TAP3 ASN.1 C

Encode/Decode API

User's Guide

Objective Systems, Inc. April 2014

Introduction

The *Objective Systems TAP3 ASN.1 C Encode/Decode API* is a dynamically-linked library (DLL) for encoding and decoding messages from the GSM TAP3 and RAP ASN.1 specifications using the Basic Encoding Rules (BER) as defined in ITU standard X.690.

This API has been developed in the C programming language. The Objective Systems ASN1C compiler was used to generate the structures and encode/decode functions. The code was compiled using Visual Studio 2008 for Windows. Since the library is distributed as a DLL, it should be compatible with applications built with other versions of Visual Studio.

Contents of the Package

The following diagram shows the directory tree structure that comprises the API:

```
tap3dll_<version>
 |
 +- doc
 |
 +- debug
 |  +- lib
 |  +- src
 |
 +- release
 |  +- lib
 |  +- src
 |
 +- rtsrc
 |
 +- rtxsrc
 |
 +- rtbersrc
 |
 +- sample
```

Where <version> would be replaced with a 3-digit version number. For example, tap3dll_v100 would be version 1.0.0.

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

- debug\lib – Contains the debug version of the TAP3 DLL and supporting library.
  debug\src – Contains the debug version of ASN1C-generated TAP3 header files.

- release\lib – Contains the release version of the TAP3 DLL and supporting library.
  release\src – Contains the release version of ASN1C-generated TAP3 header files.

- rt*src – Contains the header files for the common run-time libraries.
- doc – Contains this document.
- sample – Contains sample programs that illustrate how to use the API.

Getting Started

The package is delivered as a zipped archive file (.zip) file that will allow installation to any directory on the target system. By default, the package is installed in c:\tap3dll. The sample programs use relative directory paths, so it is not necessary to create any type of top-level environment variables.

All of the necessary object files have been compiled and installed in the lib subdirectory. The code can be tested by executing the sample programs in the sample subdirectory. These sample programs consist of a reader and writer program. The writer program populates a data variable with some data, calls an encode function that writes the encoded byte stream to a file. The reader program sets up a file stream object to read this file, decodes the data into a C structure, and then prints the decoded results.

Sample Programs

The following sample programs are included in this package:

- rap
- tap3
- tap3batch

Sample programs for Windows can be built by using the Visual C++ *nmake* utility program to execute the provided makefile. The procedure is as follows:

1. Open a shell terminal window (or Visual Studio terminal window).
2. Change directory to one of the sample directories above. For example:

   ```
   cd c:\tap3dll\sample\tap3
   ```

3. Execute make or nmake:

   ```
   nmake
   ```

The result should be *writer* and *reader* programs. The writer should be executed first. It will encode a set of test data and write the record out to a *message.dat* file. The reader program can then be executed to read the encoded file and decode the contents.

Note that in order to execute a program that uses the DLL, the operating system must be able to find the file. Two ways this can be done are as follows:

1. Modify the PATH environment variable to include the PATH where the DLL is located. For example:

   set PATH=%PATH%;c:\tap3dll\debug\lib

   alternately, the DLL file can be copied into a directory already located in the PATH.

2. Copy the DLL file into same directory in which the executable file is located.

On Linux, the sample program can be built by using the GNU *make* utility program to execute the provided makefile. The procedure is as follows:

1. Open a shell terminal window

2. Change directory to one of the sample directories above. For example:

   ```
   cd tap3dll/sample/tap3
   ```

3. Execute make:

   ```
   make
   ```

The writer and reader programs can be executed as in the Windows case above.

Note that in order to execute a program that uses the shared object on Linux, the operating system must be able to find the shared object (.so) file. Two ways this can be done are as follows:

1. Create or modify the LD_LIBRARY_PATH environment variable to include the directory where the shared object file is located. For example:

   export LD_LIBRARY_PATH=$HOME/tap3dll/debug/lib

2. The shared object file may also be copied into a system directory that is already searched for shared object files (for example, the /usr/lib directory).

General Procedure to Encode to a TAP3 Stream

This API uses stream-based encoding which efficiently encodes data to a resource such as a file, memory buffer, or socket. The BER encoding logic uses indefinite lengths in the encoding of constructed items making it possible to drop in length markers before the overall length of the components is known.

The general procedure to encode a TAP3 message is as follows:

1. Initialize a context variable for BER stream encoding.

2. Create a stream writer within the context.
3. Declare a variable of the TAP3 data type you wish to encode and populate it with data.
4. Call the generated encode function associated with the data variable.

If successful, an encoded TAP3 message will have been written to the stream. A final step would be to free the context structure once use of it is complete.

A more detailed look at each of these steps is as follows. Note that you can refer to the *sample/tap3/writer.c* file within the installation to see actual code associated with these procedures.

Initialize Context for BER Stream Encoding

A variable of type OSCTXT must first be declared. This is a general handle variable used to keep track of all internal variables involved in the encoding process. In a multithreaded application, a separate context should be declared per thread to ensure to ensure no contention between resources.

The context may then be initialized using the *berStrmInitContext* function as follows:

```
if ((stat = berStrmInitContext (&ctxt)) != 0) {
   /* Error handling code would be application dependent */
   printf ("Initialization failed, status %d\n", stat);
   rtxErrPrint (&ctxt);
   return -1;
}
```

If successful, the context will be ready for BER stream-based message processing. Note that the error handling section would be application dependent. This simple example outputs error information to standard output including what is written by the *rtxErrPrint* function. Other error handling functions are available to, for example, fetch the error text to a string buffer. See the *ASN1C C/C++ Run-time Reference Manual* for further details on other error handling functions.

Create a Stream Writer

The next step is to create a stream writer. The *rtxStream<Type>CreateWriter* function would be used for this purpose. In this name, *<Type>* would be replaced with the type of stream being created. For example, in the tap3 sample program, a file stream is created using the following call:

```
stat = rtxStreamFileCreateWriter (&ctxt, filename);
if (stat != 0) {
   rtxErrPrint (&ctxt);
   return stat;
}
```

The *filename* argument would be used to specify the name of the file to which the encoded data would be written. It is also possible to create a memory stream to write the encoded data directly to memory, or a socket stream to write to a TCP/IP socket.

Declare and Populate a TAP3 Data Variable

This API supports multiple versions of the TAP3 standard, so all of the generated data types have a prefix of *TAP3_<Version>_* which is used to distinguish the different versions. For example,

*TAP3_0312_* would indicate TAP version 3 release 12.

The following two variable declarations are made in the tap3 sample program:

```
TAP_0311_DataInterChange dataInterChange;
TAP_0311_Notification notification;
```

In this case, Notification is a sub-element of the DataInterChange CHOICE type. It is possible to declare elements on the stack in this fashion and then tie them into the main type with statements such as the following:

```
dataInterChange.t = T_TAP_0311_DataInterChange_notification;
dataInterChange.u.notification = &notification;
```

A feature of the API is that memory management is very flexible. It is possible to populate in this way, or to use the run-time *rtxMem** functions to use dynamic memory for the variable such as in the following code:

```
dataInterChange.t = T_TAP_0311_DataInterChange_notification;
dataInterChange.u.notification = rtxMemAllocType (&ctxt, TAP_0311_Notification);
.. populate dataInterChange.u.notification directly
```

When *rtxMemFree* or *rtxMemReset* or *rtFreeContext* is called to free or reset memory, it will be able to identify which items within the structure have been allocated using ASN1C memory management and free/reset them and skip over elements declared either directly on the stack or allocated using standard memory management (malloc). Note that in the latter case, it will up to the user to explicitly free these items themselves.

Call the Generated Encode Function

The encode function associated with the data type would then be invoked to encode the data:

```
stat = asn1BSE_TAP_0311_DataInterChange (&ctxt, &dataInterChange, ASN1EXPL);

rtxStreamClose (&ctxt);

if (stat != 0) {
   rtxErrPrint (&ctxt);
   berStrmFreeContext (&ctxt);
   return stat;
}
```

In most cases, the function associated with the *DataInterChange* type will be invoked as this is the main PDU type in the TAP3 standard. However, it is possible to populate and encode variables of other types as well.

The arguments to this function are as follows:

1. Context (&ctxt) – The context structure that was initialized to describe the stream.
2. Data variable (&dataInterChange) – Pointer to variable containing the data to be encoded.
3. Tagging (ASN1EXPL) – Tagging variable used to describe tagging (EXPLICIT or IMPLICIT) of items within the message. This should always be set to ASN1EXPL at the outer level.

The encode function will encode the data in BER format in a streaming fashion to the output medium. The *rtxStreamClose* or *rtxStreamFlush* function may be invoked after the encode function to ensure all data is flushed to the output stream.

The status of the encode operation can then be checked to determine if encoding was successful or not.

If successful, the encoded data will now be available on the streamed output entity. The sample program reads the file into memory and does a dump of the BER-encoded data using the XU_DUMP function.

General Procedure to Decode from a TAP3 Stream

The general procedure to decode from a TAP3 stream is as follows:

1. Initialize a context variable for BER stream decoding.
2. Create a stream reader within the context.
3. Determine TAP3 release version number for data type selection (optional).
4. Declare a variable of the TAP3 data type into which you wish to decode the stream content.
5. Call the generated BER stream decode function associated with the data variable.

If successful, the decoded TAP3 content will have been written to the data variable. A final step would be to free the context structure once use of it is complete.

A more detailed look at each of these steps is as follows. Note that you can refer to the *sample/tap3/reader.c* file within the installation to see actual code associated with these procedures. This sample reads from a TAP3 file.

Initialize Context for BER Stream Decoding

The procedure to initialize the context for stream decoding would be identical to that which was used for encoding described in the previous section.

Create a Stream Reader

The next step is to create a stream reader. The *rtxStream<Type>CreateReader* function would be used for this purpose. In this name, *<Type>* would be replaced with the type of stream being created. For example, in the tap3 sample program, a file stream is created using the following call:

```
stat = rtxStreamFileCreateReader (&ctxt, filename);
if (stat != 0) {
   rtxErrPrint (&ctxt);
   return stat;
}
```

The *filename* argument would be used to specify the name of the file from which the decoded data would be read. It is also possible to create a memory stream to read the encoded data directly from memory, or a socket stream to read from a TCP/IP socket interface.

Determine TAP3 Release Version Number

A feature of this API is that it supports multiple versions of the TAP3 standard. It does this by using generated code for multiple versions of the TAP3 ASN.1 specifications. If an application is decoding data from sources that may be of different versions, the version number of the encoded data must first be determined in order to select the appropriate version of the data type and decode function to be invoked to decode the data.

The special helper function *tap3GetReleaseVersionNumber* has been developed for this purpose. This takes advantage of the fact that all items within the TAP3 ASN.1 specifications have unique tag numbers. It is able to scan through the data to find the tag and return the release version number.

Once known, a switch statement can be set up to do separate processing based on version. This is code from the tap3 sample program that does this:

```
stat = tap3GetReleaseVersionNumber (&ctxt);

if (stat < 0) {
   printf ("tap3GetReleaseVersionNumber failed\n");
   rtxErrPrint (&ctxt);
   return stat;
}
else relVersion = stat;

switch (relVersion) {
case 9: {
   TAP_0309_DataInterChange dataInterChange;
   stat = asn1BSD_TAP_0309_DataInterChange
      (&ctxt, &dataInterChange, ASN1EXPL, 0);

   if (stat == 0) {
      if (trace) {
         printf ("Decode of DataInterChange was successful\n");
         asn1Print_TAP_0309_DataInterChange
            ("DataInterChange", &dataInterChange);
      }
   }
   break;
}

case 10: {
   ...
```

In this code, the release version number is first determined and then a switch/case statement is used to segregate the processing for each version into its own block. While somewhat redundant, it is the safest way to ensure structure integrity for different versions. Later versions of TAP3 attempted to provide greater backward compatibility with previous versions by using best practices suchs as the use of optional elements for new items; therefore it may be possible to decode messages based on older versions of the standard with the latest version.

If this is to be attempted, or if the version of the TAP3 messages to be dealt with are known, this step may be skipped. Also note the use of this method requires the capability to move to a position in the stream and reset to the previous position – something not supported in all stream types. In particular, this will not work in a socket stream.

Call the Generated Decode Function

The decode function associated with the data type would then be invoked to decode the data:

```
stat = asn1BSD_TAP_0311_DataInterChange (&ctxt, &dataInterChange, ASN1EXPL, 0);

if (stat != 0) {
   rtxErrPrint (&ctxt);
   berStrmFreeContext (&ctxt);
   return stat;
}
```

In most cases, the function associated with the *DataInterChange* type will be invoked as this is the main PDU type in the TAP3 standard. However, it is possible to populate and decode variables of other types as well.

The arguments to this function are as follows:

1. Context (&ctxt) – The context structure that was initialized to describe the stream.
2. Data variable (&dataInterChange) – Pointer to target variable to which data will be decoded.
3. Tagging (ASN1EXPL) – Tagging variable used to describe tagging (EXPLICIT or IMPLICIT) of items within the message. This should always be set to ASN1EXPL at the outer level.
4. Length (0) – This argument only has meaning if tagging is set to IMPLICIT (ASN1IMPL). It indicates the length of the component being decoded. This should always be set to zero at the outer level since tagging is set to EXPLICIT (ASN1EXPL).

If successful, the data will now be available in the data variable where it can be manipulated.

Once use of the decoded data is complete, the user should call *rtxMemFree* or *rtxMemReset* to free the memory associated with the data prior to decoding another record. Memory will also be freed when the context is closed via the *berStrmFreeContext* call.