Morse Code Over Packet Protocol (MOPP)

```
Date: June 06, 2020
Version: 1.0
Authors: Willi, OE1WKL
```

Introduction

In order to send Morse code over either LoRa or over IP, both of which transmit byte strings as packet payloads, we need a protocol to represent Morse code within these byte strings. For the Morserino-32 the following protocol (calles MOPP for short) has been established (which could of course be used by other hardware or software tools).

Design Goals

This document describes version 1 of a protocol that has the following design goals:

- Allow transmission of Morse code over data protocols that use payloads that usually consist of a number of 8-bit bytes.
- Avoid Zero bytes, as they might be interpreted as "end of string"
- Send Morse Code as faithful as possible, do not assume it is International Morse Code only - there are many languages that use their special characters in Morse, some very elaborate (Japanese). Yes, even obvious errors should be sent as they are! Therefore "transcoding" into ASCI or UTF is not really an option.
- Include information about the original speed, so that the receiver can "play" the code at the same speed as it was created.
- The encoding must be very efficient, as some packet protocols (especially LoRa, in our case) do not have a lot of bandwidth, especially when you try to obey legal requirements (only 1% uptime). Therefore sending a separate packet for each dit or dah, or even for each character, would lead to a lot of overhead and should be avoided. On the other hand, as the origin is usually created by a human operator, the chunks being sent should not interfere too much with the natural flow of Morse code data exchange, and so the latency should not become too large either.

Approach

- As a compromise between overhead and latency each word (which for our purpose is defined as a succession of characters followed by a pause that is longer than the pause between individual characters) is being transmitted as a single data packet.
- In order to encode Morse faithfully, we recognize that Morse code uses a total of four (4) different information entities or "states":
    - a short element (usually called a dot or a "dit") followed by a short pause, of the same length as the "dit" (a "dit" is called a "di" when it is not the last element of a character),
    - a long element (usually called a dash or a "dah") also followed by a short pause,
    - a longer pause (in total with the length of the preceding pause it has the length of a "dah") that signifies the end of character (EOC),
    - and even longer pause (which adds another 4 "dit"-lengths to the EOC pause, giving a total of 7 dit lengths, which signals the end of a word (EOW).
- This means with 2 bits we can encode these 4 states:
    - dit: 01b
    - dah: 10b
    - EOC: 00b
    - EOW: 11b

    By using this approach, we pack up to 4 Morse elements into an 8-bit byte.

    Note 1: At the end of a word, we immediately use EOW, and do not encode EOC first.

    Note 2: To encode EOW seems not to be necessary, as we send each word as its separate packet, and so this could be seen as redundant. But we include it nevertheless: while our encoding could end a word at any 2-bit boundary but data packets need to have the length of an integer number of 8-bit bytes, we need a method to indicate the end of the word, unless it coincides with the end of a byte. If it does not, it might have used up 2, 4 or 6 bits of a byte already, and we use 11b to encode EOW, and disregard the remaining 4 or 2 bits.
- We encode the speed at origin as an integer number between 5 and 60, using words per minute as the measure here. Obviously we need 6 bits to encode this range of numbers.
- We encode the protocol version (should we ever find a need for a different MOPP protocol) as 2 bits, only allowing the values 01b (for version 1 as described in this document), 10b and 11b. 00b is not allowed.
- To have at least a slight chance to detect invalid packets, we include into each packet a 6 bit serial number, starting with a random value, and incremented with each packet sent (we could use this for detecting a) missing packets, and/or b) noise (when the random number is not incrementing from packet to packet). Currently the serial number is not being used in the Morserino-32 software.

Complete Data Packet Format

Packet Header:

- 2 bits Protocol Version (for now always 01b)
- 6 bits Serial Number
- 6 bits Speed at origin in WpM (only 5 - 60 decimal are allowed)

Payload

The Payload starts immediately after the header, i.e. with the two least significant bits of byte #2. Payload is encoded as described above. (Words consisting just of the letter "e" or "t" would thus lead to a total packet length of 2 bytes, as the dit (for e) or dah (for t) fit into the remaining bits of byte 2.)

Example

Let's assume we want to transmit the Morse code for the word "PARIS", sent at a speed of 16 WpM. (we use binary representation throughout)

- Protocol version: 01
- Serial number: e.g. 011011
- Speed: 010000
- 'P' (di dah dah dit EOC): 01 10 10 01 00
- 'A' (di dah EOC): 01 10 00
- 'R' (di dah dit EOC): 01 10 01 00
- 'I' (di dit EOC): 01 01 00
- 'S' (di di dit EOW): 01 01 01 11

If we pack all of this into a byte string, we get the following:

```
01011011 01000001 10100100 01100001 10010001 01000101 01110000
```

(The last 4 zero bits are just here to fill up to the next byte boundary).

In hexadecimal notation this would be:

```
5B 41 A4 61 91 45 70
```

As an ASCII string this would read as:

```
[A¤a`Ep
```

We see that the encoding is quite efficient - for a word of 5 characters we need 7 bytes in this example,

including the speed information and the packet header overhead (and the fact that we give away four bits at the end of the word).

Obviously, as Morse code uses variable character lengths, this cannot be a rule that is always valid; numbers (always 5 elements long for each figure) need more space than common words (because the most frequent characters, at least in English, use an encoding that is only 1, 2 or 3 elements long).