CS 520
Characters, String & Encoding

6-bit Encoding

- CDC 6600
- Only 6 bits are available for a total of 64!
- Couldn’t encode several special characters, lowercase alphabets
- Designers weren’t concerned about text processing, not so much number crunching

ASCII table – 7 bit encoding

Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Extended ASCII – 8 bit

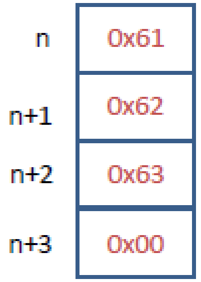
Extended ASCII

ASCII Table | Simple ASCII | Plain Text Chart

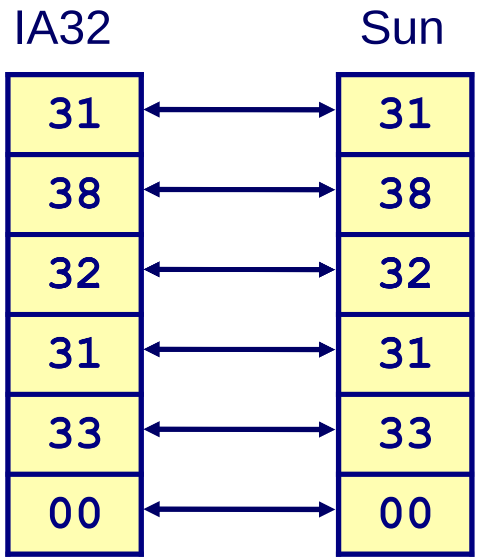
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	^{n}
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	^{2}
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Representing Strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = NULL
- Compatibility between Little Endian & Big Endian machines
- Byte ordering not an issue
- String ”abc” in memory is:




```
char S[6] = "18213";
```



Unicode

- ***Unicode is a character encoding system that covers all writing systems in use***
- ***Also includes several historical systems***

- The space of values is divided into 17 ***planes***.
- Plane 0 is the Basic Multilingual Plane (BMP)
  - Supports nearly all modern languages
  - Encodings are 0x0000-0xFFFF
  - 16 bits (2 bytes)
- Planes 1-16 are supplementary planes
  - Supports historic scripts, musical mathematical and other special symbols
  - Encodings are 0x100000-0x10FFFF
  - 21 bits
- Planes are divided into ***blocks*** or ***named ranges***

Unicode and ASCII

- ASCII is the bottom block in the BMP, known as the Basic Latin block
- ASCII values are embedded “as is” into Unicode
  - i.e. ‘a’ is 0x61 in ASCII and 0x0061 in Unicode

Special Encodings in Unicode

- The Byte-Order Mark (BOM) is used to signal endian-ness (byte order of a text file containing Unicode)
- Unicode character in general will not fit into a (8 bit) byte so Endian-ness is an issue when we store Unicode in memory
- A BOM has no other meaning and therefore usually ignored
- Encoded as 0xFEFF
- 0xFFFE is a noncharacter
  - Cannot legally appear in any exchange of Unicode
- If a text file in encoded in Unicode and the first 2 bytes is a BOM, the reader can use this to figure out the endian-ness of the file

Special Encodings in Unicode

- If the first byte is 0xFE and the next byte is 0xFF then we know that the byte order is Big Endian
- If the first byte is 0xFF and the next byte is 0xFE then we know that the byte order is Little Endian
- Since 2 bytes are not legal in Unicode, either presence 0xFEFF or 0xFFFE as the first 2 bytes can only represent the BOM
- In the absence of a BOM, Big Endian is assumed

- The **UTF-8 BOM** is a sequence of Bytes at the start of a text-stream (EF BB BF) that allows the reader to more reliably guess a file as being encoded in **UTF-8**. Normally, the **BOM** is used to signal the endianness of an encoding, but since endianness is irrelevant to **UTF-8**, the **BOM** is unnecessary

Other Non-characters

- There are a total of 66 non-characters (some of them are used for transformation of Unicode characters):
  - 0xFFEE and 0xFFFF of the BMP
  - 0x1FFEE and 0x1FFFF of plane 1
  - 0x2FFEE and 0x2FFFF of plane 2
  - -etc. up to
  - 0x10FFFE and 0x10FFFF of plane 16
  - Also 0xFDD0-0xFDDF of the BMP

UTF – UCS*Transformation Format

- **U**niversal **C**haracter **S**et – character set for Unicode
- UTF-8
  - Encodes Unicode characters in 1-4 bytes
  - ASCII gets encoded as 1 byte
  - Dominant character encoding for the WWW
- UTF-16
  - Encodes BMP characters in 2 bytes
  - Encodes non-BMP characters in 4 bytes
- UTF-32
  - Fixed-size representation of Unicode
  - Wastes some bits
  - Maximum of 21 bits needed for Unicode character but we are storing it always in 32 bits thus wasting 11 bits

UTF-8

- Take the Unicode character and throw away the leading zero bits*
- Count the remaining number of bits
- 7 bit: 0xxxxxxx
- 11 bits: 110xxxxx 10xxxxxx
- 16 bits: 1110xxxx 10xxxxxx 10xxxxxx
- 21 bits: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

*Overlong encodings are forbidden. Therefore there is a unique UTF-8 encoding for each Unicode character.

In other words, although UTF-8 theoretically allows for different representations of characters that also have a shorter one. For example, you could encode an ASCII character in two bytes by setting the MSBs to zero. The UTF-8 specification explicitly forbids this. (Attempting a 2-byte representation of 0x20 like 0xc0 0xa0 is disallowed; also disallowed is a 2-byte representation like 0xa0 0x09 since each of these are able to represented in 1 byte)

Errors in UTF-8

- Overlong encodings
- An unexpected continuation byte
- A start byte not followed by enough continuation bytes
- A 4-byte sequence starting with 0xF4 that decodes to a value greater than 0x10FFFF (i.e. you end up with a value that's outside the range of Unicode values)
- A sequence that decodes to a noncharacter
- A sequence that decodes to a value in range 0xD800-0xDFFF (these are reserved for non-BMP characters and UTF-16)

UTF-8: Self-Synchronizing

- Self-synchronizing code is a uniquely decodable code
- All 1-byte sequences contain 7 data bits
- Any 2-4 byte sequences have the correct header & continuation bytes to correctly identify the data bits

UTF-16

- 1 UTF-16 code unit (two 8 bit bytes) for each BMP character
- Byte ordering is definitely an issue with UTF-16 for BMP characters! Why?
- 2 UTF-16 code units for each non-BMP character (4 bytes in total)
- To encode a non-BMP character in UTF-16:
  - 0x10000 is subtracted from the value, leaving a 20 bit number in the range 0x00000-0xFFFFF
  - The top 10 bits are added to 0xD800 to give the first code unit, called the ***lead surrogate***
  - The low 10 bits are added to 0xDC00 to give the second code unit, called the ***trail surrogate***

Self-synchronizing

- Self-synchronizing because any coded UTF-16 value uniquely decodes to a single value
- 10 bits express values in the range 0x000-0x3FF
- Lead surrogates will be in range 0xD800+0x000 to 0xD800+0x3ff (0xD800-0xDBFF)
- Trail surrogates will be in range 0xDC00+0x000 to 0xDC00+0x3FF (0xDC00-0xDFFF)
- Remember: values 0xD800-0xDFFF are not valid Unicode characters (from UTF-8)
  - So if you see a UTF-16 in the range 0xD800-0xDFFF you know it's not BMP
  - Therefore UTF-16 BMP characters can be distinguished from UTF-16 non-BMP characters
  - **So you can tell where the Unicode character boundaries are in a UTF-16 stream**

UTF-32

- Simply take the 21-bit Unicode value and add leading zero bits to extend it to 32 bits
- Byte-order is an issue, like with UTF-16

UTF encoding examples

- ‘z’ --> ASCII $7A_{16}$ --> Unicode 7A
- UTF-8 --> ~~0~~111 1010 (7 data bits) --> 0 111 1010

  7 A
- UTF-16 --> 7A is in the BMP

  007A

  Big Endian: 00 7A

  Little Endian: 7A 00
- UTF-32 --> 00 00 00 7A

  Big Endian: 00 00 00 7A

  Little Endian: 7A 00 00 00

UTF encoding examples

- Unicode 0760_{16} (Thauna)
- UTF-8 --> 0000 0111 0110 0000

    11 data bits to be encoded as 2 bytes (16 bits) of UTF

    --> 110 11101 10 100000

    --> 1101 1101 1010 0000

    --> D D A 0
- UTF-16 --> 0760 (Big Endian 07 60; Little Endian 60 07)
- UTF-32 --> 00 00 07 60

UTF encoding examples

- Unicode $A999_{16}$ (Japanese)
- UTF-8 --> 1010 1001 1001 1001

  16 data bits to be encoded as 3 bytes (24 bits) of UTF

  --> 1110 1010 10 100110 10 011001

  --> 11101010 10100110 10011001

  --> 1110 1010 1010 0110 1001 1001

  --> E A A 6 9 9
- UTF-16 --> A999 (Big Endian A9 99; Little Endian 99 A9)
- UTF-32 --> 00 00 A9 99

UTF encoding examples

- Unicode 12345_{16}
- UTF-8 --> ~~000~~1 0010 0011 0100 0101

  17 data bits (after removing leading zeroes)

  Now try encoding using 21 bits

  --> 11110 100 10 10 0011 10 0100 0101 ??

  Problem grouping bits

  Go back to the original bits:

  0001 0010 0011 0100 0101 (20 bits in total)

  Make them 21 bits by adding leading zero

  --> 0 0001 0010 0011 0100 0101

  --> 11110 000 10 010010 10 001101 10 000101

  --> 1111 0000 1001 0010 1000 1101 1000 0101

  --> F 0 9 2 8 D 8 5

UTF encoding examples

- Unicode 12345_{16}
- UTF-16 --> 0001 0010 0011 0100 0101

  More than 2 bytes (16 bits) so we need 2 UTF-16 code units

  Subtract 10000_{16}

  --> $12345_{16} – 10000_{16} = 02345_{16}$ = 0000 0010 0011 0100 0101

  --> 0000001000 1101000101 (Divide into upper and lower 10 bits each)

  --> 008_{16} 345_{16} (upper 10 bits and lower 10 bits)

  Add $D800_{16}$ to the upper 10 bits and $DC00_{16}$ to the lower 10 bits

  --> $D800_{16} + 008_{16} = D808_{16}$ and $DC00_{16} + 345_{16} = DF45_{16}$

  --> $D808_{16}$ $DF45_{16}$ (2 separate 2-byte values)

  ($D808_{16}$ is a 2-byte value in the upper code unit;

  $DF45_{16}$ is a 2-byte value in the lower code unit)

- UTF-32 --> 00012345_{16}

Memory	Big Endian	Little Endian	
N	D8	08	D808
N+1	08	D8	
N+2	DF	45	DF45
N+3	45	DF	

UTF decoding examples

- UTF-8: EB BB AF

  --> 1110 1011 1011 1011 1010 1111

  --> 11101011 10111011 10101111

  Start from right to left and gather all the data bits 4 at a time!

  Data bits --> 1011 1110 1110 1111

  --> B E E F

  (Unicode character is from Korean!)

UTF decoding examples

- UTF-8: C0 81

  --> 1100 0000 1000 0001

  --> 1100000 10000001

  Start from right to left and gather all the data bits 4 at a time!

  Data bits --> 00 0000 0001

  Unicode 1 ?! Error!
- Unicode 1 should be encoded in 1 UTF-8 byte as 01
- Called an Overlong Encoding error

UTF decoding examples

- UTF-8 bytes in a stream: D7 B3 **ED BA AD** 44_{16}

  --> 1101 0111 1011 0011 1110 1101 1011 1010 1010 1101 0100 0100

- You can jump into the middle of a stream and try to see which is the continuation character (10) and figure out where the header is on the left of it
- In this case the ED BA AD is a stream of UTF-8 bytes that can be decoded successfully (a 3-byte encoding of a Unicode character)
- This is a property of being self-synchronizing, headers are distinguishable from each other and we can find where the boundaries are

UTF decoding examples

- UTF-16: 01 11 D8 02 DD 19 (Assume Big Endian)

  --> 0111 D802 DD19 (take 2 bytes for each UTI-16 code unit)

  0111 --> It's just from the BMP, representing that Unicode character

  D802 DD19 --> Looks like surrogate values

  D802 – D800 = 0002 (subtract lead surrogate base value)

  DD19 – DC00 = 0119 (subtract trail surrogate base value)
- Top 10 bits are from 0002: 00 0000 0010
- Bottom 10 bits are from 0119: 01 0001 1001
- Concatenate top 10 and bottom 10 bits:

  --> 00 0000 0010 01 0001 1001

  --> Group them into 4 bits each from right to left

  --> 0000 0000 1001 0001 1001

  --> 0 0 9 1 9
- Finally add the base value to it to undo the UTF-16 encoding:

  --> 00919 + 10000 = 10919
- Actually a character from the ancient Phoenician script