



De La Salle University - Manila

In Partial Fulfillment of the Course
Introduction to Computer Organization and
Architecture 2 (CSARCH2)

Unicode to UTF-8, UTF-16, and UTF-32 Converter

Technical Report

Submitted by:
Atanacio, Anjelo Patrick A.
Batac, Joel Ethan N.
Enclonar, Paul Ivan N.
Rimando, Alonzo Andrei G.

Submitted to:
Mr. Ronald Pascual

Term 3, A.Y. 2023-2024
July 31, 2024

I. Introduction

The Unicode to UTF Converter website application is designed to facilitate the conversion of Unicode characters into different Unicode Transformation Formats: UTF-8, UTF-16, and UTF-32. This tool is essential for developers and users who must ensure their text is properly encoded for various systems and applications.

Unicode

Unicode is a universal character encoding standard designed to represent text in different writing systems. It allows for consistent encoding, representation, and text handling across various platforms and applications. It was developed to address the limitations of older character encoding systems, like the ASCII (Uy, n.d.).

In Unicode, each character is assigned a unique number known as a code point, typically represented in the format "U+XXXXXX," where "XXXXXX" is a hexadecimal number. The Unicode code space ranges from U+000000 to U+10FFFF, encompassing over a million possible code points. This extensive range includes characters from virtually all writing systems, as well as a multitude of symbols and special-purpose characters.

The Unicode code space is divided into 17 planes, each containing 65,536 code points. The most significant of these is the Basic Multilingual Plane (BMP), which includes the majority of commonly used characters, such as those for modern scripts and many symbols, ranging from U+000000 to U+00FFFF. Beyond the BMP, the Supplementary Multilingual Plane (SMP), Supplementary Ideographic Plane (SIP), and Tertiary Ideographic Plane (TIP) accommodate additional characters for historical scripts, more symbols, and rare CJK (Chinese, Japanese, Korean) unified ideographs. The remaining 13 planes are currently unassigned and kept available to represent new characters in the future.

UTF-8

UTF-8, or Unicode Transformation Format - 8-bit, is a variable-length character encoding used for encoding all possible characters, or code points, in Unicode. It is widely used due to its efficiency and compatibility with ASCII, making it the dominant encoding for the web. Furthermore, UTF-8 encodes characters using one to four bytes (8-bit units). This allows it to represent a vast range of characters while minimizing the amount of space needed for common characters (Uy, n.d.).

The UTF-8 encoding scheme is structured as follows, based on the number of bits in the code point:

Bits of code point	First code point	Last code point	#of bytes	Byte 1	Byte 2	Byte 3	Byte 4
7	U+0000	U+007F	1	0xxxxxxx			
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-16

UTF-16, or Unicode Transformation Format - 16-bit, is a variable-length encoding used to represent Unicode characters. It uses one or two 16-bit code units, making it a versatile encoding scheme that can handle a wide range of characters efficiently.

The UTF-16 encoding scheme represents characters based on their code points. For code points in the range U+0000 to U+FFFF, the characters are represented as a single 16-bit code unit. These are directly encoded without any transformation (Uy, n.d.).

However, for code points in the range U+010000 to U+10FFFF, characters are represented using two 16-bit code units, known as a surrogate pair. The encoding process is as follows:

- Subtract 0x010000 from the code point, resulting in a 20-bit number in the range 0x00000 to 0xFFFFF.
- Divide the 20-bit number into two 10-bit halves:
 - Top ten bits: Represented by a number in the range 0x0000 to 0x03FF.
 - Low ten bits: Also in the range 0x0000 to 0x03FF.
- Add 0xD800 to the top ten bits to form the first 16-bit code unit (high surrogate), which will be in the range 0xD800 to 0xDBFF.
- Add 0xDC00 to the low ten bits to form the second 16-bit code unit (low surrogate), which will be in the range 0xDC00 to 0xDFFF.

UTF-32

UTF-32, or Unicode Transformation Format - 32-bit, is a fixed-length character encoding that uses exactly four bytes (32 bits) to represent each Unicode code point. This simplicity makes UTF-32 straightforward to implement and process, though it is less storage-efficient compared to UTF-8 and UTF-16 (Uy, n.d.).

Unlike UTF-8 and UTF-16, which use variable lengths to encode characters, UTF-32 uses a fixed length of four bytes for every character. This uniformity provides direct access to individual characters without the need for complex parsing logic.

In UTF-32, each character is represented by a single 32-bit code unit. The 32-bit unit can directly represent any Unicode code point in the range U+0000 to U+10FFFF. The code point value itself is used as the encoded value, ensuring a direct one-to-one correspondence between code points and their UTF-32 representation.

The website application uses the aforementioned parsing logic in converting Unicode to UTF-8, UTF-16, and UTF-32. It utilizes the javascript programming language, as well as its libraries, to translate the parsing logic of the Unicode Transformation Formats and return valid values.

II. Implementation

Tech Stack

The Unicode to UTF-8, UTF-16, and UTF-32 Converter application was designed to facilitate the seamless conversion of Unicode characters into three widely used encoding formats: UTF-8, UTF-16, and UTF-32. To achieve this, the developers selected a modern tech stack that includes React and Tailwind CSS for front-end development, JavaScript for the back-end, and Vite as the build tool.

React was chosen for its component-based architecture, and extensive community support, allowing for the modular development of user interfaces and efficient updating and rendering of components. Tailwind CSS was selected for its simplicity, as it made it easier for the developers to incorporate visual design into the website. Its built-in classes facilitated the creation of visually appealing layouts without incorporating complex styling.

On the back end, JavaScript was utilized to leverage its rich ecosystem of libraries and efficiency in handling computational logic for encoding conversions. This choice simplified the development and enhanced performance. Additionally, the developers also utilize Javascript JSX, an extension of Javascript. It enabled the developers to write and add HTML in React.

Vite, as the build tool with React, allowed for a fast development environment which helped the developers instantly visualize changes in the website. Additionally, it optimizes code compilation and execution, which can result in a better end-user experience due to faster loading times and lower resource usage. These features provided an optimized development and build experience, leading to a performant final product.

Input

The application is designed to accept two types of inputs. Firstly, users can input a Unicode point, which is automatically prefixed with "U+". This format is standard in Unicode representation, making it straightforward for those familiar with Unicode encoding to specify a character by its code point. Secondly, the application also allows users to directly input a character symbol. This is particularly useful for those who may not know the specific Unicode point of a character but want to see its encoded representations. When a character symbol is entered, the application automatically converts it into its corresponding Unicode point using the charToUnicode function. The input fields are shown below in Figure 0.

Input

Unicode

U+41

Symbol

A

Figure 0: Input Fields

Checking the Unicode Input

The converter application allows the user to input a Unicode code point via a text field. To ensure that the program works smoothly and could handle error inputs properly, the developers opted to create a function that checks the input if it is a valid Unicode code point or not. The code is shown in Figure 1.

```
function checkValid(unicode) {
  const validHex = /^(?-[1234567890abcdef])*$/i.test(unicode);
  if (!validHex) {
    return 'INVALID HEX'
  }

  if (parseInt(unicode, 16) > 0x10FFFF || parseInt(unicode, 16) < 0) {
    return 'OUT OF RANGE'
  }

  return true //return true if all tests are passed.
}
```

Figure 1: checkValid function

The checkValid function verifies if the input string is a valid Unicode point. This validation involves two main checks. First, it ensures that the input is a valid hexadecimal value. It uses a regular expression `/^(?-[1234567890abcdef])*$/i` to test if the string contains only valid hex digits (0-9, a-f). If the input fails this test, the function returns 'INVALID HEX'. Second, it checks if the hexadecimal value falls within the Unicode range, U+0000 to U+10FFFF. This is done by converting the string to its hex value using `parseInt(unicode, 16)` and comparing it against the specified range. If the value is out of range, the function returns 'OUT OF RANGE'. If both checks pass, the function returns true.

Converting the Unicode point input to UTF-8

One of the core features of the website is its ability to convert the Unicode input to UTF-8. The “unicodeToUTF8” function plays a significant role in the aforementioned feature. The function starts by checking if the input string is empty, returning an empty string if true. Then, it uses the checkValid function to validate the input. If the input is invalid, it returns the validation error.

Depending on the value of the Unicode point, it encodes the point using one of four cases:

One byte: If the Unicode point is less than or equal to 0x7F, the value is the same. However, if the Unicode point has less than two hexadecimal digits, it pads an additional 0 at the start.

```
if (parseInt(unicode, 16) <= 0x7F) {
  let binary = unicode.padStart(2, '0').toUpperCase();
  return binary //early return as this is already the valid answer
}
```

Figure 2: Logic for the case of one byte

Two bytes: If the Unicode point is between 0x80 and 0x7FF, the function converts the Unicode point to a binary number with 11 bits (padding zeroes at the start if necessary), and splits it into two groups. The first group starts with '110', followed by the first five binary digits, and the second starts with '10', followed by the last six binary digits.

```
else if (parseInt(unicode, 16) <= 0x7FF) {
  let binary = parseInt(unicode, 16).toString(2).padStart(11, '0');
  res.push('110' + binary.slice(0, 5));
  res.push('10' + binary.slice(5));
}
```

Figure 3: Logic for the case of Two bytes

Three bytes: For points between 0x800 and 0xFFFF, the function converts the Unicode point to a binary number with 16 bits (padding zeroes at the start if necessary), then split into three groups. The first group starts with '1110', and it is followed by the first four binary digits. The other two groups starts with '10', followed by the next six binary digits.

```
else if (parseInt(unicode, 16) <= 0xFFFF) {
  let binary = parseInt(unicode, 16).toString(2).padStart(16, '0');
  res.push('1110' + binary.slice(0, 4));
  res.push('10' + binary.slice(4, 10));
  res.push('10' + binary.slice(10));
}
```

Figure 4: Logic for the case of Three bytes

Four bytes: Points between 0x10000 and 0x1FFFF are converted to a binary number with 21 bits (padding zeroes at the start if necessary). Then, it is split into four groups. The first group starts with '11110', followed by the first three binary digits, and the other three groups start with '10', followed by the next six binary digits.

```
else if (parseInt(unicode, 16) <= 0x1FFFF) {
    let binary = parseInt(unicode, 16).toString(2).padStart(21, '0');
    res.push('11110' + binary.slice(0, 3));
    res.push('10' + binary.slice(3, 9));
    res.push('10' + binary.slice(9, 15));
    res.push('10' + binary.slice(15));
}
```

Figure 5: Logic for the case of Four bytes

Finally, the binary groups are converted to hexadecimal using `binaryStringToHex`, and the final UTF-8 encoded string is returned. However, if none of the conditions were met, the function returns “invalid”.

Converting the Unicode point input to UTF-16

The “`unicodeToUTF16`” function converts the Unicode point input to its UTF-16 encoding. The function starts by checking if the input string is empty or invalid using the `checkValid` function.

For inputs ranging from U+0000 to U+FFFF, the Unicode point is padded at the start with the necessary number of zeroes, so that the value still has 8 hexadecimal digits, as shown in Figure 6 below.

```
if (parseInt(unicode, 16) <= 0xFFFF) {
    let res = []
    unicode = unicode.padStart(4, '0');
    res.push(unicode.slice(0, 2))
    res.push(unicode.slice(2, 4))
    return res.join(' ').toUpperCase()
}
```

Figure 6: If input is from U+0000 to U+FFFF

For inputs ranging from U+010000 to U+10FFFF, the function calculates the high and low surrogates. It subtracts 0x10000 from the Unicode point, converts the result to a 20-bit binary string, and splits it into two 10-bit segments. The high surrogate is obtained by adding 0xD800 to the first 10 bits, and the low surrogate is obtained by adding 0xDC00 to the second 10 bits. The surrogates are then converted to hex and returned as the UTF-16 encoding. The code is shown in Figure 7 below.

```
else if (parseInt(unicode, 16) <= 0x10FFFF) {
    let res = []
    let hex = parseInt(unicode, 16) - 0x10000;
    let binary = hex.toString(2).padStart(20, '0');
    let highSurrogate = 0xD800 + parseInt(binary.slice(0, 10), 2);
    let lowSurrogate = 0xDC00 + parseInt(binary.slice(10), 2);
    res.push(highSurrogate.toString(16).toUpperCase());
    res.push(lowSurrogate.toString(16).toUpperCase());

    return res.map((hex) => [hex.slice(0, 2), hex.slice(2)]).flat().join(' ').toUpperCase();
}
```

Figure 7: If input is from U+010000 to U+10FFFF

Converting the Unicode point input to UTF-32

The “`unicodeToUTF32`” function converts a Unicode point to its UTF-32 encoding. It first checks if the input is empty or invalid using the `checkValid` function. It then converts the Unicode point to a hexadecimal number, pads it with zeroes at the start so that it becomes an 8 hexadecimal digit number, and splits it into groups of 2 characters each. The groups are joined together to form the final UTF-32 encoded string. The code is shown below in Figure 8.

```
function unicodeToUTF32(input) {
    let unicode = input

    if (unicode.length == 0) {
        return ''
    }
}
```

```
if (checkValid(unicode) !== true) {
  return checkValid(unicode);
}

unicode = parseInt(unicode, 16).toString(16) //this will remove leading 0's

let result = []

//zero extend unicode to 32 bits
unicode = unicode.padStart(8, '0')

//group unicode into two characters each then push to result
for (let i = 0; i < 8; i += 2) {
  result.push(unicode.slice(i, i + 2))
}

//convert result to string then return
return result.join(' ').toUpperCase()
}
```

Figure 8: converting to UTF-32

Output

The application provides a user-friendly interface where the UTF-8, UTF-16, and UTF-32 encodings of the inputted Unicode code point or character symbol are displayed in text fields on the website. When a user inputs a Unicode code point prefixed with "U+" or a character symbol, the application processes this input to generate its corresponding encodings. These encodings are then dynamically updated and displayed in the designated text fields. Each text field shows the encoded value in the respective format: UTF-8, UTF-16, or UTF-32. This real-time updating of text fields ensures that users can immediately see the results of their input, whether they are entering a code point or a character. This feature is particularly beneficial for users who need to verify or utilize different encoding formats, providing a clear and immediate visual representation of the encoded data. Additionally, the user can also copy the output to the clipboard of their device by clicking the corresponding buttons with a clipboard icon. The output fields are shown below in Figure 9.

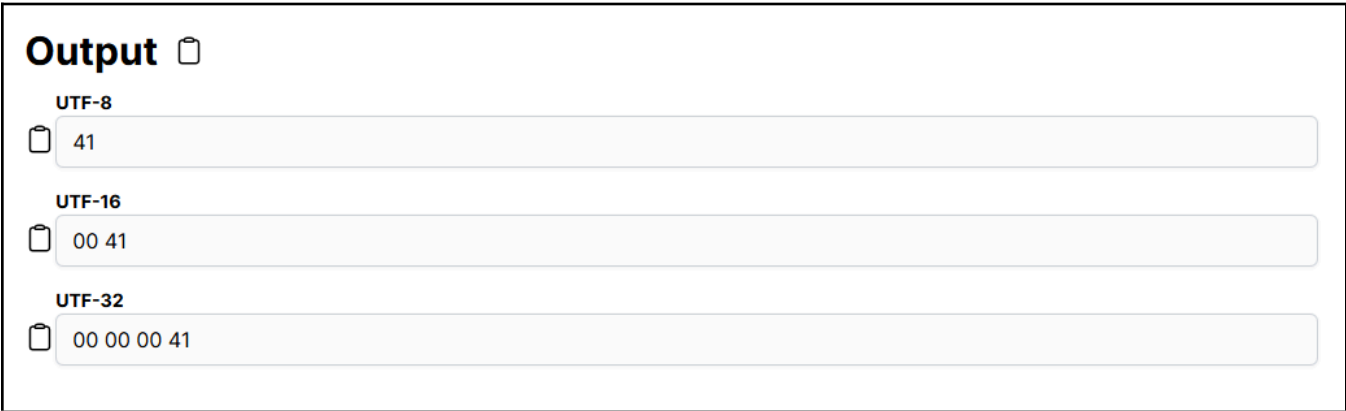


Figure 9: Output Fields

Other Implementations

Helper functions were also created with the aforementioned functions. They play a significant role in the process of converting Unicode points and characters to different encoding formats.

The `binaryStringToHex` function is responsible for converting a binary string representation of a number into its hexadecimal equivalent. It ensures that the binary string length is a multiple of 4 by padding it with leading zeros if necessary. This padding facilitates easier conversion as each group of 4 binary digits corresponds directly to a single hexadecimal digit. The function then splits the padded binary string into groups of 4 bits. Each 4-bit group is converted to its hexadecimal equivalent using `parseInt`, which parses the binary string as a base-2 number and then converts it to a hexadecimal string. Finally, these hexadecimal digits are joined together into a single string and converted to uppercase for consistency. This function is crucial in encoding processes, particularly in converting binary data derived from Unicode points into a more compact and human-readable hexadecimal format, which is used in UTF-8, UTF-16, and UTF-32 encodings.

The `unicodeToChar` function converts a given Unicode point to its corresponding character. It first checks if the input string is empty and returns an empty string if true. The function then validates the Unicode point using the `checkValid` function to ensure it is a legitimate Unicode code point. If the input passes validation, the function converts the hexadecimal Unicode point to its corresponding character using `String.fromCharCode`. This method

handles a wide range of Unicode points, including those beyond the Basic Multilingual Plane. Special cases, such as the null character, are handled explicitly to avoid issues. This function is essential for displaying the character represented by a Unicode point, which is a fundamental step in encoding and decoding processes involving character data.

The charToUnicode function converts a given character to its Unicode point. It takes a single character as input and returns its Unicode code point in hexadecimal format. The function uses codePointAt(0) to obtain the Unicode code point of the character, which it then converts to a hexadecimal string and ensures it is in uppercase for consistency. This function is useful for determining the Unicode representation of a character, which is a necessary step in various encoding processes. By converting characters to their Unicode points, it facilitates the subsequent encoding into formats like UTF-8, UTF-16, and UTF-32.

The formattedUTF function creates a formatted string containing all relevant information about a Unicode point and its corresponding encodings. It takes the Unicode point, the character represented by that point, and the encodings in UTF-8, UTF-16, and UTF-32 as input. The function then constructs a string that includes each of these elements, formatted for easy reading and copying. This formatted string provides a comprehensive view of the Unicode point and its various representations, making it a useful tool for understanding and verifying the encoding processes. It encapsulates the results of the conversions and presents them in a clear, organized manner, which is particularly helpful for debugging, documentation, and educational purposes.

Together, these functions enable the conversion of characters to their Unicode points and the subsequent encoding of these points into UTF-8, UTF-16, and UTF-32 formats, supporting comprehensive encoding and decoding operations.

III. Test Cases

This section of the paper tackles how the website’s features are tested with various inputs that are valid, invalid, and special. Additionally, it also shows how certain errors are handled, as well as edge cases.

Test Case 1: Basic Multilingual Plane (BMP) Character

- Input: ñ
- Expected Output:
 - Unicode Code Point: U+F1
 - UTF-8: C3 B1
 - UTF-16: 00 F1
 - UTF-32: 00 00 00 F1
- Actual Output:

Input

Unicode

U+F1

Symbol

ñ

Output

UTF-8

C3 B1

UTF-16

00 F1

UTF-32

00 00 00 F1

- Manual Computation:

UTF-8

- For code points from U+0080 to U+07FF, UTF-8 uses 2 bytes. U+00F1 falls in this range.
- U+00F1 in binary: 0000 0000 1111 0001
- The 2-byte format for UTF-8 is: 110xxxxx 10xxxxxx
- Split the binary representation: 1111 0001 → 00011110 000001
- Encode these into the 2-byte format:
 - Byte 1: 11000011 (0xC3)
 - Byte 2: 10110001 (0xB1)
- So, U+00F1 in UTF-8 is 0xC3 B1.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes.
- U+00F1 in binary: 0000 0000 1111 0001
- Encode directly into UTF-16: 0000 0000 1111 0001
- So, U+00F1 in UTF-16 is 0x00 F1.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+00F1 in binary: 0000 0000 0000 0000 0000 0000 1111 0001
- Encode directly into UTF-32: 0000 0000 0000 0000 0000 0000 1111 0001
- So, U+00F1 in UTF-32 is 0x00 00 00 F1.

- Pass/Fail: PASS

Test Case 2: Devanagari Letter A

- Input: अ
- Expected Output:
 - Unicode Code Point: U+0905
 - UTF-8: E0 A4 85
 - UTF-16: 09 05
 - UTF-32: 00 00 09 05
- Actual Output:

Input

Unicode

U+905

Symbol

अ

Output

UTF-8

E0 A4 85

UTF-16

09 05

UTF-32

00 00 09 05

- Manual Computation:

UTF-8

- For code points from U+0800 to U+FFFF, UTF-8 uses 3 bytes. U+0905 falls in this range.
- U+0905 in binary: 0000 1001 0000 0101
- The 3-byte format for UTF-8 is: 1110xxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 0000 1001 0000 0101 → 0000| 1001 00|00 0101
- Encode these into the 3-byte format:
 - Byte 1: 11100000 (0xE0)
 - Byte 2: 10100100 (0xA4)
 - Byte 3: 10000101 (0x85)
- So, U+0905 in UTF-8 is 0xE0 A4 85.

UTF-16


- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes.
- U+0905 in binary: 0000 1001 0000 0101
- Encode directly into UTF-16: 0000 1001 0000 0101
- So, U+0905 in UTF-16 is 0x09 05.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+0905 in binary: 0000 0000 0000 0000 0000 1001 0000 0101
- Encode directly into UTF-32: 0000 0000 0000 0000 0000 1001 0000 0101
- So, U+0905 in UTF-32 is 0x00 00 09 05.

- Pass/Fail: PASS

Test Case 3: Musical Symbol G Clef


- Input: 
- Expected Output:
 - Unicode Code Point: U+1D11E
 - UTF-8: F0 9D 84 9E
 - UTF-16: D8 34 DD 1E
 - UTF-32: 00 01 D1 1E
- Actual Output:

Input

Unicode

Symbol

U+1D11E



Output

UTF-8

F0 9D 84 9E

UTF-16

D8 34 DD 1E

UTF-32

00 01 D1 1E


- Manual Computation:

UTF-8
 - For code points from U+10000 to U+1FFFFFF, UTF-8 uses 4 bytes. U+1D11E falls in this range.
 - U+1D11E in binary: 0001 1101 0001 0001 1110
 - The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
 - Split the binary representation: 0001 1101 0001 0001 1110 → 0 00|01 1101| 0001 00|01 1110
 - Encode these into the 4-byte format:
 - Byte 1: 11110000 (0xF0)
 - Byte 2: 10011101 (0x9D)
 - Byte 3: 10000100 (0x84)
 - Byte 4: 10011110 (0x9E)
 - So, U+1D11E in UTF-8 is 0x F0 9D 84 9E

UTF-16
 - For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
 - U+1D11E in binary: 0001 1101 0001 0001 1110
 - Subtract 0x010000 from the code point: 1D11E - 10000 = 0x0D11E (0000 1101 0001 0001 1110)
 - Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 0000 1101 00 (0x34)
 - Low ten bits: 01 0001 1110 (0x11E)
 - Add 0xD800 to the top ten bits to form the high surrogate: 0x03 + 0xD800 = 0xD834
 - Add 0xDC00 to the low ten bits to form the low surrogate: 0x11E + 0xDC00 = 0xDD1E
 - So, U+1D11E in UTF-16 is 0xD8 34 DD 1E.

UTF-32
 - For any code point, UTF-32 uses 4 bytes.
 - U+1D11E in binary: 0000 0000 0001 1101 0001 0001 1110
 - Encode directly into UTF-32: 0000 0000 0001 1101 0001 0001 1110
 - So, U+1D11E in UTF-32 is 0x00 01 D1 1E.
- Pass/Fail: PASS

Test Case 4: CJK Ideograph Extension B

- Input: 
- Expected Output:
 - Unicode Code Point: U+20000
 - UTF-8: F0 A0 80 80
 - UTF-16: D8 40 DC 00
 - UTF-32: 00 02 00 00

- Actual Output:

Input

Unicode


U+20000

Symbol


𐀀

Output


UTF-8

 F0 A0 80 80

UTF-16

 D8 40 DC 00

UTF-32

 00 02 00 00

- Manual Computation:

UTF-8

- For code points from U+10000 to U+1FFFFFF, UTF-8 uses 4 bytes. U+20000 falls in this range.
- U+20000 in binary: 0010 0000 0000 0000 0000
- The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 0010 0000 0000 0000 0000 → 0 00|10 0000| 0000 00|00 0000
- Encode these into the 4-byte format:
 - Byte 1: 11110000 (0xF0)
 - Byte 2: 10100000 (0xA0)
 - Byte 3: 10000000 (0x80)
 - Byte 4: 10000000 (0x80)
- So, U+20000 in UTF-8 is 0xF0 A0 80 80.

UTF-16

- For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
- U+20000 in binary: 0010 0000 0000 0000 0000
- Subtract 0x010000 from the code point: 20000 - 10000 = 0x10000
- Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 0000 0001 00 (0x0040)
 - Low ten bits: 00 0000 0000 (0x0000)
- Add 0xD800 to the top ten bits to form the high surrogate: 0x0040 + 0xD800 = 0xD840
- Add 0xDC00 to the low ten bits to form the low surrogate: 0x0000 + 0xDC00 = 0xDC00
- So, U+20000 in UTF-16 is 0xD8 40 DC 00

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+20000 in binary: 0000 0000 0010 0000 0000 0000 0000
- Encode directly into UTF-32: 0000 0000 0010 0000 0000 0000 0000
- So, U+20000 in UTF-32 is 0x00 02 00 00

- Pass/Fail: PASS

Test Case 5: CJK Ideograph Extension B

- Input: A
- Expected Output:
 - Unicode Code Point: U+1D538
 - UTF-8: F0 9D 94 B8
 - UTF-16: D8 35 DD 38
 - UTF-32: 00 01 D5 38
- Actual Output:

Input

Unicode

Symbol

U+1D538

A

Output

UTF-8

F0 9D 94 B8

UTF-16

D8 35 DD 38

UTF-32

00 01 D5 38

- Manual Computation:

UTF-8

- For code points from U+10000 to U+1FFFFFF, UTF-8 uses 4 bytes. U+1D538 falls in this range.
- U+1D538 in binary: 0001 1101 0101 0011 1000
- The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 0001 1101 0101 0011 1000 → 0 00|01 1101| 0101 00|11 1000
- Encode these into the 4-byte format:
 - Byte 1: 11110000 (0xF0)
 - Byte 2: 10011101 (0x9D)
 - Byte 3: 10010100 (0x94)
 - Byte 4: 10111000 (0xB8)
- So, U+1D538 in UTF-8 is 0xF0 9D 94 B8.

UTF-16

- For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
- U+1D538 in binary: 0001 1101 0101 0011 1000
- Subtract 0x010000 from the code point: 1D538 - 10000 = 0x0D538 (0000 1101 0101 0011 1000)
- Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 0000 1101 01 (0x035)
 - Low ten bits: 01 0011 1000 (0x138)
- Add 0xD800 to the top ten bits to form the high surrogate: 0x0035 + 0xD800 = 0xD835
- Add 0xDC00 to the low ten bits to form the low surrogate: 0x138 + 0xDC00 = 0xDD38
- So, U+1D538 in UTF-16 is 0xD8 35 DD 38.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+1D538 in binary: 0000 0000 0001 1101 0101 0011 1000
- Encode directly into UTF-32: 0000 0000 0001 1101 0101 0011 1000
- So, U+1D538 in UTF-32 is 0x00 01 D5 38.

- Pass/Fail: PASS

Test Case 6: Astral Planes Character

- Input: ☐
- Expected Output:
 - Unicode Code Point: U+2F809
 - UTF-8: F0 AF A0 89
 - UTF-16: D8 7E DC 09
 - UTF-32: 00 02 F8 09
- Actual Output:

Input

Unicode

U+2F809

Symbol

☐

Output

UTF-8

☐ F0 AF A0 89

UTF-16

☐ D8 7E DC 09

UTF-32

☐ 00 02 F8 09

- Manual Computation:

UTF-8

- For code points from U+10000 to U+1FFFFF, UTF-8 uses 4 bytes. U+2F809 falls in this range.
- U+2F809 in binary: 00101111 10000000 1001
- The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 00101111 10000000 1001 → 0 00|10 1111| 1000 00|00 1001
- Encode these into the 4-byte format:
 - Byte 1: 11110000 (0xF0)
 - Byte 2: 10101111 (0xAF)
 - Byte 3: 10100000 (0xA0)
 - Byte 4: 10001001 (0x89)
- So, U+2F809 in UTF-8 is 0xF0 AF A0 89.

UTF-16

- For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
- U+2F809 in binary: 00101111 10000000 1001
- Subtract 0x010000 from the code point: 2F809 - 10000 = 0x1F809 (0001 1111 1000 0000 1001)
- Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 0001 1111 10 (0x007E)
 - Low ten bits: 00 0000 1001 (0x009)
- Add 0xD800 to the top ten bits to form the high surrogate: 0x007E + 0xD800 = 0xD87E
- Add 0xDC00 to the low ten bits to form the low surrogate: 0x009 + 0xDC00 = 0xDC09
- So, U+2F809 in UTF-16 is 0xD8 7E DC 09.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+2F809 in binary: 0000 0000 0010 1111 1000 0000 1001
- Encode directly into UTF-32: 0000 0000 0010 1111 1000 0000 1001
- So, U+2F809 in UTF-32 is 0x0002F809.

- Pass/Fail: PASS

Test Case 7: Smiling Face

- Input: 😊
- Expected Output:
 - Unicode Code Point: U+1F642
 - UTF-8: F0 9F 99 82
 - UTF-16: D8 3D DE 42
 - UTF-32: 00 01 F6 42

- Actual Output:

Input

Unicode

U+1F642

Symbol

😂

Output

UTF-8

F0 9F 99 82

UTF-16

D8 3D DE 42

UTF-32

00 01 F6 42

- Manual Computation:

UTF-8

- For code points from U+10000 to U+1FFFFFF, UTF-8 uses 4 bytes. U+1F642 falls in this range.
- U+1F642 in binary: 0001 1111 0110 0100 0010
- The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 0001 1111 0110 0100 0010 → 0 00|01 1111| 0110 01|00 0010
- Encode these into the 4-byte format:
 - Byte 1: 11110000 (0xF0)
 - Byte 2: 10011111 (0x9F)
 - Byte 3: 10011001 (0x99)
 - Byte 4: 10000010 (0x82)
- So, U+1F642 in UTF-8 is 0xF0 9F 99 82.

UTF-16

- For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
- U+1F642 in binary: 0001 1111 0110 0100 0010
- Subtract 0x010000 from the code point: 1F642 - 10000 = 0x0F642 (0000 1111 0110 0100 0010)
- Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 0000 1111 01 (0x3D)
 - Low ten bits: 10 0100 0010 (0x242)
- Add 0xD800 to the top ten bits to form the high surrogate: 0x003D + 0xD800 = 0xD83D
- Add 0xDC00 to the low ten bits to form the low surrogate: 0x242 + 0xDC00 = 0xDE42
- So, U+1F642 in UTF-16 is 0xD8 3D DE 42

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+1F642 in binary: 0000 0000 0001 1111 0110 0100 0010
- Encode directly into UTF-32: 0000 0000 0001 1111 0110 0100 0010
- So, U+1F642 in UTF-32 is 0x00 01 F6 42

- Pass/Fail: PASS

Test Case 8: Null Character

- Input: \0
- Expected Output:
 - Unicode Code Point: U+0000
 - UTF-8: 00
 - UTF-16: 00 00
 - UTF-32: 00 00 00 00
- Actual Output:

Input

Unicode

U+0000

Symbol

Null Character

Output

UTF-8

00

UTF-16

00 00

UTF-32

00 00 00 00

- Manual Computation:

UTF-8

- For code points from U+0000 to U+007F, UTF-8 uses 1 byte.
- U+0000 in binary: 0000 0000
- The 1-byte format for UTF-8 is: 0xxxxxxx
- Since the value is 0000 0000, it fits directly into the 1-byte format.
- So, U+0000 in UTF-8 is 0x00.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes.
- U+0000 in binary: 0000 0000 0000 0000
- Encode directly into UTF-16: 0000 0000 0000 0000
- So, U+0000 in UTF-16 is 0x00 00.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+0000 in binary: 0000 0000 0000 0000 0000 0000 0000 0000
- Encode directly into UTF-32: 0000 0000 0000 0000 0000 0000 0000 0000
- So, U+0000 in UTF-32 is 0x00 00 00 00

- Pass/Fail: PASS

Test Case 9: Maximum Code Point

- Input: U+10FFFF
- Expected Output:
 - Unicode Code Point: U+10FFFF
 - UTF-8: F4 8F BF BF
 - UTF-16: DB FF DF FF
 - UTF-32: 00 10 FF FF
- Actual Output:

Input

Unicode

U+10FFFF

Symbol

☰

Output

☐

UTF-8

☐ F4 8F BF BF

UTF-16

☐ DB FF DF FF

UTF-32

☐ 00 10 FF FF

- Manual Computation:

UTF-8

- For code points from U+10000 to U+10FFFF, UTF-8 uses 4 bytes. U+10FFFF falls in this range.
- U+10FFFF in binary: 0001 0000 1111 1111 1111 1111
- The 4-byte format for UTF-8 is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 1 0000 1111 1111 1111 1111 → 1 00|00 1111| 1111 11|11 1111
- Encode these into the 4-byte format:
 - Byte 1: 11110100 (0xF4)
 - Byte 2: 10001111 (0x8F)
 - Byte 3: 10111111 (0xBF)
 - Byte 4: 10111111 (0xBF)
- So, U+10FFFF in UTF-8 is 0xF4 8F BF BF.

UTF-16

- For code points from U+010000 to U+10FFFF, UTF-16 uses 4 bytes (2 surrogate pairs).
- U+10FFFF in binary: 0001 0000 1111 1111 1111 1111
- Subtract 0x010000 from the code point: 10FFFF - 10000 = 0x0FFFFF (1111 1111 1111 1111 1111)
- Split the resulting 20-bit number into two 10-bit halves:
 - Top ten bits: 1111 1111 11 (0x03FF)
 - Low ten bits: 11 1111 1111 (0x03FF)
- Add 0xD800 to the top ten bits to form the high surrogate: 0x03FF + 0xD800 = 0xDBFF
- Add 0xDC00 to the low ten bits to form the low surrogate: 0x03FF + 0xDC00 = 0xDFFF
- So, U+10FFFF in UTF-16 is 0xDB FF DF FF.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+10FFFF in binary: 0000 0000 0001 0000 1111 1111 1111 1111
- Encode directly into UTF-32: 0000 0000 0001 0000 1111 1111 1111 1111
- So, U+10FFFF in UTF-32 is 0x00 10 FF FF.

- Pass/Fail: PASS

Test Case 10: Invalid Code Point

- Input: U+110000
- Expected Output:
 - Unicode Code Point: U+110000
 - UTF-8: OUT OF RANGE
 - UTF-16: OUT OF RANGE
 - UTF-32: OUT OF RANGE
- Actual Output:

Input

Unicode

U+110000

Symbol

OUT OF RANGE

Output

UTF-8

OUT OF RANGE

UTF-16

OUT OF RANGE

UTF-32

OUT OF RANGE

- Pass/Fail: PASS

Test Case 11: Maximum Code Point

- Input: U+0001
- Expected Output:
 - Unicode Code Point: U+0001
 - UTF-8: 01
 - UTF-16: 00 01
 - UTF-32: 00 00 00 01
- Actual Output:

Input

Unicode

U+0001

Symbol

☐

Output

UTF-8

01

UTF-16

00 01

UTF-32

00 00 00 01

- Manual Computation:

UTF-8

- For code points from U+0000 to U+007F, UTF-8 uses 1 byte.
- U+0001 in binary: 0000 0001
- The 1-byte format for UTF-8 is: 0xxxxxxx
- Since the value is 0000 0001, it fits directly into the 1-byte format.
- So, U+0001 in UTF-8 is 0x01.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes.
- U+0001 in binary: 0000 0000 0000 0001
- Encode directly into UTF-16: 0000 0000 0000 0001
- So, U+0001 in UTF-16 is 0x00 01.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+0001 in binary: 0000 0000 0000 0000 0000 0000 0000 0001
- Encode directly into UTF-32: 0000 0000 0000 0000 0000 0000 0000 0001
- So, U+0001 in UTF-32 is 0x00 00 00 01

- Pass/Fail: PASS

Test Case 12: Bengali Letter A (Complex Script Character)

- Input: অ
- Expected Output:
 - Unicode Code Point: U+0985
 - UTF-8: E0 A6 85
 - UTF-16: 09 85
 - UTF-32: 00 00 09 85
- Actual Output:

Input

Unicode

U+985

Symbol

অ

Output

UTF-8

E0 A6 85

UTF-16

09 85

UTF-32

00 00 09 85

- Manual Computation:

UTF-8

- For code points from U+0800 to U+FFFF, UTF-8 uses 3 bytes. U+0985 falls in this range.
- U+0985 in binary: 0000 1001 1000 0101
- The 3-byte format for UTF-8 is: 1110xxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 0000 1001 1000 0101 → 0000| 1001 10|00 0101
- Encode these into the 3-byte format:
 - Byte 1: 11100000 (0xE0)
 - Byte 2: 10100110 (0xA6)
 - Byte 3: 10000101 (0x85)
- So, U+0985 in UTF-8 is 0xE0 0xA6 0x85.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes. U+0985 falls in this range.
- U+0985 in binary: 0000 1001 1000 0101
- Encode directly into UTF-16: 0000 1001 1000 0101
- So, U+0985 in UTF-16 is 0x0985.

UTF-32

For any code point, UTF-32 uses 4 bytes.

U+0985 in binary: 0000 0000 0000 0000 0000 1001 1000 0101

Encode directly into UTF-32: 0000 0000 0000 0000 0000 1001 1000 0101

So, U+0985 in UTF-32 is 0x00000985

- Pass/Fail: PASS

Test Case 13: Hebrew Letter Alef (Right-to-Left Character)

- Input: א
- Expected Output:
 - Unicode Code Point: U+05D0
 - UTF-8: D7 90
 - UTF-16: 05 D0
 - UTF-32: 00 00 05 D0
- Actual Output:

Input

Unicode

Symbol

U+5D0

א

Output

UTF-8

D7 90

UTF-16

05 D0

UTF-32

00 00 05 D0

- Manual Computation:

UTF-8

- For code points from U+0080 to U+07FF, UTF-8 uses 2 bytes. U+05D0 falls in this range.
- U+05D0 in binary: 0000 0101 1101 0000
- The 2-byte format for UTF-8 is: 110xxxxx 10xxxxxx
- Split the binary representation: 101 1101 0000 → 101 11|01 0000
- Encode these into the 2-byte format:
 - Byte 1: 11010111 (0xD7)
 - Byte 2: 10010000 (0x90)
- So, U+05D0 in UTF-8 is 0xD7 90.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes. U+05D0 falls in this range.
- U+05D0 in binary: 0000 0101 1101 0000
- Encode directly into UTF-16: 0000 0101 1101 0000
- So, U+05D0 in UTF-16 is 0x05 D0.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+05D0 in binary: 0000 0000 0000 0000 0000 0101 1101 0000
- Encode directly into UTF-32: 0000 0000 0000 0000 0000 0101 1101 0000
- So, U+05D0 in UTF-32 is 0x00 00 05 D0.

- Pass/Fail: PASS

Test Case 14: Private Use Area Character

- Input: U+E000
- Expected Output:
 - Unicode Code Point: U+E000
 - UTF-8: EE 80 80
 - UTF-16: E0 00
 - UTF-32: 00 00 E0 00
- Actual Output:

Input

Unicode

U+E000

Symbol

𐄀

Output

UTF-8

EE 80 80

UTF-16

E0 00

UTF-32

00 00 E0 00

- Manual Computation:

UTF-8

- For code points from U+0800 to U+FFFF, UTF-8 uses 3 bytes. U+E000 falls in this range.
- U+E000 in binary: 1110 0000 0000 0000
- The 3-byte format for UTF-8 is: 1110xxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 1110 0000 0000 0000 → 1110| 0000 00|00 0000
- Encode these into the 3-byte format:
 - Byte 1: 11101110 (0xEE)
 - Byte 2: 10000000 (0x80)
 - Byte 3: 10000000 (0x80)
- So, U+E000 in UTF-8 is 0xEE 0x80 0x80.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes. U+E000 falls in this range.
- U+E000 in binary: 1110 0000 0000 0000
- Encode directly into UTF-16: 1110 0000 0000 0000
- So, U+E000 in UTF-16 is 0xE0 00

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+E000 in binary: 0000 0000 0000 0000 1110 0000 0000 0000
- Encode directly into UTF-32: 0000 0000 0000 0000 1110 0000 0000 0000
- So, U+E000 in UTF-32 is 0x00 00 E0 00

- Pass/Fail: PASS

Test Case 15: Non-Character Code Point

- Input: U+FDD0
- Expected Output:
 - Unicode Code Point: U+FDD0
 - UTF-8: EF B7 90
 - UTF-16: FD D0
 - UTF-32: 00 00 FD D0
- Actual Output:

Input

Unicode

U+FDD0

Symbol

☰

Output

UTF-8

☐ EF B7 90

UTF-16

☐ FD D0

UTF-32

☐ 00 00 FD D0

- Manual Computation:

UTF-8

- For code points from U+0800 to U+FFFF, UTF-8 uses 3 bytes. U+FDD0 falls in this range.
- U+FDD0 in binary: 1111 1101 1101 0000
- The 3-byte format for UTF-8 is: 1110xxxx 10xxxxxx 10xxxxxx
- Split the binary representation: 1111 1101 1101 0000 → 1111| 1101 11|01 0000
- Encode these into the 3-byte format:
 - Byte 1: 11101111 (0xEF)
 - Byte 2: 10110111 (0xB7)
 - Byte 3: 10010000 (0x90)
- So, U+FDD0 in UTF-8 is 0xEF B7 90.

UTF-16

- For code points from U+0000 to U+FFFF, UTF-16 uses 2 bytes. U+FDD0 falls in this range.
- U+FDD0 in binary: 1111 1101 1101 0000
- Encode directly into UTF-16: 1111 1101 1101 0000
- So, U+FDD0 in UTF-16 is 0xFD D0.

UTF-32

- For any code point, UTF-32 uses 4 bytes.
- U+FDD0 in binary: 0000 0000 0000 0000 1111 1101 1101 0000
- Encode directly into UTF-32: 0000 0000 0000 0000 1111 1101 1101 0000
- So, U+FDD0 in UTF-32 is 0x00 00 FD D0

- Pass/Fail: PASS

IV. Challenges Encountered

The team faced several challenges in implementing Unicode conversion to UTF-8, UTF-16, and UTF-32 that required careful consideration and problem-solving. These include:

Special Cases

The first challenge the team addressed was the conversion of large code points, which was difficult due to the learning curve associated with the encoding schemes. Understanding how to manage code points above the Basic Multilingual Plane (BMP) required creating functions to handle surrogate pairs in UTF-16 and multi-byte sequences in UTF-8. For instance, while we knew the limits for the different planes, we had to implement functions that correctly convert inputs beyond these limits. This included correctly handling the byte shifts and ensuring accurate representation in all formats.

Another challenge was dealing with non-character code points such as U+FDD0. Although these are valid Unicode points, they are not assigned to any character and require special handling in the conversion process. This involved ensuring that the conversion functions treat these code points appropriately and consistently across all formats.

V. Conclusion

In this paper, we have explored the complexities and solutions associated with Unicode conversion across UTF-8, UTF-16, and UTF-32 encoding schemes. The implementation of these encoding formats is crucial for ensuring the accurate representation and manipulation of text across diverse platforms and applications.

Our analysis revealed several key challenges, including handling large code points and non-character code points. The solutions developed for these challenges not only addressed specific technical hurdles but also contributed to a more robust and flexible Unicode conversion process. By designing functions to manage surrogate pairs in UTF-16, multi-byte sequences in UTF-8, and ensuring consistent handling of special and invalid inputs, we enhanced the overall reliability and accuracy of the conversion process.

In conclusion, this paper underscores the significance of addressing the intricate details of Unicode conversion to facilitate seamless text processing and representation. The implementations discussed provide valuable insights into handling complex encoding scenarios, ensuring that text data is accurately represented and accessible across various systems and platforms. Future work in this field may further refine these processes and explore additional encoding challenges, contributing to the continued advancement of text encoding technologies.

VI. References

- Uy, S. (n.d.). Character Data Representation [PowerPoint slides]. College of Computer Studies, De La Salle University Manila.
https://dlsu.instructure.com/courses/169351/pages/module-5-additional-resources-2?module_item_id=4609419