode-utf-8-validation/

### 2.1 Array lengths zero to seven

A leading octet in the range `0xf8-0xff` would indicate a very short array length 0-7 long, as there is **three** bits of storage in a single byte prefix.

Examples:

- `0xf8` is a zero length array, and is a total of one octet of storage.
- `0xf9,0x78` is a one item length array, and is a total of two octets of storage.

If used to represent a NTBS, the added overhead is a worst case of 100% and a best case of 12.5%. If the trailing null isn't needed, it has the same overhead as a NTBS.

### 2.2 Array lengths eight to seventy-one

A leading octet `0xf5` would indicate a short array length 8-71 long. There is one continuation octet of the form `10xxxxxx` which means there is **six** bits of storage in a two byte prefix.

Example:

- `0xf5,0x80,0x4e,0x69,0x61,0x6c,0x6c,0x20,0x44,0x00` = `Niall D\0` is an eight item length array, and is a total of ten octets of storage.

If used to represent a NTBS, the added overhead is a worst case of 25% and a best case of 2.8%. If the trailing null isn't needed, it has a worst case of 12.5%.

### 2.3 Array lengths seventy-two to 262k

A leading octet `0xf6` would indicate an array length 72-262215 long. There are three continuation octets of the form `10xxxxxx,10yyyyyy,10zzzzzz` which means there is **eighteen** bits of storage in a four byte prefix. As with UTF-8, continuation octets are big endian in orientation.

If used to represent a NTBS, the added overhead is a worst case of 5.6% and a best case of approximately zero.

### 2.4 Array lengths 262k - 4 trillion

A leading octet `0xf7` would indicate an array length 262216-4398046773319 long. There are seven continuation octets which means there is **forty-two** bits of storage in an eight byte prefix. As with UTF-8, continuation octets are big endian in orientation.

At this scale, the added overhead over a NTBS is always approximately zero.

### 2.5 The remaining prefixes

For now, the leading octets `0xc0` and `0xc1` would be reserved for future expansion in the standard.

This author's personal preference for string views/slices would be fat pointers, however if the committee felt mutable string buffers were better and we need to additionally encode a reservation or capacity, that's what these prefixes ought to be used for.

## 3 UTF-8 support

The proposal would be that standard library functions could safely return a `char8_t *` from a `varuint8_t *` in various ways e.g.

```
// Returns the number of header octets needed for a given array length
// Returns 'MAX_SIZE' if the length is too long for a 'varuint8_t'.
size_t varuint8_header_size(size_t arraylen);

// If 'datalen' would fit inside 'buflen', place the appropriate
// length of header prefix at 'buf' and then 'data' following. Otherwise
// return nullptr.
varuint8_t *varuint8_fill(void *buf, size_t buflen,
                          const uint8_t *data, size_t datalen);

// Convenience alternative for 'varuint8_fill()'.
varuint8_t) *varuint8_fill_ntbs(void *buf, size_t buflen,
                                const char *str);

// Dynamically memory allocate the correct number of bytes to
// store 'datalen' items in the array, and copy 'data' into that array.
// You must call 'free()' on the returned pointer when done with it.
varuint8_t *varuint8_malloc(const uint8_t *data, size_t datalen);

// Convenience alternative for 'varuint8_malloc()'.
varuint8_t *varuint8_malloc_ntbs(const char *str);

// Concatenate multiple 'varuint8_t'
varuint8_t *varuint8_cat_N(void *buf, size_t buflen,
                           const varuint8_t *const *arrs, size_t arrslen);

// Concatenate two 'varuint8_t'
varuint8_t *varuint8_cat(void *buf, size_t buflen,
                         varuint8_t *arr1, varuint8_t *arr2);

// Dynamically memory allocate a clone of the 'varuint8_t'.
// You must call 'free()' on the returned pointer when done with it.
varuint8_t *varuint8_dup(const varuint8_t *arr);

// Returns the size of the array for access
size_t varuint8_length(const struct varuint8_t *arr);

// Returns the size of the array for memory copying
size_t varuint8_sizeof(const struct varuint8_t *arr);

// Return a pointer to the uint8_t at the beginning of the array
// Returns nullptr if array is zero length
uint8_t *varuint8_front(struct varuint8_t *arr);

// Return a pointer to the uint8_t at the end of the array
// Returns nullptr if array is zero length
uint8_t *varuint8_back(struct varuint8_t *arr);

// Return a pointer to the uint8_t just after the end of the array.
// Useful for iteration between front and end.
// Returns nullptr if array is zero length to match front.
```

```c
uint8_t *varuint8_end(struct varuint8_t *arr);

// Return a pointer to the uint8_t at idx into the array
// Returns nullptr if idx is outside array
uint8_t *varuint8_index(struct varuint8_t *arr, size_t idx);

// Return a NTBS if array is null terminated AND contains
// no intermediate null values, nullptr otherwise
char8_t *ntbs_from_varuint8(struct varuint8_t *arr);

// Return a pointer to the char8_t at or preceding arr[idx]
// Returns nullptr if idx is outside array or there is no
// valid UTF-8 codepoint at that index. You might use
// stdc_c8nrtoc32n() from this point to parse UTF-8.
char8_t *char8_from_varuint8_index(struct varuint8_t *arr, size_t idx);

// Copy a 'varuint8_t' into a user supplied buffer.
varuint8_t *varuint8_cpy(void *buf, size_t buflen,
                         const varuint8_t * arr);
```

I put together reference implementations for the above functions at https://github.com/ned14/wg14_strings/blob/main/src/wg14_strings/impl.c. They are straightforward and unsurprising in my opinion.

I did a trivial reference implementation and I didn't try employing SIMD optimisations. Even with trivial implementations, the benchmarks turned out to be very good.

## 4 Language support

It is not proposed in this paper, however if the language could directly support `varuint8_t` values e.g. so they could be copied around by value and `sizeof` and `arr[N]` worked out of the box, that would be great.

In particular, if `char` and `char8_t` string literals could also be encoded by the compiler as `varuint8_t`, this would enable newly recompiled code to use the length prefix for string literals instead of scanning for null termination.

Alternatively, we could simply add to the compiler automatic creation of `varuint8_t` from any input string literal like:

```c
static const varuint8_t teapot = "I am a teapot!";

static const varuint8_t file = {
#embed "blob.bin"
};
```

This would let end users upgrade their code on an opt-in rather than opt-out basis.

We can't upgrade `strcpy()` et al to use the length prefix as that would break ABI, however a new suite of safe and high performance length prefixed string manipulation functions could be added in a future paper. In my opinion, we ought to solve the length prefixed variable length array problem first, and once we have a solid foundation then we look into length prefixed strings.

## 5 Runtime overhead

The storage space overheads have already been indicated, but what sort of runtime overhead might there be?

NTBS manipulation functions are necessarily `O(N)` complexity as a full scan of the array needs to be performed every time to determine its length. This length calculation is fundamentally non-parallelisable as we cannot say in advance where valid memory might end.

As a result, real world C code usually executes `strlen()` once and stores the length next to the pointer to the array. Repeated calls to `strlen()` only really occur in any API which takes a NTBS pointer, however there are an awful lot of such APIs out there in the wild including most POSIX syscalls and library functions.

A length prefixed string has a complexity of `O(1)` and because we know the length in advance, parallel processing of the string's data using SIMD or OpenMP becomes very tractable. So how much slower is this compacted prefix header over a straight `size_t` prefix header? These are for a MacBook Pro M3 running ARM64 calling `extern` functions (i.e. no inlining is done).

For a string length zero (one byte header):

- `varuint8_fill()` takes 2.488442 nanoseconds (~10 ticks).
- `varuint8_length()` takes 0.742094 nanoseconds (~3 ticks).
- `varuint8_index()` takes 0.744102 nanoseconds (~3 ticks).

For a string length eight (two byte header):

- `varuint8_fill()` takes 2.971138 nanoseconds (~12 ticks).
- `varuint8_length()` takes 0.990406 nanoseconds (~4 ticks).
- `varuint8_index()` takes 0.992103 nanoseconds (~4 ticks).

For a string length seventy-six (four byte header):

- `varuint8_fill()` takes 3.465841 nanoseconds (~14 ticks).
- `varuint8_length()` takes 1.237405 nanoseconds (~5 ticks).
- `varuint8_index()` takes 1.238174 nanoseconds (~5 ticks).

I was curious what might happen if everything were inlined, so I also ran with `-flto`:

For a string length zero (one byte header):

- `varuint8_fill()` takes 2.4760472 nanoseconds (~10 ticks).
- `varuint8_length()` takes 0.344237 nanoseconds (~1.33 ticks).
- `varuint8_index()` takes 0.000002 nanoseconds (~0 ticks).

For a string length eight (two byte header):

- `varuint8_fill()` takes 2.970772 nanoseconds (~12 ticks).

- `varuint8_length()` takes 0.742298 nanoseconds (~3 ticks).
- `varuint8_index()` takes 0.000002 nanoseconds (~0 ticks).

For a string length seventy-six (four byte header):

- `varuint8_fill()` takes 3.219059 nanoseconds (~13 ticks).
- `varuint8_length()` takes 0.993092 nanoseconds (~4 ticks).
- `varuint8_index()` takes 0.000003 nanoseconds (~0 ticks).

The good news here is that the compiler's optimiser does realise that the length calculation can be hoisted outside the test loop, and therefore indexing runs at maximum speed.

A length prefix checking array index would cost at least two ticks due to needing to check if the index exceeds the length. I would therefore claim that this compacted way of length prefixing would be **2x slower** to safely index into the array than a simple `size_t` length prefix based design.

As with NTBSs, a developer may cache the length to avoid per-index length calculation overheads or the compiler's optimiser may silently do so behind the scenes. I think in real world code, a superscalar out of order CPU would almost certainly hide the added latency of these compressed prefix lengths, so this more compact encoding is probably 'free' except on in order CPUs.

In my own personal opinion, I think that the runtime overhead of these compact length prefixed arrays is very acceptable for the memory saved in prefixing many short strings. I look forward to hearing what the commitee's thoughts are on this tradeoff.

## 6 What happens if Unicode uses more than seventeen planes in the future?

Thanks to Thomas Köppe for raising this concern on the mailing list.

Seeing as Unicode exceeded sixteen bits in the past, might it need more encoding room in the future?

The Unicode consortium publishes a roadmap at `https://www.unicode.org/roadmaps/` and a list of new additions currently being worked upon at `https://www.unicode.org/alloc/Pipeline.html`. There is currently no expectation that seventeen planes will be exceeded in any reasonable timeframe. Furthermore, UTF-16 *by definition* cannot represent more than seventeen planes. If the Unicode consortium ever introduced eighteen planes, it would fundamentally break all UTF-16 based software. This seems highly unlikely to ever occur.

Something perhaps not obvious in Unicode is how sparsely populated it is. In Unicode 16 (published 2024), there are 154,998 defined codepoints in a defined space of 2,097,152, so only 7.4% of the available space is currently used. Unsurprisingly, everything on the pipeline and roadmaps linked above is filling in additional valid codepoints within the existing space.

Finally, planes 15 and 16 are both allocated for private encodings only i.e. codepoints within them are allocated by user software only. And planes 4 to 13 are currently unused. Whilst correct code must handle plane values 4-16, in the real world those planes will only appear in test suite

code. Therefore, in the real world, unicode codepoints between zero and `0x3ffff` are the *de facto* encoding space. Even on this measure, less than 60% of that code space currently contains defined codepoints and new additions appear to be running at about five thousand per year. This would suggest twenty-one years remain to fill planes 0-3.

# 7 References


[N3210] Uecker, Martin
*A string type for C*
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3210.pdf

[N3306] Bazley, Chris
*strb_t: A standard string buffer type*
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3306.pdf