The International Morse Code Unicode Extension (IMCUE)[1]

Introduction

On the May 13, 2005, episode of the "Tonight Show," host Jay Leno pitted two high-speed CW operators (Chip Margelli, K7JA, and Ken Miller, K6CTW), armed with their trusty paddles and a combined eighty-one years of experience with CW, against then world text-messaging champions Ben Cook and Jason Miller, armed with flip phones and youth.[2] As any amateur radio operator could have predicted, the CW operators crushed the texters.

In 2023, however, modern smartphones sport full-featured, on-screen alphanumeric keyboards. Relevant here, they offer access to many if not all of the roughly one hundred fifty thousand currently defined Unicode characters, including the roughly two thousand "emojis" (Japanese 絵文字 for "pictograph"), like 😂 (officially known as "face with tears of joy") or 🎉 (officially "party popper").[3]

The current International Morse code, on the other hand, is not nearly as rich. It defines just over fifty characters: the twenty-six Latin letters, the letter é, the ten numeric digits, and a handful of punctuation symbols and procedural signs ("prosigns").[4] Popular extensions for non-Latin-based languages add another roughly twenty letters.[5] Consequently, if a 2023 rematch of the 2005 showdown included anything but those seventy or so characters, the CW operators would be unable to even complete the challenge. Enter the International Morse code Unicode Extension ("IMCUE").

Morse code

Morse code encodes letters, numbers, punctuation, and prosigns into a combination of the two available symbols: dit (·) and dah (−). Timing typically relies on the duration of a dit:[6] A dah is as long as three dits; the space between dits and dahs within a character is one dit long; the space between letters of a word is three dits long; and the space between words is seven dits long.

Morse code then assigns sequences of dits and dahs to each character, with the shortest sequences going to the most common characters. For example, the shortest possible Morse code character, a single dit (·), represents the letter **E**. The comparatively uncommon letter **Z**, on the other hand, is represented as dah dah di dit (−−··).[7]

---

1 I'd like to thank Martin Kauppinen for his invaluable consultation. I came across Martin's blog post *https://markau.dev/posts/morse-unicode/* while doing research for this article, and he was kind enough to give me some great ideas.

2 *See http://www.arrl.org/arrlletter?issue=2005-05-20*. It is unknown whether the texters were using a non-predictive multitap system or a predictive system (like T9). *See https://en.wikipedia.org/wiki/T9_(predictive_text)*.

3 *See https://unicode-table.com/en/sets/top-emoji/*.

4 *See* Recommendation ITU-R M.1677-1 (10/2009); *https://en.wikipedia.org/wiki/Morse_code*.

5 *See https://en.wikipedia.org/wiki/Morse_code_for_non-Latin_alphabets*. In this article, I'll use "Morse code" to refer to the International Morse Code and all similar telegraph codes.

6 Jon Bloom (KE3Z), of the ARRL Laboratory, wrote a great article on Morse code timing in the April 1990 *QEX*.

7 To send prosigns, Morse code uses special characters made up of two or three letters sent without any intervening space (*i.e.*, digraphs and trigraphs), commonly represented as an overbar over the characters. So, for example, the "WAIT" procedural word ("proword") is sent as **A** plus **S** with no intervening space—(·−···)— represented as $\overline{\text{AS}}$. The Morse code equivalent of the proword "MAY DAY" is $\overline{\text{SOS}}$ **(**···−−−···). *See https://en.wikipedia.org/wiki/Prosigns_for_Morse_code*.

Unicode

Unicode describes (among other things) a set of characters that can be used to consistently represent written documents.[8] It is maintained by the Unicode Consortium.[9] The set of all characters is called the “codespace.”[10] Each Unicode character is called a “code point,” identified as **`U+0000`** through **`U+10FFFF`**, where everything after the **`U+`** is the code point’s numeric value in hexadecimal, padded on the left with zeroes to at least four digits.[11] After eliminating some reserved ones, there are about a million possible code points in the entire codespace. The current version of Unicode—version 15.0—defines about one hundred fifty thousand code points.[12]

Encoding New Characters in Morse code

Because Morse code’s earliest predecessors (c. 1830s) came along about a hundred and fifty years before Unicode (c. 1980s), Morse code lacks a mechanism to encode anything like Unicode characters. Furthermore, most of the reasonable-length combinations of dits and dahs have already been assigned. Attempting to add even dozens or hundreds, much less millions, of new individual characters would thus require the use of unreasonably long Morse code characters (perhaps as long as twenty symbols).

Other encoding systems have dealt with the problem of limited code space by reusing existing character assignments and then bracketing them with special “modified characters on” and “modified characters off” metacharacters. The ITA standards used for radioteletype (“RTTY”), for example, have characters made up of only five bits, limiting the total possible characters to thirty-two. That is obviously not enough to encode even the twenty-six Latin letters and ten numbers, much less international characters, punctuation marks, *etc*. To overcome this limitation, RTTY uses a base set of characters for the letters. Then, if the operator wants to send figures (or punctuation), he first sends the **`FIGURES`** (*i.e.*, “figures on”) metacharacter. All characters sent after that are interpreted as being from the alternate character set. When the operator is finished sending figures or punctuation, he sends the **`LETTERS`** (*i.e.*, “figures off”) metacharacter, instructing the terminal to return to interpreting characters as letters.

The Wabun code solved a similar problem faced by anyone wanting to encode Japanese words spelled with kana letters. Wabun assigns each kana letter its own Morse code character. In order to avoid a lot of new, unwieldy Morse code characters, Wabun largely reuses existing Latin and non-Latin Morse code characters. To avoid ambiguity, sequences of Wabun characters are prefixed with the “Wabun on” prosign ($\overline{\text{DO}}$ (−··−−−)) and followed by the “Wabun off” prosign ($\overline{\text{SN}}$ (···−·)).

I decided to adopt a similar method for IMCUE. Because emojis are most often sent one at a time, however, I only needed to find a single IMCUE “prefix” prosign (as opposed to separate “IMCUE on” and “IMCUE off” prosigns). Using this optimization, in order to send a Unicode character in Morse

---

8 *See https://en.wikipedia.org/wiki/Unicode*.

9 *See https://home.unicode.org/*.

10 The codespace is divided into seventeen “code planes” (numbered 0 through 16), which are then subdivided into “code point blocks.” Each code point has exactly one General Category—Letter, Mark, Number, Punctuation, Symbol, Separator, or Other—and those seven General Categories are themselves subdivided into many helpful subdivisions.

11 Some code points require multiple hex codes because of character modifiers, *etc.*

12 *See https://www.unicode.org/versions/Unicode15.0.0/*.

code, an operator only needs to send ***<prefix><cp-code>***, where ***<prefix>*** is the IMCUE prefix prosign, and ***<cp-code>*** is a special (new) code for the desired code point (discussed below). (That is, a word space terminates an IMCUE character, allowing for ***<cp-code>*** to be any number of consecutive, uninterrupted characters.)

The next step was to find the shortest Morse code character not already claimed by either the ITU standard or the most popular non-Latin extensions. Compiling a list of all of the characters currently in use revealed that between the Latin, non-Latin, and Wabun characters, all of the one-, two-, three-, four-, and five-element characters are already assigned. Fortunately, most of the six-element characters remain available. I rejected the shortest possible six-element character—*i.e.*, six dits [······]—as being too easy to confuse with the error correction character (eight dits (········)). And because the ***<prefix>*** prosign is going to have to be seven elements long anyway, $\overline{\text{UC}}$ (··−−·−·) (for "**U**ni**C**ode") seemed the most obvious choice.

Assigning Code point Codes

The last step was to assign a ***<cp-code>*** to each desired code point. I consulted the Unicode Consortium's blog to find a list of the most commonly used emojis,[13] and, in the spirit of Morse code, I assigned a one- or two-character short code to the most common one hundred emojis (with the commonest emojis getting one-character codes). In the following table, **Hex ID** is the code point's identification without the **U+**. (Note that some emojis use multiple codes (*e.g.*, a four-digit code followed by the modifier code **U+FE0F**). **E** is the emoji. **Name** is the emoji's official name. And **CPC** is the emoji's new IMCUE ***<cp-code>***:

Hex ID	E	Name	CPC
1F602	😂	face with tears of joy	J
2764 FE0F	❤️	red heart	H
1F923	🤣	rolling on the floor laughing	R
1F44D	👍	thumbs up	U
1F62D	😭	loudly crying face	C
1F64F	🙏	folded hands	P
1F618	😘	face blowing a kiss	K
1F970	🥰	smiling face with hearts	SH
1F60D	😍	smiling face with heart-eyes	E
1F60A	😊	smiling face with smiling eyes	SS
1F389	🎉	party popper	PY
1F601	😁	beaming face with smiling eyes	B
1F495	💕	two hearts	H2
1F97A	🥺	pleading face	LF
1F605	😅	grinning face with sweat	G
1F525	🔥	fire	F
263A FE0F	☺️	smiling face	S
1F926	🤦	person facepalming	FP
2665 FE0F	♥️	heart suit	HS

Hex ID	E	Name	CPC
1F937	🤷	person shrugging	PS
1F644	🙄	face with rolling eyes	RE
1F606	😆	grinning squinting face	Q
1F917	🤗	hugging face	HF
1F609	😉	winking face	W
1F382	🎂	birthday cake	BC
1F914	🤔	thinking face	TF
1F44F	👏	clapping hands	CH
1F642	🙂	slightly smiling face	LS
1F633	😳	flushed face	FF
1F973	🥳	partying face	PF
1F60E	😎	smiling face with sunglasses	FG
1F44C	👌	OK hand	OK
1F49C	💜	purple heart	PH
1F614	😔	pensive face	PN
1F4AA	💪	flexed biceps	FB
2728	✨	sparkles	SP
1F496	💖	sparkling heart	SR
1F440	👀	eyes	EY

[13] *See https://home.unicode.org/the-most-frequent-emoji/.*

Hex ID	E	Name	CPC
1F60B	😋	face savoring food	MM
1F60F	😏	smirking face	SM
1F622	😢	crying face	C1
1F449	👉	backhand index pointing right	FR
1F497	💗	growing heart	GT
1F629	😩	weary face	WY
1F4AF	💯	hundred points	ATT
1F339	🌹	rose	RS
1F49E	💞	revolving hearts	RV
1F388	🎈	balloon	BL
1F499	💙	blue heart	BH
1F603	😃	grinning face with big eyes	GR
1F621	😡	pouting face	PT
1F490	💐	bouquet	BQ
1F61C	😜	winking face with tongue	WF
1F648	🙈	see-no-evil monkey	SE
1F91E	🤞	crossed fingers	CF
1F604	😄	grinning face with smiling eyes	GS
1F924	🤤	drooling face	DF
1F64C	🙌	raising hands	RH
1F92A	🤪	zany face	ZF
2763 FE0F	❣️	heart exclamation	HX
1F600	😀	grinning face	GF
1F48B	💋	kiss mark	KS
1F480	💀	skull	SK
1F447	👇	backhand index pointing down	FD
1F494	💔	broken heart	BR
1F60C	😌	relieved face	RL
1F493	💓	beating heart	BG
1F929	🤩	star-struck	ST
1F643	🙃	upside-down face	UF

Hex ID	E	Name	CPC
1F62C	😬	grimacing face	GM
1F631	😱	face screaming in fear	FS
1F634	😴	sleeping face	SF
1F92D	🤭	face with hand over mouth	HM
1F610	😐	neutral face	NF
1F31E	🌞	sun with face	SU
1F612	😒	unamused face	UA
1F607	😇	smiling face with halo	HL
1F338	🌸	cherry blossom	CY
1F608	😈	smiling face with horns	SD
1F3B6	🎶	musical notes	M2
270C FE0F	✌️	victory hand	VH
1F38A	🎊	confetti ball	CB
1F975	🥵	hot face	HT
1F61E	😞	disappointed face	DP
1F49A	💚	green heart	GH
2600 FE0F	☀️	sun	SN
1F5A4	🖤	black heart	BK
1F4B0	💰	money bag	MB
1F61A	😚	kissing face with closed eyes	KC
1F451	👑	crown	CN
1F381	🎁	wrapped gift	WG
1F4A5	💥	collision	CO
1F64B	🙋	person raising hand	PR
2639 FE0F	☹️	frowning face	FN
1F611	😑	expressionless face	XF
1F974	🥴	woozy face	WZ
1F448	👈	backhand index pointing left	FL
1F4A9	💩	pile of poo	PP
2705	✅	check mark button	CM
1F44B	👋	waving hand	WV

For example, to send the popular "face with tears of joy" emoji 😂, one would send $\overline{\text{UC}}$**J**.

If the desired code point isn't in this table, the operator can send ***<prefix><cp-hexID>***, where ***<cp-hexID>*** is the hexadecimal identification of the code point, without the **U+**.[14] For example, the code point identification for the "high voltage"

[14] When sending hexadecimal code point identifications, operators should avoid using the common Morse code truncations for the numbers (*e.g.*, **T** for 0, **N** for 9, and so on). Four of them (**A** for 1, **E** for 5, **B**

code point ⚡ is **U+26A1**, so one would send $\overline{\text{UC}}$**26A1**. The ground symbol ⏚ is **U+23DA**, and the fuse symbol ⏛ is **U+23DB**, so one could send both by transmitting $\overline{\text{UC}}$**23DA** $\overline{\text{UC}}$**23DB**. To send multi-code code points, one repeats the ***<prefix>*** prosign before each code point identification. For example, the code point identification for the “eye” emoji 👁️ is **U+1F441 U+FE0F**, so one would send $\overline{\text{UC}}$**1F441** $\overline{\text{UC}}$**FE0F**.

In order for the table to be extended as emoji popularity changes, I intentionally did not assign all of the available 1- and 2-character ***<cp-code>*** combinations. Furthermore, there is no reason why non-emoji Unicode code points (like the fuse Unicode character) cannot be assigned their own, perhaps longer ***<cp-code>*** (like “**FUSE**”).

for 7, and **D** for 8) overlap with the letters used to represent hexadecimal numbers (*i.e.*, **A** through **F**) and would therefore be ambiguous when sending a hexadecimal number.