(Yet another scheme for using)

Morse Code as Computer Input

Bill Freeman, KE1G

Abstract

Sometimes a traditional computer keyboard is inconvenient to use. Morse code can be an effective alternative. “Standard” Morse code, however, lacks a number of the characters that may be needed when using a computer. I offer an extension scheme that should be comfortable to those who already know standard Morse code and who also use modern computer keyboards. I also suggest a relatively accurate interface using an extended iambic keyer.

Acknowledgements

I would like to thank Bill, N1BFK, and Janet, KA1JDH for their editorial assistance and ideas. I would also like to thank my non-Ham family, friends and acquaintances for putting up with my ranting and occasional obsession with this topic over the last dozen years or so.

Contents

The impatient may wish to jump to the table 1 on page 22.

You may also want to keep the glossary handy, appendix D on page 28.

1 Why I Want To Do This

I find that I'm pretty slow with the on screen keyboard of my smart phone. There are some thumb keyboard speed daemons out there, but not me. Speed with a thumb keyboard takes two hands and lot of practice. As a Ham ("Ham" (amateur) radio operator, rather than thespian) who managed to pass the Extra class code test (back when the FCC gave the tests for General and higher), I'd prefer to improve my skills with a squeeze keyer[1] (a.k.a. "iambic" keyer).

Besides, it sounds like fun. While I'm not suggesting that anyone text while driving, I envision a smart phone and/or tablet designed to be securely held by my right hand alone, screen toward my eyes, and my thumb and index finger free enough to operate strategically placed dit and dah paddles. Or, if I were to become bed ridden, I'd like the ability to drive a screen that has been placed where I can see it, by using iambic keyer paddles placed where my keyer hand naturally falls.

1.1 My objections to the existing schemes

There are a number of efforts at using Morse, or a system based on Morse, in support of the use of computers by the handicapped. But the ones I've come across tend to define an awful lot of new Morse symbols.[2] Often these are without full regard for existing assignments. One system I found on line even defines its own symbols for the numerals.

I take as "official" the assignments in the "International Telecommunications Union" (ITU) document "RECOMMENDATION ITU-R M1677 - International Morse code" (2004),[3] It has the 26 English letters (no distinction between capital and small letters), the 10 numerals (base-10 digits), 14 of the more common punctuation, some procedural symbols, and a few more. I would at least like to use those letters, numerals, and punctuation symbols for the corresponding characters.

There are some additional symbols (and one itu assignment) which are in common use to represent non-English letters and accented letters. See appendix C on page 25. I presume that Hams or others who already know Morse code, and who use a non-English language, would like to continue to use these familiar

[1] I'll try to always say "keyer" so that if I say "key" (or "keystroke") you can expect me to be talking about the keys of a computer keyboard, or the emulation of their effect.

[2] I'm using "symbol", rather than "letter" or "character" because 1. there are symbols other than letters, such as numerals, punctuation, procedural symbols, and, after I'm done, prefixes, and 2. I want to reserve "character" for something the computer puts up on the screen or otherwise recognizes.

[3] At the time of this writing, you can find a PDF on the web for "RECOMMENDATION ITU-R M1677 - International Morse code". I found it via the Wikipedia Morse code article, with a file name of "international morse code.pdf".

conventions to enter the characters of their language. I therefore avoid these assignments where I know about them, where possible and where they don't conflict with itu assignments.

The Wikipedia article on Morse code mentions three additional punctuation symbols in common use by Hams. I haven't learned these, but I avoid assigning those symbols to something else, although I could see recognizing them as those punctuation characters even if I suggest another way to encode those punctuation characters.

I'm less squeamish about reassigning some of the procedural symbols, since their usual meanings as procedural symbols aren't needed as computer input characters. Apparently the itu isn't squeamish about that either, since several of their defined punctuation characters are reassignments of procedural symbols. I presume they expect the usage to be clear from context or from extra preceding silence in actual Morse communication. In the face of the extensions above, the only unencumbered procedural symbols are "start of transmission" ($\overline{\text{KA}}$) and "error" ($\overline{\text{HH}}$).

2 Terminology

In the interests of brevity I'm going to resort to using some terms and acronyms that may not be familiar to you. I hope that I have covered all of them in the glossary (appendix D on page 28), but let me introduce some of them in context. In the sentence:

Mary had a fit.

−− ·− ·−· −·−−   ···· ·− −··   ·−   ··−· ·· − ·−·−·−

- There are 12 "symbols", the letters and the period.
- There are 35 "elements". The "t", for example has one element, while the period has six elements.
- 20 of the elements are the shorter duration "dits". The other 15 elements are the longer duration "dahs".
- The elements of any one symbol (when there are more than one) are separated from one another by the brief "inter element silence", but that's a mouthful, so I'm going to use the acronym "IES". There are 23 of these in the sentence.
- The symbols of any one word (when there are more than one) are separated from one another by the medium duration "inter symbol silence", for which I will use the acronym "ISS". There are 8 of these in the sentence.
- The words of the sentence (and in this sentence there are more than one) are separated from one another by the longer "inter word silence", acronym "IWS". There are 3 of these in the sentence.

- It will be useful to recognize interruptions in transmission as silences much longer than IWS, which I will call "ITS" for "inter transmission silence". If you count the ITS before the sentence, or the ITS after the sentence, but not both, then the sum of the various sized silences in (and before or after) the sentence is 35, the same as the number of elements.

- While "Morse" was traditionally and still can be send with a "straight key" (single button whose operation directly controls the timing of sound and silence), I'll be presuming the use of an "iambic" "keyer" in which there are a "paddle" for generating dits, and a paddle for generating dahs, where if both paddles are operated simultaneously a sequence of alternating dit and dah pairs is generated.

- At 30 "words per minute" (WPM), not including any its, using "standard" durations and the "official" definition of wpm, the sentence takes 6.0 seconds to send.

3 Details and Justifications

When I mention this plan to fellow Hams, those who seriously consider it frequently ask: "How many new symbols[4] would I have to learn?" That has led me to design a system that only requires one "new" symbol. (While I would like to *also* add new symbols for several "keys" or "modifier prefixes", they can be added compatibly, if desired, as alternative representations.)

By "new" here I mean "not in the set defined by the ITU". By this definition, symbols commonly used for non-English letters would count as new for those of us using English, or a different non-English language. So would a reuse of procedural symbols to which the ITU hasn't assigned an additional non-procedural symbol meaning. For example, symbols without any (known to me) assignment at all (such as ▪▪▬▪▬) count as new.

I'll call my one new symbol "SHIFT". It is used as a prefix to other symbols, that is, SHIFT is sent, then another symbol is sent, and the combination is given a meaning different from that of the second symbol sent.

The price for being able to get away with one new symbol is that you must memorize what it means to SHIFT the non-letter characters. (You will already know what it means to shift a letter.)

And while I only need to learn one new symbol, it is my impression that most Hams will have to learn several new ones to round out their knowledge of ITU assignments for punctuation characters. ITU defines fourteen punctuation characters. Most Hams seem to know little more than half of them. For example, do you know <*apostrophe*> or <*semicolon*>? How about the fact that an ITU alternate meaning for ▪▬▪▬▪ ($\overline{\text{AR}}$) is <*plus*>?

[4]They actually tend to say "characters" but I'm trying to be consistent in my terminology.

3.1 Symbol count

The most useful characters can be sent using a single symbol. This group includes:

- The small (lower case) English letters.
- The numerals.
- The punctuation defined by the ITU.

Many additional characters can be sent using two symbols: a SHIFT prefix symbol, followed by one of the symbols from the group above. I've attempted to choose the most commonly needed extensions for the two symbol group. This group includes:

- The capital (upper case) English letters.
- The rest of the (US PC keyboard) punctuation.
- Other commonly typed keys, such as <*enter*>, <*tab*>and <*backspace*>.
- Several additional prefixes used to change the meaning of following symbols or characters.
- A special symbol to allow sending <*space*> characters in multiples or in isolation.

Any other characters or keys will require at least three symbols. Unusual requirements sometimes exhibited by PC software, such as the need to distinguish whether a left or a right version of a modifier is used, or whether a numeric keypad key was used versus another way to type the same character, will require lots of symbols. Keys that don't even exist on all PC keyboards fall into this category.

Note that few of the keys found on the older keyboards of ASCII terminals are missing from the first two groups. This means that most "command prompt" usage, such as server management over ssh, can be done with characters of one and two symbols.

3.2 The SHIFT prefix symbol

The only new symbol that we absolutely need is the SHIFT prefix.

You might be tempted, in the interest of keeping SHIFT short, or of making it particularly easy to send with an iambic keyer, to use one of ·−·−, ··−−, −−−·, or −−−− because they aren't assigned by the ITU. Those symbols, however, are quite widely used for non-English letters, see appendix C.2 on page 26.

Though the symbol to use as SHIFT should probably be user configurable, it should default to something reasonably mnemonic, and not too slow to send.

One possibility is the "start of transmission" procedural, $\overline{\text{KN}}$ (−·−−·). Neither the ITU nor "Ham" usage have as yet given it an alternate assignment as punctuation. Unlike the <*error*> procedural, $\overline{\text{HH}}$ (········), it doesn't have an obvious use in the computer keyboard context.

There are a number of symbols that appear to be unassigned by the ITU and not in common usage. A few have five elements, many have six elements, and very many have seven (or more) elements.

I like ··–·– as SHIFT ($\overline{\text{FT}}$, thinking of "shiFT" as a mnemonic). Note that –––· takes as long to send, so stealing one of those shorter non-English letters doesn't profit you much. It is also *almost* iambic, requiring only that you wait for the start of the second dit before operating the dah paddle.

3.3 <*space*> in all of its contexts

The traditional Morse way to separate words, leaving an extra long period of silence (IWS) between symbols, will work for the sending most <*space*>s. There are times, however, when it won't be enough.

For example, many of us want to put two <*space*>s between sentences. You might argue for letting an even longer period of silence be code for two successive <*space*>s, but extending this for more than two consecutive <*space*>s seems awkward to me. You may well have need for more than two consecutive <*space*>s, such as to indent a line by three or more spaces.

Further, when using a computer, you will often want to pause between two characters, neither of which are <*space*>, for example, to read a screen full of stuff before sending <*Page Down*> again.

At other times you may want a <*space*> as the last character before a pause. Or you may want, after a pause, to begin with a <*space*>. You may even want to send only one or more <*space*>s between two pauses.

When designing a solution for these problems, one must also keep in mind that, in the world of computer input, there are circumstances in which you cannot undo the effect of having sent a <*space*>.

I propose the definition of a "character" — a sequence of two symbols unless an additional assignment is made for it — that sends a <*space*> unconditionally. Since I will also generate the traditional <*space*> for one level of extra long silence (IWS) between symbols, this new character is not quite identical to just saying "<*space*>", so I don't want to just call it <*space*>. I have chosen to instead call the new character ADDSPACE.[5] I propose that SHIFT-<*two*> be defined as ADDSPACE. (There is no great logical or mnemonic reason for this choice. It's a two symbol combination left over after my other definitions below.)

My scheme also requires that the system *avoid* sending a silence generated <*space*> if the silence is long enough to imply a pause[6] (that is, the silence is an ITS). This means that a silence generated <*space*> can't be sent (to the

[5]There may be better names, such as EXPLICITSPACE, but I'm sticking with ADDSPACE. I've already written too much text using the term ADDSPACE to comfortably switch now.

[6]When conversing with a human, there is context and perhaps other clues that let the recipient know whether a pause should be taken as including a space, but the computer will need a simpler definition.

computer) until the beginning of the first element of the following symbol, since that's as soon as the system knows that the silence was IWS rather than ITS.

We can now solve all of the problems listed above:

- You can use ADDSPACE to send multiple <*space*>s in a row. But note that you can use silence generated <*space*> for half (rounded up) of the series, since silences of <*space*> generating length placed before, between, or after ADDSPACEs still generate their own <*space*>s.
- You can pause without a preceding <*space*>, because that is how we have specified that a pause be treated.
- You can have a <*space*> before a pause by ending with ADDSPACE. Note, however, that there is an alternative for this case. You can send an IWS followed by SHIFT, and then pause. This works because pending modifiers, such as SHIFT are canceled by a pause (see section 3.5). This alternative becomes a shortcut if you haven't defined an additional new symbol for ADDSPACE, but are instead using the two symbol sequence SHIFT-<*two*> as mentioned above.
- You can begin with <*space*> after a pause by beginning with ADDSPACE.
- You can send just <*space*>s between two pauses by sending ADDSPACEs, possibly separated by IWSs if you are sending three or more <*space*>s.

3.4 System tolerance of variation

Electronic keyers often make official IES the minimum between elements, but typically offer no help with longer silences. So when the system attempts to sort silences into IES, ISS, IWS and ITS, it would be best to accept a range of times for each.

I suggest allowing that:

- An IES could be up to 0.2 ticks longer than the standard IES of one ticks[7] and still be considered to be an IES rather than an ISS.
- An ISS could be up to a tick longer than the standard ISS of three ticks and still be considered to be an ISS rather than an IWS.
- An IWS could be up to four ticks longer than the standard IWS of seven ticks and still be considered to be an IWS rather than an ITS.
- There is, of course, no limit on the length of ITS.

[7]"Standard", here, does not consider "weight". The sum of an element and its following silence is independent of weight, since weight is added to the end of the element, but subtracted from the beginning of the silence.

3.5 Some additional prefix modifiers

To do a good job of running existing PC software (I don't know how common exotic character usage is on the Mac) will often require these additional prefix modifiers. There are other potential prefix modifiers, such as <*AltGr*>, but so rarely needed that I'll defer them to later sections, and use sequences longer than two symbols to represent them.

- Use SHIFT-<*at sign*> as the ALT prefix. (Because ALT begins with **A** and there's an **a** inside the **@**. Reaching, I know.)
- Because we don't need it to get <*colon*>, we can use SHIFT-<*semi-colon*> as the CTRL prefix. (I have no mnemonic or visual reason for the choice.)
- Use SHIFT-<*question mark*> as the FUNC prefix. (I have no mnemonic or visual reason for the choice.) Note that, unlike the other modifiers, the computer is never told about the use of FUNC. The computer only knows that it received the correct character, not what we had to send to produce it.
- I'm willing to let the <*Windows*> shift consume three symbols, as FUNC-**w**. Plenty of older keyboards didn't even have it, so it must not be particularly important (I don't use it).[8]

3.5.1 Multiple prefixes

These prefixes mostly don't apply to one another, but instead to the next non-prefix character. This means that you can apply more than one of these to the same character.

If multiple prefixes need to be applied, and FUNC is one of them, FUNC must be last, since in any other placement it will be taken as implying the other hand version of the prefix that follows it, see section 7.

3.5.2 Double SHIFTs

There are several circumstances in which there will be two SHIFTs in a row. This is fine, so long as you don't get confused and omit one. Some examples would be the need to shift <*Enter*> or <*Delete*>. <*Enter*> is sent using SHIFT-<*plus*>, so to send SHIFT-<*Enter*> one sends *two* SHIFT symbols, followed by <*plus*>. <*Delete*> is sent as FUNC-**x**, so to send SHIFT-<*Delete*> one sends *two* SHIFT symbols, followed by<*question mark*>, followed by **x**.

3.5.3 Canceling prefixes

If you send one or more prefixes and then pause (delay for an ITS), the pending modification is forgotten (or you might say canceled). This provides a way to

[8] I suppose that <*Windows*> is a candidate for which ever is the least used of the three Mac modifiers, with the other two mapped onto CTRL and ALT, as seems appropriate.

undo a prefix (including SHIFT) that you sent in error.[9] It also means that when you come back after answering the phone you don't have to try to remember whether you have any prefixes pending.[10]

3.6 Capitals

Preceding a letter with SHIFT changes it from lower case to upper case, that is, from a small letter to a capital letter.

3.7 Non-ITU Punctuation

The extra punctuation is achieved by SHIFTing existing punctuation and numerals.

This first batch will seem familiar to PC (and some other) keyboard users. The character being SHIFTed is also used with the keyboard's shift key, and achieves the same result. This association is, hopefully, already somewhere in your brain and will help make these easier to learn:

- Use SHIFT-<*comma*> to get <*less than*>.
- Use SHIFT-<*period*> to get <*greater than*>.
- Use SHIFT-<*hyphen*> to get <*underscore*>.
- Use SHIFT-<*one*> to get <*exclamation point*>. Perhaps also allow the Ham practice of using −·−·−− ($\overline{\text{KW}}$).
- Use SHIFT-<*three*> to get <*number sign*>, a.k.a. <*pound*> or <*hash*>.
- Use SHIFT-<*four*> to get <*dollar sign*>. Perhaps also allow the Ham practice of using ···−··− ($\overline{\text{SDT}}$).
- Use SHIFT-<*five*> to get <*percent*>.[11]
- Use SHIFT-<*six*> to get <*circumflex*>.
- Use SHIFT-<*seven*> to get <*ampersand*>. Perhaps also allow the Ham practice of using ·−··· ($\overline{\text{AS}}$, the "wait" procedural).

[9] You might expect to be able to use the error procedural $\overline{\text{HH}}$, but we are going to use $\overline{\text{HH}}$ as <*backspace*>, and it is sometimes necessary to send things like SHIFT-<*backspace*>, CTRL-<*backspace*> or ALT-<*backspace*>, so $\overline{\text{HH}}$ can't mean to cancel prefixes.

[10] Things like <*caps lock*> and <*num lock*>, and, if playing with separating presses and releases, see section 8, keys "held down", do survive a pause in sending. I figure that the press/release stuff takes thinking time.

[11] ITU actually specifies the sequence zero-solidus-zero (0/0 — −−−−− −··−· −−−−−) as percent, but that sequence is perfectly meaningful to a computer, so we must be able to send it, rather than a <*percent*>. Further, when we get the zero it is preferable that we send the zero to the computer immediately, rather than waiting to see if we are getting this special sequence, and having already passed along the first zero and the solidus before we can know that it is the special sequence, it may not be possible to tell the computer to forget about them.

- Use SHIFT-<*eight*> to get <*asterisk*>.

This next batch is less obvious. The characters being SHIFTed are also used with the keyboard's shift key to achieve punctuation, but not the same punctuation. Using the keyboard's shift on these characters results in characters that already have Morse symbols assigned to them, so I'm using them to cover some of our leftovers:

- Because we don't need it to get <*left parenthesis*>, use SHIFT-<*nine*> to get <*left brace*>. (Because braces remind me of parentheses.)
- Because we don't need it to get <*right parenthesis*>, use SHIFT-<*zero*> to get <*right brace*>.
- Because we don't need it to get <*question mark*>, use SHIFT-<*solidus*> (a.k.a. SHIFT-<*slash*>) to get <*reverse solidus*> (\), a.k.a. <*backslash*>. (This pair is clearly related, or so it seems to me.)
- Because we don't need it in order to get <*double quote*>, use SHIFT-<*apostrophe*> to get <*grave accent*>, a.k.a. <*back tick*>. (The same sort of relation as solidus to reverse solidus.)

Finally, we have some characters which would require shift to type on a keyboard, but which have ITU symbols of their own, and can thus be SHIFTed to pick up the last four punctuation characters:

- Use SHIFT-<*left parenthesis*> to get <*left bracket*>. (Because parentheses and brackets are both grouping characters.)
- Use SHIFT-<*right parenthesis*> to get <*right bracket*>.
- Use SHIFT-<*colon*> to get <*vertical bar*>. (Because <*vertical bar*> is often depicted with a break in its middle, I can think of each of these characters as being comprised of two identical things, one over the other.)
- Use SHIFT-<*double quote*> to get <*tilde*>. (Because they're both things high above the baseline, with some width to them.)

3.8 Frequently used "Control" character keys

- Use SHIFT-<*plus*> to get <*enter*> (Because <*plus*>, $\overline{\text{AR}}$, is also the procedural "end of message", and <*enter*> in some sense marks the end of a message to the computer.)
- Use SHIFT-<*equals sign*> to get <*tab*> (Because <*equals sign*>, $\overline{\text{BT}}$, is also the procedural "message section separator", and <*tab*> often moves us from form field to form field, separate sections of the "message".)
- Use the "error" procedural, $\overline{\text{HH}}$, to get <*backspace*>. There is no SHIFT required. (This assignment is too good a fit to pass up.)

3.9 Use of the FUNC prefix

Using FUNC before ALT, CTRL, or before a SHIFT that isn't part of another prefix, is a shortcut to indicate the right hand version of the of the modifier. See section 7.

- Apply FUNC to <*one*> through <*nine*>, <*zero*>, the letter **a**, and the letter **b**, to represent, respectively, <*F1*> through <*F12*>.
- Use FUNC-**l** (small **L**) for <*Left Arrow*> (of the cursor arrows).
- Use FUNC-**r** for <*Right Arrow*>.
- Use FUNC-**u** for <*Up Arrow*>.
- Use FUNC-**d** for <*Down Arrow*>.
- Use FUNC-**p** for <*Page Up*> (**p** as in "previous" since FUNC-**u** is already used).
- Use FUNC-**n** for <*Page Down*> (**n** as in "next" since FUNC-**d** is already used).
- Use FUNC-**h** for <*Home*>.
- Use FUNC-**e** for <*End*>.
- Use FUNC-**i** for <*Insert*>.
- Use FUNC-**x** for <*Delete*> (since FUNC-**d** is already used)
- Use FUNC-**q** to get <*Escape*>. (Think of the "cancel" meaning of <*escape*> and **q** for "quit".)
- Use FUNC-**m** to enter "Mouse Mode", see section 6.

Next some less useful keys. I expect most of us to use these rarely, if at all.

- Use FUNC-**c** for the <*Caps Lock*> key. Other than perhaps managing an LED, on a PC keyboard this just sends a character to the computer, and does not, in it self, affect what is sent by the keyboard for other keys. Interpretation of the caps lock state is left to the computer.
- Use FUNC-**k** for the <*Num Lock*> key. Other than perhaps managing an LED, on a PC keyboard this just sends a character to the computer, and does not, in it self, affect what is sent by the keyboard for other keys. Interpretation of the num lock state is left to the computer. FUNC-**n** is already more profitably used, so the justification of this choice is that num lock is sort of key pad lock.
- Use FUNC-**o** as a prefix to indicate the other hand version of a key, see section 7.

- Use FUNC-<*comma*> for the <*Pause/Break*> key.
- Use FUNC-<*period*> for the <*Print Screen/System Request*> key.
- Use FUNC-**w** as the <*windows*> key. This is actually a prefix key, and is also mentioned above in section 3.5
- Use FUNC-**s** as the <*menu*> key. (S for select, since FUNC-**m** is already used.) This smells like a prefix key, but we treat it like an ordinary key. Often, tapping this on a keyboard causes a menu to appear, which is a character like action, not a prefix like action.
- FUNC-**t** (toggle) enters or exits "toggle mode". See section 8.
- FUNC-**z** is used in toggle mode to deal with the <*shift*> key, as distinct from the SHIFT prefix.
- Reserve FUNC-**g** for the <*AltGr*> (possibly used as <*compose*>). I haven't yet figured out how this is applied.

4 Optional Additional New Symbols

If you're willing to learn a few more symbols, either from the vast array of completely unassigned symbols or by stealing from the common use non-English letters, there are a few assignments that would make things faster and less confusing.

A new symbol for <*enter*> would be the most helpful, followed closely by one for <*tab*>. Not far behind are additions for FUNC and ADDSPACE. Assignments for CTRL and ALT would also be reasonably helpful. Everything else is much less important, though I'd be tempted to add an assignment for <*escape*>.

Counting SHIFT and including <*escape*>, that would make a total of eight new symbols.

We could assign some or all of these, but also keep the versions from above that require the SHIFT prefix. That would allow a user to determine how much new stuff he learns how fast.[12] Such assignments should probably be user configurable, though with hopefully mnemonic defaults from among the longer unassigned symbols.

4.1 Suggested additional assignments

I will try to live with the single SHIFT assignment for a while, mostly to confirm that it is actually usable. But I expect to add some assignments later. My current thoughts are:

[12] If, however, the additional new symbols started creeping into QSOs some effort to standardize would be in order.

<*enter*> ·−·−− Maybe think $\overline{\text{ENTT}}$. Sorry about the extra “t”. If you are going to reassign some of the non-English letters, ·−·− for $\overline{\text{ENT}}$ might be a really good choice here.

<*tab*> −·−−− Think $\overline{\text{TAM}}$ (TAb Me) or $\overline{\text{TATT}}$ (TAb To sTop?)

FUNC ··−·−· Think $\overline{\text{FN}}$ for FuNc.

ADDSPACE ·−−·− Think $\overline{\text{WA}}$ for White spAce?

ALT ·−·−·· Think $\overline{\text{AL}}$ for ALt.

CTRL −·−··· Think $\overline{\text{CI}}$ for Control It.

<*escape*> −··−− Think $\overline{\text{XT}}$ for ExiT, along with the “cancel” meaning of <*escape*>.

5 Physical Interface

Getting a computer to interpret “hand sent” Morse, that is, Morse sent using a “straight key”,[13] seems to require fancy algorithms and lots of processing power. (I remember the accuracy as being unsatisfying, but I may be behind the times. Processing power has certainly become plentiful and cheap.)

Using a modern electronic keyer that normalizes the durations of the elements and the silences between them, and using a low noise connection (rather than over 40 meters), can reduce the complexity (and improve accuracy).

You might implement the system almost entirely in software, say by feeding your keyer’s side tone into your sound card. I will leave such exploration to others.

The thing to notice about an electronic keyer is that it already knows whether it is sending a dit or a dah. Since I’m thinking of a local connection,[14] that information can be provided to the keyboard emulation system directly, rather than by converting it to a stream of key up and key down events that must be decoded back into that dit versus dah information.

My expected implementation is a micro processor with the hardware to act as a USB device or as a Bluetooth device,[15] supporting the HID keyboard protocol/profile, and behaving like a keyer in that it interfaces to a pair of paddles and provides side tone audio (speaker or earphone jack, maybe also an LED).

If, however, a manufacturer wanted to include the system in a portable computing device (notebook, tablet or smart phone) then there may be no need for a separate keyer micro controller. One could interface “paddle” buttons

[13]The occasional use of the term “key” in this section will refer to a telegraph key, rather than a computer keyboard key.

[14]In most implementations the keyer and the keyboard emulation will be sub-parts of one system.

[15]See: `http://hackaday.com/2011/08/02/bluetooth-morse-code-keyboard-for-the-disabled/` and the USB article referenced therein.

directly to the main CPU, or even just use of a couple designated spots on a touch screen as paddles. A kernel driver would provide the keyer functionality, capture of symbols, classification of "silences", conversion to scan codes, etc., and be attached to the kernel as though it were an ordinary keyboard driver.

5.1 Recognizing the four durations of silence

The keyers I've seen assure that the silence between two elements isn't shorter than standard IES, but offer no help with the timing of the longer silences.

I've toyed with the idea of making a keyer that recognizes the longer silences and enforces minimums for them as well. I suspect that some operators would be put off by the enforced minimums, wanting the first new element to begin when the paddle is operated, rather than wanting help sending "pretty" code. Thus I intend to make the enforcement of minimums optional.

Even without the enforcement of minimum silences, the keyer will still categorize the length of the silence. It is in the best position to do so, since it knows the exact code speed, a parameter that a downstream time based interface would have to infer. In the same way that the keyer can directly signal dit versus dah, it can also signal that a symbol has ended, that a <*space*> should be generated, or that we have paused.

With such an interface nothing but the keyer need be aware of code speed. It also inherently allows the keyer to offer personalized, exaggerated silence timings (Farnsworth method code), and the next stages (symbol accumulation, etc.) need be none the wiser.

5.1.1 Operator feedback

In the absence of guidance, operators vary in their timing. But with feedback, humans are very good at adapting to circumstances. What, then, if the keyer gave guidance in the timing of the silences, as well as the sounds?

For example, when a silence has become long enough that the keyer has decided that the symbol has ended, the keyer might issue a tone noticeably lower in frequency than the normal side tone. That tone might end when the silence has become long enough to generate a <*space*>, and another short chirp be emitted when the silence has indicated a pause.

Or the operator may not need help with the limit that ends a symbol, but find comfort in a tone that lasts through the period during which paddle operation will produce a generated <*space*>.

There are other possibilities, such as short chirps just before, or just after, each decision point is reached. (That, however, strikes me as too "noisy" however.)

5.2 Side tone latency

Side tone is important when sending with a keyer. The operator must know the speed and timing of the dits or dah that the keyer generates in order to time the motion of the paddles.

You might be tempted to provide side tone through the computer speaker, or over a Bluetooth link, in order to avoid having a separate speaker or ear phone, or to avoid needing a cable to connect an ear phone. But beware that the operator probably can't tolerate much latency in the side tone. A computer might offer enough priority to directly interfaced paddles that it could produce good side tone, but passing keyer data over USB or Bluetooth to tell the computer when to generate side tone is probably unsatisfactory. The side tone should probably be generated by the keyer directly.

Similarly, Bluetooth audio has noticeable latency, so a Bluetooth connection from the keyer to an ear bud may not cut it. A USB or Bluetooth connection from a keyer to a tablet and then side tone generated by the tablet and sent to a Bluetooth ear bud is even worse.

6 Mouse Mode

There are some pieces of software that are a bear to use without a mouse.

In many situations you won't need Mouse Mode because there will be a mouse available, or at least a touch screen. In fact, one system possibility is that the dit and dah paddles could be placed on the sides of a (possibly wireless) mouse, though making side buttons easy enough to avoid when just mousing is apparently hard — at least I'm always hitting mouse side buttons by accident.

But it might be nice to be able to use some of those click centric apps when all you have is a pair of keyer paddles. Therefore I propose a "Mouse Mode", entered, as detailed above, using FUNC-**m** from "normal" mode. You're not going to paint beautiful pictures, or drive a gesture recognition system using this mode, but you will be able to do basic stuff.

There are certainly other possible Mouse Mode designs. This is about a third cut design for me, and I won't be surprised to find that I want something else. Still, I believe that it is workable and reasonably efficient, if a touch ornate, so I offer it as a straw-man.

Whether or not there *is* a Mouse Mode should probably be user configurable. It is also possible to implement several designs and have the choice be configurable.

In this Mouse Mode certain symbols set a "current direction". Others specify movement distance in (or opposite to) the current direction. Still others specify clicking, double clicking, depressing for a drag, and releasing after a drag, of the several mouse buttons.

Those symbols which are not defined in Mouse Mode are mostly ignored, on the grounds that they are easily sent by mistake when trying to send a string of **e**, **t**, or a mixture, which is going to be common, see below. But a string of five or more such undefined (in Mouse Mode) symbols in a row also exits Mouse Mode, on the theory that the user got there by mistake and doesn't know how to exit Mouse Mode.

Also, particularly any <*space*> that would be generated by an IWS is ignored, not even contributing to the five invalid in a row exit count.

6.1 Direction

Use **r** (right), **u** (up), **l** (left), and **d** (down) for the cardinal directions.

You can also "make a turn" to adjust the current direction. Letter **a** will change the direction 45 degrees Anti-clockwise (counter clockwise for us Americans). **n** (Non-anti-clockwise, sorry, I want something shorter to send than **c** for clockwise) will change it 45 degrees clockwise. **v** will reVerse course, that is, turn by 180 degrees (clockwise or counter-clockwise doesn't matter in this case). **a** and **n** are chosen because they are fast to send.

Note that you can then choose an absolute diagonal direction using one of the cardinal letters followed by a 45 degree turn. For example, **u** followed by **n** starts you out northeast. Or if you can't remember **n**, you could use **l** followed by **a**.

6.2 Distance

Mouse movement is measured in some integer. Let's call its units "mickeys". Scaling this to the screen is the job of the computer, but we presume that it is usually one pixel. Note that if moving diagonally, one moves both the x and y axis, so the minimum distance moved diagonally is actually $\sqrt{2}$ mickeys, but we'll still call this moving one mickey.

Use numerals **1** through **4** for the fine scale movements, respectively, of 1, 2, 4 and 8 mickeys.

Normally we will want to move by a larger amount, which I'll call a "jog". It should probably be configurable, but let's default to 16 mickeys on the theory that this will be about the width of a small letter on modern screens. We will move longer distances as multiple jogs, and this should be convenient and fast, so I will use the very quick to send letter **e** to move by one jog.

You can use successive **e**s to move larger distances, but we will also allow **i**, **s** and **h** to move, respectively, 2, 3 and 4 jogs. Since we can't know that **i** isn't the beginning of, say **f**, and similar issues for the others, we must wait for the ISS before the movement actually occurs, so if you are in the mode of moving the final bit, intending to stop when you get there, you may want to send separate **e**s instead.

As a way to ease correction of overshoot, and for final fine positioning, **t** will cause a movement by one quarter jog[16] in the negative direction. (So you don't have to send **v** first to back up a little.)

m and **o** will back up by half and three quarters, respectively, of a jog. And **----** will back up a whole jog. As with **i**, **s**, and **h**, the motion must be deferred until the ISS is recognized.

[16]Thus it is best if jog is evenly divisible by four mickeys.

There are no symbols beginning with five dits except for <*error*> and **5**, and no symbols beginning with five dah other than **0**. Therefore we will take symbols of arbitrary length but beginning with at least five dits or with at least five dah as moving a jog for each dit in the symbol, and a negative quarter jog for each dah in the symbol. The first four moves must be deferred until we are sure that the symbol begins with five of the same element. Once the fifth dit or fifth dah is recognized those first jogs or negative quarter jogs are sent immediately, and additional dits or dah generate immediate additional jogs or negative quarter jogs.

[An alternative interpretation would be that once both directions have been used in one of these extended, began with five or more of the same element symbols, that all motions, forward or back, are quarter jogs, on the theory that you are doing final fine positioning.]

For much bigger movements we have **6**, **7**, **8** and **9** for, respectively, 16, 32, 64, and 128 jogs.

6.3 Buttons

Use **z** for the left, **x** for the middle, and **c** for the right mouse buttons. These are chosen because on my keyboard they are a group of three contiguous keys in that left to right order.[17] If you're a two button guy, don't use **x**. If you're a one button guy, only use **z**.

Just the plain letter will be single click. I considered using SHIFT and CTRL to get double clicks and drags, but software often uses things like SHIFT-<*click*> and CTRL-<*click*> to have different meanings than just a click. I'm going to use single letter prefixes instead.[18]

6.3.1 Double clicking and dragging

Prefix a button letter with **k** to Keep the (possibly second of two) press(es) of the following button held down for dragging. Or, if you were already dragging something, send **k** again to release it. Pausing (ITS) *does not* act as a release.

Note that if you were dragging, and you want to release, and pick up to drag again, possibly with a different mouse button, you will be sending two **k** symbols. The first will act as a release. The second will prefix the following button letter.

Prefix a button letter with **w** to double click it, since you probably can't reliably send the button letters fast enough to be recognized as a double click.

[17] Folks with AZERTY or Dvorak keyboards may want to customize assignments to get something they prefer, but remember that symbols consisting of all the same element are reserved for multiple jogs and multiple negative quarter jogs, and that you probably want to keep the relatively short **a** and **n** for steering. You could certainly replace the cardinal direction letters with something that forms a diamond on your keyboard if that helps. It is also possible to have multiple sets recognized simultaneously, such as also allowing **m**, <*comman*> and <*period*> for left, middle and right buttons, respectively.

[18] We could allow FUNC-**t** to work here as well, to do the simple drag, but it won't handle the double click to drag, and that would take six extra symbols (press and release) instead of two.

I'm using **w** because **d** is already used for a cardinal direction, and the name of **w** begins with "double".

You can also prefix with both **k** and **w**, in either order, to get a double click and hold the second click, still ending the drag with **k**.

If you realize that you have entered one of these prefixes by mistake, you can enter a direction setting letter, such as **r** (think of it as Reset) to cancel the prefixes. Or pausing will cancel them (but only before the following button symbol has been sent).

If you're going to SHIFT, CTRL, or ALT the button, you can add those before or after (or during a pair of) the **k** and/or **w** prefix(es).

Exit Mouse Mode using <*period*>. Mouse Mode is NOT canceled by pausing. Beware that exiting Mouse Mode does NOT end a drag.

7 Other hand keys

FUNC-**o** is used as a prefix meaning the following prefix or key is the other version.

SHIFT, CTRL, ALT, the numerals, <*period*>, <*plus*>, <*hyphen*>, <*asterisk*>, <*solidus*> and <*enter*>, used without FUNC-**o** are the left hand, or main keyboard versions, while used with FUNC-**o** as a direct prefix they are right hand, or numeric keypad versions.

The arrows, <*Home*>, <*End*>, <*Page Up*>, <*Page Down*>, <*Insert*>, and <*Delete*> are traditionally (because they predate the 101 key keyboard) considered to be on the numeric keypad, so prefixing these characters with FUNC-**o** will make them non-numeric keypad versions (such as the inverted "T" of arrow keys).[19]

Note that there is only one version of FUNC. Since the computer is never told about FUNC (see section 3.5), no piece of software could possibly use the distinction between a left and a right version of it.

Note that something like right-SHIFT-<*asterisk*> needs to be distinguished from numeric keypad <*asterisk*>. This requires the shift to be doubled for the first case, giving a total of: SHIFT-<*question mark*>-**o**-SHIFT-SHIFT-<*eight*>

In some cases FUNC-**o** can be abbreviated as just FUNC. For the right hand versions of the prefixes themselves, and of the characters which already require a FUNC prefix, the abbreviated FUNC only version of "other" can't be confused with another meaning of the added FUNC prefix. But, for example, FUNC-<*two*> means <*F2*>, so one must use the full FUNC-**o** <*two*> to indicate the numeric keypad <*two*>. Using the full FUNC-**o** version of the prefix will always work, should you not want to worry about when the cut applies.

Note that you can apply both the left and the right hand version of a prefix to the same key.

[19]For these keys the right hand is used to type both versions, so "other hand" is a misnomer. They are still "other", however, in the sense that the same funky invisible modifier scan code is sent to distinguish them, at least on PC keyboards.

This may seem like a lot to send, but how often do you care whether it's the right hand CTRL versus the left hand CTRL.

8 Overlapping key presses

There are occasions on which you want the computer to know that two or more keys were pressed at once, or that you didn't, say, release ALT during the entry of several digits.

FUNC-**t** enters and exits "toggle mode". In toggle mode, a "key" sends only a press scan code or a release scan code, whichever was not the last sent *for that key*.

You can exit toggle mode with keys "held", i.e.; when their last scan code sent was a press scan code. They continue to be held, and other keys are sent normally, as a press/release pair. A classic use for this is to "hold" the ALT key and send several <*tab*>s to select the window you want.

You can re-enter toggle mode to change which keys are held. If, however, you only want to release a key, just transmit it without re-entering toggle mode, and only its release scan code will be sent, making it no longer held. This seems a better use than sending the release code followed by the press code (which, if you really want it, can be done in toggle mode).

I can envision getting confused as to whether you are in toggle mode or not, so let SHIFT-FUNC-**t** force toggle mode off. That is, turn it off if it is on, and leave it off if it is off.

I can also imaging forgetting which keys are held, so let the sending of SHIFT-FUNC-**t** *twice in a row* mean that all keys should be released. The system would send release codes for each held key and/or an "all keys released" scan code. That might be a good thing to do if computer seems confused about what the system is sending.

If you need the SHIFT prefix to specify the character normally, you will still need it in toggle mode, and if you don't normally need the prefix, you still won't need it in toggle mode. But the system no longer guaranties that the <*shift*> key will be held or not according to the needs of the character. You will have to manage that by toggling the <*shift*> key yourself, as appropriate.

But we can't use SHIFT in toggle mode to toggle the <*shift*> key. The SHIFT prefix is needed just to specify the keys you want to toggle, such as <*circumflex*>. So the SHIFT prefix can't, by itself, stand for the <*shift*> key.

While hardly mnemonic, let's use FUNC-**z** to mean that we want to toggle the <*shift*> key. (The **z** key is at least next to one of the <*shift*> keys, at least on QWERTY keyboards.) It will be meaningful in toggle mode, but its only use in normal mode will be to release a held <*shift*> key without having to enter toggle mode to do it.

A Tabular Summary

In table 1:

The ITU column specifies the meaning of the Morse in the symbol column when it is not immediately preceded by the SHIFT or FUNC prefixes. Those assignments are consistent with ITU definition, save for the fact that ITU doesn't imply that the letters are lower case. ITU doesn't imply that the letters are upper case either, so this isn't really an inconsistency.

The SHIFTed column specifies the meaning of the Morse in the symbol column when it is immediately preceded by the SHIFT prefix. (My current favorite choice for SHIFT is ··–·– ($\overline{\text{FT}}$), which is a relatively short symbol for which I can find no common preexisting assignment.) For example, to send <*circumflex*>, send SHIFT followed by <*six*>: ··–·– –···· None of these results can be sent using ITU definitions, except that since ITU doesn't specify the case of letters, so, in a sense, it does provide a way to send the letters here. You will note that this column provides one way to send the FUNC prefix (at the end of the column).

The FUNCed column specifies the meaning of the Morse in the symbol column when it is immediately preceded by the FUNC prefix. For example, to send the <*home*> keystroke, you could send the sequence SHIFT, <*question mark*>, **h**: ··–·– ··––·· ···· (If you have defined an additional new symbol, or have reassigned a symbol with an existing assignment, to use as the FUNC prefix, then you could *also* send <*home*> using your prefix, followed by **h**, which is just two symbols long, instead of three.) None of these results can be sent using ITU definitions.

Table 1: Extended Morse

Symbol	ITU	SHIFTed	FUNCed
·–	a	A	<*F11*>
–···	b	B	<*F12*>
–·–·	c	C	<*Caps Lock*>
–··	d	D	<*Down Arrow*>
·	e	E	<*End*>
··–·	f	F	
––·	g	G	<*AltGr*>
····	h	H	<*Home*>
··	i	I	<*Insert*>
·–––	j	J	
–·–	k	K	<*Num Lock*>
·–··	l	L	<*Left Arrow*>
––	m	M	enter Mouse Mode
–·	n	N	<*Page Down*>
–––	o	O	other had prefix
·––·	p	P	<*Page Up*>
––·–	q	Q	<*Escape*>

Continued on next page

Extended Morse, continued			
Symbol	ITU	SHIFTed	FUNCed
·–·	r	R	*<Right Arrow>*
···	s	R	*<Menu>*
–	t	T	toggle press/release
··–	u	U	*<Up Arrow>*
···–	v	V	
·––	w	W	*<windows>*
–··–	x	X	*<Delete>*
–·––	y	Y	
––··	z	Z	toggle *<shift>*
·––––	1	*<exclamation pt>*	*<F1>*
··–––	2	ADDSPACE[20]	*<F2>*
···––	3	*<number sign>*[21]	*<F3>*
····–	4	*<dollar sign>*	*<F4>*
·····	5	*<percent sign>*	*<F5>*
–····	6	*<circumflex>*	*<F6>*
––···	7	*<ampersand>*	*<F7>*
–––··	8	*<asterisk>*	*<F8>*
––––·	9	*<left brace>*[22]	*<F9>*
–––––	0	*<right brace>*	*<F10>*
–·––·	*<left parenthesis>*	*<left bracket>*[23]	
–·––·–	*<right parenthesis>*	*<right bracket>*	
·–·–·–	*<period>*	*<less than>*	*<Print Screen>*
––··––	*<comma>*	*<greater than>*	*<Pause/Break>*
·––––·	*<apostrophe>*	*<grave accent>*	
·–··–·	*<double quote>*	*<tilde>*	
–––···	*<colon>*	*<vertical bar>*	
–··–·	*<solidus>*[24]	*<reverse solidus>*	
–····–	*<hyphen>*	*<underscore>*	
·–·–·	*<plus>*	*<Enter>*	
–···–	*<equals>*	*<Tab>*	
–·–·–·	*<semi-colon>*	CTRL	
·––·–·	*<at sign>*	ALT	
··––··	*<question mark>*	FUNC	

You probably want to read section 3.5 on page 10 to understand how to apply SHIFT, CTRL and/or ALT to a character whose otherwise unadorned version requires the use of FUNC. Though you may not otherwise care about the ability to specify the right hand or numeric keypad versions of keys, that capability does restrict where FUNC can occur when there are multiple prefixes.

[20] Sending *<space>* characters is a complex issue. See section 3.3 on page 8

[21] You may know number sign – # – as “pound” or “hash”.

[22] These are the squiggly ones. – { }

[23] These are the square cornered ones. – [ ]

[24] You may know solidus as “slash”, and you may know reverse solidus as “backslash”. – \

Table 2: Mouse Mode

Char	Meaning	Char	Meaning
e	jog[25]	t	-jog/4
i	2 * jog	m	-jog/2
s	3 * jog	o	-3*jog/4
h	4 * jog	ch[26]	-jog
5	5 * jog[27]	0	-5*jog/2
1	mickey[28]	2	2 * mickey
3	4 * mickey	4	8 * mickey
6	16 * jog	7	32 * jog
8	64 * jog	9	128 * jog
u	direction up	d	direction down
l	direction left	r	direction right
a	turn 45° anti-clockwise	n	turn 45° clockwise
v	turn 180°	x	click middle
z	click left	c	click right
w	double next click	k	start/stop drag[29]
<*period*>	exit Mouse Mode		

You may want to review the details of Mouse Mode in section 6.

B Technical Implementation Notes

At least in a PC world, CTRL, ALT and <*Windows*> actually wind up being reported to the computer as presses before the modified character and releases afterward. We don't expect that the computer cares about the relative order of the presses, so long as they come before the modified character, nor the order of the releases, so long as they come after the modified character. Because prefixes are canceled by a pause in sending, none of this is sent to the computer until recognition of the end of the last symbol of the character to be modified.

The computer is also told about shift presses and releases. Some characters, like <*question mark*>, are sent from a keyboard while holding the SHIFT key, but in Morse there is a symbol for <*question mark*>, so the SHIFT prefix isn't sent. Some other characters, such as <*left bracket*> are sent from a keyboard without holding the SHIFT key, but have no Morse symbol, and we use the SHIFT prefix to get them. So the presence or absence of the SHIFT prefix does not always map to whether or not we send SHIFT presses and releases to the

[25] A jog is a move in the current direction by some middling amount, such as the width of a small letter.

[26] The Spanish ch: ----

[27] Additional symbols beginning with at least five dits or beginning with at least five dahs will move a jog for each dit and -jog/4 for each dah.

[28] A mickey is the base resolution of the mouse interface

[29] If already dragging, this is stop dragging. If not dragging, and click is also doubled, this is click and hold.

computer. But, like CTRL and ALT, any necessary shift presses and releases are sent around the affected character when that character is ready to be send.

C Preexisting Assignments

The ITU assignments for letters (though of indeterminate case) can be found in the first two columns of table 1 in appendix A. Here I summarize the other assignments I respect, and am trying to avoid.

C.1 ITU Non-English letter

ITU-R M1677 makes a single concession to accented Roman characters in the form of “Accented E” (··−··). There is no guidance as to what kind of accent is intended[30]. ITU provides no help for the decoration of other characters.

C.1.1 Procedural Signs (symbols)

Table 3: Procedural Signs

Meaning	Morse
Wait	·−···
Understood	···−·
Not understood, resend[31]	··−−··
Error, I will resend the last several chars/words	········
Starting signal, opens each transmission	−·−·−
Invitation to Transmit[32]	−·−
End of Message[33]	·−·−·
End of Work	···−·−
Break, or Message Section Separator[34]	−···−
Distress Signal[35]	···−−−···

Procedural Signs are essentially out of band symbols, used to inform the receiving operator of things like:

- Intentional pauses in transmission
- Message boundaries, when multiple messages are sent in a single transmission
- Separating parts of a message like address, body, signature

[30] Common usage includes acute e and e with ogonek.
[31] Also question mark
[32] Also letter “K”.
[33] Also plus
[34] Also Double Hyphen, or equals.
[35] Yes, this is officially a single symbol, not the sequence of the three letters “SOS”.

- Switching of sending and receiving roles
- Intention to terminate operation.

You will note that some of these also have assignments as ITU punctuation or Ham common usage punctuation. One must presume that, in a message using both meanings, which meaning is clear to a human operator from the context. Since the system has no need of the procedural meaning, it doesn't have this problem, and always takes the punctuation meaning.

There are actually a few other home-grown procedural signs in use. Most of them are quite long anyway, such as −·−··−· $\overline{\text{CL}}$, and I won't bother to document them here.

C.2 Non-ITU accented or decorated letters

I am no expert on non-English usage[36], so don't consider this section to be comprehensive.

These are used by some people conversing in non-English languages. They could also be used to in an English conversation to include non-English words,[37] though I don't expect that to be common.

Table 4: Common non-English Extensions

Characters	Morse
ä, æ, a with ogonek	·−·−
à, å[38]	−··−·
ç, ĉ, ć	−·−··
ch, š, ĥ	−−−−
Eth	··−−·
ś	···−···
è, ł	·−··−
é, d with stroke, e with ogonek[39]	··−··
ĝ	−−·−·
ĵ	·−−−·
ź	−−··−·
ñ, ń	−−·−−
ö. ø, ó	−−−·
ŝ[40]	···−·
Thorn[41]	·−−··

Continued on next page

[36]And I know nothing of non-Roman assignments for languages like Japanese, or even Greek
[37]Such as the sometimes fashionable inclusion of French words in British detective stories.
[38]Conflicts with ITU Fraction Bar.
[39]ITU accented E.
[40]Conflicts with ITU procedural "Understood".
[41]If someone knows the TEX/LATEX encodings of characters with ogonek or the Eth or Thorn characters, I'd appreciate hearing from them.

Common non-English Extensions, continued	
Characters	Morse
ü, ŭ	··−−
ż	−−··−

Where there is a clear keyboarding convention to represent these characters, a system like mine could accept these assignments and transmit them faithfully to the computer. But, as you will note, the same symbol is used for different symbols in different languages, so at the very least one would have to specify which language is in use.

It may be best to have a mechanism for representing unrecognized symbols according to their length and sequence of dits and dahs, and which will be ignored by OS keyboard drivers that don't understand them. In fact, being able to put any Morse hardware into a mode where all symbols are send this way could provide for even non-Latin derived languages.

C.2.1 Amateur Radio practice

Hams have traditionally used −·−−· to mean "over to you, but only you". That is, essentially the same as −·− used as a procedural at the end of a transmission, but with the added request to others on frequency that they do not "break in". The assignment as the left parenthesis by ITU doesn't necessarily bother this, since the context at the end of a transmission is usually clear.

There was a push to use −·−·−− as exclaimation point (!), which is more reasonable than the transfer from American Morse of −−−· for this relatively infrequent character, though the latter is the more common practice. There is an extra meaning of the wait procedural ·−··· to mean ampersand (&). And ···−··− is used for the dollar sign ($).

C.3 Unassigned symbols

Steering clear of ITU and other common assignments, the 6 available five element symbols are:

- ··−·− (my choice for SHIFT)
- ·−·−−
- ·−−·−
- −··−−
- −·−−−
- −−−·−

These are candidates for SHIFT, and for any of section 4 for which one chooses to make an assignment.

I have the list of unassigned six element symbols, but I feel that it is too long to include here. I can provide it on request, and will be updating it if people report additional non-English letter or other usages.

D Glossary

dah The longer of two durations of sound (or light) used in International Morse Code. It lasts three ticks *plus* any weight adjustment. Some people call it dash.

dit The shorter of two durations of sound (or light) used in International Morse Code. It lasts one tick *plus* any weight adjustment. Some people call it dot.

element A dit or a dah.

iambic As used here, a form of keyer with separate paddles for dit and dah, where if both paddles are held operated, alternating dits and dahs are sent. You may recognize the term from poetry, as in "iambic pentameter" which is composed of alternating unstressed and stressed syllables. A single such pair is an "iamb".

IES Inter Element Silence. The shortest period of silence (or darkness) used in International Morse Code. It separates the elements of a symbol. It lasts one tick *minus* any weight adjustment. Note that the inclusion of the negative of the weight adjustment in the defined duration of silences compensates for its addition to the elements, making the time from the beginning of one element to the beginning of the next element be independent of the weight adjustment.

ISS Inter Symbol Silence. A period of silence that separates the last element of one symbol in a word from the first element of the next symbol in the same word. It lasts three ticks *minus* any weight adjustment.

ITS Inter Transmission Silence. A new concept created for this scheme, it is any period of silence too long to be IWS, and signifies that we are pausing, such as to think, or get coffee, or answer the phone. In an actual Morse conversation you might send the "wait" procedural symbol, but the computer isn't impatient.

ITU The International Telecommunications Union is a United Nations agency. It is, among other things, the "official" arbiter of the definition of International Morse Code.

IWS Inter Word Silence. A period of silence that separates the last element of the last symbol of one word from the first element of the first symbol of the next word. It lasts seven ticks *minus* any weight adjustment.

keyer A device that eases the sending of Morse code. It allows a single action to generate a whole series of "sounds" with the desired timing. The earliest were mechanical, and colloquially called "bugs", and only helped with the timing of "dits", but modern versions tend to be electronic and do much more.

Morse Morse code, named for its inventor, Samuel Finley Breese Morse, is a method of communicating text by the timing of a sequence of events. Originally the events were the raising and lowering of a pencil over a moving paper tape, producing long and short "marks" and "spaces". Mariners have extensively used a series light and darkness from a shuttered lamp. But most of us now think of it as a series of sounds and silences, and those are the terms used here. See dit, dah, IES, ISS, IWS and ITS above.

paddle The control handle or handles of a keyer. As distinct from a telegraph "key", a paddle is operated by pressing it sideways, rather than down, and either moves in both directions, selecting between dits and dahs, or there are two back to back, operated by pressing one toward the other.

prefix A symbol that changes the interpretation of what follows. Usually just the following symbol is re-interpreted, but when multiple prefixes occur in a row the effect may apply to the next non-prefix symbol, or to the result of applying the latter prefix to the next non-prefix symbol, or it can modify the immediately following prefix. Prefixes may be composed of one symbol, or of more than one symbol..

symbol A series of elements separated only by IES and surrounded by longer silences.

system My (KE1G) scheme for extending Morse code to handle computer keyboard duties, or an implementation of hardware and/or software accepting that scheme and producing the corresponding computer, etc., inputs.

tick Basic measure of time related to Morse code transmission speed. This is a personal term of mine, used in the definition of code speed, see WPM.

WPM Words Per Minute. A measure of Morse transmission speed. While words vary in length, the "official" word is "Paris", including the following word space. That's 50 ticks long, so the duration of a tick in seconds is 1.2 divided by WPM, which gives surprisingly accurate average over most English texts.