ETN

Enrico Deiana, Emanuele Del Sozzo

Introduction

This paper describes the main data structures contained in the header file *etn.h*. In particular, it focuses on *Encoder* and *Decoder* data structures and how they are related to data types.
Header *etn.h* is part of *libetn*, a C library for encoding, decoding, and verifying **ETN** types. The library's interface is built around readers and writers, which can be overloaded to encode/decode/verify memory buffers, file descriptors, etc.
Library *libetn* interacts with code generated using *eg2source*, that generates C code containing structures that describe types, and *libetn* consumes these definitions to perform encoding/decoding/verifying.

Encoder

EtnEncoder is a "writer" and can be considered a base class. This data structure contains two function pointers, a *top-level pointer* out of the *red-black tree* (to avoid a malloc for a simple type that contains no pointers or any types), a *red-black tree* and an *index*.

```
typedef struct EtnEncoder_s {
        int (*write)(struct EtnEncoder_s *e, uint8_t *data, EtnLength length);
        void (*flush)(struct EtnEncoder_s *e);
        void *topLevelPointer;
        struct rbtree addrToIndex;
        EtnLength index;
} EtnEncoder;
```

The two function pointers are one to a *write* function and one to a *flush* function. The actual implementations of both are declared in the header file *packetEncoder.h* and defined in file *packetEncoder.c*.
The *write* function writes data to a packet sending fragments as maximum size reached. The final (non-full) fragment is not sent because there may be remaining data to write into it.
The *flush* function flushes packet encoder, sending the packet as the final fragment. Note that if the write does not fill the first fragment, then nothing will be sent until flush is called.
The *red-black tree* structure is used to remember pointers to some encoded types so that we can handle type loops, e.g.:

```
typedef struct a_s{
        B* b;
}A;

typedef struct b_s{
        C* c;
}B;

typedef struct c_s{
        A* a;
```

```
}C;
```

Each node of the red-black tree contains a pointer to the encoded datum and its index (which is a unique identifier for each encoded datum of the data set that is going to be encoded). So, looking at the example we have:

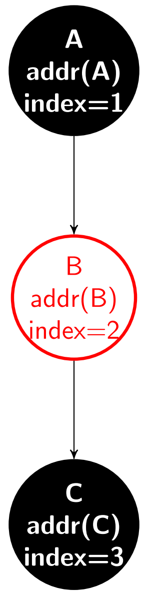

Figure 1: Type loops example

Red-black trees are not used for data of any type (it has no sense to use them for encoding data of simple type like integers, chars or structure without pointers, for instance), but only for the pointer to the type that was passed to *encode()* and all *EtnKindPtrs* and *EtnKindAnys* types. In this way, if we encounter a type already seen in a circular structure we encode it but its pointer is not added to the red-black tree (which is a sort of representation of the structure of a complex type), so we do not end in infinite loops trying to represent type loops.
The *index* is a counter incremented every time an *encode()* function (that encode each datum of the data set that is being encoded) is called.
With *encode()* function we mean the set of functions which encode each type of data supported by ETN, from integers to maps, unions, tuples etc.. They are defined in *encode.c* and an *EncoderMap* is used to call the right *encode()* function for each data type.
EtnBufferEncoder is a specialization of *EtnEncoder*. It encodes to provided memory range. In contains an *EtnEncoder* and two pointers to *uint8_t* type that point to a memory range.

```
typedef struct EtnBufferEncoder_s {
        EtnEncoder encoder;
        uint8_t *dataCurrent;
        uint8_t *dataEnd;
} EtnBufferEncoder;
```

EtnNullEncoder is a specialization of *Encoder*. It is actually a quite useless structure since it contains only an *EtnEncoder*.

```
typedef struct EtnNullEncoder_s {
        EtnEncoder encoder;
} EtnNullEncoder;
```

PacketEncoder is specialization of *EtnEncoder*. It encodes into packet and sends it on flush. It contains a *EtnBufferEncoder*, a *Packet*, the total length of the packet, a *boolean* that indicates whether a packet was sent and a *Connection*.

```
typedef struct PacketEncoder_s {
        EtnBufferEncoder encoder;
        Packet          *packet;
        uint32_t         totalLength;
        bool             packetSent;
        Connection      *connection;
} PacketEncoder;
```

Decoder

EtnDecoder is a “reader” and can be considered a base class. This data structure contains a function pointer to a *read* function, a pointer to the data, the data length and an index to the next data.

```
typedef struct EtnDecoder_s {
        int (*read) (struct EtnDecoder_s *d, uint8_t *data, EtnLength length);
        void **indexToData;
        EtnLength indexToDataLength;
        EtnLength nextIndex;
} EtnDecoder;
```

The actual implementation of *read* is defined in file *decoder.c*.
EtnBufferDecoder is a specialization of *EtnDecoder*. It decodes to a provided memory range. In contains a *EtnDecoder* and two pointers to the memory range.

```
typedef struct EtnBufferDecoder_s {
        EtnDecoder decoder;
        uint8_t *dataCurrent;
        uint8_t *dataEnd;
} EtnBufferDecoder;
```

PathDecoder is a specialization of *EtnDecoder*. It decodes from a provided path. It contains a *EtnBufferDecoder* and a pointer to the original data.

```
typedef struct EtnPathDecoder_s {
        EtnBufferDecoder bufferDecoder;
        uint8_t *dataOriginal;
} EtnPathDecoder;
```

Types

ETN types are all defined in file *types.h*. As we can see, the encoders/decoders do not interact directly with the types. Indeed, the encoder/decoder type does not contain the type to be encoded/decoded. The interaction between them comes from other data structures that contain them all.
For instance, data structure *EtnRpcHost* (defined in *erpc.c* file and used for interprocess-communication):

```
struct EtnRpcHost {
        EtnValue v;
        EtnEncoder *e;
        EtnDecoder *d;
};
```

Here an example (from file *testPointer.c*) on how a value of type *pointer* is encoded and decoded (actions needed to encode/decode other types are similar).

```
static bool _predicate(void){
        uint8_t *in = malloc (sizeof (uint8_t)), *out;
        *in = 0xbe;

        uint8_t buf[1024];
        EtnBufferEncoder *e = etnBufferEncoderNew(buf, sizeof(buf));
        EtnLength encodedSize;
        etnEncode((EtnEncoder *) e, EtnToValue(&Uint8PtrType, &in), &
            encodedSize);

        EtnBufferDecoder *d = etnBufferDecoderNew(buf, encodedSize);
        etnDecode((EtnDecoder *) d, EtnToValue(&Uint8PtrType, &out));

        free (e);
        etnBufferDecoderFree (d);

        return *in == *out;
}
```

A pointer to a *uint8_t* is allocated and initialized at the value *0xbe* (*in*), while another one is declared (*out*). An *EtnBufferEncoder* is created and the function *etnEncode()* is called to encode the value of the *in* pointer, which is going to be written into the buffer. Then, an *EtnBufferDecoder* is created using the same buffer of *EtnBufferEncoder*. The function *etnDecode()* is used to decode the content of the buffer and the result is inside *out* pointer. Eventually, the test returns true if the content of the two pointers (*in* and *out*) is the same, false otherwise.

So, we can notice the only link between the *EtnEncoder* and *EtnDecoder* is the common buffer *buf*. The *macro EtnToValue* takes in input the *type* and the value and returns a variable of type *EtnValue* that contains both the pointer to the value (*void**) and the type (*EtnType*).