utf8gen

Paul Hardy

This manual describes `utf8gen`, a utility for converting Unicode hexadecimal code points into UTF-8 as printable characters for immediate viewing and as byte sequences suitable for including in programs.

Table of Contents

1 Introduction

This document describes some typical uses for `utf8gen`, a utility to read ASCII hexadecimal numbers, interpret them as Unicode code points, and output Unicode Transformation Format – 8-bit (UTF-8).

If you have questions, please email `unifoundry@unifoundry.com`. You can check for the latest `utf8gen` news at `http://unifoundry.com/utf8gen/`.

— Paul Hardy (`unifoundry@unifoundry.com`) 2018

2 Unicode

2.1 Unicode Overview

Unicode arose out of a practical need for a common encoding to represent all of the world's languages on computers. It has grown rapidly over the past 20+ years to contain more than 100,000 glyphs (characters). These glyphs are divided over multiple Unicode *planes*: Plane 0 through Plane 16 (decimal), for a total of 17 planes. Each plane contains 64k *code points*, which in hexadecimal is 10000 code points. *Code point* is a more general term than *character*, because Unicode contains more than just visible characters; for example, Unicode contains various code points for indicating variation selection for scripts that have multiple forms of a visible character.

2.2 Unicode Planes

Plane 0 contains most of the world's modern scripts. Code points in this range are denoted as Unicode code points U+0000 through U+FFFF, inclusive. This plane is also known as the Basic Multilingual Plane, or BMP. The ASCII code points are in the beginning of the BMP, from U+0000 through U+007F. The BMP is almost entirely allocated — there are hardly any free code point ranges in the BMP for assigning new scripts. Fortunately, Unicode has 16 more planes beyone Plane 0.

Plane 1 contains many ancient scripts, and modern collections that were not assigned to Plane 0 (for example, emoji). This plane is also known as the Supplementary Multilingual Plane, or SMP. Unicode code points in the SMP are in the range U+10000 through U+1FFFF, inclusive.

Plane 2 is the Supplementary Ideographic Plane, or SIP. It contains Chinese and Japanese ideographs that were not included in Plane 0. Unicode code points in Plane 2 are in the range U+20000 through U+2FFFF, inclusive.

These are the main planes with assigned visible characters.

Plane 14 is the Supplementary Special-purpose Plane, or SSP. Its code points are in the range U+E0000 through U+EFFFF. This plane contains specialized tags and other designators.

Planes 15 and 16 are Private Use Area (PUA) planes. They can contain any user-defined characters and special-purpose codes. These planes span the Unicode range U+FFFFF through U+10FFFF.

2.3 UTF-8

Thus the valid Unicode range is U+0000 through U+10FFFF, inclusive. Encoding the entire Unicode range takes from one byte for the ASCII range to 21 bits to encode anything in Plane 16 (U+100000 through U+10FFFF).

A problem with transmitting these multi-byte numbers is that different computer architectures order bytes in a multi-byte word differently. Today there are only two common orderings: big-endian, where the largest byte is stored first, and little-endian, where the smallest byte is stored first.

When transmitting information between computers of different architectures, a standard protocol had to be defined. The Unicode encoding that modern web browsers use is called Unicode Transformation Format – 8-bit, or UTF-8. It has also become the standard encoding for text documents that contain non-ASCII characters.

UTF-8 encoding has several desirable characteristics, which are described briefly below.

The first byte in a UTF-8 encoded character begins with a series of '`1`' bits, to indicate how many bytes the character requires, except a one-byte character starts with a '`0`' bit to designate the byte as ASCII. The ASCII range, U+0000 through U+007F, is encoded the same in UTF-8, as just one byte. Each byte after the first in a multi-byte character begins with the bits '`10`'.

The number of bytes in a UTF-8 encoded Unicode code point varies from one to four bytes. Thus it is an efficient encoding compared to one that would transmit the same number of bytes for every character across the entire Unicode range.

No single-byte UTF-8 character will ever begin with the pattern '`10`', as single-byte UTF-8 characters always begin with a '`0`' bit. So string searching functions can skip bytes within a UTF-8 byte string and if a byte currently being examined begins with the bits '`10`', the search function knows it is past the beginning of a multi-byte character.

Unicode code points are published in *code charts*, available at `http://unicode.org/`. These code charts number code points using hexadecimal. These hexadecimal numbers must be converted to UTF-8 for transmission on web pages, storing in a text document, etc. Hence the creation of `utf8gen`.

3 Invoking utf8gen

3.1 Motivation

This chapter provides examples of typical uses for utf8gen for programmers and end-users.

I needed to generate hundreds of lines of source code containing different UTF-8 characters for a set of programs. My searches did not find anything that performed the conversion as I wanted, so I wrote utf8gen.

With the Unicode Standard specifying code point assignments in hexadecimal, it was natural to write software that took a hexadecimal number as input. There are numerous potential forms of output, especially considering the formatting syntax of different programming languages. The purpose of most of the options for utf8gen is to select various output options.

utf8gen reads in hexadecimal numbers, one per input line. Each number can be followed by a space and a miscellaneous string to the end of the line. That *remainder* string can optionally be printed on output; more on that later.

3.2 Printing a Character

The simplest thing an end-user might want to know is whether their computer has a font that supports a certain Unicode character. The easiest way to use utf8gen is interactively at a terminal, typing in hexadecimal numbers and looking at the character produced. To do this, run the command

```
utf8gen -c -n
```

The -c option tells utf8gen to print the input hexadecimal number as a Unicode character on the screen. The -n option tells utf8gen to *not* print the UTF-8 byte sequence as a set of formatted numbers. Just enter one hexadecimal number in the range 0 through 10FFFF, inclusive, one number per line. When finished running utf8gen interactively in this way, end your input by typing C-d.

3.3 Code Generation

3.3.1 The Usefulness of Octal

If converting hexadecimal numbers into a form that a programming language accepts, there are many possiblilties. For this reason, utf8gen accepts format strings in the style of the C printf function. This was a natural choice, as utf8gen is written in C.

With eight bits in a byte, and UTF-8 encoded characters starting either with a '0' bit for ASCII or with '10' for all but the first byte in a multi-byte sequence, it is convenient to look at Unicode code point numbers encoded as octal. If a byte in a UTF-8 byte string begins with '10', this leaves six bits for the remainder of the byte. This is conveniently viewed as two octal digits.

The default output of utf8gen is simply the sequence of octal digits in a UTF-8 character, printed in the C style of a backslash followed by three octal digits per byte. This is handy for a quick copy and paste of a single UTF-8 byte sequence into a program.

If using the C-style backslashed octal number format, it can be reassuring to see what a Unicode code point is in octal (at least it was for me, when I first wrote the program and was verifying its proper operation). A simple way of doing this is to have `utf8gen` echo the input hexadecimal number you typed in as octal, and then print the UTF-8 representation. To do this, run a command of the form

```
utf8gen -e "%03o = "
```

For example, if you enter the hexadecimal number *2134* (the Unicode code point for '`Script Small Letter O`' in the '`Letterlike Symbols`' block, `utf8gen` will generate this output:

```
20464 = \342\204\264
```

The hexadecimal number 2134 is 20464 in octal. Notice how two octal digits from the Unicode code point appear in each UTF-8 byte except for the first byte. The leading octal digit of '`2`' represents the leading two bits '`10`' in a UTF-8 multi-byte sequence. The first byte in a multi-byte UTF-8 sequence starts with a string of '`1`' bits indicating how many bytes long the encoded character is. In this case, the UTF-8 representation of U+2134 will take three bytes, so the first byte begins with the bit string '`1110`'. That corresponds to the first two octal digits ('`34`') of the first byte in the sequence, '`\342`'.

Looking at the sequence '`\342\204\264`' again, it is easy to see the placement of the octal representation of this Unicode code point, 20464. In this way, verifying the proper conversion of the hexadecimal Unicode code point to UTF-8 is straightforward.

3.3.2 Commenting Code

Commenting code is of course useful, especially when dealing with something as arcane as raw UTF-8 byte sequences. `utf8gen` provides various ways of doing this. A couple of examples should suffice to give you an idea of these capabilities.

The simplest method for creating comments might be to follow an octal sequence with the Unicode code point in its canonical form. The `-e` option *echoes* the input number to the output using the format string that follows. This will accomplish that:

```
utf8gen -e "/* U+%04X */ "
```

For the hexadecimal input number 2134, this produces the output

```
/* U+2134 */ \342\204\264
```

The expectation is that a programmer will be able to use an editor that can take a string like '`\342\204\264`' and easily convert it into a *print*-style command in the programming language of choice.

It might be preferable to print the comment after the UTF-8 byte sequence. The `-s` option allows this by *swapping* the default output string order. For example, the command

```
utf8gen -e " /* U+%04X */" -s
```

produces the output (again, using 2134 as the input number) of

```
\342\204\264 /* U+2134 */
```

It might even be useful to output the initial hexadecimal number using two different bases. This is accomplished with the `-E` option, followed by the format string for echoing the input number in two ways. For example, the command

```
utf8gen -E " /* U+%04X = 0%o */" -s
```

produces the output (with an input of '`2134`')

```
\342\204\264 /* U+2134 = 020464 */
```

3.3.3 Remainder Strings

One can only glean so much by looking at numbers though. A textual comment describing a Unicode code point can also help. `utf8gen` supports printing free-form text following an initial hexadecimal number followed by a space. This is done with the `-r` option, to print the *remainder* of the input line, using the format string that follows this option.

The Unicode Consortium makes various data files available with a free use license. The first field is usually the Unicode code point in hexadecimal. Remaining fields will contain information about each code point. For example, given the following line of input:

```
2134 SCRIPT SMALL O
```

This command

```
utf8gen -e " /* U+%04X " -s -r "%s */"
```

will produce this output:

```
\342\204\264 /* U+2134 SCRIPT SMALL O */
```

This can facilitate batch processing of large portions of a Unicode data file.

3.3.4 UTF-8 Output Format

`utf8gen` also allows specifying the format of the encoded UTF-8 bytes with the `-u` option followed by a format string. For example, suppose the programming language you use will accept bytes in hexadecimal using the form `\x` followed by a hexadecimal number. If we take the previous example input line and provide it to `utf8gen` with the comand

```
utf8gen -u "\x%02x" -r " /* %s */" -s
```

this will produce the output

```
\xe2\x84\xb4 /* SMALL SCRIPT O */
```

If the `-r` option is selected but there is nothing after the hexadecimal number on an input line, no remainder content will be printed.

Of course, you could also use the `-e` or `-E` options to echo back the input number in the desired output format(s) by adding it to the command line.

3.3.5 Input and Output Files

This information can be extracted and provided as an input file to `utf8gen` using the `-i` option to specify an input file. Output can be written to a file using the `-o` option.

3.4 Use Case Summary

The descriptions in this chapter give a brief overview of all of the `utf8gen` options and how they might be used in practice.

`utf8gen` tries to strike a balance between the basics that a programmer might find useful for bulk conversion of a large number of hexadecimal Unicode code points versus creeping featurism. While `utf8gen` won't write your program for you, it can make the bulk conversion of code points efficient.

4 utf8gen Reference

4.1 NAME

utf8gen − Generate UTF-8 output from hexadecimal input

4.2 SYNOPSIS

utf8gen [[-e *format1*] | [-E *format2*]] [-r *formatr*] [[-u *utf8_format*] | -n] [-c] [-s] [-i *input_file*] [-o *output_file*]

4.3 DESCRIPTION

utf8gen reads a list of hexadecimal ASCII values in the range 0 through 10FFFF, one per line, and prints the UTF-8 encoding of that number as a Unicode code point.

Each input line must begin with a hexadecimal number. A string may follow after that, which can be echoed to the output as the "remainder" (see the `-r` option below). The total input line length, including an ending newline, is limited to 4096 bytes.

4.4 OPTIONS

`-c` After the UTF-8 codes are printed, print a space followed by the character that the hexadecimal code point represents.

`-e` Echo the input code point in one format, using the printf(3) format string *format1*.

`-E` Echo the input code point in two formats, using the printf(3) format string *format2*.

`-n` Do *not* print the UTF-8 byte values. This can be useful if only the printed character itself is desired; see the `-c` option.

`-r` Print the remainder of the input string after the initial hexadecimal digits, using the printf(3) format string *formatr*.

`-s` Swap the order of output: print the UTF-8 output portion first, then print the input string portion. This can be useful for generating code containing a UTF-8 encoding followed by a comment that contains the input hexadecimal digits.

`-u` Print the UTF-8 encoded value of the input hexadecimal number, as numeric codes for each UTF-8 byte, using the printf(3) format string *utf8_format*. If no string is specified, a default format of a backslash followed by three octal digits is printed for each byte.

4.5 EXAMPLES

```
utf8gen -e "0x%04X " -u "\\%03o"
utf8gen -E "U+%04x = 0%02o = "
utf8gen -s -e " /* U+%04X */" -u "\\%03o"
```

4.6 FILES

Files contain lines that each begin with an ASCII hexadecimal code in the valid Unicode range 0 through 10FFFF, inclusive. This hexadecimal code may optionally be followed by a space followed by an arbitrary string ending with a newline, up to the limit of 4096 bytes per input line. An example line could be the following (with no indent):

    41 Letter 'A'

4.7 AUTHOR

utf8gen was written by Paul Hardy.

4.8 LICENSE

4.9 BUGS

No known bugs exist.