HQSL Format Specification (v1.0.0)

Eugene Medvedev (AC1PZ)

1. Introduction

QSL cards are a traditional amateur radio practice, a document certifying that a radio contact between two stations actually took place, sent by both correspondents through means other than radio. Such documents are instrumental in amateur radio competitions of any kind and are involved in the attainment of various amateur radio honors, so it is important for them to be verifiable and convenient to reference.

While in modern day, the historical paper QSL card is frequently replaced by other means of confirmation, like records in a centralized database, paper cards remain in wide use, as they still serve an important function - they’re inherently independent of any central authority, or even the original sender.

The purpose of the digitally signed QSL format described herein is not to replace paper QSL cards, nor it is to replace centralized databases, but rather, to bridge the gap between them by producing a document that can serve the purposes of both, depending on situation.

2. Conventions used in this document

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119 [1] when, and only when, they appear in all capitals, as shown here.

- **Fragment-safe US ASCII** is the range of the following US ASCII characters:
  - `0-9`, `A-Z`, `a-z`
  - `?/:@-._~!$&'()*+;=`

  To be more explicit, these are the characters with codes `0x21`, `0x24`, `0x26-0x29`, `0x2a-0x2b`, `0x2d-0x3b`, `0x3d`, `0x3f-0x5a`, `0x5f`, `0x61-0x7a`, `0x7e`, inclusive. It is the list of characters safe to use in a fragment identifier (hash) of an URL, sans the HQSL field separator (comma, `0x2c`) and the %-encoding character (`%`). This list can be derived through careful study of STD 66 ([2]). While `%`-encoding can be used to package arbitrary data into the fragment identifier, HQSL format explicitly does not make use of this. Keep in mind that this list of characters does not include any whitespace characters.
- **Sender** is the originator of the QSL card.
- **Correspondent** is the other party of a communication described by the QSL card.
- **Verifier** is a person or organization attempting to verify the authenticity of a QSL card.
- **Certifier** is a person or organization professing to certify that the Sender is licensed to use a particular call sign during a particular time period.

3. Requirements

A digital QSL card requires the following properties to be useful:

1. It must be human-readable without specialized software.
2. It must be machine-readable and format-agnostic with respect to method of delivery, i.e. function identically, whether it was delivered as a string of characters, a digital file, a printed card, or a picture of that card.
3. The data must be cryptographically signed.
4. The signature must be verifiable with as much independence from any central authority as possible.
5. The card must remain readable, parsable and verifiable for decades, preferably in perpetuity.

All of this this does not preclude the use of a centralized database, with all the benefits associated with such databases, but must exist in addition to them.

None of the prior attempts to introduce a digital QSL card distinct from a centralized database found so far (See [3], [4], [5], [6]) fit all of the above requirements.

Radio amateur practice is inherently tied to government licensing of this activity, and historically, amateur radio organizations maintaining QSO databases and other amateur infrastructure have been verifying government-issued license documents before allowing users to participate. There is no obvious way to maintain what privacy remains to an amateur radio operator without someone playing that role. As a result, a digital QSL card system is also in need of a public key infrastructure that has to have the following properties:

1. It must be possible for an arbitrary number of certifiers to exist.
2. Certifiers must be able to securely rely on each other’s license verification work if they choose to.
3. It must be possible for a verifier to establish a chain of trust, from a certifier they choose to trust, to a particular public key, using as little specialized software as possible.
4. It must be feasible for any organization or individual to take up maintenance of the public key infrastructure using readily available open source software.

4. HQSL digitally signed QSO format

An HQSL is a human readable format for storing a single-direction QSO record as would be presented on a paper QSL, meant for encoding in a machine-readable QR Code. That QR Code is then to be added to a printed QSL card, or a printable image of such a card.

Failing that, the text of the HQSL can be sent digitally by other means – as a file containing HQSL data, as an URL identical to the one packaged into a QR Code, or as a simple text string.

4.1. QSO data

Traditional definitions of what is considered a sufficient record of a QSO differ, but most converge on the following set of data, omitting at most one element:

1. The sender’s call sign.
2. The UTC date and time of the start of communication.
3. A signal report in the format appropriate to the mode of communication.
4. The mode itself.
5. The frequency, band, or both.
6. The location of the sender at the moment of communication, which is key in certain contests and diplomas.

The following definitions are motivated by the requirements of keeping this data human-readable, representing it as a QR Code that will be decipherable without software more specialized than a generic QR Code reader, and maintaining the barcode representation on paper at an appreciably small size.

4.1.1. Call sign

- Call signs MUST be in capitals. They MUST NOT contain any symbols beyond `A-Z`, `0-9`, `-` and `/`.
- They MUST include any prefixes or suffixes required by regulations for that particular QSO, like those mandated by Canada-United States Reciprocal Operating Agreement or CEPT Recommendation T/R 61-01, matching the prefixes and/or suffixes used on the air.
- They SHOULD NOT include any optional suffixes. (`/QRP`, `/MM`, etc.)

4.1.2. UTC date and time

UTC date and time MUST be written as `YYYYMMDDHHMM`, where `YYYY` is the year, `MM` is the number of the month, `HH` is the number of the hour out of 24 hours, and `MM` is the number of minutes. Example:

```
202401201604
```

All calculations that rely on the use of seconds MUST assume 0 seconds.

4.1.3. Signal report

Different communication modes historically prescribe using different formats for signal report – RST, RS, RSQ, the positive and negative dB as used in FT8, etc. As such, signal report is an arbitrary string, that MUST only contain characters in fragment-safe US ASCII range.

4.1.4. Mode

Due to the need to keep the QSL record human readable, and in the interest of future-proofing, a mode designator is also an arbitrary string. It MUST consist of characters in fragment-safe US ASCII range.

Implementations SHOULD prefer what the ADIF standard [7] calls “submodes” to what it calls “modes” and use the generally accepted mode designations.

4.1.5. Frequency

While historically, band names are used for QSL accounting more often, the list of bands available to amateurs may change in the future and did change in the past, and is dependent on local regulations. More importantly, while a band can always be computed from the frequency, the reverse is not true. As such, HQSL format stores a frequency in megahertz, rather than band, and leaves computing the band from frequency to rendering software. In cases of crossband operation, the sender MUST use the frequency they were transmitting at. In cases where the sender is an SWL, (Shortwave Listener) there is only one frequency to consider, the one they're listening at.

The frequency MUST be given in megahertz and written in a normalized form:

1. Decimal point (`.`, `0x2E`) is used as the decimal separator.
2. Frequencies above 1 MHz are given to a precision of no more than 3 digits after the decimal, truncated rather than rounded.
3. Trailing and leading zeroes are omitted. In the particular case of sub-1MHz frequencies, this means that the frequency will start with a decimal point.
4. Trailing decimal point MUST be removed.

Examples:

- 18074 kHz: `18.074`
- 1358 Hz: `.001358`
- 18050 kHz: `18.05`
- 18000 kHz: `18`
- 10050074 kHz: `10050.074`

In cases where the band is known, but the specific frequency was not recorded, whether due to deficiencies in equipment or record-keeping, it is RECOMMENDED that the frequency given in the HQSL is exactly in the middle between the start and end of appropriate band segment according to locally mandated regulations.

It is RECOMMENDED that the parsing software determine the band, if required, by computing the middle frequencies of every band and picking the one closest to the frequency given in the HQSL, thus ensuring that this computation always results in a band designation.

4.1.6. Location

The location of the sender is a Maidenhead locator [8] string, which MUST be given to a precision of at least 4 characters (2 pairs) and SHOULD be given to as much precision as is required to completely resolve the district ambiguity – that is, determine which national district subdivision (county/district/province) or special location (POTA/IOTA,etc) the transmitter was in. The precision is considered sufficient when the rectangle described by the Maidenhead locator contains the position of the actual transmitting antenna and only covers exactly one such subdivision. No reasonable geographic position should require a precision of more than 10 characters.

Implementations MUST ignore case when parsing, so that both the correct form of `AA11BB` and the more commonly seen `AA11bb` form get interpreted correctly.

4.2. HQSL Record

A full HQSL record is a sequence of fields separated by a “`,`” (comma, `0x2c`) character. To save as much of the limited QR code data volume as possible, the content of the field is identified by its order in the record, and not by any prefixes or field identifiers:

1. Sender’s call sign as per 4.1.1.
2. Sender’s location as per 4.1.6
3. Correspondent’s call sign as per 4.1.1
4. Date and time of the start of communication as per 4.1.2
5. Signal report as per 4.1.3
6. Frequency as per 4.1.5
7. Mode as per 4.1.4
8. Extra data.
9. An empty field reserved for future expansion of the standard. MUST be empty.
10. Digital signature.

Every field MUST be present. Implementations MUST be capable of reading and displaying HQSLs even if fields 5 (signal report), 8 (extra data), and 9 (reserved) are empty. They SHOULD completely reject HQSLs in which other fields are empty.

Due to size limitations of QR Codes, there is no specification for storing more than one HQSL in a single QR Code.

4.2.1. Digital signature

Digital signature is a detached binary (type `0x00`) OpenPGP signature as described in section 5.2 of RFC4880. [9]

- The signature is computed over the entire preceding HQSL record, including the field separators, but excluding the separator between the signed part and the signature itself.
- This signature MUST contain the Public Key ID packet in addition to the packets OpenPGP considers mandatory, and SHOULD contain as few other packets as feasible.

The signature is then encoded in Base 36 as described in Appendix 2 and attached to the HQSL record as an extra field, with its own field separator character.

Unsigned HQSLs are also possible, and MUST contain the text “`UNSIGNED`” in their digital signature field in place of actual signature data. (This is needed to prevent

URL parsing machinery across the Internet from mangling such HQSL URLs, see Appendix 2) Implementations MUST gracefully handle such non-signatures.

4.2.2. Extra data

Extra data field MUST contain only characters in fragment-safe US ASCII range and MAY be used for information inherent in a QSL that is not covered by other fields, like contest exchange strings, POTA and SOTA information, etc. Individual items within the extra data field SHOULD be separated with a “`;`” (semicolon, `0x3b`). Where encoding a space is required, implementations SHOULD use an `_` (underscore, `0x5f`) instead.

4.3. HQSL file storage

When desired, an HQSL MAY be saved into a file and delivered through other methods, (email, Winlink, pigeon post, sneakernet, etc.) in which case, the file SHOULD be predictably named in the following pattern:

```
<sender>_<correspondent>_<datetime>.hqsl
```

where “sender” is the sender’s call sign, “correspondent” is the correspondent’s call sign (as per 4.1.1, but the character `/`, if present, MUST be replaced with `-`), while date and time are given as in 4.1.2.

4.4. QR Code generation and printing

Applications purporting to deal with printed versions of a HQSL MUST support printing and/or reading them as QR Codes as per ISO/IEC 18004:2015. [10]

When packaged into a QR Code, HQSL data MUST be preceded by a header, thus turning it into a full-featured URL. Example:

```
https://hqsl.net/h#
```

The HQSL data itself becomes the URI fragment (hash) part of that URL. Keep in mind that the header is not part of the digitally signed data, nor part of a HQSL proper.

Implementations SHOULD ensure that the URL itself contains a web application capable of parsing the HQSL data in the URI fragment, verifying the keys and reporting to the user, thereby permitting a user to verify an HQSL without installing any software at all. In this case, which certifier keys will be trusted and which key servers to use is up to the maintainer of that web application.

In cases where the URL does not point at a web application, it MUST instead contain a static web page, describing what an HQSL is, and how it may be interpreted and verified.

Installable software MAY, in turn, ignore the contents of the header entirely, and use whichever trusted certifier keys and key servers the user chooses to configure. Implementations which accept HQSL data by routes other than decoding a QR Code MUST be able to handle HQSLs both with a header and without.

It is RECOMMENDED that implementations encode the signature field as a separate “alphanumeric” mode data block while the entire preceding HQSL record and the URL header are encoded as a “bytes” mode block, which is explicitly a feature of the QR code standard. This is the entire reason a Base 36 encoding was specified, and improves the space efficiency of the QR code by as much as 10%. See [11] for more technical detail on the method.

Libraries capable of manually producing mixed mode QR Codes or automatically optimizing QR Code data to make use of mixed modes are available in multiple languages:

- Javascript: https://github.com/soldair/node-qrcode
- Python: https://github.com/lincolnloop/python-qrcode
- Go: https://github.com/piglig/go-qr
- PHP: https://github.com/chillerlan/php-qrcode/
- Java, TypeScript, JavaScript, Python, C++: https://www.nayuki.io/page/qr-code-generator-library

5. HQSL Public Key Infrastructure

OpenPGP standard has been the target of much criticism in recent years. However, with the requirements and intended use of HQSL given in section 3, most of the objections typically seen (See [12] for a summary) only apply to scenarios involving encrypted communication within an email-based network. Since HQSL exclusively concerns itself with signatures, most of the objections listed are either not a detriment or follow directly from the requirements. Using a well-established standard, in turn, provides a selection of ready-made software that can be used to build the key infrastructure.

As such, HQSL format requires the use of OpenPGP-compliant secret and public keys, and an OpenPGP-based key certification system. The remaining drawbacks of OpenPGP are mitigated by decoupling from email entirely: User IDs mandated by HQSL are deliberately not email addresses and do not contain personal names. Connections between them and actual individuals can thus be confined to government-run databases, outside the scope of amateur radio infrastructure maintained by the amateurs themselves.

In an ideal world, key certification systems would be run by the same organizations that issue amateur radio call signs, or, failing that, by national amateur radio unions. However, in practice, most likely, only major amateur radio QSO databases will simultaneously have the resources, reputation and expertise to certify public keys. It is the aim of this document to provide a common language allowing them to operate compatible, interconnected public key servers, leaving it to the individual verifiers to decide which key certifications they will trust.

You can consult [13] for a deeper understanding of OpenPGP internals.

5.1 OpenPGP key generation for HQSL

It is the expectation that implementations use ready-made libraries to handle OpenPGP and don't attempt to build the entire system from scratch. There are no explicit limits on the kinds of keys used, and implementations SHOULD keep up with the standard as it is updated to the changing cryptography landscape.

However, implementations MUST be able to handle `ed25519/cv25519` key pairs, (as prescribed in [14]), and MUST be able to use `SHA-256` hashes for signing, and OpenPGP v4 key and signature formats, as that is selected as the baseline for HQSL. Implementations SHOULD use `ed25519` keys and EdDSA for all signing and certification operations, unless something markedly superior has emerged since the time this document was written and became widely enough supported.

Signer public keys used with HQSL MUST have a User ID in the following format:

```
Amateur Radio Callsign: <callsign>
```

where "callsign" is the sender call sign written as per section 4.1.1. Callsigns in User IDs MUST be given without any additional prefixes or suffixes – the intention is that a callsign owner needs only one public key, no matter how many prefixes or suffixes they have to operate with at different times. There is no requirement this be a primary or only User ID, so a signer can use the same key for all callsigns they are licensed to operate with. Example:

```
Amateur Radio Callsign: AC1PZ
```

Should `AC1PZ` operate in Canada, and thus be required to give their callsign on air as, for example, `AC1PZ/VE3`, the HQSL for such a QSO would contain `AC1PZ/VE3` in the sender field, but the User ID on the signing key would remain `AC1PZ`.

- Signer keys MAY additionally have any other User IDs in any desired format, but only User IDs in this format are the subject of HQSL key certification.
- One key may have as many callsign User IDs as needed, but certification applies to individual User IDs, rather than the entire key.

It is RECOMMENDED that senders use dedicated keys for signing HQSL, and only use them to identify themselves as radio amateurs, rather than owners of email addresses, even though the option to do otherwise remains open.

5.2 Key certification

It is expected and normal that multiple keys with identical User IDs as per section 5.1 may exist, if only because a single call sign may belong to multiple people and organizations over time. The purpose of key certification is to establish a correspondence between a time period, a call sign, and a specific key pair, vouched for by a given trusted authority. The certifier is making a signed claim that the person in control of a particular private key is legally entitled to operate under a specific amateur radio callsign during a specific time period, and encodes this time period in the certification signature.

OpenPGP standard permits the addition of notation metadata to signatures, and this is how certifiers make their claims of correspondence. (See section 5.2.3.16 of [9]) Compatible implementations MUST have certifiers sign the section 5.1 conformant User IDs on signer keys (rather than the key itself or any other part of OpenPGP key data structure) with a published certifier key that will be available to verifiers, adding a notation named `qsl@hqsl.net`, with the value formatted in the following way:

```
<callsign>,<start datetime>,<end datetime>
```

Where “callsign” is the callsign being certified, written as on the User ID being signed, and start and end dates are given as per 4.1.2, in UTC. Example:

```
AC1PZ,202309190500,203309190500
```

In this example case, the start and end datetimes are the start and end dates of the amateur license grant, converted to UTC.

Pairs of start/end datetimes may recur as much as needed in cases where the period being certified is not a single consecutive stretch of time. Example:

```
AC1PZ,202309190500,203309190500,203309210500,204309210500
```

It is the responsibility of the certifier to decide which dates they certify and how to investigate whether the signer is actually licensed to operate during this time period. The procedures the certifier employs to verify that the signer key is owned by that person or organization are also outside the scope of this document.

OpenPGP standard does not go into much detail on the semantics of the signatures, so it is important to strictly define their meanings for the purposes of

HQSL, as well as define what features of the OpenPGP standard are allowed for use in HQSL certification signatures:

- In all cases, verifiers MUST examine certifier signatures on signer User IDs in chronological order as per the dates given within the signature structure, (and not the order of appearance) and only consider the latest valid signature by any given certifier key. The only exception is a certification revocation: the presence of a valid revocation invalidates all other certification signatures on the same User ID by the same certifier key, no matter when they were made.
- A certification signature MUST only contain exactly one notation named `qsl@hqsl.net`. (OpenPGP standard actually allows an arbitrary number of identically named notation packets, which is far less widely supported than it should be.) The behavior of a verifier when encountering multiple notations with that name is undefined. (Depending on the library used, the verifier programmer might not even have an option to detect that more than one such notation exists.)
- The certifier's signature MUST NOT have an expiration time and MUST NOT be irrevocable. Verification software MAY ignore expiration times or irrevocability flags on signatures entirely if encountered.

Implementations MUST accept the signatures on individual HQSLs as valid only when all the following conditions are met:

1. The signature made by the sender key on the HQSL data is valid.
2. The sender key is valid and has not been revoked.
3. The date on the signature itself overlaps the validity period of the sender key. (See 5.2.3.6 in [9])
4. The sender public key has a valid certification signature by a trusted certifier key, applied to the user ID matching the call sign of the HQSL sender, which has prefixes and suffixes removed for this comparison.
5. The certifier key is itself valid and has not been revoked.
6. The said certifier signature contains a `qsl@hqsl.net` notation in the correct format.
7. The HQSL QSO datetime is between the start and end datetimes given in that notation, inclusive.

5.3 Key distribution

When delivered as a QR code, an HQSL is explicitly an URL which points to a web server.

This web server SHOULD, in addition to delivering a web application capable of decoding and verifying an HQSL, also respond as an HKP-over-TLS key server (as described in [15], with `hkps://<domain>` and `https://<domain>` being equivalent) and MAY additionally respond on the HKP port (`hkp://`, port 11371). Failing that, the web application itself MUST be aware of the location of a working keyserver and perform key updates during verification.

Such key servers SHOULD NOT accept key removal requests for HQSL signing keys, as that would compromise the primary purpose of the system. HQSL format deliberately does not require any personal data to be present in the keys to make it unnecessary to permanently remove keys to comply with GDPR and other data protection laws – a call sign does not inherently identify an individual without a record in a government licensing database.

If the aforementioned key server is maintained by a certifier organization or individual, it SHOULD additionally publish the public key (or keys) that can be used to verify their own certifications at a well-known URL `https://<domain>/.well-known/hqsl-certifier.asc` in ASCII armor format.

5.4. Key and certification revocation

The primary purpose of this key infrastructure is signatures, not encryption. Revocation strategies based on the idea that a key may be valid until a given point in time, but stop being valid afterwards, are ineffective, since it is far from abnormal to send out QSLs years and occasionally decades after the fact.

As such, implementations SHOULD reserve key revocation for actual breaches, like keys known to be stolen.

For similar reasons, it is also RECOMMENDED that key expiration is avoided: In an email-centric, encryption-first setting, periodic key expiration serves to prevent people from sending email to addresses which are no longer current. With callsigns, it simply does not apply.

At the same time, certifiers SHOULD, where possible, maintain the correctness of their certification by adding new signatures on sender keys to reflect changing circumstances, like license prolongation or revocations.

Appendix 1: HQSL Example

Figure 1: Example QR code containing an HQSL

Decoding this should result in the following HQSL (hard-wrapped to the width of 78 characters):

```
https://hqsl.net/h#AC1PZ,FN42gv,W1KOT,202402081323,+00,18.101,FT8,59_05,,19H4V
9DABY5VH3WE05MV34Z5JBEBJRD9Q7VTLB98L789GFL79P56QWFX0JHV3U6VSEXRODMYLOZ40UM798E
V4FSPVY8YVMQ0WLZA66Q38VW0G6PV23O6Y65PK94NZE5B381MHOPR4NJJU67QC25JW85JL23V644BL
P0HD8KBY2MODEBRICTZ5C0LC
```

To clarify, the same in hexadecimal:

```
00000000  68 74 74 70 73 3a 2f 2f  68 71 73 6c 2e 6e 65 74  |https://hqsl.net|
00000010  2f 68 23 41 43 31 50 5a  2c 46 4e 34 32 67 76 2c  |/h#AC1PZ,FN42gv,|
00000020  57 31 4b 4f 54 2c 32 30  32 34 30 32 30 38 31 33  |W1KOT,2024020813|
00000030  32 33 2c 2b 30 30 2c 31  38 2e 31 30 31 2c 46 54  |23,+00,18.101,FT|
00000040  38 2c 35 39 5f 30 35 2c  2c 31 39 48 34 56 39 44  |8,59_05,,19H4V9D|
00000050  41 42 59 35 56 48 33 57  45 30 35 4d 56 33 34 5a  |ABY5VH3WE05MV34Z|
00000060  35 4a 42 45 42 4a 52 44  39 51 37 56 54 4c 42 39  |5JBEBJRD9Q7VTLB9|
00000070  38 4c 37 38 39 47 46 4c  37 39 50 35 36 51 57 46  |8L789GFL79P56QWF|
00000080  58 30 4a 48 56 33 55 36  56 53 45 58 52 4f 44 4d  |X0JHV3U6VSEXRODM|
00000090  59 4c 4f 5a 34 30 55 4d  37 39 38 45 56 34 46 53  |YLOZ40UM798EV4FS|
000000a0  50 56 59 38 59 56 4d 51  30 57 4c 5a 41 36 36 51  |PVY8YVMQ0WLZA66Q|
000000b0  33 38 56 57 30 47 36 50  56 32 33 4f 36 59 36 35  |38VW0G6PV23O6Y65|
000000c0  50 4b 39 34 4e 5a 45 35  42 33 38 31 4d 48 4f 50  |PK94NZE5B381MHOP|
000000d0  52 34 4e 4a 4a 55 36 37  51 43 32 35 4a 57 38 35  |R4NJJU67QC25JW85|
000000e0  4a 4c 32 33 56 36 34 34  42 4c 50 30 48 44 38 4b  |JL23V644BLP0HD8K|
000000f0  42 59 32 4d 4f 44 45 42  52 49 43 54 5a 35 43 30  |BY2MODEBRICTZ5C0|
00000100  4c 43                                             |LC|
00000102
```

Appendix 2: Base 36 Encoding

RFC9285 [16] recommends that binary data packaged into a QR code be encoded in Base 45 format, which then can be encoded as an alphanumeric data block, resulting in an optimal use of space. However, the HQSL data record also constitutes a valid URL with a fragment and is meant to be transferred as such. It cannot use Base 45 encoding as is, because it requires characters which have special meanings in URLs: `%` (`0x25`) and space (`0x20`). Additionally, many online applications, social networks and instant messengers will mis-parse URLs when they end in any character other than `0-9A-Z`, even if they have no complaints about encountering other punctuation inside an URL.

To deal with that, HQSL specifies a Base 36 encoding, for which, the Base 58 algorithm (as used in Bitcoin addresses) is used, due to being easy to generalize for arbitrary alphabets of arbitrary length under 256 characters.

The alphabet used for Base 36 in HQSL is as follows:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Encoding a sequence of octets as Base 36 is performed like this:

1. Leading zero octets are counted and removed from the sequence.
2. The remaining octets are interpreted as a single big-endian integer.
3. The integer is converted to base 36, and every digit is replaced by the corresponding letter of the chosen alphabet.
4. Leading zero octets are reproduced in the output as the corresponding number of the zeroth character of the alphabet, then the rest of the converted digits are output.

Decoding is performed in the reverse:

1. Leading zeroth characters of the alphabet are counted and removed from the sequence.
2. The remaining characters are interpreted as digits of a large base 36 number.
3. Leading zero octets as counted just prior are reproduced in the output, followed by the octets of the recovered base 36 number converted to bytes.

Example Python code for encoding and decoding Base 36

```
#!/usr/bin/env python

"""
Example Base36 encoder and decoder functions.

This implementation is basically somewhat simplified and commented
code from https://github.com/keis/base58/
and is presented here for illustrative purposes.
"""

# The actual alphabet we're using, as a line of bytes.
ALPHABET = b"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

# Derived value of the number base, in our case, 36.
BASE = len(ALPHABET)

# A map of character to its index in the alphabet.
MAP = {chr(char): index for index, char in enumerate(ALPHABET)}


def encode_int(that_integer: int) -> bytes:
    """
    Encode an integer as a Base43 string.
    """

    # We start with an empty buffer of bytes.
    buffer = b""

    # While that_integer remains
    while that_integer:
        # Divide the integer by BASE, producing
        # the division result and the modulo.
        that_integer, idx = divmod(that_integer, BASE)
        # The division result goes on to the next loop,
        # while the modulo is the number of character in our alphabet.
        buffer = ALPHABET[idx : idx + 1] + buffer

    return buffer


def encode(source: bytes) -> str:
    """
    Encode a sequence of bytes as a Base43 string
    """
```

```python
    # A zero length sequence of bytes encodes to an empty string,
    # so this is a trivial optimization.
    if len(source) == 0:
        return ""

    # Zero bytes at the beginning are stripped and counted.
    origlen = len(source)
    source = source.lstrip(b"\0")
    newlen = len(source)

    # What remains is converted to a big bigendian integer.
    acc = int.from_bytes(source, byteorder="big")

    # And then converted as if it were an integer.
    output = encode_int(acc)

    # The initial zeroes are encoded as the first character of the alphabet
    # and restored, the encoded integer follows.
    return (ALPHABET[0:1] * (origlen - newlen) + output).decode("utf-8")


def decode_int(source: str) -> int:
    """
    Decode an encoded string as an integer.
    """

    # We start with a zero.
    decimal = 0

    # This is exactly the reverse of encode_int:
    # Every character is converted to its index in the alphabet,
    # and summed up into the big integer.
    for char in source:
        decimal = decimal * BASE + MAP[char]
    return decimal


def decode(source: str) -> bytes:
    """
    Decode an encoded string into bytes.
    """

    # Identify and count the leading zeroes.
    origlen = len(source)
    source = source.lstrip(str(ALPHABET[0:1]))
    newlen = len(source)

    # Decode the rest as a single integer
    acc = decode_int(source)

    # Convert the resulting integer to bytes, restoring the leading zeroes
    # through allocating a buffer longer than the bits of the
    # integer we produced.
    return acc.to_bytes(origlen - newlen + (acc.bit_length() + 7) // 8, "big")
```

Readily made implementations of this algorithm that can be used with an arbitrary alphabet and will handle Base 36 if given the alphabet used in HQSL exist for multiple languages:

- JavaScript: https://github.com/cryptocoinjs/base-x
- Python: https://github.com/keis/base58/
- C#: https://github.com/ssg/SimpleBase

References

[1] S. O. Bradner, “Key words for use in RFCs to Indicate Requirement Levels,” RFC 2119; RFC Editor, Standard 2119, Mar. 1997. doi: 10.17487/RFC2119. Available: https://www.rfc-editor.org/info/rfc2119

[2] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” RFC 3986; RFC Editor, Standard 3986, Jan. 2005. doi: 10.17487/RFC3986. Available: https://www.rfc-editor.org/info/rfc3986

[3] Chris R. Burger ZS6EZ, “A perspective on electronic QSLing.” Mar. 2002. Available: https://zs6ez.org.za/articles/e-qsl.htm

[4] C3 Committee, “QR Code on QSL Cards,” in *IARU region 1 general conference*, 2017. Available: http://www.oh3ac.fi/IARU2017/LA17_C3_47 DARC - QR Code on QSL Cards.pdf

[5] Logbook of the World, “Developer Information: Digitally Signed Log File Structure.” 2017. Available: https://lotw.arrl.org/lotw-help/developer-tq8/

[6] Song Guo BG6TOE, “Digital signature of QSL cards.” Jun. 2023. Available: https://blog.matsu.dev/post/2023/06/digital-qsl-en/

[7] “Amateur Data Interchange Format,” Standard. Available: https://adif.org/adif

[8] Wikipedia contributors, “Maidenhead locator system — Wikipedia, the free encyclopedia.” 2023. Available: https://en.wikipedia.org/w/index.php?title=Maidenhead_Locator_System&oldid=1192700741

[9] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw, “OpenPGP Message Format,” RFC 4880; RFC Editor, Standard 4880, Nov. 2007. doi: 10.17487/RFC4880. Available: https://www.rfc-editor.org/info/rfc4880

[10] “Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification,” International Organization for Standardization, Geneva, CH, Standard, Mar. 2015. Available: https://www.iso.org/standard/62021.html

[11] Nayuki, “Optimal text segmentation for QR Codes,” Available: https://www.nayuki.io/page/optimal-text-segmentation-for-qr-codes

[12] Latacora LLC, “The PGP Problem.” 2019. Available: https://www.latacora.com/blog/2019/07/16/the-pgp-problem/

[13] H. Schaefer, P. Schaub, and T. L. Coles, *OpenPGP for application developers*. 2024. Available: https://openpgp.dev/book

[14] P. Wouters, D. Huigens, J. Winter, and N. Yutaka, “OpenPGP Message Format,” Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-ietf-openpgp-rfc4880bis-10, Jan. 2024. Available: https://datatracker.ietf.org/doc/draft-ietf-openpgp-crypto-refresh/

[15] D. Shaw and A. Gallagher, “OpenPGP HTTP Keyserver Protocol,” Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-gallagher-openpgp-hkp-03, Dec. 2023. Available: https://datatracker.ietf.org/doc/draft-gallagher-openpgp-hkp/03/

[16] P. Fältström, F. Ljunggren, and D.-W. van Gulik, “The Base45 Data Encoding,” RFC 9285; RFC Editor, Standard 9285, Aug. 2022. doi: 10.17487/RFC9285. Available: https://www.rfc-editor.org/info/rfc9285