T.38 PER Encode/Decode API
User's Guide

Objective Systems, Inc. September 2002

Introduction

The T.38 PER Encode/Decode API is a package of functions for encoding and decoding messages from the T.38 ASN.1 specification using the Packed Encoding Rules (PER) as defined in ITU standard X.691. The T.38 specification describes a two-phase protocol – one phase for the primary and secondary IFP packets and a second phase for the UDPTL packet end packet.

This API has been developed in the C programming language. The Objective Systems ASN1C compiler was used to generate the initial structures and encode/decode functions. The source code was then edited to include only what was necessary to meet the requirements. The final package is comprised of minimal ANSI standard C code that should be portable to a wide-range of operating environments (including embedded).

Contents of the Package

The following diagram shows the directory tree structure that comprises the T.38 PER encode/decode package:

```
t38v5xx
 |
 +- build
 |
 +- doc
 |
 +- lib
 |
 +- src
 |
 +- sample
```

The top-level directory indicates this is T.38 version 5.xx (where xx would be replaced with a version number). Note that the version number corresponds to the version of ASN1C that was used to create it. It does not correspond to the version number of the T.38 standard that was used.

The purpose and contents of the various subdirectories are as follows:

- build – this directory contains the main makefile required to build the run-time library
- doc – this directory contains this document as well as the *ASN1C C/C++ User's Manual*
- lib – this directory contains the run-time library
- src – this directory contains all of the C source files (both compiler generated and low-level)
- specs – contains the ASN.1 specification source files used to build the generated code
- sample – this directory contains a writer and reader sample program to illustrate the encoding and decoding of a T.38 message

The base package was built using the GNU gcc compiler (version 3.2.2) on a Linux PC platform. The only changes required to build on different Linux/UNIX systems should be the editing of the *platform.mk* file in the top-level directory to contain the specification of compiler/linker options for the compiler to be used.

For Windows, the package should be ported over using FTP in ASCII mode to convert all source files from UNIX to DOS format. The *platform.w32* file in the top-level directory should then be renamed to *platform.mk* (the existing *platform.mk* file for GNU can be either renamed or deleted – it is not needed on the Windows platform). The *makefile* in the build subdirectory can then be executed using the Microsoft Visual C++ *nmake* utility program.

Getting Started

The package is delivered as a .tar.gz archive file that can be unpacked in any directory on the development system. All makefiles and internal sample programs use relative directory paths, so it is not necessary to create any type of top-level environment variables.

If you are using Linux as your base system with gcc 3.2.2, then the library files delivered with the package can be used. Otherwise, all files from the lib subdirectory should be deleted and the makefile in the build subdirectory executed to rebuild the library for the specific environment. Note that definitions in the platform.mk file may need to be edited to change compiler/linker settings.

After the library is built, it can be tested by building and executing the sample programs in the sample subdirectory. To build the sample programs, execute the makefile in the sample subdirectory. This should cause *writer* and *reader* executable files to be built. If you now execute *writer*, it should encode a sample record and write its contents to an output file (*message.dat*), and then provide a human-readable dump of the results to standard output.

The *reader* program can then be executed. This program will read the *message.dat* file, decode the data, and then print the contents of each filed in the data structures to standard output.

The following sections provide greater detail on the procedure of encoding and decoding T.38 packets. The sample writer and reader programs should be used as a guide when reading these sections.

Encoding T.38 Packet Structures

The T.38 ASN.1 specification specifies a two-phase protocol. In order to encode a message of this type, the following steps must be performed:

1. A primary IFP packet and, optionally, secondary IFP packets must be encoded, and

2. The results must be plugged into a UTPTL packet structure and then that is encoded to produce the finished message.

The user should use the writer program (*writer.c*) in the T.38 sample directory (*t38v5xx/sample*) as a guide when reading the rest of the procedure.

Encoding Primary and Secondary IFP Packets

To encode a primary IFP packet, a variable of the *IFPPacket* C structure must first be populated with data. This structure is as follows:

```
typedef struct EXTERN IFPPacket {
   struct {
      unsigned data_fieldPresent : 1;
   } m;
```

```
   Type_of_msg  type_of_msg;
   Data_Field  data_field;
} IFPPacket;
```

This structure has two fields: *type_of_msg* and *data_field*. The *type_of_msg* field is of type *Type_of_msg* and is defined as follows:

```
#define T_Type_of_msg_t30_indicator     1
#define T_Type_of_msg_t30_data          2

typedef struct EXTERN Type_of_msg {
   int t;
   union {
      /* t = 1 */
      Type_of_msg_t30_indicator  t30_indicator;
      /* t = 2 */
      Type_of_msg_t30_data  data;
   } u;
} Type_of_msg;
```

This is a choice of two alternatives: *t30_indicator* or *t30_data*. Setting the *t* field member of the generated structure specifies the choice alternative. The value *T_Type_of_msg_t30_indicator* specifies the T30 indicator option whereas *T_Type_of_msg_t30_data* specifies data. The data for the structure is set by filling in the union value *u* with data of the selected choice option. Both options are enumerated values that are represented by C enum value types.

The second field of the IFP packet structure is *data_field*. This is an optional field. To specify its presence, the *m.data_fieldPresent* bit must be set to one in the main structure. The *data_field* type is of type *Data_Field*. This type is used to represent an ASN.1 SEQUENCE OF construct as follows:

```
typedef struct EXTERN Data_Field {
   ASN1UINT n;
   Data_Field_element *elem;
} Data_Field;
```

In this structure, *n* is used to hold the number of elements in the data field array and the *elem* field is used to hold an array of the elements to be encoded.

The *Data_Field_element* type holds each of the array elements. This is defined as follows:

```
typedef struct EXTERN Data_Field_element {
   struct {
      unsigned field_dataPresent : 1;
   } m;
   Data_Field_element_field_type  field_type;
   ASN1DynOctStr  field_data;
} Data_Field_element;
```

This type is a structure containing the *Data_Field_field_type* enumerated type and the *field_data* octet string type. The *field_data* field is optional so if this is to be included, the *m.field_dataPresent* bit must be set.

The *IFPPacket* structure described above is used to represent both the primary and secondary packets that go into the final *UDPTLPacket* structure. Each packet must be encoded into a separate message buffer using different contexts. The general procedure to do this is as follows (see *writer.c* for details):

1. Declare an *ASN1CTXT* variable and initialize it using the *rtInitContext* function,

2. Populate the *IFPPacket* structure with the data to encoded as described above,

3. Use the *pu_setBuffer* function to specify the buffer the message is to be encoded into,

4. Invoke the *asn1PE_IFPPacket* function to encode the packet, and

5. Use the *pe_GetMsgPtr* function to get the start address and length of the encoded message component. These items will be used later to populate the UDPTL packet structure.

Encoding a UDPTL Packet

Once the primary and secondary IFP packets have been encoded, a UDPTL packet can be encoded. It is necessary to have complete encodings of the other packets before this point because these components are used to produce the encoded UDPTL packet.

The structure of the UDPTL packet is as follows:

```
typedef struct EXTERN UDPTLPacket {
   ASN1INT   seq_number;
   ASN1OpenType  primary_ifp_packet;
   UDPTLPacket_error_recovery  error_recovery;
} UDPTLPacket;
```

All of the secondary structures within this type are represented by *ASN.1 Open Types*. These types hold references to previously encoded IFP packet components. To populate the *primary_ifp_packet* field, the message start address and length from the encoded primary IFP packet obtained in the previous step must be used. A code snippet from the writer.c sample program showing how this is done is as follows:

```
udptlPacket.primary_ifp_packet.numocts = len;
udptlPacket.primary_ifp_packet.data = msgptr;
```

The *msgptr* and *len* variables were obtained from the encoded *IFPPacket* structure using the *pe_GetMsgPtr* function.

The secondary error recover packets are encoded in the same fashion. In this case, additional substructures exist for describing the packets, but the packets themselves are ASN.1 open types.

Once the packet structure is populated, the procedure to encode the UDPTL packet is as follows:

1. Declare an *ASN1CTXT* variable and initialize it using the *rtInitContext* function. This must a different context then those used to encode the IFP packets,

2. Use the *pu_setBuffer* function to specify the buffer the message is to be encoded into,

3. Invoke the *asn1PE_UDPTLPacket* function to encode the packet, and

4. Use the *pe_GetMsgPtr* function to get the start address and length of the final encoded message component.

Decoding T.38 Packet Structures

The T.38 ASN.1 specification specifies a two-phase protocol. In order to fully decode packets within these messages, two distinct steps are generally needed:

1. The main UDPTL message structure is decoded, and

2. The primary IFP packets and secondary IFP error record packets are decoded.

This is the inverse of the encoding procedure presented earlier. The user should use the reader program (*reader.c*) in the T.38 sample directory (*t38v5xx/sample*) as a guide when reading the rest of the procedure.

Decoding the UDPTL Packet

The procedure to decode the UDPTL packet structure is as follows:

1. Declare an *ASN1CTXT* variable and initialize it using the *rtInitContext* function,

2. Use the *pu_setBuffer* function to set the pointer and length of the buffer containing the UDPTL message structure to be decoded,

3. Invoke the *asn1PD_UDPTLPacket* function to decode the data,

4. Access the *UDPTLPacket* structure to get at the decoded data fields.

The C structures that are used to describe a UDPTL packet were shown in the section on encoding.

Decoding Primary and Secondary IFP Packets

The *UDPTLPacket* structure in its decoded form contains references to encoded primary and secondary IFP packet components. To access data within these components, further decoding must be done.

The following line within the *UTPTLPacket* SEQUENCE construct in the T.38 ASN.1 module specifies the primary IFP packet:

```
primary-ifp-packet TYPE-IDENTIFIER.&Type(IFPPacket)
```

This maps to an ASN.1 Open Type structure within the generated code for the UDPTL packet:

```
ASN1OpenType  primary_ifp_packet;
```

This structure describes an encoded IFP packet. To decode it, the following must be done:

1. An ASN.1 context structure (*ASN1CTXT*) is declared to hold the decoding parameters. This must be initialized using the *rtInitContext* function,

2. The *pu_setBuffer* function is then used to set the message buffer pointer and length to point at the open type fields:

```
msgptr = udptlPacket.primary_ifp_packet.data;
len = udptlPacket.primary_ifp_packet.numocts;
pu_setBuffer (&c2, msgbuf, len, aligned);
```

3. The *asn1PD_IFPPacket* function is then invoked to decode the IFP packet data.

The decoded results can then be accessed from the *IFPPacket* variable that was passed to the decode function.

Decoding the secondary IFP packets within the error CHOICE construct is done in a similar fashion. See the reader.c program for further details.