Erl8583 User Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**

1.1 What erl8583 is . . . 1

1.2 What erl8583 is not . . . 1

1.3 Overview . . . 1

2 ISO 8583 **1**

2.1 Message Type Identifier (MTI) . . . 1

2.2 Bitmap . . . 2

2.3 Data Elements . . . 2

2.4 Encoding ISO 8583 Messages . . . 3

3 Erl8583 **4**

3.1 Installing erl8583 . . . 4

3.2 The **erl8583_message** module . . . 4

3.3 Encoding rules modules . . . 5

3.4 Marshalling modules . . . 6

3.5 The **erl8583_convert** module . . . 7

4 Example Code **7**

4.1 The **erl8583_message** module . . . 7

4.2 Marshalling a message with the **erl8583_marshaller_xml** module . . . 8

4.3 A minimal ISO 8583 client . . . 9

4.4 Using the generic **erl8583_marshaller** module . . . 11

4.5 An unmarshaller for American Express messages . . . 14

4.6 Unmarshalling a Postilion message . . . 16

4.7 Writing a field encoder for a Postilion message . . . 19

5 Concluding comments **21**

1 Introduction

1.1 What erl8583 is

Erl8583 is an Erlang library that provides abstractions of ISO 8583 messages and functions to pack and unpack messages as byte streams. The library supports a number of packing formats (ASCII, binary, XML and EBCDIC) that are in use. This user guide shows that it can be used to encode messages in formats used by a number of frameworks and transaction processors including jPOS, iso8583py, American Express and Postilion.

Erl8583 was influenced by jPOS, an open-source Java framework for processing ISO 8583 messages. JPOS's API is, unsurprising, quite different from erl8583's though.

1.2 What erl8583 is not

Erl8583 is not a framework for processing ISO 8583 messages. There is probably a case to be made for writing an Erlang/OTP application that can process or switch financial transactions because of OTP's scalability and robustness. Unfortunately, erl8583 is not that application but, with luck, erl8583 may be useful in developing such switches and transaction processors.

1.3 Overview

The following material is covered in this user guide:

- Section 2 gives a brief introduction to ISO 8583 messages. This section can be skipped if you have experience with the ISO 8583 standard.
- Section 3 describes the modules of erl8583 and how the library marshals (packs) and unmarshals (unpacks) messages.
- Section 4 provides several examples of how to use erl8583 for marshalling and unmarshalling messages..

2 ISO 8583

ISO 8583 is a standard that defines message types and content for electronic transactions using payment cards like credit and debit cards. There are three versions of the standard from 1987, from 1993 and from 2003. The 1987 standard is the most widely adopted.

An ISO 8583 message consists of three parts:

- A message type identifier (MTI)
- One or more bitmaps that indicate what data elements are present in the message
- Data elements that contain values relevant to the transaction, e.g. the transaction date and amount

The next three sections cover the parts in a little detail.

2.1 Message Type Identifier (MTI)

The MTI is a 4 (decimai) digit number that encodes:

- The version of the standard. The first digit of the MTI can have the value '0', '1' or '2' indicating that the 1987, 1993 or 2003 version is used.

- The message class. The second digit specifies the purpose of the message, e.g. ’1’ denotes an authorization message, ’2’ indicates a financial message and ’8’ specifies a network administration message. See the ISO specifcation for the full list of message classes.
- The message function. The third digit indicates whether the message is a request (’0’), a response (’1’), an advice (’2’), etc. See the ISO 8583 specification for all message functions.
- The message origin. The fourth digit indicates where the message originated and whether the message is a repeat. For example, ’0’ indicates that the message originated from an acquiring institution while ’1’ denotes that the message is a repeat from an acquiring institution.

2.2 Bitmap

The 1987 specification of ISO 8583 defines 128 data elements that may be present in a message. The 1993 and 2003 versions define 192 fields. Each data element is identified by an integer value in the range 1-128 for the 1987 specification or 1-192 for the later specifications. The sender indicates which data elements are present in a message by one or more bitmaps. A bitmap is a collection of 64 bits; if a bit is set in the bitmap it indicates that a corresponding data element is present.

The primary bitmap is used to indicate the presence or absence of data elements 1-64. The secondary bitmap is used for fields 65-128. If no data elements with identifiers 65 or greater are present, there is no need for the secondary bitmap. The secondary bitmap is actually data element 1, so if bit 1 is set in the primary bitmap it indicates that the message contains the secondary bitmap.

The tertiary bitmap is used for fields 129-192. The tertiary bitmap is data element 65, so if bit 65 in the secondary bitmap is set it indicates that the tertiary bitmap is used.

2.3 Data Elements

An ISO 8583 message contains fields that carry the transaction information. Specific examples of data elements that might exist in a message are (from the 1987 specification):

- Field 1, the secondary bitmap
- Field 2, the primary account number (PAN)
- Field 4, the transaction amount
- Field 35, track 2 data (data read from a card’s magnetic stripe)
- Field 98, the payee

The fields listed above contain quite different data:

- Field 1 is binary data containing a sequence of bits.
- Fields 2 and 4 contain numeric amounts.
- Field 35 is encoded in a 4-bit format.
- Field 98 contains a string with alphabetic characters.

ISO 8583 provides a format for describing acceptable contents for a field and for whether the field has fixed length or variable. The following abbreviations are used to describe the contents of fields:

- *b* for binary data.
- *n* for numeric data.
- *a*, *an* and *ans* for various forms of strings.

- *z* for track 2 data.

To describe whether the field is of fixed or variable length, the following descriptors are used:

- LLVAR, the field is of variable length and the length can be encoded in two decimal digits.
- LLLVAR, the field is of variable length and the length can be encoded in three decimal digits.
- fixed, the field length is constant.

The table below shows the permitted content for some fields from the 1987 specification.

Table 1: Formats for some ISO 8583 data elements

Field	Name	Format	Content	(Maximum) Length
2	Primary Account Number	LLVAR	*n*	19
3	Processing Code	Fixed	*n*	6
35	Track 2 Data	LLVAR	*z*	37
42	Card Acceptor Identification Code	Fixed	*ans*	15
63	Reserved Private	LLLVAR	*ans*	999
64	Message Authentication Code	Fixed	*b*	64

The lengths of a field in Table 1 are expressed in terms of the field content and not in bytes. For example, a field of length 15 containing alphabetic characters might be encoded as 15 bytes if the string contains ASCII characters but a binary field of length 64 bits might be encoded as 8 bytes if all 8 bits are used in each byte.

It may happen that the contents of a fixed length field is less than the prescribed length of the ISO 8583 standard. If the value is shorter than the prescribed length:

- The contents must be padded with trailing spaces for fields that contain strings, e.g. for field 42 in Table 1.
- The contents must be padded with leading zeroes for numeric fields, e.g. for field 3 in Table 1.

2.4 Encoding ISO 8583 Messages

ISO 8583 defines the acceptable content for the fields of an ISO 8583 message; it does not describe how the fields must be encoded. This has led to different implementations of ISO 8583 adopting incompatible encodings. Here are some examples:

- A numeric (*n*) field of length 6 might be encoded as 6 BCD digits in 3 bytes or in 6 bytes as a string of 6 ASCUU characters.
- A string field might use the ASCII character set or EBCDIC characters.
- A binary (*b*) field might encode 8 bits of the value in 8 bits or the value might be converted to an ASCII hexadecimal string in which case the resultant encoding needs twice as many bytes as the original value.
- While ISO 8583 specifies that a message consists of an MTI, a bitmap and data elements, some implementations precede the message with a length field containing the length of the message. Other implementations append padding characters to the end of the message so that all messages have the same length.

One of the goals of erl8583 is to make it simple to encode ISO 8583 messages irrespective of how some other party expects messages to be encoded.

3 Erl8583

Erl8583 is a library that can be used to construct and parse ISO 8583 messages for several encoding schemes used in practice. This section starts by describing how to install erl8583. The remainder of this section describes the modules that make up the erl8583 library.

The definitive references for the erl8583 API are the edocs in the distribution. This guide provides a high-level overview of the modules to help you get started. The modules can be classified into four types:

- The **erl8583_message** module that provides a data structure for an ISO 8583 message.
- Encoding rules modules that specify the domain of ISO 8583 data elements.
- Marshalling modules that are used to encode ISO 8583 messages or to decode byte streams into ISO 8583 messages.
- The **erl8583_convert** module that provides low-level conversions between data representations.

3.1 Installing erl8583

To install erl8583, unzip the erlang archive into some directory accessible from your Erlang runtime (e.g. /usr/lib/erlang/lib). The archive adheres to the standard directory structure for an OTP application:

- beam files are in the ebin directory.
- Header files are in the include directory.
- EDoc API documentation files are in the doc directory.

The source code is included in the src directory and the EUnit unit tests are found in the test directory. Example source code used in this guide is in the src_examples directory.

On Linux you should be able to build the archive by running the make.sh bash script.

3.2 The erl8583_message module

The erl8583_message module provides a data type representing an ISO 8583 message. Some of the functions exposed by the module are:

- **new** that creates an ISO 8583 message data structure.
- **set_mti** and **get_mti** for setting and getting the MTI of an ISO 8583 message.
- **set** and **get** for setting and getting the value of a specified data element specified by its ID in the range 0-192. ISO 8583 defines data elements with IDs in the range 1-192; ID 0 is an alias for the MTI.
- **update** for changing the value of a data element.
- **get_fields** that returns a list of the data element IDs that have been set for a message.

See the EDoc documentation for the complete list of functions and the arguments that are passed.

Valid types for data element values are: strings, binaries and nested message instances. The API does not prevent you setting the value of a data element to be an atom, a tuple, an integer or a float but those types will cause an exception to be raised if you try to marshal (see section 3.3) the message. Note in particular that numeric (*n*) values should be encoded as strings; e.g. set the MTI as "0200" not as the integer 200.

It may seem strange allowing data elements to be nested messages since ISO 8583 does not describe this scenario. However it is quite common to see data elements that are reserved for private use (e.g. field 127) to be nested messages. In such cases, one can refer to, say, field 127.3 which denotes sub field 3 in data element 127. The example in section 4.7 looks at such a case.

3.3 Encoding rules modules

Section 2.3 described how ISO 8583 defines valid content for the data elements of a message. A data element is constrained by the permissible content (e.g. *b*, *n*, *ans*, etc), its format (fixed, LLVAR or LLVAR) and its length. Erl8583 has three modules (erl8583_fields, erl8583_fields_1993 and erl8583_fields_2003) that specify the permissible content for a data element based on the 1987, 1993 and 2003 specifications (note that erl8583 currently doesn't support the 2003 specification). Each module exposes a function **get_encoding** that returns the valid content as a 3-tuple. The first element of the tuple is an atom that describes the content type (e.g. *b*, *n* or *ans*, etc). The second element of the tuple is an atom describing the format (*fixed*, *llvar* or *lllvar*). The third element of the tuple is the length specified as an integer. For the LLVAR and LLLVAR formats, the length denotes the maximum allowed length for the element.

The code snippet below shows some of the implementation of the erl8583_fields module.

```
-module(erl8583_fields).

%%
%% Include files
%%
%% @headerfile "../include/erl8583_types.hrl"
-include("erl8583_field_ids.hrl").
-include("erl8583_types.hrl").

%%
%% Exported Functions
%%
-export([get_encoding/1]).

%%
%% API Functions
%%

%% @doc Returns how a field is encoded as a triple consisting of the content
%%      (e.g. ans, b, z, etc), the format (e.g. llvar, lllvar or fixed) and
%%      the maximum length.
%%
%% @spec get_encoding(FieldId::integer()) -> field_encoding()
-spec(get_encoding(integer()) -> field_encoding()).

get_encoding(?MTI) ->
  {n, fixed, 4};
get_encoding(?BITMAP_EXTENDED) ->
  {b, fixed, 8};
get_encoding(?PAN) ->
  {n, llvar, 19};
get_encoding(?PROC_CODE) ->
  {n, fixed, 6};
%%
%% Code omitted...
%%
get_encoding(?TRACK_2_DATA) ->
  {z, llvar, 37};
get_encoding(?TRACK_3_DATA) ->
  {ans, lllvar, 104};
get_encoding(?RETRIEVAL_REF_NUM) ->
  {an, fixed, 12};
%%
%% Code omitted...
%%
```

The macros like **MTI**, **BITMAP_EXTENDED**, **TRACK_2_DATA**, etc. in the code listing are defined in the erl8583_field_ids.hrl header file.

3.4 Marshalling modules

The ISO 8583 specification provides an abstraction of messages that must be transformed to a list of bytes. Implementers of the ISO 8583 standard encode messages differently and there are a multitude of ways to marshal a message into a wire-level encoding. To cater for the most common encodings of messages, erl8583 provides the following four modules:

- **erl8583_marshaller_ascii** Transforms a message to a list of ASCII characters. Binaries and bitmaps are marshalled as ASCII hex strings.
- **erl8583_marshaller_binary** Transforms a message to a list of bytes. Binaries and strings are encoded using the values set in the message. Numeric data is transformed to BCD.
- **erl8583_marshaller_ebcdic** Transforms a message to a list of EBCDIC characters. Binaries and bitmaps are marshalled as EBCDIC hex strings.
- **erl8583_marshaller_xml** Transforms a message to an XML document. The XML schema is (intended to be) the same as that used by jPOS.

Unfortunately, the above four marshalling modules are usually insufficient. Erl8583 provides a generic marshaller, **erl8583_marshaller**, that can be used to implement other marshallers. The generic marshaller can be passed a list of modules that will be called back to marshal:

- The MTI.
- The bitmap.
- The data elements.

Additionally, the marshaller can be supplied with modules that will be called:

- Before marshalling the MTI.
- After marshalling all the data elements.
- To get how a data element is encoded (e.g. to specify that the 1987 version of the ISO 8583 standard must be used).
- That specifies the order in which data elements must be marshalled.

The implementation of **erl8583_marshaller**'s marshal function is shown below. The structure of the code shows that code is called before marshalling the MTI, then the MTI is marshalled, next the bitmap is marshalled, then the data elements are encoded and, finally, a function that completes the marshalling is called.

```
%% @doc Marshals an ISO 8583 message into a byte sequence.
%%
%% @spec marshal(iso8583message(), list(marshal_handler())) -> list(byte())
-spec(marshal(iso8583message(), list(marshal_handler())) -> list(byte())).

marshal(Message, MarshalHandlers) ->
  OptionsRecord = parse_options(MarshalHaendlers, #marshal_options{}),
  {Marshalled1, Message1} = init_marshalling(OptionsRecord, Message),
  Marshalled2 = Marshalled1 ++ encode_mti(OptionsRecord, Message1),
  {MarshalledBitmap, Message2} = encode_bitmap(OptionsRecord, Message1),
  Marshalled3 = Marshalled2 ++ MarshalledBitmap,
  Marshalled4 = Marshalled3 ++ encode_fields(OptionsRecord, Message2),
  end_marshalling(OptionsRecord, Message2, Marshalled4).
```

Four of the code examples in section 4 show how to use the **erl8583_marshaller** to implement various marshalling schemes.

As an aside, the ASCII, binary, EBCDIC and XML marshaller use the generic marshaller; these marshallers invoke the generic marshaller but pass themselves as the modules to be called back for marshalling the MTI, bitmap, etc.

3.5 The erl8583_convert module

The **erl8583_convert** module provides functions to convert between representations of data. Functions are exported to:

- Convert ASCII characters to EBCDIC (and vice-versa).
- Convert a string of decimal digits to BCD (and vice-versa).
- Pad a string with trailing spaces.
- Pad a string of digits with leading zeroes.
- Convert lists of bytes to ASCII hex strings (and vice-versa).

In practice, you may never need to call these functions yourself; the functions are used mainly by the marshallers and you will invoke the marshalling functions instead. See the EDocs for complete details of the module.

4 Example Code

The first example in this section shows the use of the most commonly used functions in the **erl8583_message** module. The remaining six samples show how to marshal and unmarshal messages.

4.1 The erl8583_message module

The sample code below shows how to create a message. After creating the message we set the value of data elements of various types:

- Field 0 (the MTI) is a numeric value.
- Field 43 is a string.
- Field 64 is a binary value.
- Field 127 is another message containing subfields 2 and 12. The subfields can be referenced as the lists [127, 2] and [127, 12].

```
%% An example that demonstrates setting and getting data elements in an
%% iso8583message().
-module(example_1_message).

%%
%% Include files
%%
-include_lib("erl8583/include/erl8583_field_ids.hrl").

%%
%% Exported Functions
%%
-export([test/0]).

%%
%% API Functions
%%
test() ->
  % Create a message.
  Msg1 = erl8583_message:new(),

  % Set the MTI (field 0), card acceptor name/location (field 43), a MAC (field 64) and
  % field 127 is a private use field whose contents is not defined by ISO.
```

```
  % This code shows how a data element can be a numeric value (field 0), a string
  % (field 43), a binary (field 64) or another message (field 127) containing
  % subfields (fields 127.2 and 127.3).
  Msg2 = erl8583_message:set_mti("0200", Msg1),
  Msg3 = erl8583_message:set(?CARD_ACCEPTOR_NAME_LOCATION, "ZIB Head Office ATM    V/I  ↩
      Lagos    01NG", Msg2),
  Msg4 = erl8583_message:set(?MESSAGE_AUTHENTICATION_CODE, <<1,2,3,4,5,6,7,8>>, Msg3),
  % Field 127 is a message containing subfields 127.2 and 127.12.
  % Notice that we use two different ways of setting the value of a subfield.
  % The way we set subfield 127.12 is the preferred idiom.
  SubMsg1 = erl8583_message:new(),
  SubMsg2 = erl8583_message:set(2, "0000387020", SubMsg1),
  Msg5 = erl8583_message:set(127, SubMsg2, Msg4),
  Msg6 = erl8583_message:set([127, 12], "ZIBeTranzSnk", Msg5),

  % Display the fields defined for the message.
  io:format("Fields: ~w~n", [erl8583_message:get_fields(Msg6)]),

  % Display fields 0, 43, 64, 127.2 and 127.12.
  io:format("MTI:                          ~s~n", [erl8583_message:get(0, Msg6)]),
  io:format("Card acceptor name/location: ~s~n", [erl8583_message:get(43, Msg6)]),
  io:format("MAC:                          ~p~n", [erl8583_message:get(64, Msg6)]),
  io:format("Field 127.2:                  ~s~n", [erl8583_message:get([127, 2], Msg6)]),
  io:format("Field 127.12:                 ~s~n", [erl8583_message:get([127, 12], Msg6)]).
```

Notice that we included the **erl8583_field_ids.hrl** header so that we could refer to the fields using macro identifiers (e.g. **MTI**) rather than numeric values.

Running the code produces the following output

```
Fields:  [0,43,64,127]

MTI: 0200

Card acceptor name/location:  ZIB Head Office ATM V/I Lagos 01NG

MAC: <<1,2,3,4,5,6,7,8>>

Field 127.2:  0000387020

Field 127.12:  ZIBeTranzSnk

ok
```

4.2 Marshalling a message with the erl8583_marshaller_xml module

The code below shows how to marshal the message using the **erl8583_marshaller_xml** module. The XML marshaller is intended to be compatible with jPOS's XML packager. In practice, marshalling a message into an XML document is not very useful. However the XML representation is human-readable which cannot be said of the other encodings.

```
%% An example that demonstrates marshalling an
%% iso8583message().
-module(example_2_xml_marshaller).

%%
%% Include files
%%
-include_lib("erl8583/include/erl8583_field_ids.hrl").

%%
%% Exported Functions
%%
```

```
-export([test/0]).

%%
%% API Functions
%%
test() ->
  % Create a message.
  Msg1 = erl8583_message:new(),

  % Set some fields.
  Msg2 = erl8583_message:set_mti("0200", Msg1),
  Msg3 = erl8583_message:set(?CARD_ACCEPTOR_NAME_LOCATION, "ZIB Head Office ATM    V/I  ↩
      Lagos    01NG", Msg2),
  Msg4 = erl8583_message:set(?MESSAGE_AUTHENTICATION_CODE, <<1,2,3,4,5,6,7,8>>, Msg3),
  Msg5 = erl8583_message:set([127, 2], "0000387020", Msg4),
  Msg6 = erl8583_message:set([127, 12], "ZIBeTranzSnk", Msg5),

  % Marshal the message and display it.
  Marshalled = erl8583_marshaller_xml:marshal(Msg6),
  io:format("~s~n", [Marshalled]).
```

Running the above program produces the output underneath. Notice that the binary field (field 64) has attribute type="binary" and that the complex data element (field 127) is encapsulated in an isomsg tag.

```
<isomsg>
<field id="0" value="0200" />
<field id="43" value="ZIB Head Office ATM V/I Lagos 01NG" />
<field id="64" value="0102030405060708" type="binary" />
<isomsg id="127">
<field id="2" value="0000387020" />
<field id="12" value="ZIBeTranzSnk" />
</isomsg>
</isomsg>
ok
```

4.3 A minimal ISO 8583 client

jPOS is a popular Java framework for processing ISO 8583 messages. It supports a host of ways to marshal messages. In this example we use an ASCII packer (the jPOS term for what is called a marshaller in erl8583). This serves two purposes:

- It shows that erl8583 can be used to exchange messages.
- It shows that ISO 8583 involves more than marshalling messages; there are issues around networking as well.

We start by writing a minimal jPOS server. To get this code to compile and run, you will need to download jPOS from http://www.jpos.org. The program listens on port 8000 for ISO 8583 messages and echoes them back to the client after changing the MTI to indicate that the message is a response.

```
import org.jpos.iso.*;
import org.jpos.iso.channel.ASCIIChannel;
import org.jpos.iso.packager.ISO87APackager;

public class Example3Server implements ISORequestListener {

  public static void main(String[] args) throws Exception {
```

```
    ServerChannel channel = new ASCIIChannel(new ISO87APackager());
    ISOServer server = new ISOServer(8000, channel, null);
    server.addISORequestListener(new Example3Server());
    new Thread(server).start();
  }

  @Override
  public boolean process(ISOSource source, ISOMsg m) {
    try {
      m.setResponseMTI();
      source.send(m);
      return true;
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }

}
```

The above line

```
ServerChannel channel = new ASCIIChannel(new ISO87APackager());
```

creates an ISO87APackager that can marshal and unmarshal ASCII encoded messages. The ASCIIChannel class takes care of a transport layer detail: Identifying the end of the message. This class expects that the ASCII encoded message will be preceded by four ASCII digits that contain the length of the ISO 8583 message.

The code below is an implementation of our client. We create an "0800" message, marshal the message, prepend four characters encoding the length and send the resultant data to the server. The do_recv function listens for the response and waits until it has all the response bytes, strips the first four bytes encoding the message length and returns the marshalled response data. The response is then unmarshalled and the MTI is displayed.

```
-module(example_3_client).
-export([test/0]).

test() ->
  % Create a message.
  Msg1 = erl8583_message:new(),
  Msg2 = erl8583_message:set_mti("0800", Msg1),
  Msg3 = erl8583_message:set(3, "300000", Msg2),
  Msg4 = erl8583_message:set(24, "045", Msg3),
  Msg5 = erl8583_message:set(41, "11111111", Msg4),
  Msg6 = erl8583_message:set(42, "222222222222222", Msg5),
  Msg7 = erl8583_message:set(63, "This is a Test Message", Msg6),

  % Marshal the message using the ASCII marshaller.
  AsciiMessage = erl8583_marshaller_ascii:marshal(Msg7),
  io:format("Sending:~n~s~n~n", [AsciiMessage]),

  % Our jPOS server expects a four digit length to be sent before the message.
  % We use an erl8583_convert function to create the header.
  % Send the request.
  LengthHeader = erl8583_convert:integer_to_string(length(AsciiMessage), 4),
  {ok, Sock} = gen_tcp:connect("localhost", 8000, [list, {packet, 0}, {active, false}]),
  ok = gen_tcp:send(Sock, LengthHeader ++ AsciiMessage),

  % Get the response to the request and unmarshal it.
  AsciiResponse = do_recv(Sock, []),
  io:format("Received:~n~s~n", [AsciiResponse]),
  Response = erl8583_marshaller_ascii:unmarshal(AsciiResponse),
```

```
  % Display the MTI from the response.
  io:format("~nMTI: ~s~n", [erl8583_message:get(0, Response)]).

% A pretty standard function to read data from a socket.
do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
      UpdatedBs = Bs ++ B,
      if
        % There's a 4 byte length header. Use it to check if
        % we have received the whole response.
        length(UpdatedBs) < 4 ->
          do_recv(Sock, UpdatedBs);
        true ->
          {LenStr, Rest} = lists:split(4, UpdatedBs),
          Len = list_to_integer(LenStr) + 4,
          if
            Len >= length(UpdatedBs) ->
              % Got the whole response, return the response
              % but not the length header.
              Rest;
            true ->
              % Haven't got all the response data.
              do_recv(Sock, UpdatedBs)
          end
      end
    end.
```

If we start the Java server and run the Erlang client, the client produces the following output showing the ASCII marshalled request and response and the MTI of the response message:

```
Sending:
08002000010000C000023000000451111111122222222222222022This is a Test Message


Received:
08102000010000C000023000000451111111122222222222222022This is a Test Message


MTI: 0810
ok
```

4.4 Using the generic erl8583_marshaller module

In the previous section we wrote an ISO 8583 server using jPOS and used the erl8583_marshaller_ascii module to marshal and unmarshal our messages. In this section we write a minimal server in Python using the iso8583py library (available from http://code.google.com/p/iso8583py). The server code below is modified from code written by Igor Custodio and licensed under the GNU GPL. The original code can be found on the iso8583py site hosted by googlecode.

```
from ISO8583.ISO8583 import ISO8583
from ISO8583.ISOErrors import *
from socket import *

# Configure the server
serverIP = "127.0.0.1"
serverPort = 8583
maxConn = 5
```

```
# Create a TCP socket
s = socket(AF_INET, SOCK_STREAM)
# bind it to the server port
s.bind((serverIP, serverPort))
# Configure it to accept up to N simultaneous Clients waiting...
s.listen(maxConn)


# Run forever
while 1:
        #wait new Client Connection
        connection, address = s.accept()
        while 1:
                # receive message
                isoStr = connection.recv(2048)
                if isoStr:
                        pack = ISO8583()
                        #parse the iso
                        try:
                                pack.setNetworkISO(isoStr)
                                pack.getBitsAndValues()
                        except InvalidIso8583, ii:
                                print ii
                                break
                        except:
                                print 'Something happened!!!!'
                                break

                        #send answer
                        pack.setMTI('0810')
                        ans = pack.getNetworkISO()
                        connection.send(ans)

                else:
                        break
        # close socket
        connection.close()
```

The above code starts a server that listens on port 8583 and echoes any request after changing the MTI to 0810. This server, like the one in the previous section, expects data to be ASCII encoded. Unlike the previous server though where the length of messages was encoded as 4 ASCII digits this server expects the length of the marshalled data to be encoded as two (big-endian) bytes. It would be easy to modify only three or four lines of the previous client code to work with this server. However we'll make more extreme changes and implement a custom marshaller for marshalling the messages for this server. Our marshaller will prepend the length of the marshalled message at the beginning of the message. The listing below for the example_4_marshaller module shows the implementation:

```
-module(example_4_marshaller).

-export([marshal/1, marshal_end/2]).

% Our marshal function uses the ASCII marshaller to marshal the MTI,
% the bitmap and the data elements but also specifies that the
% marshal_end function in this module must be called after marshalling
% all data elements.
marshal(Message) ->
  MarshallingOptions = [{mti_marshaller, erl8583_marshaller_ascii},
              {bitmap_marshaller,  erl8583_marshaller_ascii},
              {field_marshaller,  erl8583_marshaller_ascii},
              {end_marshaller, ?MODULE}],
  erl8583_marshaller:marshal(Message, MarshallingOptions).

% After marshalling the message, we prepend the message with the length of
```

```
% the message encoded in two bytes.
marshal_end(_Message, Marshalled) ->
  Length = length(Marshalled),
  [Length div 256, Length rem 256] ++ Marshalled.
```

The marshalling code uses the generic marshaller to call back the ASCII marshaller when marshalling the MTI, the bitmap and the data elements. After marshalling the data elements, the generic marshaller calls the marshal_end function of the example_4_marshaller module (as specified by the {end_marshaller, ?MODULE} option passed to the generic marshaller's marshal function). The marshal_end function prepends the marshalled message with two bytes that encode the length of the marshalled data.

Our client code, example_4_client, is now rewritten to use our custom marshaller. Notice that we no longer have code that prepends the length of the message to the marshalled data:

```
-module(example_4_client).
-export([test/0]).

test() ->
  % Create a message.
  Msg1 = erl8583_message:new(),
  Msg2 = erl8583_message:set_mti("0800", Msg1),
  Msg3 = erl8583_message:set(3, "300000", Msg2),
  Msg4 = erl8583_message:set(24, "045", Msg3),
  Msg5 = erl8583_message:set(41, "11111111", Msg4),
  Msg6 = erl8583_message:set(42, "222222222222222", Msg5),
  Msg7 = erl8583_message:set(63, "This is a Test Message", Msg6),

  % Marshal the message using our custom marshaller and send the result.
  AsciiMessage = example_4_marshaller:marshal(Msg7),
  {ok, Sock} = gen_tcp:connect("localhost", 8583, [list, {packet, 0}, {active, false}]),
  io:format("Sending:~n~p~n~n", [AsciiMessage]),
  ok = gen_tcp:send(Sock, AsciiMessage),

  % Get the response to the request and unmarshal it.
  AsciiResponse = do_recv(Sock, []),
  io:format("Received:~n~s~n", [AsciiResponse]),
  Response = erl8583_marshaller_ascii:unmarshal(AsciiResponse),

  % Display the MTI from the response.
  io:format("~nMTI: ~s~n", [erl8583_message:get(0, Response)]).

% A pretty standard function to read data from a socket.
do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
      UpdatedBs = Bs ++ B,
      if
        % There's a 2 byte length header. Use it to check if
        % we have received the whole response.
        length(UpdatedBs) < 2 ->
          do_recv(Sock, UpdatedBs);
        true ->
          {[Len1, Len2], Rest} = lists:split(2, UpdatedBs),
          Len = Len1 * 256 + Len2 + 2,
          if
            Len >= length(UpdatedBs) ->
              % Got the whole response, return the response
              % but not the length header.
              Rest;
            true ->
              % Haven't got all the response data.
              do_recv(Sock, UpdatedBs)
```

```
          end
      end
    end.
```

If we run the server and the client, the client displays the following:

```
Sending:
[0,77,48,56,48,48,50,48,48,48,48,49,48,48,48,48,67,48,48,48,48,50,51,48,48,48,
48,48,48,52,53,49,49,49,49,49,49,49,49,50,50,50,50,50,50,50,50,50,50,50,50,
50,50,50,48,50,50,84,104,105,115,32,105,115,32,97,32,84,101,115,116,32,77,
101,115,115,97,103,101]

Received:
08102000010000c000023000000451111111122222222222222022This is a Test Message

MTI: 0810
ok
```

The first two bytes of the marshalled data [0, 77] indicate that 77 bytes of data follow.

Writing a marshaller to prepend the length of the data to follow is NOT recommended; the length is part of the transport protocol not the application layer. The code for adding the length should be done in a function that sends the data over the network. The purpose of this code was primarily to show how to write a very basic marshaller by extending an existing one.

4.5 An unmarshaller for American Express messages

American Express provide a very detailed specification on how to exchange ISO 8583 messages with them. Unusually, they encode numeric and string data elements (including the MTI) using EBCDIC rather than ASCII. Unfortunately, we cannot use the erl8583_marshaller_ebcidc module for marshalling messages directly since they use binary encodings for the bitmap and binary fields. In this example, we write code capable of unmarshalling AMEX messages:

```
-module(example_5_unmarshaller).

-export([unmarshal/1, unmarshal_field/3]).

unmarshal(Marshalled) ->
  % We use the EBCDIC marshaller to unmarshal the MTI,
  %        the binary marshaller to unmarshal the bitmap,
  %        this module unmarshals the data elements
  MarshallingOptions = [{mti_marshaller, erl8583_marshaller_ebcdic},
              {bitmap_marshaller,  erl8583_marshaller_binary},
              {field_marshaller,  ?MODULE}],
  erl8583_marshaller:unmarshal(Marshalled, MarshallingOptions).

% We use the binary marshaller to unmarshal binary data elements
% and the EBCDIC marshaller to unmarshal all other data elements.
unmarshal_field(FieldId, Marshalled, EncodingRules) ->
  % We use the encoding rules to get whether a data element is binary
  % or anything else.
  case EncodingRules:get_encoding(FieldId) of
    {b, _, _} ->
      erl8583_marshaller_binary:unmarshal_field(FieldId, Marshalled, EncodingRules);
    {_, _, _} ->
      erl8583_marshaller_ebcdic:unmarshal_field(FieldId, Marshalled, EncodingRules)
  end.
```

The unmarshal function above specifies that the **erl8583_marshaller** should use the EBCDIC marshaller to unmarshal the MTI, the binary marshaller to unmarshal the bitmap and use the **example_5_unmarshaller** module to unmarshal data elements. When a data element needs to be unmarshalled, the unmarshal_field function of **example_5_unmarshaller** is called. Three arguments (a field ID, marshalled data and a module that specifies how a data element is encoded) are passed to the field unmarshalling function. The implementation uses the encoding rules module to get whether the data element to be unmarshalled is binary or some other type. If the data element is binary, our unmarshaller delegates the unmarshalling to the binary marshaller otherwise the EBCDIC marshaller is used.

The **example_5_amex_message** module exercises our unmarshaller:

```
-module(example_5_amex_message).

-export([test/0]).

test() ->
  % The marshalled message to unmarshal.
  MarshalStr = "F1F1F0F0703425C000408000F1F5F3F7F0F0F1F2F3F4F5F6F1" ++
           "F2F3F4F5F0F0F4F0F0F0F0F0F0F0F0F0F0F0F1F0F0F0" ++
           "F0F0F0F0F1F0F9F0F1F0F0F0F0F0F0F0F1F3F0F1F8F4" ++
           "F0F1F0F1F1F5F0F6F0F0F1F2F0F1F8F0F1F2F3F4F0F7F4" ++
           "F2F0F0F0F0F0F0F0F0F1F2F3F4F5F6F7F8F8F4F0",
  Marshalled = binary_to_list(erl8583_convert:ascii_hex_to_binary(MarshalStr)),

  % Unmarshal the message using our unmarshaller.
  Message = example_5_unmarshaller:unmarshal(Marshalled),

  % Get the field IDs and display the values.
  FieldIds = erl8583_message:get_fields(Message),
  F = fun(FieldId) ->
        FieldValue = erl8583_message:get(FieldId, Message),
        io:format("Field ~p: ~s~n", [FieldId, FieldValue])
    end,
  lists:map(F, FieldIds),
  ok.
```

Running our test on the test message in the above code produces the following output:

```
Field 0:  1100
Field 2:  370012345612345
Field 3:  004000
Field 4:  000000000100
Field 11:  000001
Field 12:  090100000000
Field 14:  1301
Field 19:  840
Field 22:  101150600120
Field 24:  180
Field 25:  1234
Field 26:  0742
Field 42:  000000012345678
Field 49:  840
ok
```

4.6 Unmarshalling a Postilion message

Postilion is a commercial ISO 8583 framework widely deployed for processing and switching messages. A commenter on a jPOS newsgroup had difficulty unpacking the following message from a Postilion 'node':

```
30323030F23E049508E0810000000000040000223136303537303031303531 32
32393839383433313030303030303030303030303030303030313031323133 3437
33353030303031313133343733353130313230373132313031313031303130 3043
30303030303030304330303030303030303036363237363239303030303033 3033
38373032303130353730303131303031202020202020202020202020205A 494220
48656164204F66666963652041544D20202020562F49204C61676F7320202020 20
30314E473536363030343135313031303430393031323636353930313532 3131
32303132303331343430303230303031313560 1C10000000000313030303030 3033
3338373032305A656E69746841544D7363725A4942655472616E7A536E6B3030
30303032303030303031315A656E6974685447202020203132 5A4942655472616E
7A536E6B303132333431303030303120202035363 65A454E4954482042323030
3630393231
```

There are three points about the above message (the first point is obvious and the last two aren't):

- The message is encoded as ASCII hex which is not a marshalling scheme inherently supported by erl8583.
- Field 127 is encoded with a 6 digit length rather than a 3 digit length as in the ISO 8583 specification. Effectively the format is LLLLLLVAR rather than LLLVAR. This seems to have been the cause of the poster's problem.
- Field 127 is itself a message containing a bitmap followed by data elements; i.e.field 127 contains subfields.

In this section we write an unmarshaller that is capable of parsing the above message and attends to the first two issues. In the next section we extend our unmarshaller so that it unpacks field 127 into a message.

The code below implements our unmarshaller:

```
-module(example_6_unmarshaller).

-export([unmarshal/1, unmarshal_init/2, unmarshal_field/3]).

% Use:
%      The ASCII marshaller to unmarshal the MTI
%      The binary marshaller to unmarshal the bitmap
%      This module to unmarshal data elements
% Also call the unmarshal_init method of this module before
% trying to unmarshal the message.
unmarshal(Marshalled) ->
  MarshallingOptions = [{mti_marshaller, erl8583_marshaller_ascii},
              {bitmap_marshaller, erl8583_marshaller_binary},
              {field_marshaller, ?MODULE},
              {init_marshaller, ?MODULE}],
  erl8583_marshaller:unmarshal(Marshalled, MarshallingOptions).

% The marshalled message is encoded as ASCII hex. Convert the ASCII hex
% to a list of bytes before unmarshalling further.
unmarshal_init(Message, Marshalled) ->
  MarshalledBin = erl8583_convert:ascii_hex_to_binary_list(Marshalled),
  {Message, MarshalledBin}.

% This function is called when a data element needs to be unmarshalled.
```

```
%
% Special handling for field 127.
% The length of field 127 is encoded in 6 bytes rather than 3.
unmarshal_field(127, Marshalled, _EncodingRules) ->
  {LenStr, Rest} = lists:split(6, Marshalled),
  Len = list_to_integer(LenStr),
  {Value, MarshalledRem} = lists:split(Len, Rest),
  {Value, MarshalledRem, []};
% Use the binary marshaller to unmarshal binary fields and
% the ASCII marshaller to unmarshal all other fields.
unmarshal_field(FieldId, Marshalled, EncodingRules) ->
  case EncodingRules:get_encoding(FieldId) of
    {b, _, _} ->
      erl8583_marshaller_binary:unmarshal_field(FieldId, Marshalled, EncodingRules);
    {_, _, _} ->
      erl8583_marshaller_ascii:unmarshal_field(FieldId, Marshalled, EncodingRules)
  end.
```

The unmarshalling code's unmarshal function specifies modules that must be called before unmarshalling starts and to unmarshal the MTI, bitmap and data elements. The unmarshal_init function is called and converts a string of ASCII hex characters to a list of bytes so that we can use the default unmarshallers to do the bulk of unmarshalling. To unmarshal the MTI and bitmap the ASCII and binary unmarshallers are used. The above module provides the code to unmarshal data elements:

- We implement a function clause specifically for unmarshalling field 127 because of its non-standard encoding.
- Binary data elements are unmarshalled using the binary marshaller.
- All other data elements are unmarshalled using the ASCII marshaller.

We exercise our unmarshaller by calling the test function of the example_6_postilion_message module:

```
-module(example_6_postilion_message).

-export([test/0]).

test() ->
  % Unmarshal this message which caused problems for
  % someone on a jPOS forum.
  Marshalled = "30323030F23E049508E0810000000000" ++
      "0400002231363035373030313035313 2" ++
      "3239383938343331303030303030303 0" ++
      "3030303030303030313031323133343 7" ++
      "3335303030303131313333433733531 30" ++
      "3132303731323130313130303130304 3" ++
      "3030303030303030433030303030303 0" ++
      "3030363632373632393030303030303 3" ++
      "3837303230313035373030313130303 1" ++
      "2020202020202020202020205A494220" ++
      "48656164204F666669636520415 44D20" ++
      "202020562F49204C61676F7320202020" ++
      "30314E4735363630303431353130313 0" ++
      "3430393031323636353930313532313 1" ++
      "3230313230333134343030323030303 1" ++
      "3135601C1000000000003130303030 30" ++
      "3338373032305A656E69746841544D73" ++
      "63725A4942655472616E7A536E6B3030" ++
      "3030303230303030313 15A656E697468" ++
      "5447202020203132 5A4942655472616E" ++
      "7A536E6B3031323334313030303120" ++
      "20203536365A454E4954482042323030" ++
      "3630393231",
```

```
Message = example_6_unmarshaller:unmarshal(Marshalled),

% Display the fields and their values
FieldIds = erl8583_message:get_fields(Message),
F = fun(FieldId) ->
    FieldValue = erl8583_message:get(FieldId, Message),
    io:format("Field ~p: ~s~n", [FieldId, FieldValue])
  end,
lists:map(F, FieldIds),
ok.
```

Running the test function produces:

```
Field 0:  0200
Field 1:  ^@^@^@^@^D^@^@"
Field 2:  0570010512298984
Field 3:  310000
Field 4:  000000000000
Field 7:  1012134735
Field 11:  000011
Field 12:  134735
Field 13:  1012
Field 14:  0712
Field 15:  1011
Field 22:  001
Field 25:  00
Field 28:  C00000000
Field 30:  C00000000
Field 32:  627629
Field 37:  000000387020
Field 41:  10570011
Field 42:  001
Field 43:  ZIB Head Office ATM V/I Lagos 01NG
Field 49:  566
Field 56:  1510
Field 102:  4090126659
Field 123:  211201203144002
Field 127:  `^\^P^@^@^@^@^@100000387020ZenithATMscrZIBeTranzSnk000002000011ZenithTG
12ZIBeTranzSnk01234100001 566ZENITH B20060921
ok
```

The most noteable feature of the above is the apparent garbage in field 127 that encodes the bitmap. In the next section, we extend our unmarshaller to extract the subfields of field 127.

4.7 Writing a field encoder for a Postilion message

Data element 127 of the Postilion message of the previous section contains subfields. To unpack this field, we need to define how the subfields are encoded. We implement an encoding rules module, example_7_field127_rules:

```
-module(example_7_field127_rules).

-export([get_encoding/1]).

% We provide clauses only for field IDs 2, 3, 12, 13, 14 and 20
% since these are the only (sub)fields needed for example 7.
% For completeness we could add clauses for other fields.
get_encoding(2) ->
  {ans, llvar, 32};
get_encoding(3) ->
  {ans, fixed, 48};
get_encoding(12) ->
  {ans, llvar, 25};
get_encoding(13) ->
  {ans, fixed, 17};
get_encoding(14) ->
  {ans, fixed, 8};
get_encoding(20) ->
  {n, fixed, 8}.
```

The types defined in our encoding rules module were obtained from the same web page where the problem of unmarshalling the message was reported. The page also specifies the encoding of other subfields of field 127 but we don't implement them since they're not relevant to our example.

Now that we have the encoding rules implemented, we rework the unmarshaller of the previous section to use it. The result is the example_7_unmarshaller module:

```
-module(example_7_unmarshaller).

-export([unmarshal/1, unmarshal_init/2, unmarshal_field/3]).

unmarshal(Marshalled) ->
  MarshallingOptions = [{mti_marshaller, erl8583_marshaller_ascii},
              {bitmap_marshaller, erl8583_marshaller_binary},
              {field_marshaller, ?MODULE},
              {init_marshaller, ?MODULE}],
  erl8583_marshaller:unmarshal(Marshalled, MarshallingOptions).

unmarshal_init(Message, Marshalled) ->
  MarshalledBin = erl8583_convert:ascii_hex_to_binary_list(Marshalled),
  {Message, MarshalledBin}.

% This function is called when a data element needs to be unmarshalled.
%
% Special handling for field 127.
% The length of field 127 is encoded in 6 bytes rather than 3.
% After extracting the value of field 127 we unmarshal it since
% it contains subfields.
unmarshal_field(127, Marshalled, _EncodingRules) ->
  {LenStr, Rest} = lists:split(6, Marshalled),
  Len = list_to_integer(LenStr),
  {Value, MarshalledRem} = lists:split(Len, Rest),

  % Notice:
  % 1. That we need to specify how the subfields of field 127 are
  %    encoded.
  % 2. We don't specify a marshaller for the MTI (since there isn't
```

```
  %    a message type).
  MarshallingOptions = [{bitmap_marshaller, erl8583_marshaller_binary},
              {field_marshaller, ?MODULE},
              {encoding_rules, example_7_field127_rules}],
  Message127 = erl8583_marshaller:unmarshal(Value, MarshallingOptions),

  % Return the unmarshalled message as the value of field 127.
  {Message127, MarshalledRem, []};

% Use the binary marshaller to unmarshal binary fields and
% the ASCII marshaller to unmarshal all other fields.
unmarshal_field(FieldId, Marshalled, EncodingRules) ->
  case EncodingRules:get_encoding(FieldId) of
    {b, _, _} ->
      erl8583_marshaller_binary:unmarshal_field(FieldId, Marshalled, EncodingRules);
    {_, _, _} ->
      erl8583_marshaller_ascii:unmarshal_field(FieldId, Marshalled, EncodingRules)
  end.
```

The unmarshal function clause to unmarshal field 127 now uses the erl8583_marshaller to unmarshal the value of field 127 using our encoding rules module and the existing unmarshal function. We do not specify a module to unmarshal the MTI since the subfield does not contain a message type.

The code below exercises our unmarshaller and displays the subfields of data element 127:

```
-module(example_7_postilion_message).

-export([test/0]).

test() ->
  % Unmarshal this message which caused problems for
  % someone on a jPOS forum.
  Marshalled = "30323030F23E049508E0810000000000" ++
           "040000223136303537303030130353132" ++
           "323938393834333130303030303030303030" ++
           "303030303030303030313031323133343 7" ++
           "333530303030313131333343733353130" ++
           "313230373132313031313030313030343" ++
           "303030303030303030433030303030303030" ++
           "303036363237363239303030303030303033" ++
           "383730323031303537303031313030313 1" ++
           "202020202020202020202020205A494220" ++
           "48656164204F66666963652041544D20" ++
           "202020562F49204C61676F7320202020" ++
           "30314E473536363030343135313031303130" ++
           "343039303132363635393031353231313 1" ++
           "323031323033313434303032303030313 1" ++
           "3135601C1000000000000313030303030" ++
           "3338373032305A656E69746841544D73" ++
           "63725A4942655472616E7A536E6B3030" ++
           "303030323030303031315A656E697468" ++
           "544720202020313 25A4942655472616E" ++
           "7A536E6B303132333431303030303120" ++
           "20203536365A454E4954482042323030" ++
           "3630393231",
  Message = example_7_unmarshaller:unmarshal(Marshalled),

  % Get field 127 and display its subfields
  Field127 = erl8583_message:get(127, Message),
  F = fun(FieldId) ->
        FieldValue = erl8583_message:get(FieldId, Field127),
        io:format("Field 127.~p: ~s~n", [FieldId, FieldValue])
```

```
  end,
Field127Subfields = erl8583_message:get_fields(Field127),
lists:map(F, Field127Subfields),
ok.
```

Calling the test function of example_7_postilion_message produces:

```
Field 127.2:  0000387020
Field 127.3:  ZenithATMscrZIBeTranzSnk000002000011ZenithTG
Field 127.12:  ZIBeTranzSnk
Field 127.13:  01234100001 566
Field 127.14:  ZENITH B
Field 127.20:  20060921
ok
```

5 Concluding comments

The erl8583 library provides modules for encapsulating ISO 8583 messages and for packing and unpacking messages for transmission over a network. The library does not provide functions for processing ISO 8583 messages. However, there is a project, node8583, also hosted on googlecode that has message processing as its goal and for which erl8583 was written as a first step.