

Packedobjects Reference Manual

Table of Contents

1	Introduction	1
1.1	Background	1
1.2	What is Packedobjects?	1
1.3	Features	1
1.4	Limitations	2
1.5	Installing Packedobjects	2
1.6	The manual	2
1.7	Further reading	2
1.8	Quick start	2
2	Domain Specific Language	4
2.1	Integers	4
2.2	Strings	4
2.3	Sequences	4
2.4	Choice	5
2.5	Enumeration	5
2.6	Boolean and null	5
2.7	Real	5
2.8	Protocol grammar	5
2.9	Data grammar	6
3	Example protocols	7
3.1	A phonebook	7
3.2	An abstract example: Shopping for food and drinks	8
3.3	A practical example: UDP time server	9
4	Integer Encoding Rules	11
4.1	integer	11
4.1.1	length encoding	11
4.1.2	unconstrained integers	11
4.1.3	semi-constrained integers	12
4.1.4	constrained integers	12
4.2	string	13
4.2.1	semi-constrained string	13
4.2.2	constrained string	13
4.2.3	fixed length string	14
4.3	bit-string	14
4.4	octet-string	15
4.5	hex-string	15
4.6	numeric-string	16
4.7	enumerated	16
4.8	boolean	16

4.9	null	17
4.10	sequence	17
4.11	sequence-optional	17
4.12	sequence-of	18
4.13	choice	18
5	Extensibility	20
6	Working from C	21
6.1	Simple example	21
6.2	An extensible example	23
7	Embedded Linux	24
7.1	OpenWrt	24
8	Future work	26
8.1	Performance	26
8.2	JSON version	26
	Index	27

1 Introduction

1.1 Background

The International Standards Organisation (OSI) 7 Layer reference model provided an academic framework for the design of network protocols and standards. In comparison to the OSI model, the TCP/IP model adopted a more simplified approach where amongst other changes the Presentation Layer was consumed by the Application Layer. As such, a network applications programmer needs to consider how to structure their data when transferring it across an internet. A number of competing technologies have been developed and continue to develop in this area. When it comes to structuring data in a human-readable way, XML has dominated. Approaches to structuring binary data range from serialising native data structures to transforming an abstract syntax into a more concise binary form. The later approach often requires code generation. In many cases text-based protocols suffice. However, binary protocols continue to play an important role in supporting network applications, including use in networks outside of the Internet.

1.2 What is Packedobjects?

Packedobjects is a data encoding tool (or serialization tool) which utilises the “data is code” principle of symbolic expressions available in Lisp-like languages to allow the scripting of tightly-packed binary network protocols. This dynamic approach provides specific flexibility when working on embedded systems as it reduces the amount of cross compilation and deploy cycles that occur following more traditional methods which require code generation. In addition, the separation of how the data is encoded from the compiled application facilitates a concept known as extensibility of the network protocol without requiring special handling. In short, Packedobjects provides high-level bit-packing on low-level devices.

Packedobjects is available as a module for GNU Guile version 2:

<http://www.gnu.org/software/guile/>

The ability to manipulate data with the Scheme language is a powerful feature of the tool. You have the choice of working completely from Scheme or you can embed Scheme into your existing C application.

1.3 Features

- Very efficient encoding size
- Expressive language for describing contents of a message
- Fully dynamic including ability to alter protocol at runtime
- Simple API with two function calls
- Easy integration to C using Guile as an extension language
- Supported on embedded Linux

1.4 Limitations

Packedobjects is not a general purpose tool. It uses bit manipulation to tightly pack data in a way that has no respect for byte alignment. This type of encoding is based on the unaligned variant of Packed Encoding Rules. Encoding this concisely comes at a price. In a worst case scenario you may have a large chunk of *octet-string* data which starts at an offset of 1 bit. This 1 bit would throw the chunk of 8 bit data out of alignment and require bit manipulation. Therefore, Packedobjects is best suited to encode highly structured data that is not dominated by strings. If your data is dominated by strings, applying standard lossless compression might be better. Although, applying compression to XML comes with its own performance hit. Another alternative would be to use something like Protocol Buffers which tries to offer the best of both worlds: encoding speed and concisely encoded data. Packedobjects encodes more concisely than **Protocol Buffers** and is more dynamic. It does not require code generation to take place to convert a protocol into native language routines. This means it is well suited to embedded Linux applications as it eliminates cross compilation headaches and provides protocol extensibility free of charge. Being completely dynamic also comes with a runtime performance cost. Packedobjects tries to make sure all data is correct at encode time including checking the contents of strings. As a result, encode time is significantly slower than decode time. Runtime performance of the tool is marked for future work.

1.5 Installing Packedobjects

To install from the latest source:

```
git clone git://gitorious.org/packedobjects/packedobjects.git
cd packedobjects/
autoreconf -i
./configure
make
sudo make install
```

1.6 The manual

A PDF version of this manual is available:

<http://zedstar.org/packedobjects/packedobjects.pdf>

1.7 Further reading

Moore, J. "**Everything counts in small amounts**", International Workshop on Dynamic languages for Robotic and Sensor systems (DYROS), November 2010

1.8 Quick start

To load the module:

```
(use-modules (packedobjects packedobjects))
```

Packedobjects has only two functions with the following signatures:

```
encode : protocol data1 -> pdu
decode : protocol pdu -> data2
```

where:

- protocol is a list describing the way data is encoded
- data1 is a list of values to be encoded
- pdu is a string containing encoded binary data
- data2 is the list of values obtained from the decode process which is equal to data1

The canonical Hello World! example:

```
(use-modules (packedobjects packedobjects))
(define protocol '(message string))
(define data1 '(message "Hello World!"))
(define pdu (encode protocol data1))
(define data2 (decode protocol pdu))
```

Within the REPL:

```
guile> data1
(message "Hello World!")
guile> data2
(message "Hello World!")
```

2 Domain Specific Language

The language we represent using an s-expression can be referred to as a Domain Specific Language (DSL) with its purpose to specify structured data for communication across a computer network. The subsections that follow will introduce the DSL by formally specifying the grammar for the protocol description language together with the grammar for the data that needs to be communicated.

The DSL used by Packedobjects consists of two categories of data type: atomic and compound. An atomic data type specifies a single value to be encoded whereas a compound data type consists of one or more atomic and/or compound data types. The compound data types include the various sequence types and the choice type. The following subsections will introduce all the data types available.

2.1 Integers

The integer data type is a core type. All other types are transformed into this type before being mapped onto the encoder. The integer type uses visible subtype constraints to optimise the encoding. Subtyping in this case is used to restrict the range of values allowed for an integer value. The ability to customise data types produces efficient encodings. Not only are less bits sent across the communications link but also more optimised encoder/decoder implementations can be built to handle specific protocols. Constraints are specified using the range syntax.

2.2 Strings

There are various string types that differ according to the type of characters they represent and therefore the amount of bits they need when encoded. For example, a string containing only the characters one and zero requires just 1 bit, whereas a string containing characters which can represent hexadecimal requires 4 bits per character when encoded. The default string type encodes in 7 bits. There is also an 8 bit string type which could be used to contain non-string data. As with integers, the various string types employ subtyping to optimise the encodings. This time the constraints are specified using the size syntax and are used to restrict the length of strings.

2.3 Sequences

The sequence type provides a useful way of logically grouping together named values so that each belongs to a unique name space. Although it has no impact on the data encoded it is an important mechanism for structuring data. It may not be feasible to can encode every value within a sequence. To handle this optionality we must use a variation of the sequence type that informs the encoder to include the required extra information. In addition to encoding a small amount of extra data, the added flexibility of optionality will result in a loss of performance as we need to determine what is present in the sequence at runtime. Another important feature of sequences is they may repeat. From a low-level encoding point of view this is straight forward. All we need to know is how many times the sequence repeats. From a higher-level perspective we must employ special handling of our values to

group together each individual sequence. We do this by grouping each repeating sequence with an extra pair of brackets.

2.4 Choice

Typically a network protocol will consist of a selection of different messages or Protocol Data Units (PDUs). In addition, within a PDU itself decisions may need to be made that selectively encode only parts of the message. The choice data type gives us this flexibility.

2.5 Enumeration

Enumeration is common in many high-level languages. The DSL used by Packedobjects restricts the sequence of values to be of type symbol or integer.

2.6 Boolean and null

Boolean and null types are both simple to use. A boolean type encodes a true or false value and a null type encodes no value. Although a null type encodes no value its significance comes from its context. There are specific circumstances where no extra data needs to be encoded to convey information. An analogous scenario would be the acknowledgement system employed by the Transmission Control Protocol (TCP). It is possible that a TCP acknowledgement is explicitly sent across a network where no actual TCP data is communicated other than the TCP header itself. With our DSL we could have a choice of PDUs and one particular choice represents an acknowledgement. By making this choice we already have all the information we need.

2.7 Real

Packedobjects does not directly support a real data type. The user can supply custom definitions which help optimise the encoding of values they frequently use. However, an example of a general definition might be:

```
(pi sequence
  (mantissa integer)
  (base enumerated (2 10))
  (exponent integer))
```

We could then encode our value of π as follows:

```
(pi
  (mantissa 314159265)
  (base 10)
  (exponent -8))
```

2.8 Protocol grammar

```
<protocol> ::= <form>

<form> ::= <atomic type> | <compound type>

<atomic type> ::= '(' 'integer' <range>? ') '
| '(' <id> 'string' <size>? ') '
| '(' <id> 'bit-string' <size>? ') '
| '(' <id> 'octet-string' <size>? ') '
| '(' <id> 'hex-string' <size>? ') '
```

```

| '(' <id> 'numeric-string' <size>? ')'
| '(' <id> 'enumerated' <enum> ')'
| '(' <id> 'boolean' ')'
| '(' <id> 'null' ')'

<compound type> ::= '(' <id> 'sequence' <form>+ ')'
| '(' <id> 'sequence-optional' <form>+ ')'
| '(' <id> 'sequence-of' <form>+ ')'
| '(' <id> 'choice' <form>+ ')'

<range> ::= '(' 'range' 'min' 'max' ')'
| '(' 'range' 'min' <integer> ')'
| '(' 'range' <integer> 'max' ')'
| '(' 'range' <integer> <integer> ')'

<size> ::= '(' 'size' 'min' 'max' ')'
| '(' 'size' 'min' <integer> ')'
| '(' 'size' <integer> 'max' ')'
| '(' 'size' <integer> <integer> ')'
| '(' 'size' <integer> ')'

<enum> ::= '(' <symbol>+ | <number>+ ')'

<id> ::= <symbol>

```

2.9 Data grammar

```

<data> ::= <form>

<form> ::= <atomic value> | <compound value>

<atomic value> ::= '(' <id> <string> ')'
| '(' <id> <number> ')'
| '(' <id> <symbol> ')'
| '(' <id> <boolean> ')'
| '(' <id> ')'

<compound value> ::= '(' <id> <form>+ ')'
| '(' <id> '[' <form>+ ']'* ')'

<id> ::= <symbol>

```

Constructing a list of values is straight forward, however, special attention is required for *sequence-of*. This requires extra bracketing to delimit the values which belong to each sequence.

3 Example protocols

Code for all examples are available in the repository:

<http://gitorious.org/packedobjects/packedobjects/trees/master/examples/>

3.1 A phonebook

This example will show how the *sequence-optional* type can be used to selectively include values. It also demonstrates how the *sequence-of* type is used for repeating items. First let's include the appropriate modules and define our protocol.

```
(use-modules (packedobjects packedobjects))
(use-modules (ice-9 pretty-print))

(define phonebook
  '(person sequence-optional
    (name string (size 1 100))
    (id integer (range 1 max))
    (email string (size 3 320))
    (phone-number sequence-of
      (number numeric-string (size 8 20))
      (type enumerated (mobile home work))))))
```

We can now define some phonebook entries

```
(define entry1
  '(person
    (name "John Doe")
    (id 1234)
    (email "johnd@example.com")))

(define entry2
  '(person
    (name "Fred Blogs")
    (id 9999)
    (email "fredb@example.com")
    (phone-number
      ((number "42424242")
       (type home))))))

(define entry3
  '(person
    (name "Jane Smith")
    (id 42)
    (email "janes@example.com")
    (phone-number
      ((number "123456780")
       (type work))
      ((number "969696969")
       (type mobile))))))
```

We can now encode and decode these entries from the REPL

```
guile> (define pdu (encode phonebook entry1))
guile> (pretty-print (decode phonebook pdu))
(person
  (name "John Doe")
  (id 1234)
  (email "johnd@example.com"))
```

```

guile> (define pdu (encode phonebook entry2))
guile> (pretty-print (decode phonebook pdu))
(person
  (name "Fred Blogs")
  (id 9999)
  (email "fredb@example.com")
  (phone-number ((number "42424242") (type home))))
guile> (define pdu (encode phonebook entry3))
guile> (pretty-print (decode phonebook pdu))
(person
  (name "Jane Smith")
  (id 42)
  (email "janes@example.com")
  (phone-number
    ((number "123456780") (type work))
    ((number "969696969") (type mobile))))
guile>

```

3.2 An abstract example: Shopping for food and drinks

We will describe a real life activity such as shopping for food and drinks and encode our representation into bytes. As a network protocol this example is contrived, however, it does illustrate how much flexibility we have using our s-expression based DSL. In particular it will highlight powerful inbuilt data manipulation techniques such as quasiquote, unquote and unquote-splicing available as part of the Scheme language.

```

(define booze
  '(sequence-of
    (beer null)
    (nibbles null)))

```

We start by defining a sequence-of type containing null types. The sequence-of type is a compound data type which consists of a repeating sequence of other data types, in this case a sequence of two null types. The null type is one of several atomic data types available. It is an unusual data type in that it requires no value.

```

(define grub
  '(sequence
    (pizza null)
    (salad null)))

```

In addition to our drinks we should have some food. In this case we use the sequence data type which specifies both pizza and salad.

```

(define trolley
  '(trolley sequence-optional
    (drink ,@booze)
    (food ,@grub)))

```

```

(define basket
  '(basket choice
    (food ,@grub)
    (drink ,@booze)))

```

To carry our food and drink we could use a trolley or use a basket. The trolley is large enough to carry both but we must choose between the food or drink if we use a basket. The sequence-optional data type is a flexible type which is similar to the sequence type but each member is optional. Therefore, using the trolley we could decide to only pack some food. Using the basket we must choose between either the food or drink. The choice data

type is a compound data type which enforces this restriction. In both cases, we have used unquote-splicing to reuse our definitions of food and drink and therefore are able to produce concise protocol descriptions. Having defined a protocol we must specify values according to their description.

```
(define thirsty
  '(basket
    (drink
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles))))))

(define hungry
  '(basket
    (food
      (pizza)
      (salad))))

(define thirsty+hungry
  '(trolley
    ,(cadr thirsty)
    ,(cadr hungry)))
```

Depending on our mood, we might be thirsty, hungry or both. In the case of being both thirsty and hungry we will use a trolley. This example illustrates the use of unquote to reuse our definitions of being hungry and thirsty. Note how we apply *cadr* to symbolise removing the basket from the items so we can place them in the trolley instead. From the REPL we can examine our definitions.

```
guile> (use-modules (ice-9 pretty-print))
guile> (pretty-print thirsty+hungry)
(trolley
 (drink ((beer) (nibbles))
        ((beer) (nibbles))
        ((beer) (nibbles))
        ((beer) (nibbles)))
 (food (pizza) (salad)))
```

Lets encode the values by allocating 10 bytes to the encode process.

```
guile> (define pdu (encode trolley thirsty+hungry #:size 10))
guile> (string-length pdu)
2
```

We can see that only 2 bytes was required so lets decode those 2 bytes.

```
guile> (pretty-print (decode trolley pdu #:size 2))
(trolley
 (drink ((beer) (nibbles))
        ((beer) (nibbles))
        ((beer) (nibbles))
        ((beer) (nibbles)))
 (food (pizza) (salad)))
```

We end up with the same values we originally encoded.

3.3 A practical example: UDP time server

GNU Guile has built-in support for networking including support for UDP communication. In this example we build a time server which binary encodes its response using Packedob-

jects. After the client sends a request to the server it displays the current date from the encoded data.

We will take a look at the simple client/server protocol used:

```
(define protocol
  '(pdu choice
    (time-request null)
    (time-response sequence
      (seconds integer (range 0 59))
      (minutes integer (range 0 59))
      (hours integer (range 0 23))
      (day-of-the-month integer (range 1 31))
      (month integer (range 0 11))
      (year integer (range 109 119))
      (day-of-the-week integer (range 0 6))
      (day-of-the-year integer (range 0 365))
      (day-light-saving enumerated (yes no unknown))
      (time-zone-offset integer (range -46800 43200))
      (time-zone string (size 1 4))))))
```

We can see the protocol uses the *choice* data type to indicate we will be encoding differently depending if we are the client or the server. For example, the client will send a very simple request to the server which will respond with a structured message containing date components. More specifically the *time-request* message is defined as the *null* data type meaning it does not require any additional value to be encoded. The *time-response* message uses the compound data type *sequence* to contain a series of *integer* types and a *string*. These atomic types have constraints such as (*range 0 59*) to restrict the range of valid values. There is also an *enumerated* type which allows a single item to be selected from the list of items. Running the programs:

```
./client.scm
Wed Aug 4 15:51:02 2010
```

The binary data received from the client is converted to a string then displayed.

```
./server.scm
Listening for clients in pid: 26964
rx: 1 bytes from client 127.0.0.1:55362
tx: 11 bytes to client 127.0.0.1:55362
```

The server indicates receiving a single byte from the client and then responds back with 11 bytes. To transfer the date as a string would have consumed 24 bytes. Moreover, receiving numeric data as strings may require conversion to integer form to carry out calculations.

4 Integer Encoding Rules

The process of encoding involves combining a protocol and some data and transforming this into an integer form. The integer form is then transformed into a core form before being supplied to the low-level encoder. In the following sections we will describe how each data type is first mapped to an integer form then mapped to its corresponding core form. We call this transformation Integer Encoding Rules. The integer form can be summarised as follows

```
(integer (range x y) n)
```

where x and y restrict the range of values n may take. We can then determine whether we need to encode signed or unsigned values and how many bits are required to encode this range. The resulting core form can be expressed as

```
(signed (bits z) n)
(unsigned (bits z) n)
```

where we show a choice between the signed or unsigned representation and z which represents the number of bits required to encode value n .

4.1 integer

Integers may be unconstrained, semi-constrained or constrained. In addition, each kind comes in two variants: signed and unsigned. All unconstrained and semi-constrained integers require a length encoding to represent the number of bytes required to encode the value. In the subsections that follow we will first describe this length encoding and then go on to show by example how the different data types are encoded.

4.1.1 length encoding

A length encoding requires 2 bits. Integers are then encoded within 8, 16 or 32 bit ranges. This choice of 3 categories can be encoded in 2 bits. Currently Packedobjects supports 32 bit architectures however, the 2 bits used to encode the length category can support an additional value for 64 bit platforms.

4.1.2 unconstrained integers

Given the following protocol

```
(foo integer)
```

and corresponding data

```
(foo 1066)
```

we combine the protocol and data together to form

```
;; Normal form
((foobar integer (range min max) 1066))

;; Integer form
((integer (range min max) 1066))

;; Core form
((unsigned (bits 2) 1) (signed (bits 16) 1066))
```

As this example has no range it is unconstrained and will need to be encoded as a signed value. The first item in the list represents the length encoding obtained from applying

the length encoding function. The second item in the list is the value encoded within the number of bits previously determined by length encoding category.

4.1.3 semi-constrained integers

Encoding a semi-constrained integer follows a similar approach to an unconstrained integer except that a lower bound restricts the size of the value we encode. This offset value will always be positive and is therefore encoded as a unsigned variant. For example, the protocol

```
(foo integer (range -1000 max))
```

is combined with the value

```
(foo -1000)
```

to produce

```
;; Normal form
((foo integer (range -1000 max) -1000))

;; Integer form
((integer (range -1000 max) -1000))

;; Core form
((unsigned (bits 2) 0) (unsigned (bits 8) 0))
```

Instead of encoding the value -1000 we first subtract the lower bound. This allows us to encode the result as a positive value.

4.1.4 constrained integers

Encoding a constrained integer bypasses the need to provide a length encoding. We determine the number of bits required to encode the value based on the *range* using the Guile function *integer-length* as follows:

```
(lambda (lb ub)
  (let ((range (- ub lb)))
    (if (zero? range)
        1
        (integer-length range))))
```

For example the protocol

```
(foo integer (range -100 100))
```

is combined with the value

```
(foo 100)
```

to produce

```
;; Normal form
((foo integer (range -100 100) 100))

;; Integer form
((integer (range -100 100) 100))

;; Core form
((unsigned (bits 8) 200))
```

As with semi-constrained integers, the lower-bound means we will always encode positive values.

4.2 string

Depending on size constraints all strings must be semi-constrained, constrained or fixed in length. Only fixed length strings do not require a length encoding.

4.2.1 semi-constrained string

A semi-constrained string has no size constraint on the upper bound. For example, the protocol

```
(foo string)
```

is combined with the value

```
(foo "foobar")
```

to produce

```
;; Normal form
((foo string (size min max) "foobar"))
```

```
;; Integer form
((integer (range 0 max) 6)
 (integer (range 0 127) 102)
 (integer (range 0 127) 111)
 (integer (range 0 127) 111)
 (integer (range 0 127) 98)
 (integer (range 0 127) 97)
 (integer (range 0 127) 114))
```

```
;; Core form
((unsigned (bits 2) 0)
 (unsigned (bits 8) 6)
 (unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))
```

As there is no size constraint a length encoding is required. We transform our string into integer form with the length plus the characters in decimal form. The first item in the list is the length encoding represented as a semi-constrained integer.

4.2.2 constrained string

A constrained string has a more concise length encoding because it uses a constrained integer. For example, the protocol

```
(foo string (size 1 10))
```

is combined with the value

```
(foo "foobar")
```

to produce

```
;; Normal form
((foo string (size 1 10) "foobar"))
```

```
;; Integer form
((integer (range 1 10) 6)
 (integer (range 0 127) 102)
 (integer (range 0 127) 111))
```

```

(integer (range 0 127) 111)
(integer (range 0 127) 98)
(integer (range 0 127) 97)
(integer (range 0 127) 114))

;; Core form
((unsigned (bits 4) 5)
 (unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))

```

4.2.3 fixed length string

A fixed length string is the most concise as it requires no length encoding. For example, the protocol

```
(foo string (size 6))
```

is combined with the value

```
(foo "foobar")
```

to produce

```

;; Normal form
((foo string (size 6) "foobar"))

;; Integer form
((integer (range 0 127) 102)
 (integer (range 0 127) 111)
 (integer (range 0 127) 111)
 (integer (range 0 127) 98)
 (integer (range 0 127) 97)
 (integer (range 0 127) 114))

;; Core form
((unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))

```

The transformation to core form requires no length encoding and only contains values for each character.

4.3 bit-string

A bit-string encoding follows the same techniques as a string encoding, however, each character can be encoded within 1 bit. For example, the protocol

```
(foo bit-string (size 1 10))
```

is combined with the value

```
(foo "101010")
```

to produce

```

;; Normal form
((foo bit-string (size 1 10) "101010"))

```

```

;; Integer form
((integer (range 1 10) 6)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0))

;; Core form
((unsigned (bits 4) 5)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0))

```

Note how we must convert the character version of one and zero to its numeric equivalent. From this form we simply apply the same transformation techniques as previously described.

4.4 octet-string

An octet-string encoding is exactly the same as a string encoding but instead of using 7 bits we use 8 bits. For example, the protocol

```
(foo octet-string (size 6))
```

is combined with the value

```
(foo "foobar")
```

to produce

```

;; Normal form
((foo octet-string (size 6) "foobar"))

;; Integer form
((integer (range 0 255) 102)
 (integer (range 0 255) 111)
 (integer (range 0 255) 111)
 (integer (range 0 255) 98)
 (integer (range 0 255) 97)
 (integer (range 0 255) 114))

;; Core form
((unsigned (bits 8) 102)
 (unsigned (bits 8) 111)
 (unsigned (bits 8) 111)
 (unsigned (bits 8) 98)
 (unsigned (bits 8) 97)
 (unsigned (bits 8) 114))

```

4.5 hex-string

A hex-string encoding follows a very similar method to a bit-string encoding except that we have a larger range of values to encode corresponding to the valid characters of hexadecimal. For example, the protocol

```
(foo hex-string (size 1 10))
```

is combined with the value

```
(foo "AFAFAF")
```

to produce

```
;; Normal form
((foo hex-string (size 1 10) "AFAFAF"))
```

```
;; Integer form
((integer (range 1 10) 6)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15))
```

```
;; Core form
((unsigned (bits 4) 5)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15))
```

4.6 numeric-string

The encoding of a numeric-string follows the same principles previously described and transforms into a similar core form as a hex-string. The only difference being a numeric-string has an integer range restriction specified as (*range 0 9*).

4.7 enumerated

An enumerated encoding is very similar to a choice encoding, however we count the first item from 0. For example, if we have the following protocol

```
(foobar enumerated
 (foo bar baz))
```

and supply

```
(foobar bar)
```

we obtain

```
;; Normal form
((foobar enumerated 1 2))
```

```
;; Integer form
((integer (range 0 2) 1))
```

```
;; Core form
((unsigned (bits 2) 1))
```

4.8 boolean

A boolean encoding concisely maps to 1 bit. For example, if we have the following protocol

```

    (foo boolean)
and supply
    (foo #f)
we obtain
    ;; Normal form
    ((foo boolean #f))

    ;; Integer form
    ((integer (range 0 1) 0))

    ;; Core form
    ((unsigned (bits 1) 0))

```

4.9 null

A null encoding does not require any call to the encoder.

4.10 sequence

A sequence has no impact on the encoding output and therefore does not map to a core form.

4.11 sequence-optional

A sequence-optional encoding requires an additional value to be encoded to represent which items of the sequence have been supplied. For example, if we have the following protocol

```

    (foobar sequence-optional
      (foo boolean)
      (bar boolean)
      (baz boolean))
and supply
    (foobar
      (foo #t)
      (baz #t))
we obtain
    ;; Normal form
    ((foobar sequence-optional 5 7)
     (foo boolean #t)
     (baz boolean #t))

    ;; Integer form
    ((integer (range 0 7) 5)
     (integer (range 0 1) 1)
     (integer (range 0 1) 1))

    ;; Core form
    ((unsigned (bits 3) 5)
     (unsigned (bits 1) 1)
     (unsigned (bits 1) 1))

```

The first item of the list encodes the value 5 as a constrained integer to represent the bitmap *101* which informs us that the second item in the sequence was not supplied.

4.12 sequence-of

A sequence-of encoding requires a value to represent how many times the sequence repeats. This value is encoded as a semi-constrained integer. For example, if we have the following protocol

```
(foobar sequence-of
  (foo boolean)
  (bar boolean))
```

and supply

```
(foobar
  ((foo #t)
   (bar #t))
  ((foo #f)
   (bar #f))
  ((foo #t)
   (bar #f)))
```

we obtain

```
;; Normal form
((foobar sequence-of 3)
 (foo boolean #t)
 (bar boolean #t)
 (foo boolean #f)
 (bar boolean #f)
 (foo boolean #t)
 (bar boolean #f))

;; Integer form
((integer (range 0 max) 3)
 (integer (range 0 1) 1)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0))

;; Core form
((unsigned (bits 2) 0)
 (unsigned (bits 8) 3)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0))
```

Note how the first item informs us that the sequence repeats 3 times and is encoded without an upper bound. The first two items represent a semi-constrained integer encoding of the value 3. The remaining items represent the boolean values repeatedly encoded as part of the sequence.

4.13 choice

A choice encoding requires an index value to be encoded corresponding to the position of the chosen item in the sequence. For example, if we have the following protocol

```
(foobar choice
```

```
(foo boolean)
(bar boolean))
```

and we supply

```
(foobar
 (bar #f))
```

we obtain

```
;; Normal form
((foobar choice 2 2) (bar boolean #f))

;; Integer form
((integer (range 1 2) 2) (integer (range 0 1) 0))

;; Core form
((unsigned (bits 1) 1) (unsigned (bits 1) 0))
```

The first item in the list indicates the second choice was made in the sequence. The choice index is encoded as a constrained integer.

5 Extensibility

The concept of extensibility can be confusing, especially to those who have worked exclusively with text-based network protocols that highly structure the data they communicate. With this extra structure comes flexibility. It allows an application to receive a message and silently ignore parts of the message it does not understand or recognise. XML is a good example of a technology which facilitates this approach. The structure or tags placed around the data within the message provide all the information required to understand the real payload. This approach is in direct conflict for a protocol designer who strives to minimise every bit of information communicated. Although binary protocols can still follow a similar tag based approach it is common to try and further optimise the solution so only the minimal amount of data required to be decoded successfully is actually communicated. The challenge is producing binary protocols which are not fragile or easily broken by simple changes in the protocol definition. This future proof design approach is known as extensibility. Thus, extensibility refers to the way that network communication can continue between parties A and B even though party B may have updated the way it communicates. The following will illustrate a simple example. Party A uses the following protocol:

```
(define protocol-version-1
  '(foo choice
    (message-A boolean)
    (message-B boolean)))
```

Party B updates its protocol to the following:

```
(define protocol-version-2
  '(foo choice
    (message-A boolean)
    (message-B boolean)
    (message-C boolean)))
```

At this point parties A and B may produce incompatible encodings. For example, it is not possible for party B to communicate message-C with party A because party A has no knowledge of such a message. Party A would be expecting an encoding based on a choice between 2 messages. Encoding standards such as Packed Encoding Rules (PER) handle this situation using special notation in the protocol syntax to indicate the likelihood that specific parts of the specification will be extended and then encode some extra structure to facilitate this. Packedobjects does not require this. What is required is that both parties obtain the same version-2 protocol. In this case, even though party A will never use message-C, it can still receive the message and can choose to silently ignore it. The party A program does not need to be recompiled because the protocol is available via a Scheme script which can be dynamically loaded or bootstrapped over a simple HTTP request. This provides a more stable upgrade option for deployed devices allowing them to continue working with restricted functionality until a software upgrade can be authorised by the user. The ability to maintain communication across mass-deployed devices can be a key goal in the domain of embedded communication technologies.

6 Working from C

Code for all examples are available in the repository:

<http://gitorious.org/packedobjects/packedobjects/trees/master/examples/>

6.1 Simple example

We will describe the function calls used in the main function.

```
#include <assert.h>
#include <libguile.h>

int main()
{
    SCM batting_average, player, player_batting_average;
    SCM scm_encoded_data, scm_decoded_data;
    char *c_encoded_data;
    size_t bytes;

    // setup guile
    scm_init_guile();

    // load scheme code
    scm_c_primitive_load("bbcard.scm");

    // make batting average
    batting_average = make_batting_average(250, 10, -3);

    // make player
    player = make_player("Casey", "Mudville Nine", 32, "left field", "ambidextrous");

    // combine player and batting average
    player_batting_average = make_player_batting_average(player, batting_average);

    // make scheme string of encoded data
    scm_encoded_data = make_encoded_data(player_batting_average);

    // convert to c string
    c_encoded_data = scm_to_locale_stringn(scm_encoded_data, &bytes);

    // decode the c string
    scm_decoded_data = make_decoded_data(c_encoded_data, bytes);

    // check decoded data matches original
    assert(scm_is_true(scm_equal_p(scm_decoded_data, player_batting_average)));

    // free c string
    free(c_encoded_data);

    return 0;
}
```

We first have to initialize Guile and load our Scheme code:

```
// setup guile
scm_init_guile();

// load scheme code
```

```
scheme_c_primitive_load("bbcard.scm");
```

The Scheme code first loads the Packedobjects module and defines the protocol:

```
(use-modules (packedobjects packedobjects))

(define protocol
  '(bbcard sequence
    (name string (size 1 60))
    (team string (size 1 60))
    (age integer (range 1 100))
    (position string (size 1 60))
    (handedness enumerated (left-handed right-handed ambidextrous))
    (batting-average sequence
      (mantissa integer)
      (base enumerated (2 10))
      (exponent integer))))
```

We use 3 functions to help create the values that we will supply to our encoder.

```
(define (make-batting-average mantissa base exponent)
  '(batting-average
    (mantissa ,mantissa)
    (base ,base)
    (exponent ,exponent)))

(define (make-player name team age position handedness)
  '(bbcard
    (name ,name)
    (team ,team)
    (age ,age)
    (position ,position)
    (handedness ,handedness)))

(define (make-player-batting-average player average)
  (append player (list average)))
```

We call these 3 functions from C as follows:

```
// make batting average
batting_average = make_batting_average(250, 10, -3);

// make player
player = make_player("Casey", "Mudville Nine", 32, "left field", "ambidextrous");

// combine player and batting average
player_batting_average = make_player_batting_average(player, batting_average);
```

Calling the encode routine returns a Scheme string of encoded data.

```
// make scheme string of encoded data
scm_encoded_data = make_encoded_data(player_batting_average);
```

We will use this data to supply back to the decoder but first we need to turn it into a C string.

```
// convert to c string
c_encoded_data = scm_to_locale_stringn(scm_encoded_data, &bytes);
```

Now we call the decode routine with the string

```
// decode the c string
scm_decoded_data = make_decoded_data(c_encoded_data, bytes);
```

If everything worked the decoded data should equal the original values

```
// check decoded data matches original
assert(scm_is_true(scm_equal_p(scm_decoded_data, player_batting_average)));
```

6.2 An extensible example

We already introduced the concept of extensibility but we will now try and illustrate an example. Although Packedobjects is intended for communicating data across a network it could also be used as a form of persistent storage. As with network communication, the data is encoded in a portable and efficient way. In this example we have two versions of a C program which store phone records. The first version works as follows:

```
Usage:
  phonedb1 [OPTION...] - phone db v1

Help Options:
  -h, --help      Show help options

Application Options:
  --dump          dump output
  --id            id
  --name          name
  --number        number
  --type          type
```

To add a record to the database we can run the program

```
./phonedb1 --id 1 --name john --number 99999999 --type home
```

and examine what is stored in the database with

```
./phonedb1 --dump
```

which in this case produces

```
(phonedb
 ((person
  (id 1)
  (name "john")
  (phone-number (number "99999999") (type home))))))
```

We now decide to make a new version of the phonedb program which also stores email addresses. This involves changing the protocol (schema is a better name in this case) to add the additional field. We will add a new record using the new version as follows

```
./phonedb2 --id 2 --name joe --email joe@bloggs.com --number 11111111 --type work
```

Now because the schema is stored outside of the C program it is possible to read it without modifying and recompiling the old version of phonedb. Thus we can still read the contents of the database with the old program as follows:

```
./phonedb1 --dump
```

which displays

```
(phonedb
 ((person
  (id 1)
  (name "john")
  (phone-number (number "99999999") (type home))))
 ((person
  (id 2)
  (name "joe")
  (email "joe@bloggs.com")
  (phone-number (number "11111111") (type work))))))
```

Thus, two different versions of the program can work on the database.

7 Embedded Linux

Packedobjects depends on GNU Guile which can be easily built using toolchains from embedded Linux distributions such as OpenEmbedded and OpenWrt.

So far the software has been run on the following embedded devices:

- Sharp Zaurus C1000
- Openmoko Neo 1973
- Openmoko Freerunner
- Beagleboard Rev B
- Ben NanoNote

7.1 OpenWrt

The following is an example Makefile which will package Packedobjects as an ipk on OpenWrt based devices.

```
# author: jmoore@zedstar.org
# modified: 16/08/2010

include $(TOPDIR)/rules.mk

PKG_NAME:=packedobjects
PKG_VERSION:=0.4.1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.gz
PKG_SOURCE_URL:=http://zedstar.org/tarballs/

PKG_BUILD_DIR:=$(BUILD_DIR)/$(PKG_NAME)-$(PKG_VERSION)

PKG_INSTALL:=1

include $(INCLUDE_DIR)/package.mk

define Package/packedobjects
    SECTION:=libs
    CATEGORY:=Libraries
    DEPENDS:=+guile
    TITLE:=packedobjects
endef

define Package/packedobjects/install
$(INSTALL_DIR) $(1)/usr/lib
$(INSTALL_DIR) $(1)/usr/share/guile/site

$(CP) \
$(PKG_INSTALL_DIR)/usr/lib/libpackedobjects*.so* \
$(1)/usr/lib/
$(CP) \
$(PKG_INSTALL_DIR)/usr/share/guile/site/* \
$(1)/usr/share/guile/site/
endef
```

```
$(eval $(call BuildPackage,packedobjects))
```

A binary package built for the Ben NanoNote is available for testing:

http://zedstar.org/ipk/packedobjects_0.4.1_xburst.ipk

8 Future work

8.1 Performance

Encoding performance can be improved. Currently it goes through various stages which can be seen by supplying a debug flag to the encode function such as:

```
scheme@(guile-user)> (encode '(foo boolean) '(foo #t) #:debug 7)
;; Expanded data
(foo #t)

;; Keys
((foo))

;; Combined data
(((foo boolean) (foo #t)))

;; Normal form
((foo boolean #t))

;; Integer form
((integer (range 0 1) 1))

;; Core form
((unsigned (bits 1) 1))

$1 = "\x80"
```

Note, different values to debug can alter how much information is output with 7 being the most verbose.

In general there will be areas of code which might benefit from tail optimisation.

8.2 JSON version

It should be possible to implement Integer Encoding Rules with JSON as the protocol/schema language. This would allow ports to other languages such as Python. The [Apache Avro](#) project is doing something along these lines.

Index

A

ASN.1 3

D

decode 3

E

encode 3

I

Installing 3

O

OpenWrt 24

P

Protocol Buffers 3