C Encode/Decode API for 3GPP Non-Access Stratum (NAS) and Radio Resource Control (RRC)

Add-on Kit for ASN1C

Table of Contents

Introduction

The NAS C Encode/Decode API supports encoding and decoding 3GPP Layer 3 messages. These messages are described in the following 3GPP technical specifications:

- TS 24.007 - Mobile radio interface signaling layer 3; General Aspects
- TS 24.008 - Mobile radio interface Layer 3 specification; Core network protocols; Stage 3
- TS 24.011 - Point-to-Point (PP) Short Message Service (SMS) support on mobile radio interface
- TS 24.301 - Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3
- TS 44.060 - Mobile Station (MS) - Base Station System (BSS) interface; Radio Link Control / Medium Access Control (RLC/MAC) protocol
- TS 44.018 - Mobile radio interface layer 3 specification; Radio Resource Control (RRC) protocol

This document explains how to build and use this API using the add-on package available for the ASN1C Compiler.

Methodology

This API has been developed in the C programming language, using Objective Systems' ASN1C compiler to generate the structures and encode/decode functions. The code was generated from a mixture of ASN.1 and CSN.1 specifications. In cases where the relevant technical specifications did not use either ASN.1 or CSN.1, we approximated the given message or information element using one of these notations. Since this was an approximation, it was also necessary to use a combination of configuration directives and custom code to achieve the desired results. The configuration directives are made effective by using the '-3gl3' command-line option with ASN1C. Our white paper, "Using ASN.1 to Describe 3GPP Messages" [http://www.obj-sys.com/docs/UsingASNtoDescribe3GPPMessages.pdf], describes how messages were approximated using ASN.1. (Note that this paper does not reflect our more recently capability to directly compile CSN.1 notation.)

The end result is an API that consisted of C types and structures similar to what a user would get by compiling a standard ASN.1 specification. The other benefit of this approach is that in addition to encode/decode functions, supporting functions such as print, copy, compare, etc. can be generated from the definitions.

Note that one or more ASN.1 modules are associated with each of the above 3GPP specifications. In some of these modules, PDU (“Protocol Definition Unit”) types are defined. These PDU types are used to represent an entire group of (or possibly all) messages from that module. Common fields are also factored out into the PDU type.

The prefixes for each of the types in the API are given in the following table:

Table 1. Module Prefixes

TS24008IE	prefix for the information elements defined in 24.008
TS24008Msg	prefix for the messages defined in 24.008
TS24011IE	prefix for the information elements defined in 24.011
TS24011Msg	prefix for the messages defined in 24.011
TS24301IE	prefix for the information elements defined in 24.301
TS24301Msg	prefix for the messages defined in 24.301
TS44018IE	prefix for the information elements defined in 44.018

TS44018Msg	prefix for the messages defined in 44.018
TS44060IE	prefix for the information elements defined in 44.060

The PDU types we have defined are given in the following table:

Table 2. PDUs

TS24008Msg_PDU	PDU for the messages defined in 24.008
TS24011Msg_CP_PDU	PDU for the CP messages defined in 24.011
TS24011Msg_RP_PDU	PDU for the RP messages defined in 24.011
TS24301Msg_PDU	PDU for the messages defined in 24.301
TS44018Msg_L2_PSEUDO_LEN_PDU	PDU for 44.018 RRC messages using an L2 Pseudo Length
TS44018Msg_RR_Short_PD_PDU	PDU for 44.018 RRC messages using a short protocol discriminator
TS44018Msg_DL_DCCH_PDU	PDU for 44.018 RRC standard L3 messages transmitted on the main DCCH from network to MS (downlink)
TS44018Msg_UL_DCCH_PDU	PDU for 44.018 RRC standard L3 messages transmitted on the main DCCH from MS to network (uplink)
TS44018Msg_UL_SACCH_PDU	PDU for 44.018 RRC standard L3 messages transmitted on the SACCH from MS to network (uplink)

Note that for 44.018 RRC messages, you must choose the PDU type according to the message format and (in some cases) the channel and message direction.

Contents of the Package

The following diagram shows the directory tree structure that comprises the add-on:

```
nas_dev
|
+- custsrc
+- doc
|
+- debug
|  +- build
|  +- lib
|
+- release
|  +- build
|  +- lib
|
+- src
+- src2
+- test_ts24008_ies
+- test_ts24008_msgs
+- test_ts24301_msgs
+- test_ts44018_ies
+- test_ts44018_msgs
```

```
+- util
```

This is installed under the 'c' subdirectory of the ASN1C installation.

In addition, 3GPP NAS and RRC ASN.1 and CSN.1 specifications are added to the specs subdirectory in folder 3GPP-NAS.

Building the API

After installation of the ASN1C 3GPP add-on package is complete, a new directory named 'nas_dev' will have been formed under the 'c' subdirectory. This contains all of the development files needed to build the API.

To build on Windows using the default version of Visual Studio for the version of ASN1C where you installed the NAS SDK, the following would be done:

1. Open the file osys3gpp_a.vcxproj with the Visual Studio IDE. This project file is a Visual Studio 2010 project. Using the IDE retarget the project to the IDE's version.

2. Build the project from within the IDE.

That should be all that is necessary. It will cause the ASN1C compiler to be invoked to generate C source code for all of related ASN.1 and CSN.1 specifications. It will then compile the resulting files with Visual Studio C/C++ compiler. Resulting libraries in both debug and release configurations may be found in the debug\lib and release\lib subdirectories respectively.

If you wish to build the API with a different version of Visual Studio instead of the default version, all that needs to be done is moving the nas_dev folder from under the 'c' subdirectory to a different 'c_' subdirectory. So, for example, if you wanted to build with Visual Studio 2012, you would move the nas_dev folder under c_vs2012 and execute the procedure above. If the version of Visual Studio is newer than Visual Studio 2010, retarget the project file to the newer version.

To build on Linux, the procedure would be similar:

1. From a terminal window, change directory (cd) to the c/nas_dev subdirectory within the installation.

2. Execute the top-level makefile using the 'make' command:

```
make
```

Getting Started

The easiest way to get started is by examining the test_* subdirectories within the package. These contains test programs for encoding and decoding all of the different message types defined in the standards. Typically, code from within these samples can be used to form larger programs that can encode or decode larger message sets.

Sample Programs

Numerous sample programs are included in this package.

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 Visual Studio terminal window.

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

```
cd c:\<acroot>\c\nas_dev\test_ts24008_msgs\Abort
```

3. Execute nmake:

```
nmake
```

The procedure would be similar on Linux except that the make utility would be used.

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 (via reader -v) to read the encoded file and decode the contents.

Encoding Messages

As mentioned above, the ASN.1 modules define one or more PDU ("Protocol Definition Unit") types to encompass all, or a subset of, the messages for the corresponding specification. Generally, the procedure for encoding is the same for all PDU types. However, `TS24301Msg_PDU` does have some additional requirements related to NAS security.

Encoding Messages Other Than TS24301Msg_PDU

We'll use the PDU for TS 24.008 as our example. File TS24008Msgs.asn contains a type defined as follows:

```
PDU ::= SEQUENCE {
    -- L3 header, octet 1, bits 5 thru 8
    l3HdrOpts CHOICE {
        skipInd INTEGER(0..15),
        transId SEQUENCE {
            flag BOOLEAN,
            value INTEGER(0..255)
        }
    },
    protoDiscr   NAS-PROTOCOL-CLASS.&protoDiscr ({TS24008-IE-OBJECTSET}),
    sendSeqNum   INTEGER (0..3) OPTIONAL,
    msgType      NAS-PROTOCOL-CLASS.&msgType ({TS24008-IE-OBJECTSET}),
    data         NAS-PROTOCOL-CLASS.&Value
                    ({TS24008-IE-OBJECTSET}{@protoDiscr,@msgType})
}
```

The first 4 elements within this definition (`l3HdrOpts`, `protoDiscr`, `sendSeqNum`, and `msgType`) describe header fields as defined in the TS 24.007 and TS 24.008 documents. The final data field is a variable type field that defines the contents for all of the different message types. The combination of protocol discriminator and message type serve to specify the message type.

The general procedure to encode a message of this type is as follows:

1. Declare variables of the generated PDU type (e.g. `TS24008Msg_PDU`) and the specific message type to be sent (e.g. `TS24008Msg_IdentityRequest`).

2. Populate the types. The address of the specific message structure would be stored within the PDU union structure. The generated `asn1SetTC_*` (set table constraint) functions can be used to set fixed value fields (protocol discriminator and message type) and the pointer to the message data in one call.

3. Initialize the context structure and set the context buffer pointer.

4. Call the PDU encode function

5. Get the message pointer and length to work with the binary message.

Before a NAS encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling the `rtInitContext` function.

Only memory-buffer based encoding is supported because the message sizes are generally small (normally less than 256 bytes).

To do memory-based encoding, the `rtxInitContextBuffer` function would be called. This can be used to specify use of a static or dynamic memory buffer. Specification of a dynamic buffer is possible by setting the buffer address argument to null and the buffer size argument to zero.

The PDU encode function can then be called to encode the message. If the return status indicates success (0), then the message will have been encoded in the given buffer. The length of the encoded message can be obtained by calling the `rtxCtxtGetMsgLen` run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the `rtxCtxtGetMsgPtr` run-time function can be used to obtain the start address of the message. This routine will also return the length of the encoded message.

A program fragment that could be used to encode a 3G NAS Identity Request message is as follows:

```
#include "rt3gppsrc/TS24008Msgs.h" /* include file generated by ASN1C */

main ()
{
    TS24008Msg_PDU pdu;
    TS24008Msg_IdentityRequest idReq;
    OSCTXT ctxt;
    OSOCTET msgbuf[256], *msgptr;
    int i, len, stat;
    const char* filename = "message.dat";

    /* Initialize context structure */

    stat = rtInitContext (&ctxt);
    if (0 != stat) {
        printf ("rtInitContext failed; status = %d\n", ret);
        rtxErrPrint (&ctxt);
        return ret;
    }

    /* Populate C structure */

    pdu.l3HdrOpts.u.skipInd = 0;
    asn1SetTC_TS24008Msg_PDU_obj_IdentityRequest (&ctxt, &pdu, &idReq);

    OSCRTLMEMSET (&idReq, 0, sizeof(idReq));
    idReq.value.typeOfIdent = TS24008IE_IdentityTypeValue_typeOfIdent_imei;

    /* Encode */

    rtxCtxtSetBufPtr (&ctxt, msgbuf, sizeof(msgbuf));
```

```
    stat = NASEnc_TS24008Msg_PDU (&ctxt, &pdu);
    if (0 != stat) {
        printf ("encode PDU failed; status = %d\n", ret);
        rtxErrPrint (&ctxt);
        return ret;
    }

    msgptr = rtxCtxtGetMsgPtr (&ctxt);
    len = rtxCtxtGetMsgLen (&ctxt);

    ...
}
```

Encoding TS24301Msg_PDU

The procedure for encoding `TS24301Msg_PDU` is basically the same as given above. When the message is to be encoded with security protection, however, there are a few additional requirements. These steps are *not* necessary when encoding a message *without* security protection.

For security protected messages, including `SERVICE REQUEST` messages (which always have integrity protection):

1. Initialize the NAS security context
2. Specify the algorithm and keys to use
3. Set NAS security parameters
4. Assign security header fields in the PDU.
5. Free the NAS security context when finished.

The following code illustrates these steps. It encodes an `ACTIVATE DEDICATED EPS BEARER CONTEXT ACCEPT` message with both integrity and confidentiality protection.

```
#include "TS24301Msgs.h"
#include "rt3gppsrc/rt3gppNasSec.h"

int main (int argc, char** argv)
{
    TS24301Msg_PDU pdu;
    TS24301Msg_ActvDedEPSBearerCtxtAcc data;
    TS24301Msg_ActvDedEPSBearerCtxtAcc* pvalue = &data;
    OSCTXT ctxt;
    OSOCTET msgbuf[256], *msgptr;
    OSSIZE       len;
    int  i, ret;
    OSCrypt128Key integKey = { ... };
    OSCrypt128Key encryptKey = { ... };


    /* Initialize context structure */

    ret = rtInitContext (&ctxt);
    if (0 != ret) {
```

```
    printf ("rtInitContext failed; status = %d\n", ret);
    rtxErrPrint (&ctxt);
    return ret;
}

/* Initialize the NAS Security context */
ret = rtx3gppSecInitialize(&ctxt);
if (0 != ret) {
    rtxErrPrint (&ctxt);
}


/* Specify the integrity and encryption algorithms and keys.
This only needs to be done the first time or when these values change
*/
ret = rtx3gppAssignAlgorithmKeys(&ctxt, &integKey, &encryptKey,
OS3GPPSecAlgorithm_AES, OS3GPPSecAlgorithm_AES);
if (0 != ret) {
    rtxErrPrint (&ctxt);
}

/* Assign count, bearerId, direction in the NAS security context.
These act as input into the security algorithms.
*/
ctxt.p3gppSec->count = 15;
ctxt.p3gppSec->bearerId = 0;
ctxt.p3gppSec->direction = 0; /*uplink*/

/* Populate data structure */

asn1Init_TS24301Msg_PDU (&pdu);

/* Specify the security header is present. Populate the security header
fields. The security header's protocol discriminator is always EPS
Mobility management. The msgAuthCode field will be filled in by function
NASEnc_TS24301Msg_PDU.
*/
pdu.m.secHdrPresent = 1;
pdu.secHdr.secHdrType = TS24007L3_SecHdrType_integProtAndCipher;
pdu.secHdr.protoDiscr = TS24007L3_ProtoDiscr_epsMobMgmt;
pdu.secHdr.seqNumber  = 1;

/* The skipIndicator is used for the message's ESP bearer identity;
it is always zero. The PDU's protocol discriminator (unlike the security
header's protocol discriminator above) is set according to the message
being protected.
*/
pdu.l3HdrOpts.t = T_TS24007L3_L3HdrOptions_skipInd;
pdu.l3HdrOpts.u.skipInd = 0;
pdu.protoDiscr = TS24007L3_ProtoDiscr_epsSessMgmt;
pdu.msgType = ASN1V_ts24301Msg_mt_ActvDedEPSBearerCtxtAcc;

...
```

```
    /* Encode data.
    NASEnc_TS24301Msg_PDU will encrypt the message and compute the message
    authentication code according to the security settings chosen above.
    */

    rtxCtxtSetBufPtr (&ctxt, msgbuf, sizeof(msgbuf));

    ret = NASEnc_TS24301Msg_PDU (&ctxt, &pdu);
    if (0 != ret) {
        printf ("encode PDU failed; status = %d\n", ret);
        rtxErrPrint (&ctxt);
        return ret;
    }

    ...

    /* Free the NAS security context */

    rtx3gppSecFree(&ctxt);

    ...
}
```

When the message being security protected is a `SERVICE REQUEST` message, the above procedure is basically the same, but a few differences do exist:

- `TS24301Msg_ServiceRequest` has its own `secHdr` field that is simply the security header type. This means there are *two* fields (this one and `TS24301Msg_PDU.secHdr.secHdrType`) that have the security header type; *both* should be set to `TS24007L3_SecHdrType_secHdrForSvcReq`.
- `TS24301Msg_ServiceRequest` has its own MAC field, `shortMAC`. This does not need to be set. It will be assigned by `NASEnc_TS24301Msg_PDU` (`TS24301Msg_PDU.secHdr.msgAuthCode` will not be assigned).

Decoding Messages

As mentioned above, the ASN.1 modules define one or more PDU ("Protocol Definition Unit") types to encompass all, or a subset of, the messages for the corresponding standard. Generally, the procedure for decoding is the same for all PDU types. However, `TS24301Msg_PDU` does have some additional requirements related to NAS security.

Decoding Messages Other Than TS24301Msg_PDU

The following are the basic steps to decode with a PDU decode function:

1. Prepare a context variable for decoding
2. Initialize the data structure to receive the decoded data
3. Call the desired PDU decode function to decode the message
4. Free the context after use of the decoded data is complete to free allocated memory structures

Before a NAS decode function can be called, the user must first initialize a context block structure. The context block is initialized by calling the `rtInitContext` function.

Only memory-buffer based encoding is supported for 3GPP layer 3 because the message sizes are generally small (normally less than 256 bytes).

To do memory-based decoding, the `rtxInitContextBuffer` function would be called. The message to be decoded must reside in memory. The arguments to this function would then specify the message buffer in which the data to be decoded exists.

The PDU variable that is to receive the decoded data must then be initialized. This can be done by either initializing the variable to zero using memset, or by calling the ASN1C generated initialization function.

The PDU decode function can then be called to decode the message. If the return status indicates success (0), then the message will have been decoded into the PDU type variable. The decode function may automatically allocate dynamic memory to hold variable length variables during the course of decoding. This memory will be tracked in the context structure, so the programmer does not need to worry about freeing it. It will be released when the either the context is freed or explicitly when the `rtxMemFree` or `rtxMemReset` function is called.

The final step of the procedure is to free the context block. This must be done regardless of whether the block is static (declared on the stack and initialized using `rtInitContext`), or dynamic (created using `rtNewContext`). The function to free the context is `rtFreeContext`.

A program fragment that could be used to decode a 3G NAS PDU is as follows:

```
#include "rt3gppsrc/TS24008Msgs.h" /* include file generated by ASN1C */

main ()
{
    TS24008Msg_PDU data;
    OSCTXT ctxt;
    OSOCTET* msgbuf;
    const char* filename = "message.dat";
    int stat;
    OSSIZE len;

    /* step 1: initialize context */

    stat = rtInitContext (&ctxt);

    if (stat != 0) {
        printf (“rtInitContext failed (check license)\n“);
        rtErrPrint (&ctxt);
        return stat;
    }

    /* step 2: read input file into a memory buffer */

    stat = rtxFileReadBinary (&ctxt, filename, &pMsgBuf, &len);
    if (0 == stat) {
        stat = rtxInitContextBuffer (&ctxt, pMsgBuf, len);
    }
    if (0 != stat) {
        rtxErrPrint (&ctxt);
        rtFreeContext (&ctxt);
        return stat;
    }
```

```
    /* step 3: set protocol version number */

    rtxCtxtSetProtocolVersion (&ctxt, 8);

    /* step 4: call the decode function */

    stat = NASDec_TS24008Msg_PDU (&ctxt, &data);

    if (stat == 0)
    {
        process received data..
    }
    else {
        /* error processing... */
        rtxErrPrint (&ctxt);
    }

    /* step 5: free the context */
    rtFreeContext (&ctxt);
}
```

Decoding TS24301Msg_PDU

The procedure for decoding `TS24301Msg_PDU` is basically the same as given above. When the encoded message is security protected, however, there are a few additional requirements. These steps are *not* necessary when decoding a message *without* security protection (but, unless you are certain the message is not security protected, you will need to follow these requirements).

For security protected messages, including `SERVICE REQUEST` messages (which always have integrity protection):

1. Initialize the NAS security context

2. Specify the algorithm and keys to use

3. Set NAS security parameters

4. Free the NAS security context when finished.

Each of these steps is illustrated in the encoding example above. The `NASDec_TS24301Msg_PDU` function will:

- verify the message authentication code of an integrity protected message. The MAC (or short MAC) will be decoded into either `TS24301Msg_PDU.secHdr.msgAuthCode` or `TS24301Msg_ServiceRequest.shortMAC`. If MAC verification fails, `RTERR_INVMAC` is returned.

- decrypt and further decode a confidentiality protected message

NAS Security

Functions `NASDec_TS24301Msg_PDU` and `NASDec_TS24301Msg_PDU` support security protected messages as described in TS 24.301. This section provides some important details about this support.

First, TS 24.301 specifies multiple security algorithms that can be used for security protection. This API supports two of them: the null algorithm and the AES-based algorithms. The algorithms based on SNOW3G and ZUC are not currently supported.

Finally, by design, you may provide your own implementation of the security functions exported by the asn1rt3gpp library. This would be useful in any of the following situations:

- You want to use a different implementation of the AES algorithms than we have provided.
- You want to implement the algorithms we have not implemented (SNOW3G, ZUC).

If you choose to provide your own implementation, the functions to implement (declared and documented in `rt3gppNasSec.h`) are:

- `rtx3gppAssignAlgorithmKeys`
- `rtx3gppCipher`
- `rtx3gppComputeMAC`

Working with GERAN RRC Messages

As noted in the Methodology section, there are several PDU types defined for GERAN RRC messages (3GPP TS 44.018). You must choose the correct PDU type to use based on the message format, message channel, and message direction. Having chosen the correct PDU type, you proceed with encoding/decoding in the same way as described in the sections on encoding and decoding messages "other than TS24301Msg_PDU". The following paragraphs give an overview of the PDU types for RRC and a few useful tips.

RR Short Protocol Discriminator

For messages that use the RR Short Protocol Discriminator format, use the following PDU:

```
RR-Short-PD-PDU ::= SEQUENCE {
    rr-short-PD INTEGER (0..1) DEFAULT 0,
    msgType RR-SHORT-CLASS.&msgType ({RR-Short-PD-Messages}),
    short-layer-2-header INTEGER (0..3) DEFAULT 0,
    data RR-SHORT-CLASS.&Value ({RR-Short-PD-Messages}{@msgType})
}
```

The `msgType` field determines the type of the value in the `data` field.

L2 Pseudo Length

For messges using the L2 pseudo length format, use the following PDU:

```
L2-PSEUDO-LEN-PDU ::= SEQUENCE {
    l2PseudoLen L2-Pseudo-Length,
    l3HdrOpts  L3HdrOptions, -- L3 header, octet 1, bits 5 thru 8
    protoDiscr L2-PSEUDO-LEN-CLASS.&protoDiscr ({L2-PSEUDO-LEN-PDU-Messages}),
    msgType L2-PSEUDO-LEN-CLASS.&msgType ({L2-PSEUDO-LEN-PDU-Messages}),
    data    L2-PSEUDO-LEN-CLASS.&Value ({L2-PSEUDO-LEN-PDU-Messages}{@protoDiscr,@msgType}
    restOctets  L2-PSEUDO-LEN-CLASS.&RestOctets ({L2-PSEUDO-LEN-PDU-Messages}{@protoDiscr,
}
```

Fields `protoDiscr` and `msgType` determine the types of the values in `data` and `restOctets`.

Standard L3 Formatted messages

For messages using the standard L3 format, you must choose the PDU type based on the channel and/or direction. There are three PDU types, all basically the same except that they have different sets of messages allowed. In each case, the fields `protoDiscr` and `msgType` determine the type of the value in `data`.

For downlink messages on the main DCCH:

```
DL-DCCH-PDU ::= SEQUENCE {
    l3HdrOpts  L3HdrOptions, -- L3 header, octet 1, bits 5 thru 8
    protoDiscr NAS-PROTOCOL-CLASS.&protoDiscr ({DL-DCCH-Messages}),
    msgType NAS-PROTOCOL-CLASS.&msgType ({DL-DCCH-Messages}),
    data    NAS-PROTOCOL-CLASS.&Value ({DL-DCCH-Messages}{@protoDiscr,@msgType})
    }
```

For uplink messages on the main DCCH:

```
UL-DCCH-PDU ::= SEQUENCE {
  l3HdrOpts  L3HdrOptions, -- L3 header, octet 1, bits 5 thru 8
  protoDiscr NAS-PROTOCOL-CLASS.&protoDiscr ({UL-DCCH-Messages}),
  msgType NAS-PROTOCOL-CLASS.&msgType ({UL-DCCH-Messages}),
  data    NAS-PROTOCOL-CLASS.&Value ({UL-DCCH-Messages}{@protoDiscr,@msgType})
}
```

For uplink messages on the SACCH:

```
UL-SACCH-PDU ::= SEQUENCE {
    l3HdrOpts  L3HdrOptions, -- L3 header, octet 1, bits 5 thru 8
    protoDiscr NAS-PROTOCOL-CLASS.&protoDiscr ({UL-SACCH-Messages}),
    msgType NAS-PROTOCOL-CLASS.&msgType ({UL-SACCH-Messages}),
    data    NAS-PROTOCOL-CLASS.&Value ({UL-SACCH-Messages}{@protoDiscr,@msgType})
}
```

Tips

- All of the supported RRC messages have example code in `test_ts44018_msgs`
- There are a few messages that consist of nothing more than a single octet value (e.g. "Channel Request"). We haven't defined a type for these messages.
- There are two messages that don't use any of the above formats and don't carry a message type. They are "Synchronization Channel Information" and "COMPACT Synchronization Channel Information". For these, there is no PDU type; you only need the message types to work with: `TS44018Msg_SynchChannelInfo` and `TS44018Msg_CompactSynchChannelInfo`.
- `src2/TS44018Misc.h` declares several helper functions for working with ARFCN lists.