



ASN2TXT v2.4

User's Manual

The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement.

Copyright Notice

Copyright ©1997–2014 Objective Systems, Inc. All rights reserved.

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information

Comments, suggestions, and inquiries regarding this product may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

Revision History.....	7
Overview of ASN2TXT.....	9
Using ASN2TXT.....	11
Installation.....	11
Installing on a Windows System.....	11
Installing on a UNIX System.....	11
Command-line Options.....	12
Common Options.....	13
XML Options.....	15
CSV Options.....	16
Using the GUI.....	17
Trimming CSV Output.....	23
Configuration Files.....	25
Using the ASN2TXT DLL.....	27
Setup.....	27
Cleanup.....	27
Licensing.....	28
ASN.1 to XML Type Mappings.....	30
General Mapping without ASN.1 Schema Information.....	30
General Mapping with ASN.1 Schema Information.....	31
Specific ASN.1 Type to Value Mappings.....	32
ASN.1 to CSV Type Mappings.....	41
Mapping Top-Level Types.....	41
Mapping Simple Types.....	43
Mapping Complex Types.....	44
CHOICE.....	44
Simple SEQUENCES and SETs.....	45
Mapping Nested Types.....	45
SEQUENCE in a SEQUENCE.....	46
CHOICE in a SEQUENCE.....	46
SEQUENCE OF in a SEQUENCE.....	47
Data Conversion.....	47
SEQUENCE OF in a SEQUENCE.....	48
Other Nested Data Types.....	48
OPTIONAL and DEFAULT Elements.....	49

Revision History

- January 2014 – Initial release of ASN2TXT documentation (2.4.0).
- February 2014 – Augmented information regarding the use of the ASN2TXTDLL (2.4.1).

Overview of ASN2TXT

ASN2TXT is a command-line tool and embeddable library that translates ASN.1 data encoded in the Basic, Canonical, Distinguished, or Packed encoding rules into various text formats suitable for ingestion into spreadsheets, databases, or other text processing tools. At the time of initial release, ASN2TXT supports converting ASN.1 data to XML and comma-separated value (CSV) data formats.

Conversions to XML support both an Objective Systems custom format as well as the XML Encoding Rules standard as described in ITU-T standard X.693. Conversions from all ASN.1 binary encodings (BER, CER, DER, PER) are supported by ASN2TXT.

Conversions to CSV are done by a custom transformation, since no standard for converting ASN.1-encoded data to CSV exists. Conversions from BER, CER, and DER are supported at the time of this release; PER is not supported.

Users can run ASN2TXT as a command-line application or build their own applications using the shared library provided in the package. The library API is documented separately, but examples of its use are provided in this document as well.

Using ASN2TXT

Installation

ASN2TXT comes packaged as an executable installation program for Windows or a `.tar.gz` archive for UNIX systems. The package is comprised of the following directory tree:

```
asn2txt_v24x
|
+-asn1specs
|
+-bin
|
+-doc
|
+-sample
|
+-sample_per
```

The `bin` subdirectory contains the `asn2txt` executable. The `asn1specs` directory contains specifications used by the sample programs in the `sample` directory. This document is found in the `doc` directory.

Installing on a Windows System

To install ASN2TXT on a Windows system, simply double-click the executable installer program. Selecting the default installation options will install ASN2TXT in `c:\asn2txt_v23x`.

Installing on a UNIX System

To install ASN2TXT on a UNIX system, simply unzip and untar the `.tar.gz` archive. The program may be unpacked in any directory in which the user has permissions. No installation program is available to install ASN2TXT to `/usr/local` or other common installation paths, but it is not difficult to manually add links if needed.

Command-line Options

Invoking `asn2txt` without any options will show a usage message that contains the command-line options:

ASN2TXT, version 2.4.x

ASN.1 to text formatter

Copyright (c) 2013 Objective Systems, Inc. All Rights Reserved.

Usage: `asn2txt <input files> options`

`<input files>` ASN.1 message file name (wildcards are okay)

options:

`-xml` Output to XML
`-csv` Output to CSV

Common options:

`-schema <filename>` ASN.1 definition file name(s)
`-config <filename>` Configuration file name
`-I <directory>` Import ASN.1 files from `<directory>`
`-ber` Use basic encoding rules (BER)
`-pdu <typename>` Message PDU type name
`-bcdhandling <default|none|bcd|tbcd>`
Define handling of OCTET STRINGS declared to be
binary coded decimal (BCD) strings
`-oidformat <oidDefValue|namedNumbers|numbersOnly>`
Define format of Object Identifier display
`-noopentype` Disable automatic open type decoding
`-paddingbyte <hexbyte>` Additional padding byte
`-bitsfmt <hex|bin>` BIT STRING content output format
`-inputFileType <binary|hextext|base64>`
Format of data in input file
`-skip <num>` Skip `<num>` bytes between messages
`-headerOffset <num>` Skip the first `<num>` bytes in a data file
`-lickey <key>` License key to activate

XML options:

- per Use aligned packed encoding rules (PER)
- uper Use unaligned packed encoding rules (U-PER)
- xer Output XML in ASN.1 XER format
- o <filename> Output XML filename (use "<base>.xml" for batch output)
- ascii Print out ASCII for printable hex values
- emptyoptionals Insert empty XML elements in place of missing optional elements
- emptydefault Insert XML elements with default values in place of missing elements with default values
- nowhitespace Remove all whitespace between elements
- rootElement <element> Root Element Name
- skip-bad-records Skip messages which could not be decoded

CSV options:

- minLevel <num> Set the minimum output depth
- maxLevel <num> Set the maximum output depth
- outdir <directory> Specify the output directory
- noquotes Do not quote strings in output file
- padFields Pad fields with data that would otherwise be empty
- prefix Prefix output filenames with input filenames
- q Turn off all output except errors
- s <separator> Field separator
- seqsep <separator> SEQUENCE OF field separator (defaults to '|')
- separate-lines Output fields of a SEQUENCE OF on separate lines

The following sections summarize the command-line options.

Common Options

The following options are common to both CSV and XML transformations.

Option	Arguments	Description
-csv		Selects CSV output.
-xml		Selects XML output.

Option	Arguments	Description
<filename>		<filename> is the name of the input message to decode. This element is <i>required</i> . The use of wildcards (e.g.* and ?) is supported.
-schema	<filename>	This option is <i>required</i> when using CSV or decoding PER data. When converting BER data to XML, a schema is not required; ASN2TXT will convert the data using tag names.
-config	<filename>	This option allows an Obj-Sys configuration file to be used in translation. Configuration settings can be used to apply options to specific items within a schema.
-bcdhandling	<default none bcd tbcd>	Define handling of OCTET STRINGs declared to be binary coded decimal (BCD) strings. By default, types declared as BCD or TBCD strings will be translated as such. <none> forces translation to be performed as usual, while <bcd> and <tbcd> force their respective formatting on such OCTET STRINGs.
-bitsfmt	<hex bin>	-bitsfmt may be used to specify how BIT STRING items are formatted. By default they are expressed as hexadecimal strings; use bin to express them as binary strings instead.
-inputFileType	<binary hextext base64>	-inputFileType may be used to tell ASN2TXT how the input data are formatted. By default ASN2TXT will assume that the input data are binary, but it can also decode hexadecimal or base64 encoded data. Whitespace in the input is ignored when hextext is specified.
-lickey	<key>	In Linux, Macintosh, and Windows systems, license checking is performed by the Reprise License Manager. This option permits command-line license activation. This option is compiled out for operating systems that do not support the RLM license.
-noopentype		This option disables the conversion of open types in the output. This is the default behavior when converting BER to CSV.
-oidformat	<oidDefValue namedNumbers numbersOnly>	Define format of Object Identifier display. By default (or using <oidDefValue>), the value is displayed as it was defined. Using

Option	Arguments	Description
		<namedNumbers> or <numbersOnly>, the value is displayed as such (for example “iso(1)” or “1”, respectively) and includes as many arcs as possible.
-paddingbyte	<hexbyte>	<hexbyte> is the hexadecimal value of a padding byte that may appear in the input message. Call data records (CDRs) are commonly continuously dumped to files by telephony equipment. If no information is available, the records are often padded by 0x00 or 0xFF bytes. The default padding byte is 0x00. <hexbyte> may be formatted with or without a 0x prefix.
-pdu	<typename>	<typename> is the name of the PDU data type to be decoded. This option is necessary when the top-level data type is ambiguous. It is also required when converting PER data.

XML Options

The following options can be used when converting to XML.

Option	Arguments	Description
-ber		Selects the use of Basic Encoding Rules for decoding.
-per		Selects the use of the Packed Encoding Rules (aligned) for decoding.
-uper		Selects the use of the Packed Encoding Rules (unaligned) for decoding.
-ascii		If all bytes in a decoded value are within the ASCII character range, display as standard text. Otherwise display as formatted hexadecimal text. This option only has meaning if BER data is decoded without a schema file.
-emptyDefault		Insert an element with a default value as specified in the schema at the location of a missing element in the instance.
-emptyOptionals		Insert an empty element at the location of a missing element in the schema that was declared to be optional.
-nowhitespace		Do not generate any whitespace (blanks and newline

Option	Arguments	Description
		characters) between elements. This makes the generated XML document more compact at the expense of readability.
-o	<filename>	Specify the output XML <filename> instead of writing output to standard out. Set <filename> to “<base>.xml” to specify batch output when converting multiple files.
-rootElement	<name>	Specify the root element <name> used to wrap the entire XML message at the outer level. This makes it possible to create an XML document for an ASN.1 file containing multiple individually encoded binary messages (a common feature of many Call Detail Record ASN.1 formats).
-skip-bad-records		This option enables more thorough detection of badly formed records and attempts to skip such records. This can occur if an unrecognized tag is encountered, for example. In some cases, it is impossible to continue translation after a bad record, such as when an incorrect length value was encoded.

CSV Options

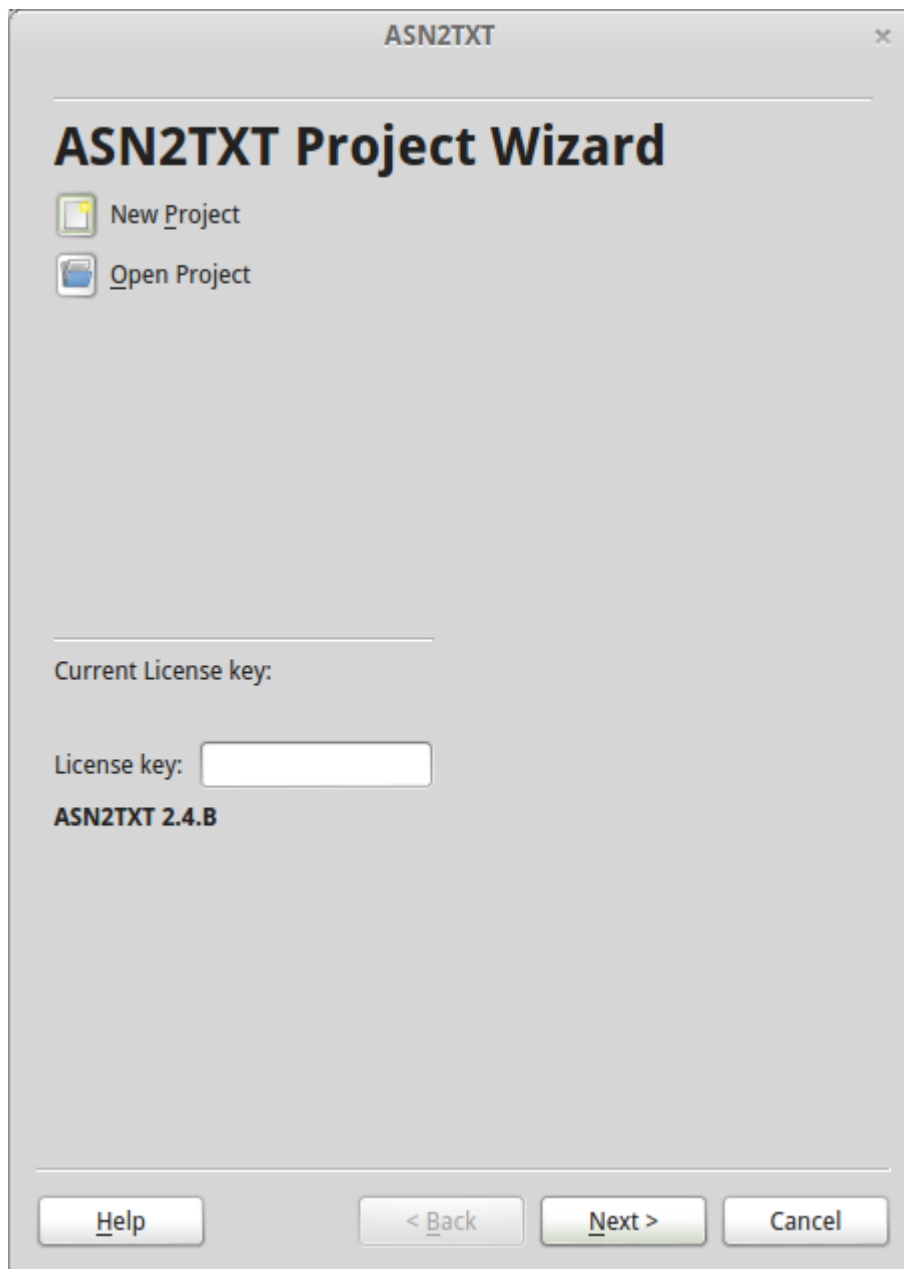
The following options can be used when converting to CSV.

Option	Arguments	Description
-maxLevel	<level>	By default, all entries in the input file will be dumped to the output file. Deeply nested types may result in excessive output, however. The -maxLevel switch causes ASN2TXT to skip outputting data after <level> levels have been processed.
-minLevel	<level>	The -minLevel option causes ASN2TXT to unwrap top-level data types <level> levels deep.
-noquotes		By default, ASN2TXT will quote all of the fields to ensure that they are processed as text by spreadsheet programs to avoid converting numeric fields into scientific notation. Using this option suppresses that behavior.
-outdir	<output directory>	Specifies the directory to which ASN2TXT will output the resulting CSV files.

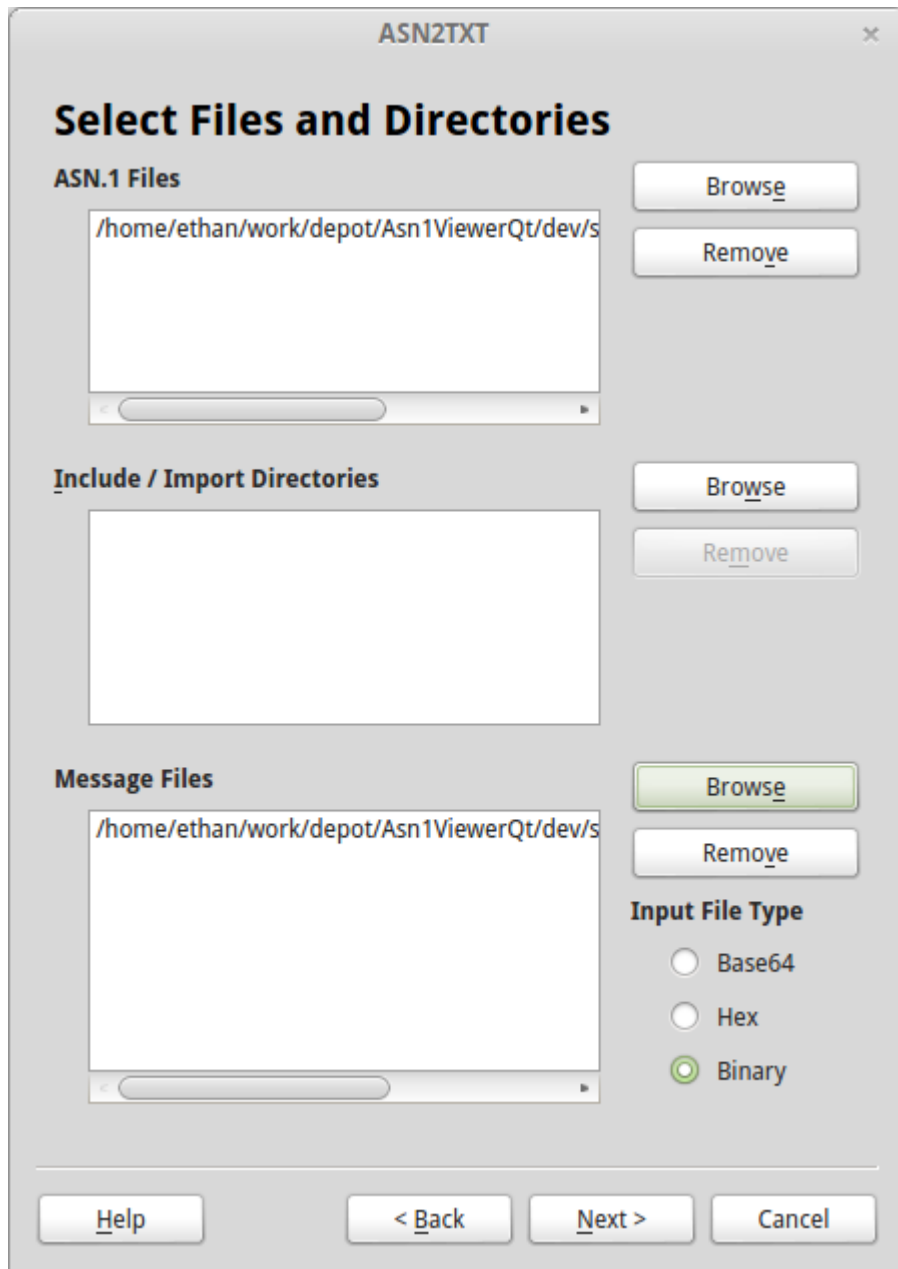
Option	Arguments	Description
-padfields		ASN2TXT will omit fields that would normally be duplicated in output files. Using this option will output these fields. It produces larger files, but is more explicit and may simplify ingesting the data.
-prefix		ASN2TXT will normally output all records to a predetermined filename based on the module name and PDU. When the -prefix option is selected, the output filenames will be prefixed with the input message filename.
-q		This option causes ASN2TXT to operate in a “quiet” mode more suitable for batch processing. Informational messages are limited and only error output will be reported.
-s	<separator>	By default, ASN2TXT assumes the record separator will be a comma. When this conflicts with output data (e.g., a field may consist of City, State), users may use the -s switch to specify a different separator such as a tab or a pipe. Enclosing the separator in quotation marks is necessary when using a tab or other whitespace character.
-seqsep	<separator>	ASN2TXT will separate elements in a SEQUENCE OF type using a vertical bar (or pipe) by default. This option allows users to adjust the separator in case it conflicts with their expected output data format.
-separate-lines		ASN2TXT places elements in a SEQUENCE OF type on the same line if they are primitive types; this conserves space in the output file and makes it easier to process the results with text processing tools. This option ensures that all SEQUENCE OF elements regardless of type will be outputted on separate lines.

Using the GUI

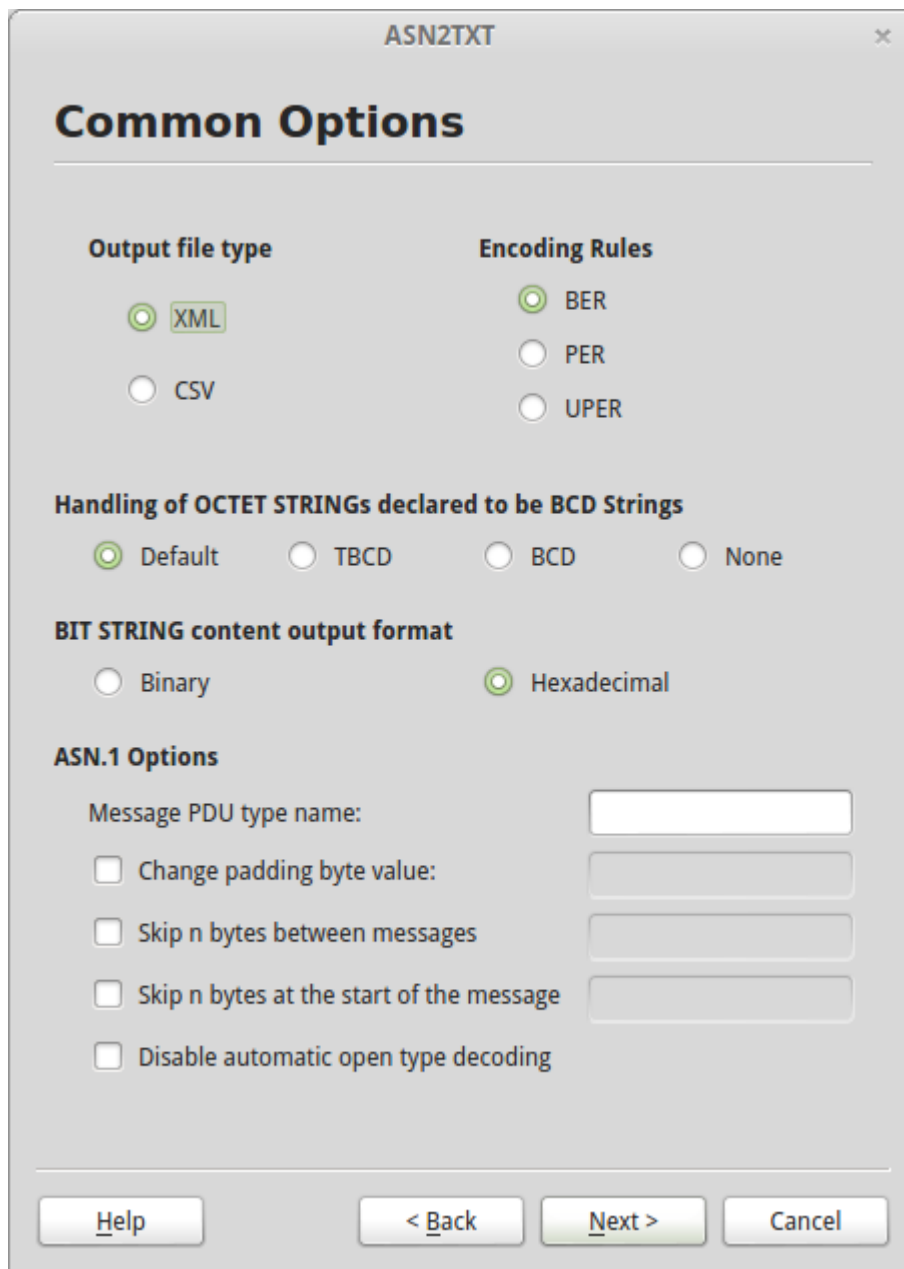
ASN2TXT is equipped with a graphical user interface designed to allow users to set common options and create projects for commonly-used files and specifications.



The initial screen contains options for users who wish to open or create a new project, but this is not necessary to use the software. The next screen is used to select the input specifications and message data used for decoding.

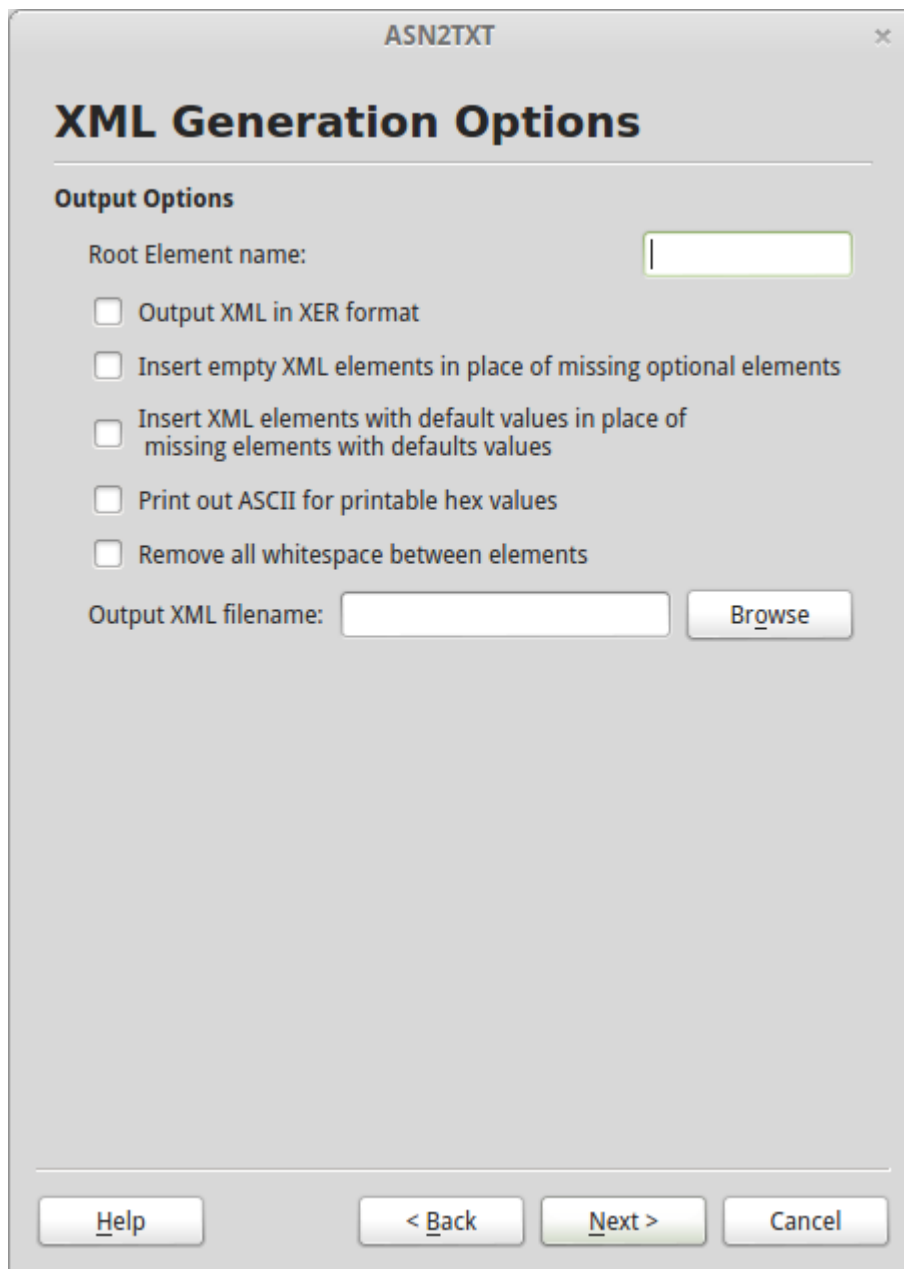


The next button in the file selection window will not activate until both the input message and input specification have been provided. After this is done, the following common options can be set:



When selecting CSV output, the GUI will automatically disable the PER input options and open type decoding. Conversions to CSV do not support either PER or open type decoding at this time.

If XML is the selected output format, the following screen will appear:



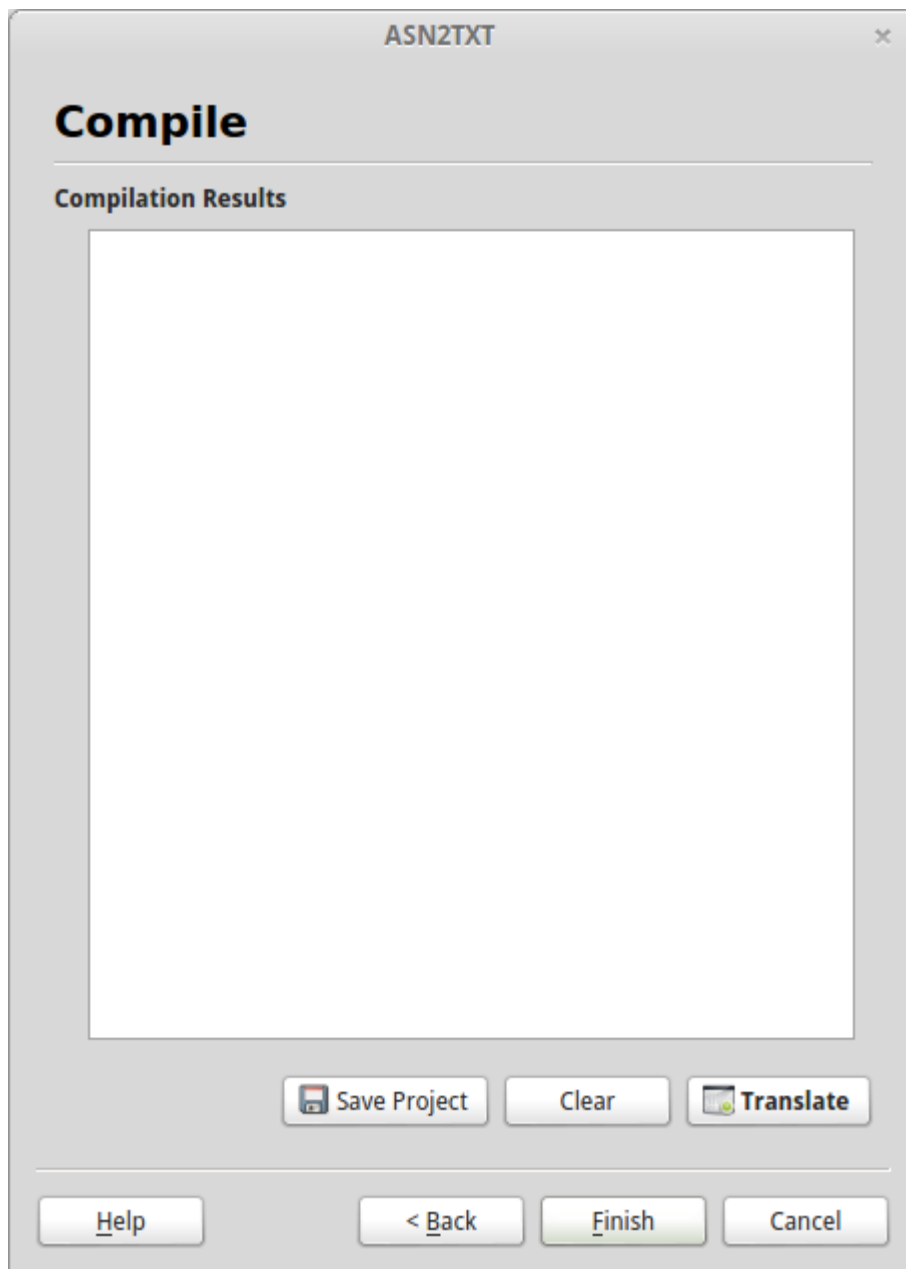
Users have two options for how to direct their XML output: it can be dumped to the GUI directly or else to a file. When the XML output filename is not provided, users will see the decoded XML output in the compilation window.

If output to CSV is requested, the following screen will appear instead:

The image shows a dialog box titled "ASN2TXT" with a close button (X) in the top right corner. The main heading is "CSV Generation Options". Below this, there is a section titled "Output Options" which contains five text input fields: "Field Separator", "Sequence of Field Separator", "Minimum Output Level", "Maximum Output Level", and "Output Directory". Below these fields are four unchecked checkboxes with the following labels: "Do not quote fields in the output files", "Pad fields that otherwise would be empty", "Add a prefix to the output files", and "Emit SEQUENCE OF items on separate lines". At the bottom of the dialog box, there are four buttons: "Help", "< Back", "Next >", and "Cancel".

Unlike XML output, CSV output is always directed to a file (or, more likely, several files). The output in the compilation window is therefore a little different than what is seen when XML output is selected without an output filename. This behavior is normal.

When the appropriate options for either type have been selected, the following screen is presented for compilation:



Clicking the “Translate” button will invoke ASN2TXT's translator. Output may be directed to the compilation results window, to a file, or to several files depending on the choices made in previous windows.

Trimming CSV Output

CSV output can often be more verbose than is needed for many applications, so users may wish to isolate or trim the CSV output using the `-minLevel` and `-maxLevel` options. TAP parsing provides a common example: the `TransferBatch` type, a `SEQUENCE`, contains several different elements that we

may wish to separate to make it easier to ingest the content into a database. The use of the `-minLevel` option ensures that the outer types are “unwrapped” such that the constituent elements are dumped to their own files.

An example run of the `asn2txt` command line tool might look like this:

```
$ asn2txt [filename] -schema tap3.asn1 -csv
ASN2TXT, version 2.4.0
ASN.1 to text formatter
Copyright (c) 2012-2013 Objective Systems, Inc. All Rights Reserved.
Processing [filename]
Decoded 1 messages.
Files written:
TAP-0311_transferBatch.csv
```

The results may be difficult to adequately process since the transfer batch will contain audit control information, control information, and other features. Use of the `-minLevel` switch helps to fix this problem:

```
$ asn2txt [filename] -schema tap3.asn1 -csv -minLevel 3
ASN2TXT, version 2.4.0
ASN.1 to text formatter
Copyright (c) 2012-2013 Objective Systems, Inc. All Rights Reserved.
Processing [filename]
Decoded 1 messages.
Files written:
TAP-0311_TransferBatch_batchControlInfo.csv
TAP-0311_TransferBatch_networkInfo.csv
TAP-0311_TransferBatch_callEventDetails.csv
TAP-0311_TransferBatch_auditControlInfo.csv
```

In this way, the outer layers of data can be unwrapped and the contents put into individual files. The use of `maxLevel` can help to trim data on the other side so that users can focus on narrower sets of data that don't consume as much space.

Configuration Files

ASN2TXT provides the option of including a configuration file during the translation. This allows the user to set certain options for specific items in the schema. Currently, only the `displayFormat` option is available, which can be used at either the production or element level.

Configuration files are formatted as XML and use the top-level tag `<asn1config>`. Below the top level, `<module>`, `<production>`, and `<element>` tags can be nested, one within the other. At each level, the tag must include the “name” attribute.

So for example, given a schema like this:

```
MyData DEFINITIONS ::= BEGIN

    IPAddress ::= OCTET STRING (SIZE (4))

    TwoIPAddresses ::= SEQUENCE {
        ipAddress1    IPAddress,
        ipAddress2    IPAddress
    }

END
```

A configuration file might look like this:

```
<asn1config>
  <module name="MyData">
    <production name="TwoIPAddresses">
      <element name="ipAddress1">
        <displayFormat>ipv4</displayFormat>
      </element>
    </production>
  </module>
</asn1config>
```

In this case, the configuration specifies `displayFormat` at the element level for the `ipAddress1` element. Note that the `<element>` tag must be nested within a `<production>` tag (which must also be nested within a `<module>` tag). Then, whenever a `TwoIPAddresses` is translated, it would be output like this (in XML, for example):

```
<message>
  <!-- message 1 -->

  <TwoIPAddresses>
    <ipAddress1>123.45.67.89</ipAddress1>
    <ipAddress2>7B2D4359</ipAddress2>
  </TwoIPAddresses>
</message>
```

Since the `displayFormat` option was set at the element level and only for `ipAddress1`, only that element is affected. In order to apply such formatting for all `IPAddress` types, the option can be set at the production level like so:

```
<asn1config>
  <module name="MyData">
    <production name="IPAddress">
      <displayFormat>ipv4</displayFormat>
    </production>
  </module>
</asn1config>
```

Note that in this case, the `<production>` tag's `name` attribute specifies the type which will be affected, rather than the type containing the affected element, as above.

Using the ASN2TXT DLL

Users who wish to create their own applications for converting ASN.1 binary formats into text may use the ASN2TXT DLL, which is included in the package as a dynamic library (`asn2txt.dll` or `libasn2txt.so`). Supporting header files are included. The API described by these files is documented in the `doc` subdirectory of the installation, and an annotated sample is provided in `sample_dll`.

Setup

The DLL may be used from a C++ program by including the `Asn2Text.h` file and instantiating the `Asn2Text` class. The application must be linked against the DLL, and users may need to set various system variables in order to find the library. (Windows users can put the library on the system `PATH` or in the same directory as their application, and UNIX users can set the `LD_LIBRARY_PATH` environment variable.)

A small program might look like this:

```
#include "Asn2Text.h"
#include <cstdio>
using namespace std;

int main(void) {
    Asn2Text *asn2text = new Asn2Text();
    printf ("ASN2TEXT object was created.\n");
    delete asn2text;
}
```

This program doesn't actually do anything, so it isn't very useful; but it does illustrate how the object should be created and deleted.

Cleanup

The DLL is responsible for allocating memory for some of its conversion tasks, and it may be necessary to force it to release memory and reset its internal state when performing multiple operations.

This is especially pertinent for GUI-driven applications that link against the DLL, where care must be exercised to isolate translations from each other. In these cases, a call to the `cleanup` method is required:

```
asn2txt->cleanup();
```

If memory leaks are observed, users should ensure that the `cleanup` method is being properly called during the application execution.

The API documentation clearly indicates those rare circumstances where users will be expected to free memory that is allocated by the DLL.

Licensing

Like ASN2TXT, the DLL must have access to a valid license in order for applications linked against it to run properly. The ASN2TXT application uses the `-lickey` command-line option to set the license key, which produces a file called `a2t_act.lic`. This file (or one provided by Objective Systems) can be copied into one of several places:

1. To a directory on the system-wide `PATH`.
2. To the directory indicated by the `RLM_LICENSE` environment variable.
3. To the directory from which the application is being executed.

The licensing API is documented in the accompanying license documentation, but of special note is the `setExeName` method. Invoking this method is only necessary when users wish to call their applications from an absolute path that differs from the current one.

We illustrate by way of a short example:

```
#include "Asn2Text.h"

int main (int argc, char **argv) {
    Asn2Text *asn2txt = new Asn2Text();
    asn2txt->setExeName (argv[0]);
}
```

The first command-line parameter (i.e., `argv[0]`) is provided to tell the Reprise License modules where they can look for an appropriate license file.

It's generally fail safe to set the `RLM_LICENSE` environment variable, and users are recommended to use that method if problems arise.

ASN.1 to XML Type Mappings

This chapter describes the mapping between ASN.1 encoded data values and XML for each of the ASN.1 types defined in the X.680 standard.

General Mapping without ASN.1 Schema Information

A BER, DER, or CER encoded data stream may be translated to XML format without providing associated ASN.1 schema information. In this case, XML element names are derived from built-in ASN.1 tag information contained within the message and values are encoded as either hexadecimal text, ASCII text, or as specific data-typed values if universal tag information is present.

XML element names derived from ASN.1 tag names for all tags except known universal tags is in the following general form:

`<TagClass_TagValue>`

where TagClass is the tag class name (APPLICATION, CONTEXT, or PRIVATE) and TagValue is the numeric tag value. For example, an [APPLICATION 1] tag would be printed as `<APPLICATION_1>` and a [0] tag (context-specific zero) would be printed as `<CONTEXT_0>`.

In the case of known universal tags, the tag value is derived using the name of the known type. In general, this is the type name defined in the ASN.1 standard with an underscore character used in place of embedded whitespace if it exists. The following table shows the XML tag names for the known types:

Tag	XML Element Name
UNIVERSAL 1	BOOLEAN
UNIVERSAL 2	INTEGER
UNIVERSAL 3	BIT_STRING
UNIVERSAL 4	OCTET_STRING
UNIVERSAL 5	NULL
UNIVERSAL 6	OBJECT_IDENTIFIER
UNIVERSAL 7	OBJECT_DESCRIPTOR
UNIVERSAL 8	EXTERNAL

Tag	XML Element Name
UNIVERSAL 9	REAL
UNIVERSAL 10	ENUMERATED
UNIVERSAL 12	EMBEDDED_PDV
UNIVERSAL 13	RELATIVE_OID
UNIVERSAL 16	SEQUENCE
UNIVERSAL 17	SET
UNIVERSAL 18-22, 25-30	Character string
UNIVERSAL 23	UTCTIME
UNIVERSAL 24	GENERALIZEDTIME

Element content will be formatted in one of three ways: hexadecimal text, ASCII (character) text, or specific-typed value.

Hexadecimal text is the default format for untyped content. ASCII text will be used if the `-ascii` command-line switch is specified and all byte values within a particular field are determined to be printable ASCII characters. A specific-type value encoding will be done if a known universal tag is found. The mapping in this case will be as described in the "Specific ASN.1 Type to XML Value Mapping" section below.

General Mapping with ASN.1 Schema Information

ASN.1 schema information is used if one or more ASN.1 schema files are specified on the command-line using the `-schema` command-line switch. In this case, element names as specified in the schema file are used for the XML element names and the content is decoded based on the specific type.

It is possible to use the `-pdu` command-line switch to force the association of a type within the specification to the message. This is only necessary if the ASN.1 files contain multiple types with the same start tag as the message type. Otherwise, the program will be able to determine on its own which type to use by matching tags. This is true for BER/DER/CER messages only: for PER, it is necessary to specify the PDU type along with the schema.

Specific ASN.1 Type to Value Mappings

This section defines the type-to-value mapping for each of the specific ASN.1 types. By default, these mappings are not in the form defined in the ASN.1 XML Encoding Rules (XER) standard (ITU-T X.693).

When a schema is provided using the `-schema` option, the output may be adjusted to conform to XER if desired by using the `-xer` option. XER is more verbose and less validation-friendly than our native XML export. It is provided for those occasions when strict conformance is required. Differences between the two formats are provided along with the schemaless mappings below.

BOOLEAN. An ASN.1 boolean value is transformed into the keyword 'true' or 'false'. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a `<BOOLEAN>` tag is added.

b BOOLEAN ::= TRUE	
Schemaless	<code><BOOLEAN>TRUE</BOOLEAN></code>
XML Mode	<code>true</code>
XER Mode	<code><TRUE/></code>

INTEGER. An ASN.1 integer value is transformed into numeric text. The one exception to this rule is if named number identifiers are specified for the integer type. In this case, if the number matches one of the declared identifiers, the identifier text is used.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an `<INTEGER>` tag is added.

i INTEGER ::= 35	
Schemaless	<code><INTEGER>35</INTEGER></code>
With schema	<code><i>35</i></code>

ENUMERATED. An ASN.1 enumerated value is transformed into the enumerated identifier text value. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an `<ENUMERATED>` tag is added.

colors ENUMERATED {r, g, b} ::= g	
Schemaless	<ENUMERATED>1</ENUMERATED>
XML Mode	<colors>g</colors>
XER Mode	<colors><g/></colors>

BIT STRING. An ASN.1 bit string value is transformed into one of three forms:

1. Binary Text (0's and 1's)
2. Hexadecimal text
3. Named bit identifiers

Binary text is the default output format. This is used if the bit string type contains no named bit identifiers and if specification of hexadecimal output was not specified on the `asn2txt` command-line.

Hexadecimal text is displayed when the `-bitsfmt hex` command-line option is used. Any unused bits in the last octet are set to zero. Note that the other bits are displayed in most-significant bit order as they appear in the string in the last byte (i.e., they are not right shifted). For example, if the last byte contains a bit string value of 1010xxxx (where x denotes an unused bit), the string is displayed as A0 in the XML output, not 0A.

Named bit identifiers are used in the case of a bit string declared with identifiers. In this case, the XML content is a space-separated list of identifier values corresponding to the bits that are set. It is assumed that bits in the string all have corresponding identifier values.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <BIT_STRING> tag is added.

bs BIT STRING {z(0), a(1), b(2), c(3)} ::= '100100'B	
Schemaless	<BIT_STRING>100100</BIT_STRING>
With Schema	<bs>100100</bs>

OCTET STRING. An ASN.1 octet string value is transformed into one of two forms:

1. Hexadecimal text

2. ASCII character text

Hexadecimal text is the default display type. ASCII text will be used for the content when the `ascii` command-line option is used and the field contains are printable ASCII characters. A special case of OCTET STRING handling is for declared binary-coded decimal (BCD) data types. This is discussed in a later section.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a `<OCTET_STRING>` tag is added.

os OCTET STRING ::= '3031'H	
Schemaless	<code><OCTET_STRING>3031</OCTET_STRING></code>
With schema	<code><os>3031</os></code>
With -ascii	<code><os>01</os></code>

NULL. An ASN.1 null value is displayed as an empty XML element. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a `<NULL>` tag is added.

n NULL ::= NULL	
Schemaless	<code><NULL/></code>
XML Mode	<code><n/></code>
XER Mode	<code><n><NULL/></n></code>

OBJECT IDENTIFIER and RELATIVE OID. An ASN.1 object identifier value is mapped into space-separated list of identifiers in numeric and/or named-number format. The identifiers are enclosed in curly braces (`{ }`). Numeric identifiers are simply numbers. The named-number format is a textual identifier followed by the corresponding numeric identifier in parentheses. It is used in cases where the identifier can be determined from the schema or is a well known identifier as specified in the ASN.1 standard.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, an `<OBJECT_IDENTIFIER>` tag is added.

oid OBJECT IDENTIFIER ::= { 1 2 840 113549 1 1 2 }

Schemaless	<OBJECTIDENTIFIER>{1 2 840 113549 1 1 2} </OBJECTIDENTIFIER>
With schema	<oid>{ 1 2 840 113549 1 1 2 } </oid>

The mapping for RELATIVE OID is the same as that for OBJECT IDENTIFIER.

Character String. An ASN.1 value of any of the known character string types is transformed into the character string text in whatever the default encoding for that type is. For example, an IA5String would contain an ASCII text value whereas a BMPString would contain a Unicode value.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a tag is added which is the name of the character string type as defined in the ASN.1 standard in angle brackets. For example, the default tag for a UTF8String type would be <UTF8String>.

str UTF8String ::= "testing"	
Schemaless	<UTF8String>TRUE</UTF8String>
With schema	<str>testing</str>

Binary-coded Decimal String. Binary-Coded Decimal (BCD) strings and Telephony Binary-Coded Decimal (TBCD) strings are not part of the ASN.1 standard, but their use is prevalent in many telephony-related ASN.1 specifications. Conversion of these types into standard numeric text strings is supported.

In general, BCD strings pack two numeric digits into a single byte value by using a four-bit nibble to hold each digit. By convention, the digits are reversed in TBCD strings, but there are no official standards for this encoding.

The `-bcdhandling` command-line option can be used to force a certain type of conversion if an encoding does not follow the usual conventions. The default handling is to reverse digits in strings determined to be TBCD strings and not reverse digits in BCD strings. The `bcd` option is used for no reversal of digits in all of these types of strings. The `tbcd` option instructs ASN2TXT to reverse the digits for all BCD strings.

If no processing is desired, `-bcdhandling none` can be used to instruct ASN2TXT not to process these strings.

REAL. An ASN.1 real value is transformed into numeric text in exponential number format. If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <REAL> tag is added.

r REAL ::= 137.035999074	
Schemaless	<REAL>137.035999074</REAL>
With schema	<r>137.035999074</r>

SEQUENCE and SET. An ASN.1 sequence value is transformed into an XML value containing an element wrapper with each of the XML element encoded values inside.

name ::= SEQUENCE { first UTF8String, middle UTF8String OPTIONAL, last UTF8String }	name Name ::= { first "Joe", last "Jones" }
Schemaless	<SEQUENCE> <CONTEXT_0> <UTF8String>Joe</UTF8String> </CONTEXT_0> <CONTEXT_2> <UTF8String>Jones</UTF8String> </CONTEXT_2> </SEQUENCE>
With schema	<name> <first>Joe</first> <last>Jones</last> </name>
With -emptyOptionals	<name> <first>Joe</first> <middle/> <last>Jones</last> </name>

When a SET is used instead, the outer SEQUENCE tag is replaced with SET. The mappings are otherwise identical.

SEQUENCE OF / SET OF. The representation of a repeating value in XML varies depending on the type of the element value.

If the value being translated is a sequence of an atomic primitive type, the XML content is a space-separated list of values. The definition of "atomic primitive type" is any primitive type whose value may not contain embedded whitespace. This includes BOOLEAN, INTEGER, ENUMERATED, REAL, BIT STRING, and OCTET STRING values.

If the value being translated is a constructed type or if it may contain whitespace, the value is wrapped in a tag which is either the name of the encapsulating type (defined or built-in) or the SEQUENCE OF element name if this form of the type was used.

If BER/DER/CER data is being decoded without a schema and the universal tag for this type is parsed, a <SEQUENCE> or <SET> tag is added. That is because the tag value (hex 30 or 31) is the same for SEQUENCE OF or SET OF as it is for SEQUENCE or SET.

<code>soi SEQUENCE OF INTEGER ::= {1, 2, 3}</code>	
Schemaless	<pre><SEQUENCE> <INTEGER>1</INTEGER> <INTEGER>2</INTEGER> <INTEGER>3</INTEGER> </SEQUENCE></pre>
With schema	<pre><soi> <INTEGER>1</INTEGER> <INTEGER>2</INTEGER> <INTEGER>3</INTEGER> </soi></pre>

<code>sos SEQUENCE OF UTF8String ::= { "test 1", "test 2" }</code>	
Schemaless	<pre><SEQUENCE> <UTF8STRING>test 1</UTF8STRING> <UTF8STRING>test 2</UTF8STRING> </SEQUENCE></pre>

With schema	<pre><sos> <UTF8String>test 1</UTF8String> <UTF8String>test 2</UTF8String> </sos></pre>
-------------	---

<pre>Name ::= SEQUENCE { first UTF8String, middle UTF8String OPTIONAL, last UTF8String }</pre>	<pre>son SEQUENCE OF Name ::= { { first 'Joe', last 'Jones' }, { first 'John', middle 'P', last 'Smith' } }</pre>
--	---

Schemaless	<pre><SEQUENCE> <SEQUENCE> <UTF8STRING>Joe</UTF8STRING> <UTF8STRING>Jones</UTF8STRING> </SEQUENCE> <SEQUENCE> <UTF8STRING>John</UTF8STRING> <UTF8STRING>P</UTF8STRING> <UTF8STRING>Smith</UTF8STRING> </SEQUENCE></pre>
------------	---

<p>With schema. This example shows the results with <code>-emptyOptionals</code> selected. If it were not, the first <code><middle/></code> element would be omitted.</p>	<pre><son> <Name> <first>Joe</first> <middle/> <last>Jones</last> </Name> <Name> <first>John</first> <middle>P</middle> <last>Smith</last> </Name> </son></pre>
---	---

CHOICE. The mapping of an ASN.1 CHOICE value is the alternative element tag followed by the value translated to XML format.

<pre> C CHOICE ::= { a INTEGER, b OCTET STRING, c UTF8String } </pre>	<pre> c C ::= { a : 42 } </pre>
Schemaless	<INTEGER>42</INTEGER>
With schema	<pre> <C> <a>42 </C> </pre>

Open Type. The mapping of an ASN.1 open type value depends on whether the actual type used to represent the value can be determined. ASN2TXT attempts to determine the actual type using the following methods (in this order):

1. Table constraints
2. Tag lookup in all defined schema types (BER/DER/CER only)
3. Universal tag lookup (BER/DER/CER only)

If the type can be determined, an XML element tag containing the type name is first added followed by the translated content of the value.

If the type cannot be determined, the open type content is translated into hexadecimal text from of the encoded value. This will also be done if the `-noopentype` command-line switch is used.

As an example, consider the `AlgorithmIdentifier` type used in the `AuthenticationFramework` and other related security specifications:

```

AlgorithmIdentifier ::= SEQUENCE {
  algorithm ALGORITHM.&id({SupportedAlgorithms}),
  parameters ALGORITHM.&Type({SupportedAlgorithms}{@algorithm})
  OPTIONAL
}

```

In this case, the `parameters` element references an open type that is tied to a type value based on the value of the `algorithm` key. Without getting into the details of the use of the accompanying information object sets, it is known that for an `algorithm` value of object identifier { 1 2 840 113549 1 1 2

}, the type of the parameters field is NULL (i.e. there are no associated parameters). The XML translation in this case will be the following:

```
<algorithm>{ 1 2 840 113549 1 1 2 }</algorithm>  
<parameters>  
  <NULL/>  
</parameters>
```


ASN.1 to CSV Type Mappings

Converting ASN.1 types to CSV output is not always very straightforward. It is akin to normalizing a database, except that there is only one table. For complex types, it is necessary to duplicate information across several rows.

There exists no standard for converting ASN.1 data to CSV. BER, CER, and DER data are encoded in a hierarchical format that lends itself to translation to other hierarchical formats such as XML. CSV, on the other hand, is flat data format: there are no structured types or children, and all data in a CSV file are displayed on single lines. This complicates the translation of ASN.1 to CSV, since structured data types like SEQUENCES can be nested to an arbitrary depth or repeated an arbitrary number of times.

While these limitations make conversion a difficult problem, CSV offers some advantages over XML. CSV files are usually considerably smaller than XML, since no markup is necessary to distinguish elements. Many databases import CSV data directly into tables, so no intermediate transformations are required. CSV files can be easier to manipulate procedurally; no external XML parsers are required to read the files, and many scripting languages have built-in facilities for working with comma-delimited data.

We may divide conversion into roughly two steps: collecting the column headers and then outputting the column data. Header information comes from parsing the input specification, while the column data are found in the actual encoded content. This documentation is primarily concerned with how the column headers are collected.

Mapping Top-Level Types

PDU data types are stored in their own CSV files, usually in the form of `ModuleName_ProductionName.csv`. There are three main top-level data types of interest:

1. SEQUENCE / SEQUENCE OF
2. SET / SET OF
3. CHOICE

The list types (SEQUENCE and SET OF) are the same as the unit types. The content is repeated when needed on multiple rows of the CSV file.

Simple types may be used as top-level data types, but in practice this is rare. Translation in this case proceeds as described in the following sections.

As an example, the following SEQUENCE would be dumped to `MyModule_Type1.csv`:

```
MyModule DEFINITIONS ::= BEGIN

Type1 ::= SEQUENCE {
    ...
}

END
```

If the input file type had two such SEQUENCES, the resulting files would be `MyModule_Type1.csv` and `MyModule_Type2.csv`.

When a CHOICE is used as the top-level data type, the typename for the CHOICE is ignored and the files are generated using the typenames in the CHOICE. For example, the following specification would generate the same output as the one with two top-level SEQUENCES named `Type1` and `Type2`:

```
MyModule DEFINITIONS AUTOMATIC TAGS ::= BEGIN

Type1 ::= SEQUENCE {
    ...
}

Type2 ::= SEQUENCE {
    ...
}

PDU ::= CHOICE {
    t1 Type1,
    t2 Type2
}
```

When a SEQUENCE or SET OF type is used as the top level, the underlying unit type is referenced instead. For example, the following ASN.1 specification would create the file `MyModule_Type1.csv`:

```
MyModule DEFINITIONS ::= BEGIN

Type1 ::= SEQUENCE {
    ...
}

PDU ::= SEQUENCE OF Type1

END
```

In this case, the PDU type carries no extra information for outputting the data; the contents of `Type1` are outputted on separate lines.

One of the implications of this kind of translation is that the message structure cannot be reconstructed from the output data files. A top-level data type of a CHOICE, SEQUENCE, or SEQUENCE OF may result in exactly the same output files, even though the bytes of the message may differ. Such ambiguity should not cause any problems since a specification is required for decoding the ASN.1 data.

Mapping Simple Types

Simple types in ASN.1 consist of the following:

- BOOLEAN
- INTEGER
- BIT STRING
- OCTET STRING
- NULL
- OBJECT IDENTIFIER
- REAL
- ENUMERATED
- Character strings
- RELATIVE-OID
- UTCTime

- GeneralizedTime
- GraphicString
- VisibleString
- GeneralString

Each simple type is mapped to a corresponding string representation of the input data. This is a relatively straightforward conversion. Of special note, we use the BOOLEAN values "TRUE" (for any hex octet not equal to 0x00) and "FALSE" (for any hex octet equal to 0x00). NULL values are outputted simply as "NULL."

Simple type mappings require no extra logic for output. Their textual representations are generally quite straightforward. Mapping complex types, however, is more difficult.

Mapping Complex Types

Complex types of interest include the following:

- SEQUENCE
- SEQUENCE OF
- SET
- SET OF
- CHOICE

Complex types are more difficult to transform than simple types. They can be self-referential and nested, which complicates transformation. CSV is a flat file format that cannot properly represent nested types in a fixed number of columns, so care must be taken in transforming the data to ensure that it is properly represented. This process is very similar to a first-order database normalization.

CHOICE

As explained in the previous section, the CHOICE at the top level is effectively ignored: the elements of the CHOICE are used to generate the output of a file instead. In the routine case where the CHOICE is contained in another data type or stands alone, the mapping is slightly different.

Take for example the following CHOICE:

```
C ::= CHOICE {  
    i INTEGER,  
    b BOOLEAN,  
    s UTF8String  
}
```

The elements contained in the CHOICE will be used as the column names. The name of the CHOICE itself will be ignored. The resulting column names from this example would look like this:

```
i,b,s
```

Simple SEQUENCES and SETs

This section describes the transformation of SEQUENCE data types. The SET data type is analogous to the SEQUENCE. The SEQUENCE OF and SET OF types are likewise equivalent.

The only significant difference between SEQUENCE and SET is that elements may be encoded in any order in a SET. ASN2TXT will order SET elements in the order they appear in the specification. The SEQUENCES considered in this section contain only simple types to simplify the collection of header data. Other cases are considered in the next sections.

Take, for example, the following SEQUENCE specification:

```
S ::= SEQUENCE {  
    i INTEGER ,  
    s UTF8String,  
    b BIT STRING  
}
```

Each element of the SEQUENCE will be represented by an item in the output CSV file as follows:

```
i,s,b
```

Mapping Nested Types

When a SEQUENCE or SET contains other complex data types, it is said to be *nested*. Types may be nested to an arbitrary depth in ASN.1, so the resulting output can be extremely verbose in complex specifications. Moreover, these nested types can be repeating. The following sections describe how ASN2TXT handles nested types. A SEQUENCE is exactly the same as a SET to ASN2TXT; the two types are used interchangeably in the following sections.

SEQUENCE in a SEQUENCE

One form of nested data occurs when a SEQUENCE type contains another, as in the following example:

```
A ::= SEQUENCE {
  a INTEGER,
  b SEQUENCE { aa INTEGER, bb BOOLEAN },
  c BIT STRING
}
```

In this case, the following columns would be generated in the output CSV:

```
a, aa, bb, c
```

ASN2TXT removes all references to the SEQUENCE named b. Instead, the inner data (aa and bb) is collapsed into the main data type. It is as though we have instead provided the following specification:

```
A ::= SEQUENCE {
  a INTEGER,
  aa INTEGER,
  bb BOOLEAN,
  b BIT STRING
}
```

While the BER encoding of the two specifications is different, they are functionally equivalent to ASN2TXT.

CHOICE in a SEQUENCE

When a CHOICE appears in a SEQUENCE, each of the elements in the CHOICE is represented in the output CSV file, even though only one will be selected in any given message.

For example, take the following specification:

```
A ::= SEQUENCE {
  a INTEGER,
  b CHOICE { aa INTEGER, bb BOOLEAN },
  c BIT STRING
}
```

The resulting columns will appear as though the CHOICE were actually a SEQUENCE:

```
a, aa, bb, c
```

SEQUENCE OF in a SEQUENCE

The last data type to consider is the SEQUENCE OF. This is handled very much like a SEQUENCE: the SEQUENCE OF is ignored and its contents are represented for the column headers as in the following example:

```
A ::= SEQUENCE {
  a INTEGER,
  b SEQUENCE OF INTEGER,
  c BIT STRING
}
```

In this case, the columns will be straightforwardly translated:

```
a,b,c
```

It is possible that the repeated data type is not primitive, but rather complex. For example:

```
A ::= SEQUENCE {
  a INTEGER,
  b SEQUENCE OF SEQUENCE {
    aa INTEGER,
    bb BOOLEAN
  },
  c BIT STRING
}
```

In this case, the innermost data are represented in the output CSV files, but the actual SEQUENCE OF will be ignored as before:

```
a,aa,bb,c
```

The exact same columns would be represented if a CHOICE were used instead of a SEQUENCE. ASN2TXT will always do its best to collapse nested data types, drilling down to the innermost data to collect the column headers.

Data Conversion

Having collected column headers for the output CSV, the second and final step is to output the actual data from the decoded BER message. Fortunately this is considerably more straightforward than collapsing the data structures in the specification.

The main case to consider is that in which data types are repeated: when a SEQUENCE OF is nested inside of a SEQUENCE. Some brief comments follow for other nested data types.

SEQUENCE OF in a SEQUENCE

Take for example the simple case previously seen:

```
A ::= SEQUENCE {
    a INTEGER,
    b SEQUENCE OF INTEGER,
    c BIT STRING
}
```

Let us assume for sake of argument that there are two integers in the inner SEQUENCE OF. In this case, the resulting CSV file will have two rows in addition to the header row.

The common data, columns a and c, will be represented once in the output file (unless `-padFields` is specified), while the repeated element b will change. For example:

```
a,b,c
1,97823789324,010010
,18927481,
```

If you have chosen to pad the fields, the output will look like this:

```
a,b,c
1,97823789324,010010
1,18927481,010010
```

While this example is very simple, it is possible to nest data types to an arbitrary depth, and the representation of columns and their data can be quite large. In pathological instances, the CSV output may be larger than the output generated by other tools like ASN2XML.

Other Nested Data Types

The other nested data types, SEQUENCE and CHOICE, are relatively trivial to convert once the columns have been assembled as described in the previous section. A single row may be used to output a message without repeating types.

The CHOICE data type bears some explanation. The following specification is the same used in the previous section:

```
A ::= SEQUENCE {
    a INTEGER,
    b CHOICE { aa INTEGER, bb BOOLEAN },
    c BIT STRING
}
```


Some example output data follows:

```
a,aa,bb,c
1,,FALSE,101010
2,137,,100001
```

The output lines will contain data in either the aa or bb columns but not both. Only the selected data should be represented in the output line.

OPTIONAL and DEFAULT Elements

Optional primitive elements that are missing in an input message will result in a blank entry in the output CSV file. Take, for example, the following specification:

```
A ::= SEQUENCE {
    a INTEGER,
    b UTF8String OPTIONAL,
    c BIT STRING
}
```

This might result in the following output:

```
a,b,c
1,test string,100100
2,,100101
3,another test,100100
```

In this example, the second message does not contain the optional UTF8String, so it is omitted from the output.

Elements marked DEFAULT are handled differently in the output. If an element is missing in the input specification, the default value is copied into the output CSV file. The following specification is used to demonstrate:

```
A ::= SEQUENCE {
    a INTEGER,
    b UTF8String DEFAULT "test",
    c BIT STRING
}
```

In this case, we might have the following output:

```
a,b,c
1,test string,100100
2,test,100101
3,another test,100100
```

Like the previous example, the input data omitted the default UTF8String. Instead of a blank entry, however, the output CSV data contains `test`.