Using the Open Source ASN.1 Compiler

Documentation for asn1c version 0.9.29

Lev Walkin <vlm@lionet.info>

November 11, 2017

Contents

Chapter 1

Quick start examples

1.1 A "Rectangle" converter and debugger

One of the most common need is to create some sort of analysis tool for the existing ASN.1 data files. Let's build a converter for existing Rectangle binary files between BER, OER, PER, and XER (XML).

1. Create a file named **rectangle.asn** with the following contents:

   ```
   RectangleModule DEFINITIONS ::= BEGIN

   Rectangle ::= SEQUENCE {
       height  INTEGER,
       width   INTEGER
   }

   END
   ```

2. Compile it into the set of .c and .h files using `asn1c` compiler:

   ```
   asn1c -no-gen-example rectangle.asn
   ```

3. Create the converter and dumper:

   ```
   make -f Makefile.am.example
   ```

4. Done. The binary file converter is ready:

   ```
   ./converter-example -h
   ```

1.2 A "Rectangle" Encoder

This example will help you create a simple BER and XER encoder of a "Rectangle" type used throughout this document.

1. Create a file named **rectangle.asn** with the following contents:

```
RectangleModule DEFINITIONS ::= BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER,
    width   INTEGER
}

END
```

2. Compile it into the set of .c and .h files using asn1c compiler [ASN1C]:

```
asn1c −no−gen−example rectangle.asn
```

3. Alternatively, use the Online ASN.1 compiler [AONL] by uploading the **rectangle.asn** file into the Web form and unpacking the produced archive on your computer.

4. By this time, you should have gotten multiple files in the current directory, including the **Rectangle.c** and **Rectangle.h**.

5. Create a main() routine which creates the Rectangle_t structure in memory and encodes it using BER and XER encoding rules. Let's name the file **main.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <Rectangle.h>   /* Rectangle ASN.1 type */

/* Write the encoded output into some FILE stream. */
static int write_out(const void *buffer, size_t size, void *app_key) {
    FILE *out_fp = app_key;
    size_t wrote = fwrite(buffer, 1, size, out_fp);
    return (wrote == size) ? 0 : -1;
}

int main(int ac, char **av) {
    Rectangle_t *rectangle; /* Type to encode        */
    asn_enc_rval_t ec;      /* Encoder return value  */

    /* Allocate the Rectangle_t */
    rectangle = calloc(1, sizeof(Rectangle_t)); /* not malloc! */
```

```
    if(!rectangle) {
        perror("calloc() failed");
        exit(1);
    }

    /* Initialize the Rectangle members */
    rectangle->height = 42;  /* any random value */
    rectangle->width  = 23;  /* any random value */

    /* BER encode the data if filename is given */
    if(ac < 2) {
        fprintf(stderr, "Specify filename for BER output\n");
    } else {
        const char *filename = av[1];
        FILE *fp = fopen(filename, "wb");   /* for BER output */

        if(!fp) {
          perror(filename);
          exit(1);
        }

        /* Encode the Rectangle type as BER (DER) */
        ec = der_encode(&asn_DEF_Rectangle, rectangle, write_out, fp);
        fclose(fp);
        if(ec.encoded == -1) {
          fprintf(stderr, "Could not encode Rectangle (at %s)\n",
              ec.failed_type ? ec.failed_type->name : "unknown");
          exit(1);
        } else {
          fprintf(stderr, "Created %s with BER encoded Rectangle\n", filename);
        }
    }

    /* Also print the constructed Rectangle XER encoded (XML) */
    xer_fprint(stdout, &asn_DEF_Rectangle, rectangle);

    return 0; /* Encoding finished successfully */
}
```

6. Compile all files together using C compiler (varies by platform):

   ```
   cc -I. -o rencode *.c
   ```

7. Done. You have just created the BER and XER encoder of a Rectangle type, named **rencode**!

1.3 A "Rectangle" Decoder

This example will help you to create a simple BER decoder of a simple "Rectangle" type used throughout this document.

1. Create a file named **rectangle.asn** with the following contents:

```
RectangleModule DEFINITIONS ::= BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER,
    width   INTEGER
}

END
```

2. Compile it into the set of .c and .h files using asn1c compiler [ASN1C]:

```
asn1c —no—gen—example rectangle.asn
```

3. Alternatively, use the Online ASN.1 compiler [AONL] by uploading the **rectangle.asn** file into the Web form and unpacking the produced archive on your computer.

4. By this time, you should have gotten multiple files in the current directory, including the **Rectangle.c** and **Rectangle.h**.

5. Create a main() routine which takes the binary input file, decodes it as it were a BER-encoded Rectangle type, and prints out the text (XML) representation of the Rectangle type. Let's name the file **main.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <Rectangle.h>   /* Rectangle ASN.1 type */

int main(int ac, char **av) {
    char buf[1024];      /* Temporary buffer      */
    asn_dec_rval_t rval; /* Decoder return value  */
    Rectangle_t *rectangle = 0; /* Type to decode. Note this 0[1]! */
    FILE *fp;            /* Input file handler    */
    size_t size;         /* Number of bytes read  */
    char *filename;      /* Input file name */

    /* Require a single filename argument */
```

[1] Forgetting to properly initialize the pointer to a destination structure is a major source of support requests.

```
        if(ac != 2) {
            fprintf(stderr, "Usage: %s <file.ber>\n", av[0]);
            exit(1);
        } else {
            filename = av[1];
        }

        /* Open input file as read-only binary */
        fp = fopen(filename, "rb");
        if(!fp) {
            perror(filename);
            exit(1);
        }

        /* Read up to the buffer size */
        size = fread(buf, 1, sizeof(buf), fp);
        fclose(fp);
        if(!size) {
            fprintf(stderr, "%s: Empty or broken\n", filename);
            exit(1);
        }

        /* Decode the input buffer as Rectangle type */
        rval = ber_decode(0, &asn_DEF_Rectangle, (void **)&rectangle, buf, size);
        if(rval.code != RC_OK) {
            fprintf(stderr, "%s: Broken Rectangle encoding at byte %ld\n", filename,
                (long)rval.consumed);
            exit(1);
        }

        /* Print the decoded Rectangle type as XML */
        xer_fprint(stdout, &asn_DEF_Rectangle, rectangle);

        return 0; /* Decoding finished successfully */
    }
```

6. Compile all files together using C compiler (varies by platform):

   ```
   cc -I. -o rdecode *.c
   ```

7. Done. You have just created the BER decoder of a Rectangle type, named **rdecode**!

1.4 Adding constraints to a "Rectangle"

This example shows how to add basic constraints to the ASN.1 specification and how to invoke the constraints validation code in your application.

1. Create a file named **rectangle.asn** with the following contents:

```
RectangleModuleWithConstraints DEFINITIONS ::= BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER (0..100), -- Value range constraint
    width   INTEGER (0..MAX)  -- Makes width non-negative
}

END
```

2. Compile the file according to procedures shown in section 1.3 on page 7.

3. Modify the Rectangle type processing routine (you can start with the main() routine shown in the section 1.3 on page 7) by placing the following snippet of code *before* encoding and/or *after* decoding the Rectangle type:

```
int ret;           /* Return value */
char errbuf[128];  /* Buffer for error message */
size_t errlen = sizeof(errbuf);  /* Size of the buffer */

/* ... here goes the Rectangle decoding code ... */

ret = asn_check_constraints(&asn_DEF_Rectangle, rectangle, errbuf, &errlen);
/* assert(errlen < sizeof(errbuf)); // you may rely on that */
if(ret) {
    fprintf(stderr, "Constraint validation failed: %s\n", errbuf);
    /* exit(...); // Replace with appropriate action */
 }

/* ... here goes the Rectangle encoding code ... */
```

4. Compile the resulting C code as shown in the previous chapters.

5. Test the constraints checking code by assigning integer value 101 to the **.height** member of the Rectangle structure, or a negative value to the **.width** member. The program will fail with "Constraint validation failed" message.

6. Done.

Chapter 2

ASN.1 Compiler

2.1 The asn1c compiler tool

The purpose of the ASN.1 compiler is to convert the specifications in ASN.1 notation into some other language, such as C.

The compiler reads the specification and emits a series of target language structures (C structs, unions, enums) describing the corresponding ASN.1 types. The compiler also creates the code which allows automatic serialization and deserialization of these structures using several standardized encoding rules (BER, DER, OER, PER, XER).

Let's take the following ASN.1 example[2]:

```
RectangleModule DEFINITIONS ::= BEGIN

Rectangle ::= SEQUENCE {
    height  INTEGER,        -- Height of the rectangle
    width   INTEGER         -- Width of the rectangle
}

END
```

The asn1c compiler reads this ASN.1 definition and produce the following C type:

```
typedef struct Rectangle_s {
    long height;
    long width;
} Rectangle_t;
```

[2]Chapter 5 on page 43 provides a quick reference on the ASN.1 notation.

The asn1c compiler also creates the code for converting this structure into platform-independent wire representation and the decoder of such wire representation back into local, machine-specific type. These encoders and decoders are also called serializers and deserializers, marshallers and unmarshallers, or codecs.

Compiling ASN.1 modules into C codecs can be as simple as invoking `asn1c`: may be used to compile the ASN.1 modules:

```
asn1c <modules.asn>
```

If several ASN.1 modules contain interdependencies, all of the files must be specified altogether:

```
asn1c <module1.asn> <module2.asn> ...
```

The compiler **-E** and **-EF** options are used for testing the parser and the semantic fixer, respectively. These options will instruct the compiler to dump out the parsed (and fixed, if **-F** is involved) ASN.1 specification as it was understood by the compiler. It might be useful to check whether a particular syntactic construct is properly supported by the compiler.

```
asn1c -EF <module-to-test.asn>
```

The **-P** option is used to dump the compiled output on the screen instead of creating a bunch of .c and .h files on disk in the current directory. You would probably want to start with **-P** option instead of creating a mess in your current directory. Another option, **-R**, asks compiler to only generate the files which need to be generated, and supress linking in the numerous support files.

Print the compiled output instead of creating multiple source files:

```
asn1c -P <module-to-compile-and-print.asn>
```

2.2 Compiler output

The `asn1c` compiler produces a number of files:

- A set of .c and .h files for each type defined in the ASN.1 specification. These files will be named similarly to the ASN.1 types (**Rectangle.c** and **Rectangle.h** for the Rectangle-Module ASN.1 module defined in the beginning of this document).
- A set of helper .c and .h files which contain the generic encoders, decoders and other useful routines. Sometimes they are referred to by the term *skeletons*. There will be quite a few of them, some of them are not even always necessary, but the overall amount of code after compilation will be rather small anyway.
- A **Makefile.am.libasncodecs** file which explicitly lists all the generated files. This makefile can be used on its own to build the just the codec library.
- A **converter-example.c** file containing the *int main()* function with a fully functioning encoder and data format converter. It can convert a given PDU between BER, XER, OER and PER. At some point you will want to replace this file with your own file containing the *int main()* function.
- A **Makefile.am.example** file which binds together **Makefile.am.libasncodecs** and **converter-example.c** to build a versatile converter and debugger for your data formats.

It is possible to compile everything with just a couple of instructions:

```
asn1c -pdu=Rectangle *.asn
make -f Makefile.am.example                    # If you use 'make'
```

or

```
asn1c *.asn
cc -I. -DPDU=Rectangle -o rectangle.exe *.c    # ... or like this
```

Refer to the chapter 1 on page 4 for a sample *int main()* function if you want some custom logic and not satisfied with the supplied *converter-example.c*.

2.3 Command line options

The following table summarizes the `asn1c` command line options.

Stage Selection Options	Description
`-E`	Stop after the parsing stage and print the reconstructed ASN.1 specification code to the standard output.
`-F`	Used together with `-E`, instructs the compiler to stop after the ASN.1 syntax tree fixing stage and dump the reconstructed ASN.1 specification to the standard output.
`-P`	Dump the compiled output to the standard output instead of creating the target language files on disk.
`-R`	Restrict the compiler to generate only the ASN.1 tables, omitting the usual support code.
`-S` *<directory>*	Use the specified directory with ASN.1 skeleton files.
`-X`	Generate the XML DTD for the specified ASN.1 modules.

Warning Options	Description
`-Werror`	Treat warnings as errors; abort if any warning is produced.
`-Wdebug-parser`	Enable the parser debugging during the ASN.1 parsing stage.
`-Wdebug-lexer`	Enable the lexer debugging during the ASN.1 parsing stage.
`-Wdebug-fixer`	Enable the ASN.1 syntax tree fixer debugging during the fixing stage.
`-Wdebug-compiler`	Enable debugging during the actual compile time.

Language Options	Description
`-fbless-SIZE`	Allow SIZE() constraint for INTEGER, ENUMERATED, and other types for which this constraint is normally prohibited by the standard. This is a violation of an ASN.1 standard and compiler may fail to produce the meaningful code.

`-fcompound-names`	Use complex names for C structures. Using complex names prevents name clashes in case the module reuses the same identifiers in multiple contexts.
`-findirect-choice`	When generating code for a CHOICE type, compile the CHOICE members as indirect pointers instead of declaring them inline. Consider using this option together with `-fno-include-deps` to prevent circular references.
`-fincludes-quoted`	Generate #include lines in ”double” instead of <angle> quotes.
`-fknown-extern-type=<name>`	Pretend the specified type is known. The compiler will assume the target language source files for the given type have been provided manually.
`-fline-refs`	Include ASN.1 module's line numbers in generated code comments.
`-fno-constraints`	Do not generate the ASN.1 subtype constraint checking code. This may produce a shorter executable.
`-fno-include-deps`	Do not generate the courtesy #include lines for non-critical dependencies.
`-funnamed-unions`	Enable unnamed unions in the definitions of target language's structures.
`-fwide-types`	Use the wide integer types (INTEGER_t, REAL_t) instead of machine's native data types (long, double).

Codecs Generation Options	**Description**
`-no-gen-OER`	Do not generate the Octet Encoding Rules (OER, X.696) support code.
`-no-gen-PER`	Do not generate the Packed Encoding Rules (PER, X.691) support code.
`-no-gen-example`	Do not generate the ASN.1 format converter example.

`-pdu={`**`all`**`|`**`auto`**`|`*`Type`*`}`	Create a PDU table for specified types, or discover the Protocol Data Units automatically. In case of `-pdu=`**`all`**, all ASN.1 types defined in all modules wil form a PDU table. In case of `-pdu=`**`auto`**, all types not referenced by any other type will form a PDU table. If *`Type`* is an ASN.1 type identifier, it is added to a PDU table. The last form may be specified multiple times.

Output Options	**Description**
`-print-class-matrix`	When `-EF` options are given, this option instructs the compiler to print out the collected Information Object Class matrix.
`-print-constraints`	With `-EF`, this option instructs the compiler to explain its internal understanding of subtype constraints.
`-print-lines`	Generate "`-- #line`" comments in `-E` output.

Chapter 3

API reference

The functions desribed in this chapter are to be used by the application programmer. These functions won't likely change change or get removed until the next major release.

The API calls not listed here are not public and should not be used by the application level code.

3.1 ASN_STRUCT_FREE() macro

Synopsis

```
#define ASN_STRUCT_FREE(type_descriptor, struct_ptr)
```

Description

Recursively releases memory occupied by the structure described by the **type_descriptor** and referred to by the **struct_ptr** pointer.

Does nothing when **struct_ptr** is NULL.

Return values

Does not return a value.

Example

```
Rectangle_t *rect = ...;
ASN_STRUCT_FREE(asn_DEF_Rectangle, rect);
```

3.2 ASN_STRUCT_RESET() macro

Synopsis

```
#define ASN_STRUCT_RESET(type_descriptor, struct_ptr)
```

Description

Recursively releases memory occupied by the members of the structure described by the **type_descriptor** and referred to by the **struct_ptr** pointer.

Does not release the memory pointed to by **struct_ptr** itself. Instead it clears the memory block by filling it out with 0 bytes.

Does nothing when **struct_ptr** is NULL.

Return values

Does not return a value.

Example

```
struct my_figure {          /* The custom structure */
    int flags;              /* <some custom member> */
    /* The type is generated by the ASN.1 compiler */
    Rectangle_t rect;
    /* other members of the structure */
};

struct my_figure *fig = ...;
ASN_STRUCT_RESET(asn_DEF_Rectangle, &fig->rect);
```

3.3 asn_check_constraints()

Synopsis

```
int asn_check_constraints(
    const asn_TYPE_descriptor_t *type_descriptor,
    const void *struct_ptr, /* Target language's structure */
    char *errbuf,           /* Returned error description */
    size_t *errlen          /* Length of the error description */
);
```

Description

Validate the structure according to the ASN.1 constraints. If errbuf and errlen are given, they shall be pointing to the appropriate buffer space and its length before calling this function. Alternatively, they could be passed as NULLs. If constraints validation fails, errlen will contain the actual number of bytes used in errbuf to encode an error message, properly 0-terminated.

Return values

This function returns 0 in case all ASN.1 constraints are met and -1 if one or more ASN.1 constraints were violated.

Example

```
Rectangle_t *rect = ...;

char errbuf[128];  /* Buffer for error message */
size_t errlen = sizeof(errbuf);  /* Size of the buffer */
```

```
int ret = asn_check_constraints(&asn_DEF_Rectangle, rectangle, errbuf, &errlen);
/* assert(errlen < sizeof(errbuf)); // Guaranteed: you may rely on that */
if(ret) {
    fprintf(stderr, "Constraint validation failed: %s\n", errbuf);
}
```

3.4 asn_decode()

Synopsis

```
asn_dec_rval_t asn_decode(
    const asn_codec_ctx_t *opt_codec_parameters,
    enum asn_transfer_syntax syntax,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,    /* Data to be decoded */
    size_t size            /* Size of that buffer */
);
```

Description

The **asn_decode()** function parses the data given by the **buffer** and **size** arguments. The encoding rules are specified in the **syntax** argument and the type to be decoded is specified by the **type_descriptor**.

The **struct_ptr_ptr** must point to the memory location which contains the pointer to the structure being decoded. Initially the ***struct_ptr_ptr** pointer is typically set to 0. In that case, **asn_decode()** will dynamically allocate memory for the structure and its members as needed during the parsing. If ***struct_ptr_ptr** already points to some memory, the **asn_decode()** will allocate the subsequent members as needed during the parsing.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,          /* Decoded successfully */
        RC_WMORE,       /* More data expected, call again */
        RC_FAIL         /* Failure to decode data */
    } code;             /* Result code */
    size_t consumed;    /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

RC_OK	Decoded successfully and completely
RC_WMORE	More data expected, call again
RC_FAIL	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

The **.consumed** value is in bytes, even for PER decoding. For PER, use **uper_decode()** in case you need to get the number of consumed bits.

Restartability

Some transfer syntax parsers (such as ATS_BER) support restartability.

That means that in case the buffer has less data than expected, the **asn_decode()** will process whatever is available and ask for more data to be provided using the RC_WMORE return **.code**.

Note that in the RC_WMORE case the decoder may have processed less data than it is available in the buffer, which means that you must be able to arrange the next buffer to contain the unprocessed part of the previous buffer.

The **RC_WMORE** code may still be returned by parser not supporting restartabilty. In such cases, the partially decoded structure shall be discarded and the next invocation should use the extended buffer to parse from the very beginning.

Example

```
Rectangle_t *rect = 0;    /* Note this 0[1]! */
asn_dec_rval_t rval;
rval = asn_decode(0, ATS_BER, &asn_DEF_Rectangle, (void **)&rect, buffer, buf_size);
switch(rval.code) {
```

[1]Forgetting to properly initialize the pointer to a destination structure is a major source of support requests.

```
case RC_OK:
    asn_fprint(stdout, &asn_DEF_Rectangle, rect);
    ASN_STRUCT_FREE(&asn_DEF_Rectangle, rect);
    break;
case RC_WMORE:
case RC_FAIL:
default:
    ASN_STRUCT_FREE(&asn_DEF_Rectangle, rect);
    break;
}
```

See also

asn_fprint() at page 22.

3.5 asn_encode()

3.6 asn_encode_to_buffer

Example

```
uint8_t buffer[128];
size_t buf_size = sizeof(buffer);
asn_enc_rval_t er;
er = asn_encode_to_buffer(0, ATS_DER, &asn_DEF_Rectangle, buffer, buf_size);
if(er.encoded > buf_size) {
   fprintf(stderr, "Buffer of size %zu is too small for %s, need %zu\n",
       buf_size, asn_DEF_Rectangle.name, er.encoded);
}
```

3.7 asn_encode_to_new_buffer

Example

```
asn_encode_to_new_buffer_result_t res;
res = asn_encode_to_new_buffer(0, ATS_DER, &asn_DEF_Rectangle, buffer, buf_size);
if(res.buffer) {
    /* Encoded successfully. */
    free(res.buffer);
} else {
    fprintf(stderr, "Failed to encode %s, estimated %zd bytes\n",
        asn_DEF_Rectangle.name, res.result.encoded);
}
```

3.8 asn_fprint()

Synopsis

```
int asn_fprint(FILE *stream,     /* Destination file */
    const asn_TYPE_descriptor_t *type_descriptor,
    const void *struct_ptr       /* Structure to be printed */
);
```

Description

The **asn_fprint()** function prints human readable description of the target language's structure into the file stream specified by **stream** pointer.

The output format does not conform to any standard.

The **asn_fprint()** function attempts to produce a valid output even for incomplete and broken structures, which makes it more suitable for debugging complex cases than **xer_fprint()**.

Return values

0	Output was successfully made
-1	Error printing out the structure

Example

```
Rectangle_t *rect = ...;
asn_fprint(stdout, &asn_DEF_Rectangle, rect);
```

See also

xer_fprint() at page 32.

3.9 asn_random_fill()

Synopsis

```
int asn_random_fill(
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure’s ptr */
    size_t approx_max_length_limit
);
```

Description

Create or initialize a structure with random contents, according to the type specification and optional member constraints.

For best results the code should be generated without `-no-gen-PER` option to `asn1c`. Making PER constraints code available in runtime will make **`asn_random_fill`** explore the edges of PER-visible constraints and sometimes break out of extensible contstraints' ranges.

The **`asn_random_fill()`** function has a bias to generate edge case values. This property makes it useful for debugging the application level code and for security testing, as random data can be a good seed to fuzzing.

The **`approx_max_length_limit`** specifies the approximate limit of the resulting structure in units closely resembling bytes. The actual result might be several times larger or smaller than the given length limit. A rule of thumb way to select the initial value for this parameter is to take a typical structure and use twice its DER output size.

Return values

0	Structure was properly initialized with random data
-1	Failure to initialize the structure with random data

3.10 ber_decode()

Synopsis

```
asn_dec_rval_t ber_decode(
    const asn_codec_ctx_t *opt_codec_ctx,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,    /* Data to be decoded */
    size_t size            /* Size of that buffer */
);
```

Description

Decode BER, DER and CER data (Basic Encoding Rules, Distinguished Encoding Rules, Canonical Encoding Rules), as defined by ITU-T X.690.

DER and CER are different subsets of BER.

Consider using a more generic function **asn_decode(ATS_BER)**.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,          /* Decoded successfully */
        RC_WMORE,       /* More data expected, call again */
        RC_FAIL         /* Failure to decode data */
    } code;             /* Result code */
    size_t consumed;    /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

RC_OK	Decoded successfully and completely
RC_WMORE	More data expected, call again
RC_FAIL	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

The **.consumed** value is in bytes.

Restartability

The **ber_decode()** function is restartable (stream-oriented). That means that in case the buffer has less data than expected, the decoder will process whatever is available and ask for more data to be provided using the RC_WMORE return **.code**.

Note that in the RC_WMORE case the decoder may have processed less data than it is available in the buffer, which means that you must be able to arrange the next buffer to contain the unprocessed part of the previous buffer.

See also

der_encode() at page 25.

3.11 der_encode

See also

ber_decode() at page 24, **asn_decode(ATS_BER)** at page 19.

3.12 der_encode_to_buffer

3.13 oer_decode()

Synopsis

```
asn_dec_rval_t oer_decode(
    const asn_codec_ctx_t *opt_codec_ctx,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,    /* Data to be decoded */
    size_t size            /* Size of that buffer */
);
```

Description

Decode the BASIC-OER and CANONICAL-OER (Octet Encoding Rules), as defined by ITU-T X.696.

Consider using a more generic function **asn_decode(ATS_BASIC_OER)**.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,          /* Decoded successfully */
        RC_WMORE,       /* More data expected, call again */
        RC_FAIL         /* Failure to decode data */
    } code;             /* Result code */
    size_t consumed;    /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

RC_OK	Decoded successfully and completely
RC_WMORE	More data expected, call again
RC_FAIL	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

The **.consumed** value is in bytes.

Restartability

The **oer_decode()** function is restartable (stream-oriented). That means that in case the buffer has less data than expected, the decoder will process whatever is available and ask for more data to be provided using the RC_WMORE return **.code**.

Note that in the RC_WMORE case the decoder may have processed less data than it is available in the buffer, which means that you must be able to arrange the next buffer to contain the unprocessed part of the previous buffer.

3.14 oer_encode

3.15 oer_encode_to_buffer

3.16 uper_decode()

Synopsis

```
asn_dec_rval_t uper_decode(
    const asn_codec_ctx_t *opt_codec_ctx,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,   /* Data to be decoded */
    size_t size,          /* Size of the input data buffer, bytes */
    int skip_bits,        /* Number of unused leading bits, 0..7 */
    int unused_bits       /* Number of unused tailing bits, 0..7 */
);
```

Description

Decode the Unaligned BASIC or CANONICAL PER (Packed Encoding Rules), as defined by ITU-T X.691

Consider using a more generic function **asn_decode(ATS_UNALIGNED_BASIC_PER)**.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,          /* Decoded successfully */
        RC_WMORE,       /* More data expected, call again */
        RC_FAIL         /* Failure to decode data */
    } code;             /* Result code */
    size_t consumed;    /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

`RC_OK`	Decoded successfully and completely
`RC_WMORE`	More data expected, call again
`RC_FAIL`	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

Note that the **.consumed** value is in bits. Use **(.consumed+7)/8** to convert to bytes.

Restartability

The **uper_decode()** function is not restartable. Failures are final.

3.17 uper_decode_complete()

Synopsis

```
asn_dec_rval_t uper_decode_complete(
    const asn_codec_ctx_t *opt_codec_ctx,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,    /* Data to be decoded */
    size_t size            /* Size of data buffer */
);
```

Description

Decode a "Production of a complete encoding", according to ITU-T X.691 (08/2015) #11.1.

Consider using a more generic function **asn_decode(ATS_UNALIGNED_BASIC_PER)**.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,              /* Decoded successfully */
        RC_WMORE,           /* More data expected, call again */
        RC_FAIL             /* Failure to decode data */
    } code;                 /* Result code */
    size_t consumed;        /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

RC_OK	Decoded successfully and completely
RC_WMORE	More data expected, call again
RC_FAIL	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

The the **.consumed** value is returned in bytes.

Restartability

The **uper_decode_complete()** function is not restartable. Failures are final.

The complete encoding contains at least one byte, so on success **.consumed** will be greater or equal to 1.

3.18 uper_encode

3.19 uper_encode_to_buffer

3.20 uper_encode_to_new_buffer

3.21 xer_decode()

Synopsis

```
asn_dec_rval_t xer_decode(
    const asn_codec_ctx_t *opt_codec_ctx,
    const asn_TYPE_descriptor_t *type_descriptor,
    void **struct_ptr_ptr,/* Pointer to a target structure's ptr */
    const void *buffer,    /* Data to be decoded */
    size_t size            /* Size of data buffer */
);
```

Description

Decode the BASIC-XER and CANONICAL-XER (XML Encoding Rules) encoding, as defined by ITU-T X.693.

Consider using a more generic function **asn_decode(ATS_BASIC_XER)**.

Return values

Upon unsuccessful termination, the ***struct_ptr_ptr** may contain partially decoded data. This data may be useful for debugging (such as by using **asn_fprint()**). Don't forget to discard the unused partially decoded data by calling **ASN_STRUCT_FREE()** or **ASN_STRUCT_RESET()**.

The return value is returned in a compound structure:

```
typedef struct {
    enum {
        RC_OK,          /* Decoded successfully */
        RC_WMORE,       /* More data expected, call again */
        RC_FAIL         /* Failure to decode data */
    } code;             /* Result code */
    size_t consumed;    /* Number of bytes consumed */
} asn_dec_rval_t;
```

The **.code** member specifies the decoding outcome.

RC_OK	Decoded successfully and completely
RC_WMORE	More data expected, call again
RC_FAIL	Failed for good

The **.consumed** member specifies the amount of **buffer** data that was used during parsing, irrespectively of the **.code**.

The **.consumed** value is in bytes.

Restartability

The **xer_decode()** function is restartable (stream-oriented). That means that in case the buffer has less data than expected, the decoder will process whatever is available and ask for more data to be provided using the RC_WMORE return **.code**.

Note that in the RC_WMORE case the decoder may have processed less data than it is available in the buffer, which means that you must be able to arrange the next buffer to contain the unprocessed part of the previous buffer.

3.22 xer_encode

3.23 xer_fprint()

Synopsis

```
int xer_fprint(FILE *stream,     /* Destination file */
    const asn_TYPE_descriptor_t *type_descriptor,
    const void *struct_ptr       /* Structure to be printed */
);
```

Description

The **xer_fprint()** function outputs XML-based serialization of the given structure into the file stream specified by **stream** pointer.

The output conforms to BASIC-XER, as defined by ITU-T X.693.

Return values

0	XML output was successfully made
-1	Error printing out the structure

Since the **xer_fprint()** function attempts to produce a conforming output, it will likely break on partial structures by writing incomplete data to the output stream and returning -1. This makes it less suitable for debugging complex cases than **asn_fprint()**.

Example

```
Rectangle_t *rect = ...;
xer_fprint(stdout, &asn_DEF_Rectangle, rect);
```

See also

asn_fprint() at page 22.

Chapter 4

API usage examples

Let's start with including the necessary header files into your application. Normally it is enough to include the header file of the main PDU type. For our *Rectangle* module, including the *Rectangle.h* file is sufficient:

```
#include <Rectangle.h>
```

The header files defines a C structure corresponding to the ASN.1 definition of a rectangle and the declaration of the ASN.1 *type descriptor*. A type descriptor is a special globally accessible object which is used as an argument to most of the API functions provided by the ASN.1 codec. A type descriptor starts with *asn_DEF_...*. For example, here is the code which frees the Rectangle_t structure:

```
Rectangle_t *rect = ...;

ASN_STRUCT_FREE(asn_DEF_Rectangle, rect);
```

This code defines a *rect* pointer which points to the Rectangle_t structure which needs to be freed. The second line uses a generic **ASN_STRUCT_FREE()** macro which invokes the memory deallocation routine created specifically for this Rectangle_t structure. The *asn_DEF_Rectangle* is the type descriptor which holds a collection of routines and operations defined for the Rectangle_t structure.

4.1 Generic encoders and decoders

Before we start describing specific encoders and decoders, let's step back a little and check out a simple high level way.

The asn1c runtime supplies (see *asn_application.h*) two sets of high level functions, **asn_encode*** and **asn_decode***, which take a transfer syntax selector as an argument. The transfer syntax selector is defined as this:

```
/*
 * A selection of ASN.1 Transfer Syntaxes to use with generalized encoders and decoders.
 */
enum asn_transfer_syntax {
    ATS_INVALID,
    ATS_NONSTANDARD_PLAINTEXT,
    ATS_BER,
    ATS_DER,
    ATS_CER,
    ATS_BASIC_OER,
    ATS_CANONICAL_OER,
    ATS_UNALIGNED_BASIC_PER,
    ATS_UNALIGNED_CANONICAL_PER,
    ATS_BASIC_XER,
    ATS_CANONICAL_XER,
};
```

Using this encoding selector, encoding and decoding becomes very generic:

Encoding:

```
uint8_t buffer[128];
size_t buf_size = sizeof(buffer);
asn_enc_rval_t er;

er = asn_encode_to_buffer(0, ATS_DER, &asn_DEF_Rectangle, buffer, buf_size);

if(er.encoded > buf_size) {
    fprintf(stderr, "Buffer of size %zu is too small for %s, need %zu\n",
        buf_size, asn_DEF_Rectangle.name, er.encoded);
}
```

Decoding:

```
Rectangle_t *rect = 0;     /* Note this 0[1]! */

... = asn_decode(0, ATS_BER, &asn_DEF_Rectangle, (void **)&rect, buffer, buf_size);
```

4.2 Decoding BER

The Basic Encoding Rules describe the most widely used (by the ASN.1 community) way to encode and decode a given structure in a machine-independent way. Several other encoding rules (CER, DER) define a more restrictive versions of BER, so the generic BER parser is also

[1] Forgetting to properly initialize the pointer to a destination structure is a major source of support requests.

capable of decoding the data encoded by the CER and DER encoders. The opposite is not true.

The ASN.1 compiler provides the generic BER decoder which is capable of decoding BER, CER and DER encoded data.

The decoder is restartable (stream-oriented). That means that in case the buffer has less data than expected, the decoder will process whatever is available and ask for more data to be provided using the RC_WMORE return **.code**.

Note that in the RC_WMORE case the decoder may have processed less data than it is available in the buffer, which means that you must be able to arrange the next buffer to contain the unprocessed part of the previous buffer.

Suppose, you have two buffers of encoded data: 100 bytes and 200 bytes.

- You can concatenate these buffers and feed the BER decoder with 300 bytes of data, or
- You can feed it the first buffer of 100 bytes of data, realize that the ber_decoder consumed only 95 bytes from it and later feed the decoder with 205 bytes buffer which consists of 5 unprocessed bytes from the first buffer and the additional 200 bytes from the second buffer.

This is not as convenient as it could be (the BER encoder could consume the whole 100 bytes and keep these 5 bytes in some temporary storage), but in case of existing stream based processing it might actually fit well into existing algorithm. Suggestions are welcome.

Here is the example of BER decoding of a simple structure:

```
Rectangle_t *
simple_deserializer(const void *buffer, size_t buf_size) {
    asn_dec_rval_t rval;
    Rectangle_t *rect = 0;    /* Note this 0[1]! */

    rval = asn_DEF_Rectangle.op->ber_decoder(0,
          &asn_DEF_Rectangle,
          (void **)&rect,    /* Decoder changes the pointer */
          buffer, buf_size, 0);

    if(rval.code == RC_OK) {
        return rect;          /* Decoding succeeded */
    } else {
```

[1] Forgetting to properly initialize the pointer to a destination structure is a major source of support requests.

```
            /* Free the partially decoded rectangle */
            ASN_STRUCT_FREE(asn_DEF_Rectangle, rect);
            return 0;
        }
    }
```

The code above defines a function, *simple_deserializer*, which takes a buffer and its length and is expected to return a pointer to the Rectangle_t structure. Inside, it tries to convert the bytes passed into the target structure (rect) using the BER decoder and returns the rect pointer afterwards. If the structure cannot be deserialized, it frees the memory which might be left allocated by the unfinished *ber_decoder* routine and returns 0 (no data). (This **freeing is necessary** because the ber_decoder is a restartable procedure, and may fail just because there is more data needs to be provided before decoding could be finalized). The code above obviously does not take into account the way the *ber_decoder()* failed, so the freeing is necessary because the part of the buffer may already be decoded into the structure by the time something goes wrong.

A little less wordy would be to invoke a globally available *ber_decode()* function instead of dereferencing the asn_DEF_Rectangle type descriptor:

```
rval = ber_decode(0, &asn_DEF_Rectangle, (void **)&rect, buffer,
   buf_size);
```

Note that the initial (asn_DEF_Rectangle.op->ber_decoder) reference is gone, and also the last argument (0) is no longer necessary.

These two ways of BER decoder invocations are fully equivalent.

The BER de*coder* may fail because of (*the following RC_... codes are defined in ber_decoder.h*):

- RC_WMORE: There is more data expected than it is provided (stream mode continuation feature);
- RC_FAIL: General failure to decode the buffer;
- ... other codes may be defined as well.

Together with the return code (.code) the asn_dec_rval_t type contains the number of bytes which is consumed from the buffer. In the previous hypothetical example of two buffers (of 100 and 200 bytes), the first call to ber_decode() would return with .code = RC_WMORE and .consumed = 95. The .consumed field of the BER decoder return value is **always** valid, even if the decoder succeeds or fails with any other return code.

Look into ber_decoder.h for the precise definition of ber_decode() and related types.

4.3 Encoding DER

The Distinguished Encoding Rules is the *canonical* variant of BER encoding rules. The DER is best suited to encode the structures where all the lengths are known beforehand. This is probably exactly how you want to encode: either after a BER decoding or after a manual fill-up, the target structure contains the data which size is implicitly known before encoding. Among other uses, the DER encoding is used to encode X.509 certificates.

As with BER decoder, the DER encoder may be invoked either directly from the ASN.1 type descriptor (asn_DEF_Rectangle) or from the stand-alone function, which is somewhat simpler:

```
/*
 * This is the serializer itself.
 * It supplies the DER encoder with the
 * pointer to the custom output function.
 */
ssize_t
simple_serializer(FILE *ostream, Rectangle_t *rect) {
    asn_enc_rval_t er;  /* Encoder return value */

    er = der_encode(&asn_DEF_Rect, rect, write_stream, ostream);
    if(er.encoded == -1) {
        fprintf(stderr, "Cannot encode %s: %s\n",
            er.failed_type->name, strerror(errno));
        return -1;
    } else {
        /* Return the number of bytes */
        return er.encoded;
    }
}
```

As you see, the DER encoder does not write into some sort of buffer. It just invokes the custom function (possible, multiple times) which would save the data into appropriate storage. The optional argument *app_key* is opaque for the DER encoder code and just used by *_write_stream()* as the pointer to the appropriate output stream to be used.

If the custom write function is not given (passed as 0), then the DER encoder will essentially do the same thing (i. e., encode the data) but no callbacks will be invoked (so the data goes nowhere). It may prove useful to determine the size of the structure's encoding before actually doing the encoding[1].

[1] It is actually faster too: the encoder might skip over some computations which aren't important for the size

Look into der_encoder.h for the precise definition of der_encode() and related types.

4.4 Encoding XER

The XER stands for XML Encoding Rules, where XML, in turn, is eXtensible Markup Language, a text-based format for information exchange. The encoder routine API comes in two flavors: stdio-based and callback-based. With the callback-based encoder, the encoding process is very similar to the DER one, described in section 4.3 on the preceding page. The following example uses the definition of write_stream() from up there.

```
/*
 * This procedure generates an XML document
 * by invoking the XER encoder.
 * NOTE: Do not copy this code verbatim!
 *       If the stdio output is necessary,
 *       use the xer_fprint() procedure instead.
 *       See section 4.7 on page 41.
 */
int
print_as_XML(FILE *ostream, Rectangle_t *rect) {
    asn_enc_rval_t er;  /* Encoder return value */

    er = xer_encode(&asn_DEF_Rectangle, rect,
        XER_F_BASIC, /* BASIC-XER or CANONICAL-XER */
        write_stream, ostream);

    return (er.encoded == -1) ? -1 : 0;
}
```

Look into xer_encoder.h for the precise definition of xer_encode() and related types.

See section 4.7 on page 41 for the example of stdio-based XML encoder and other pretty-printing suggestions.

determination.

4.5 Decoding XER

The data encoded using the XER rules can be subsequently decoded using the xer_decode() API call:

```
Rectangle_t *
XML_to_Rectangle(const void *buffer, size_t buf_size) {
    asn_dec_rval_t rval;
    Rectangle_t *rect = 0;    /* Note this 0[1]! */

    rval = xer_decode(0, &asn_DEF_Rectangle, (void **)&rect,
       buffer, buf_size);

    if(rval.code == RC_OK) {
        return rect;          /* Decoding succeeded */
    } else {
        /* Free partially decoded rect */
        ASN_STRUCT_FREE(asn_DEF_Rectangle, rect);
        return 0;
    }
}
```

The decoder takes both BASIC-XER and CANONICAL-XER encodings.

The decoder shares its data consumption properties with BER decoder; please read the section 4.2 on page 35 to know more.

Look into xer_decoder.h for the precise definition of xer_decode() and related types.

4.6 Validating the target structure

Sometimes the target structure needs to be validated. For example, if the structure was created by the application (as opposed to being decoded from some external source), some important information required by the ASN.1 specification might be missing. On the other hand, the successful decoding of the data from some external source does not necessarily mean that the data is fully valid either. It might well be the case that the specification describes some subtype constraints that were not taken into account during decoding, and it would actually be useful to perform the last check when the data is ready to be encoded or when the data has just been decoded to ensure its validity according to some stricter rules.

[1] Forgetting to properly initialize the pointer to a destination structure is a major source of support requests.

The asn_check_constraints() function checks the type for various implicit and explicit constraints. It is recommended to use asn_check_constraints() function after each decoding and before each encoding.

Look into constraints.h for the precise definition of asn_check_constraints() and related types.

4.7 Printing the target structure

There are two ways to print the target structure: either invoke the print_struct member of the ASN.1 type descriptor, or using the asn_fprint() function, which is a simpler wrapper of the former:

```
asn_fprint(stdout, &asn_DEF_Rectangle, rect);
```

Look into constr_TYPE.h for the precise definition of asn_fprint() and related types.

Another practical alternative to this custom format printing would be to invoke XER encoder. The default BASIC-XER encoder performs reasonable formatting for the output to be useful and human readable. To invoke the XER decoder in a manner similar to asn_fprint(), use the xer_fprint() call:

```
xer_fprint(stdout, &asn_DEF_Rectangle, rect);
```

See section 4.4 on page 39 for XML-related details.

4.8 Freeing the target structure

Freeing the structure is slightly more complex than it may seem to. When the ASN.1 structure is freed, all the members of the structure and their submembers are recursively freed as well. The ASN_STRUCT_FREE() macro helps with that.

But it might not always be feasible to free the whole structure. In the following example, the application programmer defines a custom structure with one ASN.1-derived member (rect).

```
struct my_figure {        /* The custom structure */
    int flags;            /* <some custom member> */
    /* The type is generated by the ASN.1 compiler */
    Rectangle_t rect;
    /* other members of the structure */
};
```

This member is not a reference to the Rectangle_t, but an in-place inclusion of the Rectangle_t structure. If there's a need to free the **rect** member, the usual procedure of freeing everything must not be applied to the **&rect** pointer itself, because it does not point to the beginning of memory block allocated by the memory allocation routine, but instead lies within a block allocated for the my_figure structure.

To solve this problem, in addition to ASN_STRUCT_FREE() macro, the asn1c skeletons define the ASN_STRUCT_RESET() macro which doesn't free the passed pointer and instead resets the structure into the clean and safe state.

```
/* 1. Rectangle_t is defined within my_figure */
struct my_figure {
    Rectangle_t rect;
} *mf = ...;
/*
 * Freeing the Rectangle_t
 * without freeing the mf->rect area.
 */
ASN_STRUCT_RESET(asn_DEF_Rectangle, &mf->rect);

/* 2. Rectangle_t is a stand-alone pointer */
Rectangle_t *rect = ...;
/*
 * Freeing the Rectangle_t
 * and freeing the rect pointer.
 */
ASN_STRUCT_FREE(asn_DEF_Rectangle, rect);
```

It is safe to invoke both macros with the target structure pointer set to 0 (NULL). In this case, the function will do nothing.

Chapter 5

Abstract Syntax Notation: ASN.1

This chapter defines some basic ASN.1 concepts and describes several most widely used types. It is by no means an authoritative or complete reference. For more complete ASN.1 description, please refer to Olivier Dubuisson's book [Dub00] or the ASN.1 body of standards itself [ITU-T/ASN.1].

The Abstract Syntax Notation One is used to formally describe the data transmitted across the network. Two communicating parties may employ different formats of their native data types (e. g., different number of bits for the native integer type), thus it is important to have a way to describe the data in a manner which is independent from the particular machine's representation. The ASN.1 specifications are used to achieve the following:

- The specification expressed in the ASN.1 notation is a formal and precise way to communicate the structure of data to human readers;
- The ASN.1 specifications may be used as input for automatic compilers which produce the code for some target language (C, C++, Java, etc) to encode and decode the data according to some encoding formats. Several such encoding formats (called Transfer Encoding Rules) have been defined by the ASN.1 standard.

Consider the following example:

```
Rectangle ::= SEQUENCE {
    height  INTEGER,
    width   INTEGER
}
```

This ASN.1 specification describes a constructed type, *Rectangle*, containing two integer fields. This specification may tell the reader that there exists this kind of data structure and that some entity may be prepared to send or receive it. The question on *how* that entity is going to send or receive the *encoded data* is outside the scope of ASN.1. For example, this data structure may be encoded according to some encoding rules and sent to the destination using the TCP protocol. The ASN.1 specifies several ways of encoding (or “serializing”, or “marshaling”) the data: BER, PER, XER and others, including CER and DER derivatives from BER.

The complete specification must be wrapped in a module, which looks like this:

```
RectangleModule1
    { iso org(3) dod(6) internet(1) private(4)
      enterprise(1) spelio(9363) software(1)
      asn1c(5) docs(2) rectangle(1) 1 }
    DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

-- This is a comment which describes nothing.
Rectangle ::= SEQUENCE {
    height  INTEGER,        -- Height of the rectangle
    width   INTEGER         -- Width of the rectangle
}

END
```

The module header consists of module name (RectangleModule1), the module object identifier ({...}), a keyword “DEFINITIONS”, a set of module flags (AUTOMATIC TAGS) and “::= BEGIN”. The module ends with an “END” statement.

5.1 Some of the ASN.1 Basic Types

5.1.1 The BOOLEAN type

The BOOLEAN type models the simple binary TRUE/FALSE, YES/NO, ON/OFF or a similar kind of two-way choice.

5.1.2 The INTEGER type

The INTEGER type is a signed natural number type without any restrictions on its size. If the automatic checking on INTEGER value bounds are necessary, the subtype constraints must be used.

```
SimpleInteger ::= INTEGER

-- An integer with a very limited range
SmallPositiveInt ::= INTEGER (0..127)

-- Integer, negative
NegativeInt ::= INTEGER (MIN..0)
```

5.1.3 The ENUMERATED type

The ENUMERATED type is semantically equivalent to the INTEGER type with some integer values explicitly named.

```
FruitId ::= ENUMERATED { apple(1), orange(2) }

-- The numbers in braces are optional,
-- the enumeration can be performed
-- automatically by the compiler
ComputerOSType ::= ENUMERATED {
    FreeBSD,            -- acquires value 0
    Windows,            -- acquires value 1
    Solaris(5),         -- remains 5
    Linux,              -- becomes 6
    MacOS               -- becomes 7
}
```

5.1.4 The OCTET STRING type

This type models the sequence of 8-bit bytes. This may be used to transmit some opaque data or data serialized by other types of encoders (e. g., video file, photo picture, etc).

5.1.5 The OBJECT IDENTIFIER type

The OBJECT IDENTIFIER is used to represent the unique identifier of any object, starting from the very root of the registration tree. If your organization needs to uniquely identify something (a router, a room, a person, a standard, or whatever), you are encouraged to get your own identification subtree at http://www.iana.org/protocols/forms.htm.

For example, the very first ASN.1 module in this Chapter (RectangleModule1) has the following OBJECT IDENTIFIER: 1 3 6 1 4 1 9363 1 5 2 1 1.

```
ExampleOID ::= OBJECT IDENTIFIER

rectangleModule1-oid ExampleOID
  ::= { 1 3 6 1 4 1 9363 1 5 2 1 1 }

-- An identifier of the Internet.
internet-id OBJECT IDENTIFIER
  ::= { iso(1) identified-organization(3)
        dod(6) internet(1) }
```

As you see, names are optional.

5.1.6 The RELATIVE-OID type

The RELATIVE-OID type has the semantics of a subtree of an OBJECT IDENTIFIER. There may be no need to repeat the whole sequence of numbers from the root of the registration tree where the only thing of interest is some of the tree's subsequence.

```
this-document RELATIVE-OID ::= { docs(2) usage(1) }

this-example RELATIVE-OID ::= {
    this-document assorted-examples(0) this-example(1) }
```

5.2 Some of the ASN.1 String Types

5.2.1 The IA5String type

This is essentially the ASCII, with 128 character codes available (7 lower bits of an 8-bit byte).

5.2.2 The UTF8String type

This is the character string which encodes the full Unicode range (4 bytes) using multibyte character sequences.

5.2.3 The NumericString type

This type represents the character string with the alphabet consisting of numbers ("0" to "9") and a space.

5.2.4 The PrintableString type

The character string with the following alphabet: space, "'" (single quote), "(", ")", "+", "," (comma), "-", ".", "/", digits ("0" to "9"), ":", "=", "?", upper-case and lower-case letters ("A" to "Z" and "a" to "z").

5.2.5 The VisibleString type

The character string with the alphabet which is more or less a subset of ASCII between the space and the "~" symbol (tilde).

Alternatively, the alphabet may be described as the PrintableString alphabet presented earlier, plus the following characters: "!", """, "#", "$", "%", "&", "*", ";", "<", ">", "[", "\", "]", "^", "_", "`" (single left quote), "{", "|", "}", "~".

5.3 ASN.1 Constructed Types

5.3.1 The SEQUENCE type

This is an ordered collection of other simple or constructed types. The SEQUENCE constructed type resembles the C "struct" statement.

```
Address ::= SEQUENCE {
    -- The apartment number may be omitted
    apartmentNumber      NumericString OPTIONAL,
    streetName           PrintableString,
    cityName             PrintableString,
    stateName            PrintableString,
    -- This one may be omitted too
    zipNo                NumericString OPTIONAL
}
```

5.3.2 The SET type

This is a collection of other simple or constructed types. Ordering is not important. The data may arrive in the order which is different from the order of specification. Data is encoded in the order not necessarily corresponding to the order of specification.

5.3.3 The CHOICE type

This type is just a choice between the subtypes specified in it. The CHOICE type contains at most one of the subtypes specified, and it is always implicitly known which choice is being decoded or encoded. This one resembles the C "union" statement.

The following type defines a response code, which may be either an integer code or a boolean "true"/"false" code.

```
ResponseCode ::= CHOICE {
    intCode     INTEGER,
    boolCode    BOOLEAN
}
```

5.3.4 The SEQUENCE OF type

This one is the list (array) of simple or constructed types:

```
-- Example 1
ManyIntegers ::= SEQUENCE OF INTEGER

-- Example 2
ManyRectangles ::= SEQUENCE OF Rectangle

-- More complex example:
-- an array of structures defined in place.
ManyCircles ::= SEQUENCE OF SEQUENCE {
                            radius INTEGER
                            }
```

5.3.5 The SET OF type

The SET OF type models the bag of structures. It resembles the SEQUENCE OF type, but the order is not important. The elements may arrive in the order which is not necessarily the same as the in-memory order on the remote machines.

```
-- A set of structures defined elsewhere
SetOfApples :: SET OF Apple

-- Set of integers encoding the kind of a fruit
FruitBag ::= SET OF ENUMERATED { apple, orange }
```

Bibliography

[ASN1C] The Open Source ASN.1 Compiler. http://lionet.info/asn1c

[AONL] Online ASN.1 Compiler. http://lionet.info/asn1c/asn1c.cgi

[Dub00] Olivier Dubuisson — *ASN.1 Communication between heterogeneous systems* — Morgan Kaufmann Publishers, 2000. http://asn1.elibel.tm.fr/en/book/. ISBN:0-12-6333361-0.

[ITU-T/ASN.1] ITU-T Study Group 17 — Languages for Telecommunication Systems http://www.itu.int/ITU-T/studygroups/com17/languages/