Workgroup: Internet Engineering Task Force
Internet-Draft: draft-devault-bare-07
Published: 11 May 2022
Intended Status: Informational
Expires: 12 November 2022
Authors: D. DeVault
SourceHut

Binary Application Record Encoding (BARE)

Abstract

The Binary Application Record Encoding (BARE) is a data format used to represent application records for storage or transmission between programs. BARE messages are concise and have a well-defined schema, and implementations may be simple and broadly compatible. A schema language is also provided to express message schemas out-of-band.

Comments

Comments are solicited and should be addressed to the mailing list at ~sircmpwn/public-inbox@lists.sr.ht and/or the author(s).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

The purpose of the BARE message encoding, like hundreds of others, is to encode application messages. The goals of such encodings vary (leading to their proliferation); BARE's goals are the following:

- Concise messages
- A well-defined message schema
- Broad compatibility with programming environments
- Simplicity of implementation

This document specifies the BARE message encoding, as well as a schema language that may be used to describe the layout of a BARE message. The schema of a message must be agreed upon in advance by

each party exchanging a BARE message; message structure is not encoded into the representation. The schema language is useful for this purpose but not required.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Use-cases

The goals of a concise, binary, strongly-typed, and broadly-compatible structured message encoding format support a broad number of use-cases. Examples include:

* Self-describing authentication tokens for web services
* Opaque messages for transmitting arbitrary state between unrelated internet services
* A representation for packets in an internet protocol
* A structured data format for encrypted or signed application messages
* A structured data format for storing data in persistent storage

The conciseness of a BARE-encoded message enables representing structured data under strict limitations on message length in a large variety of contexts. The simple binary format may also be easily paired with additional tools, such as plain-text encodings, compression, or cryptography algorithms, as demanded by the application's needs, without increasing the complexity of the message encoding. A BARE message has a comparable size and entropy to the underlying state it represents.

The BARE schema language also provides a means of describing the format of BARE messages without implementation-specific details. This encourages applications that utilize BARE to describe their state in a manner that other programmers can easily utilize for application interoperation. The conservative set of primitives offered by BARE aids in making such new implementations easy to write.

2. Specification of the BARE Message Encoding

A BARE message is a single value of a pre-defined type, which may be of an aggregate type enclosing multiple values. Unless otherwise specified, there is no additional container or structure around the value; it is encoded plainly.

A BARE message does not necessarily have a fixed length, but the schema author may make a deliberate choice to constrain themselves to types of well-defined lengths if this is desired.

The names for each type are provided to establish a vocabulary for describing a BARE message schema out-of-band, by parties who plan to exchange BARE messages. The type names used here are provided for this informative purpose, but are more rigourously specified by the schema language specification in Section 3.

2.1. Primitive Types

Primitive types represent exactly one value.

uint
: A variable-length unsigned integer encoded using the Unsigned Little Endian Base 128 (ULEB128). Every octet of the encoded value has the most-significant bit set, except for the last octet. The remaining bits are the zero-extended integer value in 7-bit groups, the least-significant group first.

  The encoder MUST encode uint using the minimum necessary number of octets, and the decoder SHOULD raise an error if it encounters the opposite.

  The maximum precision of such a number is 64-bits. The maximum length of an encoded uint is therefore 10 octets.

  Numbers that require all ten octets will have 6 bits in the final octet that do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

int
: A signed integer with variable-length encoding. Signed integers are represented as uint using a "zig-zag" encoding: positive values x are written as 2x + 0, negative values as -2x - 1. Another way of looking at it is that negative numbers are complemented, and whether to complement is encoded in bit 0.

  The encoder MUST encode int using the minimum necessary number of octets, and the decoder SHOULD raise an error if it encounters the opposite.

The maximum precision of such a number is 64-bits. The maximum length of an encoded int is therefore 10 octets.

Numbers that require all ten octets will have 6 bits in the final octet that do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

u8, u16, u32, u64
: Unsigned integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first).

i8, i16, i32, i64
: Signed integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first), with two's complement notation.

f32, f64
: Floating-point numbers represented with the IEEE 754 [IEEE.754.1985] binary32 and binary64 floating point number formats.

bool
: A boolean value, either true or false, encoded as a u8 type with a value of one or zero, respectively representing true or false.

  If a value other than one or zero is found in the u8 representation of the bool, the message is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

str
: A string of text. The length of the text in octets is encoded first as a uint, followed by the text data represented with the UTF-8 encoding [RFC3629].

If the data is found to contain invalid UTF-8 sequences, it is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

data
Arbitrary data of a variable length. The length (in octets) is encoded first as a uint, followed by the data itself encoded literally.

data[length]
Arbitrary data of a fixed "length", e.g. data[16]. The length (in octets) is not encoded into the message. The data is encoded literally in the message.

void
A type with zero length. It is not encoded into BARE messages.

enum
An unsigned integer value from a set of named values agreed upon in advance, encoded with the uint type.

An enum whose uint value is not a member of the values agreed upon in advance is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

Note that this makes the enum type unsuitable for representing several enum values that have been combined with a bitwise OR operation.

Using uint for enum value makes it possible to encode named values with different number of octets. Constant-length enum can be achieved when all the enum values are encoded by uints with the same number of octets.

2.2. Aggregate Types

Aggregate types may store zero or more primitive or aggregate values.

optional<type>
A value of "type" that may or may not be present, e.g. optional<u32>. Represented as either a u8 with a value of zero, indicating that the optional value is unset; or a u8 with a value of one, followed by the encoded data of the optional type.

An optional value whose initial u8 is set to a number other than zero or one is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

list<type>
A variable-length list of "type" values, e.g. list<str>. The length of the list (number of values) is encoded as a uint, followed by the encoded values of each member of the list concatenated.

list<type>[length]
A list of "length" values of "type", e.g. list<uint>[10]. The length is not encoded into the message. The encoded values of each member of the list are concatenated to form the encoded list.

map<type A><type B>
A mapping of "type B" values keyed by "type A" values, e.g. map<u32><str>. The encoded representation of a map begins with the number of key/value pairs encoded as a uint, followed by the encoded key/value pairs concatenated. Each key/value pair is encoded as the encoded key concatenated with the encoded value.

A message with repeated keys is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

union
A tagged union whose value may be one of any type from a set of types agreed upon in advance. Every type in the set is assigned a numeric identifier. The value is encoded as the selected type's identifier represented with the uint encoding, followed by the encoded value of that type.

A union with a tag value that does not have a corresponding type assigned is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

struct
A set of values of arbitrary types concatenated in an order agreed upon in advance. Each value is called "field", and the field has a name and type.

2.3. User-Defined Types

A user-defined type gives a name to another type. This creates a distinct type whose representation is equivalent to the named type. An arbitrary number of user-defined types may be used for the same underlying type; each is distinct from the other.

2.4. Invariants

The following invariants are specified:

* Any type that is ultimately a void type (either directly or via a user-defined type) MUST NOT be used as an optional type, list value, map key, map value, or struct field type. Void types may only be used as members of the set of types in a tagged union.

* Enums MUST have at least one named value, and each named value of an enum MUST be unique.

* The lengths of fixed-length data and fixed-length list types MUST be at least one and MUST NOT be longer than 18,446,744,073,709,551,615 octets (the maximum value of a u64).

* Any map key type (directly or via a user-defined type) MUST be of a primitive type that is not f32, f64, data, data[length], or void.

* Unions MUST have at least one type, and each type of a union MUST be unique.

* Structs MUST have at least one field, and each field of a struct MUST have a unique name.

* Any user-defined type MUST be defined before used. Any user-defined type MUST NOT be defined recursively (directly or indirectly).

3. BARE Schema Language Specification

The use of the schema language is optional. Implementations SHOULD support decoding arbitrary BARE messages without a schema document, by defining the schema in a manner that utilizes more native tools available from the programming environment.

However, it may be useful to have a schema document for use with code generation, documentation, or interoperability. A domain-specific language is provided for this purpose.

3.1. Lexical Analysis

During lexical analysis, "#" is used for comments; if encountered, the "#" character and any subsequent characters are discarded until a line feed (%x0A) is found.

3.2. ABNF Grammar

The syntax of the schema language is provided here in Augmented Backus-Naur Form [RFC5234]. However, this grammar differs from [RFC5234] in that literal text strings are case-sensitive (e.g. "type" does not match "TypE").

```
schema          =  [WS] user-types [WS]

user-types      =  user-type [WS user-types]
user-type       =  "type" WS user-type-name WS any-type
user-type-name  =  UPPER *(ALPHA / DIGIT) ; first letter is uppercase

any-type        =  "uint" / "u8"  / "u16" / "u32" / "u64"
any-type        =/ "int" / "i8"  / "i16" / "i32" / "i64"
any-type        =/ "f32" / "f64"
any-type        =/ "bool"
any-type        =/ "str"
any-type        =/ "data" [length]
any-type        =/ "void"
any-type        =/ "enum" [WS] "{" [WS] enum-values [WS] "}"
any-type        =/ "optional" type
any-type        =/ "list" type [length]
any-type        =/ "map" type type
any-type        =/ "union" [WS] "{" [[WS] "|"] [WS] union-members [WS] [
any-type        =/ "struct" [WS] "{" [WS] struct-fields [WS] "}"

length          =  [WS] "[" [WS] integer [WS] "]"
integer         =  1*DIGIT

enum-values     =  enum-value [WS enum-values]
enum-value      =  enum-value-name [[WS] "=" [WS] integer]
enum-value-name =  UPPER *(UPPER / DIGIT / "_")

type            =  [WS] "<" [WS] any-type [WS] ">"

union-members   =  union-member [[WS] "|" [WS] union-members]
union-member    =  any-type [[WS] "=" [WS] integer]

struct-fields   =  struct-field [WS struct-fields]
struct-field    =  1*ALPHA [WS] ":" [WS] any-type

UPPER           =  %x41-5A ; uppercase ASCII letters, i.e. A-Z
ALPHA           =  %x41-5A / %x61-7A ; A-Z / a-z
DIGIT           =  %x30-39 ; 0-9

WS              =  1*(%x0A / %x09 / " ") ; whitespace
```

See Appendix B.1 for an example schema written in this language.

3.3. Semantic Elements

The names of fields and user-defined types are informational: they are not represented in BARE messages. They may be used by code generation tools to inform the generation of field and type names in the native programming environment.

Enum values are also informational. Values without an integer token are assigned automatically in the order that they appear, starting from zero and incrementing by one for each subsequent unassigned value. If a value is explicitly specified, automatic assignment continues from that value plus one for subsequent enum values.

Union type members are assigned a tag in the order that they appear, starting from zero and incrementing by one for each subsequent type. If a tag value is explicitly specified, automatic assignment continues from that value plus one for subsequent values.

4. Application Considerations

Message authors who wish to design a schema that is backwards- and forwards-compatible with future messages are encouraged to use union types for this purpose. New types may be appended to the members of a union type while retaining backwards compatibility with older message types. The choice to do this must be made from the first message version -- moving a struct into a union *does not* produce a backwards-compatible message.

The following schema provides an example:

```
type Message union {MessageV1 | MessageV2 | MessageV3}

type MessageV1 ...

type MessageV2 ...

type MessageV3 ...
```

An updated schema that adds a MessageV4 type would still be able to decode versions 1, 2, and 3.

If a message version is later deprecated, it may be removed in a manner compatible with future versions 2 and 3 if the initial tag is specified explicitly.

```
type Message union {MessageV2 = 1 | MessageV3}
```

5. Future Considerations

To ensure message compatibility between implementations and backwards- and forwards-compatibility of messages, constraints on vendor extensions are required. This specification is final, and new types or extensions will not be added in the future. Implementors MUST NOT define extensions to this specification.

To support the encoding of novel data structures, the implementor SHOULD make use of user-defined types in combination with the data or data[length] types.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

Message decoders are common vectors for security vulnerabilities. BARE addresses this by making the message format as simple as possible. However, the decoder MUST be prepared to handle a number of error cases when decoding untrusted messages, such as a union type with an invalid tag, or an enum with an invalid value. Such errors may also arise by mistake, for example when attempting to decode a message with the wrong schema.

Support for data types of an arbitrary, message-defined length (lists, maps, strings, etc) is commonly exploited to cause the implementation to exhaust its resources while decoding a message. However, legitimate use-cases for extremely large data types (possibly larger than the system has the resources to store all at once) do exist. The decoder MUST manage its resources accordingly, and SHOULD provide the application a means of providing their own decoder implementation for values that are expected to be large.

There is only one valid interpretation of a BARE message for a given schema, and different decoders and encoders should be expected to provide that interpretation. If an implementation has limitations imposed from the programming environment (such as limits on numeric precision), the implementor MUST document these limitations, and prevent conflicting interpretations from causing undesired behavior.

8. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[IEEE.754.1985] Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE Standard 754, August 1985.

Appendix A. Example Values

This section lists example values in decimal, as string, or as named value (left or top), and their encoded representation in hexadecimal (right or bottom).

uint

0	00
1	01
126	7e
127	7f
128	80 01
129	81 01
255	FF 01

int

0	00
1	02
-1	01
63	7e
-63	7d
64	80 01
-64	7f
65	82 01
-65	81 01
255	FE 03
-255	FD 03

u32

0	00 00 00 00
1	01 00 00 00
255	FF 00 00 00

i16

0	00 00
1	01 00
-1	FF FF
255	FF 00
-255	01 FF

f64

0.0	00 00 00 00 00 00 00 00
1.0	00 00 00 00 00 00 f0 3f
2.55	66 66 66 66 66 66 04 40
-25.5	00 00 00 00 00 80 39 C0

bool

true	01
false	00

str

"BARE"	04 42 41 52 45

data

Example value is in hexadecimal.

```
   aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb

   10 aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc
   bb
```

data[16]

```
   Example value is in hexadecimal.

   aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb

   aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb
```

void

```
   Not encoded.
```

enum {FOO BAR = 255 BUZZ}

```
   FOO                 00
   BAR                 FF 01
   BUZZ                80 02
```

optional<u32>

```
   (unset)             00
   0                   01 00 00 00 00
   1                   01 01 00 00 00
   255                 01 FF 00 00 00
```

list<str>

```
   "foo" "bar" "buzz"

   03 03 66 6f 6f 03 62 61 72 04 62 75 7A 7A
```

list<uint>[10]

```
   0 1 254 255 256 257 126 127 128 129

   00 01 FE 01 FF 01 80 02 81 02 7E 7F 80 01 81 01
```

map<u32><str>

```
   0 => "zero"
   1 => "one"
   255 => "two hundreds and fifty five"
```

```
03 00 00 00 00 04 7A 65 72 6F 01 00 00 00 03 6F
6E 65 FF 00 00 00 1B 74 77 6F 20 68 75 6E 64 72
65 64 73 20 61 6E 64 20 66 69 66 74 79 20 66 69
76 65
```

union {int | uint = 255 | str}

```
0                   00 00
1                   00 02
1                   FF 01 01
-1                  00 01
255                 00 FE 03
255                 FF 01 FF 01
-255                00 FD 03
"BARE"              80 02 04 42 41 52 45
```

struct {foo: uint bar: int buzz: str}

```
foo => 255
bar => -255
buzz => "BARE"

FF 01 FD 03 04 42 41 52 45
```

Appendix B. Example Company

An example company that uses BARE to encode data about customers and employees.

B.1. Message Schema

The following is an example of a schema written in the BARE schema language.

```
type PublicKey data[128]
type Time str # ISO 8601

type Department enum {
  ACCOUNTING
  ADMINISTRATION
  CUSTOMER_SERVICE
  DEVELOPMENT

  # Reserved for the CEO
  JSMITH = 99
}

type Address list<str>[4] # street, city, state, country

type Customer struct {
  name: str
  email: str
  address: Address
  orders: list<struct {
    orderId: i64
    quantity: i32
  }>
  metadata: map<str><data>
}

type Employee struct {
  name: str
  email: str
  address: Address
  department: Department
  hireDate: Time
  publicKey: optional<PublicKey>
  metadata: map<str><data>
}

type TerminatedEmployee void

type Person union {Customer | Employee | TerminatedEmployee}
```

B.2. Encoded Messages

Some basic example messages in hexadecimal are provided for the schema specified in Appendix B.1.

A "Person" value of type "Customer" with the following values:

name James Smith

email jsmith@example.org

address
123 Main St; Philadelphia; PA; United States

orders (1) orderId: 4242424242; quantity: 5

metadata (unset)

Encoded BARE message:

```
00 0b 4a 61 6d 65 73 20 53 6d 69 74 68 12 6a 73
6d 69 74 68 40 65 78 61 6d 70 6c 65 2e 6f 72 67
0b 31 32 33 20 4d 61 69 6e 20 53 74 0c 50 68 69
6c 61 64 65 6c 70 68 69 61 02 50 41 0d 55 6e 69
74 65 64 20 53 74 61 74 65 73 01 b2 41 de fc 00
00 00 00 05 00 00 00 00
```

Encoded BARE message, but characters of strings are decoded:

```
00 0b  J  a  m  e  s     S  m  i  t  h 12  j  s
 m  i  t  h  @  e  x  a  m  p  l  e  .  o  r  g
0b  1  2  3     M  a  i  n     S  t 0c  P  h  i
 l  a  d  e  l  p  h  i  a 02  P  A 0d  U  n  i
 t  e  d     S  t  a  t  e  s 01 b2 41 de fc 00
00 00 00 05 00 00 00 00
```

A "Person" value of type "Employee" with the following values:

name Tiffany Doe

email tiffanyd@acme.corp

address 123 Main St; Philadelphia; PA; United States

department ADMINISTRATION

hireDate 2020-06-21T21:18:05Z

publicKey (unset)

metadata (unset)

Encoded BARE message:

```
01 0b 54 69 66 66 61 6e 79 20 44 6f 65 12 74 69
66 66 61 6e 79 64 40 61 63 6d 65 2e 63 6f 72 70
0b 31 32 33 20 4d 61 69 6e 20 53 74 0c 50 68 69
6c 61 64 65 6c 70 68 69 61 02 50 41 0d 55 6e 69
74 65 64 20 53 74 61 74 65 73 01 14 32 30 32 30
2d 30 36 2d 32 31 54 32 31 3a 31 38 3a 30 35 5a
00 00
```

Encoded BARE message, but characters of strings are decoded:

```
01 0b  T  i  f  f  a  n  y     D  o  e 12  t  i
 f  f  a  n  y  d  @  a  c  m  e  .  c  o  r  p
0b  1  2  3     M  a  i  n     S  t 0c  P  h  i
 l  a  d  e  l  p  h  i  a 02  P  A 0d  U  n  i
 t  e  d     S  t  a  t  e  s 01 14  2  0  2  0
 -  0  6  -  2  1  T  2  1  :  1  8  :  0  5  Z
00 00
```

A "Person" value of type "TerminatedEmployee".

Encoded BARE message:

```
02
```

Appendix C. Complex Data

BARE schema examples for complex data structures.

C.1. Simple Hierarchical Data

Recursive data types are forbidden in BARE. The following examples show how linked list and binary tree, widely used recursive data types, can be encoded in BARE messages.

As BARE supports variable-length lists, encoding of linked list is straightforward.

```
type Element struct {
  str: what
}

type LinkedList list<Element>
```

A binary tree can be encoded to BARE's variable-length list with 2x + 1 and 2x + 2 indexing.

```
type Node optional<struct {
  str: what
}>

type BinaryTree list<Node>
```

C.2. JSON Schema

Sometimes it is needed to deal with generic format of data. When the use-case for recursive types is encountered, each element to encode needs to be identified.

```
type ElementId uint

type False void
type True void
type Null void

type Object map<str><ElementId>
type Array list<ElementId>

type Element union {
  | False
  | True
  | Null
  | f64
  | str
  | Object
  | Array
}

type JSONDocument list<Element>
```

C.3. Graph

It is not possible to encode pointers in BARE. However, an arbitrary graph can be encoded in the lists of nodes and connections.

```
type NodeId uint

type Node struct {
  what: str
}

type Connection struct {
  from: NodeId
  to: NodeId
  why: str
}

type Graph struct {
  nodes: map<NodeId><Node>
  edges: list<Connection>
}
```

Appendix D. Design Decisions

This section documents the reasoning behind the decisions made during BARE specification process.

f32 and f64 are fully compliant with IEEE 754 [IEEE.754.1985]

The use-case is a sensor sending NaN values or encoding of infinity in scientific applications.

The consequences are that encoded values of f32 and f64 types are not canonical, and therefore forbidden as map keys.

Types of a union needs to be unique

However, user-defined types are distinct types, so it is not a problem overall.

Recursive types are forbidden

Recursive types bring the possibility to encode arbitrary tree data structures for the price of:

1. runtime errors for cyclic references,
2. possible stack overflows during encoding/decoding when recursive encoders/decoders are used,
3. confusion because they do not come with pointers, although data to encode usually uses pointers.

It is not worth it.

The consequence is that recursive types need to be mapped to non-recursive types when used.

Namespaces or imports are not used

It would increase complexity. BARE schema language is simple.

There is no bitmap type

Use data[length] instead.

There is no date/time type

Use u64 for timestamp or str.

There is no ordered map type

Ordered maps are not widely supported in programming environments. Users that want to use ordered maps can use a list of pairs:

```
list<struct {
  key: KeyType
  val: ValType
}>
```

Author's Address

Drew DeVault
SourceHut

454 E. Girard Ave #2R
Philadelphia, PA 19125
United States of America

Phone: +1 719 213 5473
Email: sir@cmpwn.com
URI: https://sourcehut.org/