ASCII2TAP (v1.1, May 2011)

Prerequisites

- Programming experience with C
- Familiarity with TAP grammars

Introduction

Organizations having to generate and parse TAP files are usually also faced with the problem of interworking between ASN.1-based TAP formats and other proprietary ASCII formats. For example, mediation devices typically produce call-related information (including rating and charging information) in some ASCII format. Most billing systems can also be configured to output call data in some ASCII format which can be customizable. This customization allows for instance the advanced operator to select which database columns are to be used when exporting as well as the position, width and justification of the exported fields.

Therefore a utility to import ASCII file formats into TAP files could not be missing from the Roaming Components toolkit: this is ASCII2TAP. There is also a utility that implements the reverse process (TAP2ASCII) and which is described in a separate whitepaper.

A simple approach

Since TAP files are binary files, encoding and decoding them can be quite an issue. Even though there is a multitude of tools operating on ASCII files (the UNIX text tools are a good example) there is no comparable toolchain for binary files. An initial approach to the problem is therefore to mitigate it to something simpler: instead of directly converting between the desired ASCII format and TAP, first convert to an intermediate, easy to generate ASCII format, and then use tools that automatically transform the intermediate format to the binary data that is TAP. This intermediate ASCII format will have to be able to represent the tree structure of TAP – and what better format to handle this than the ubiquitous XML?

```
.....
<CallEventDetail>
  <mobileTerminatedCall>
    <basicCallInformation>
        <fraudMonitorIndicator>1</fraudMonitorIndicator>
        <chargeableSubscriber>
          <simChargeableSubscriber>
            <imsi>202052251044710</imsi>
            <msisdn>306947322118</msisdn>
          </simChargeableSubscriber>
        </chargeableSubscriber>
        <rapFileSequenceNumber>09999</rapFileSequenceNumber>
        <networkType>1</networkType>
        <callOriginator>
          <callingNumber>306944184684</callingNumber>
        ...
    </basicCallInformation>
    <locationInformation>
      <networkLocation>
        <recEntityCode>0</recEntityCode>
```

Both Roaming Studio and Roaming Components offer support for this XML output. Roaming Components' TAP2XML and XML2TAP are the utilities used to convert to and from this format,

while in Roaming Studio, importing and exporting from XML is done through the “File/Import From” and “File/Export To” menu options. The problem (from the ASCII2TAP viewpoint) now morphs into transforming whatever input ASCII format into this XML file (through Perl/Python scripting, XSLT transformations, or any other form of manual coding).

Since both source and destination targets are ASCII formats, this is substantially easier than the original problem. It still needs to be tackled carefully though. TAP files require specific business logic to be valid, business logic that has to be applied when the data are inserted into the tree. For example, in the XML section in the listing above, one can see recEntityCode; one of the 7 different lookup codes that must be carefully handled (utcTimeOffsetCode, exchangeRateCode, etc.) as it is pointing to a specific lookup table maintained inside the TAP file.
Therefore the XML “transformation code”, unless simple, can become unintelligible. Nevertheless, for many cases:

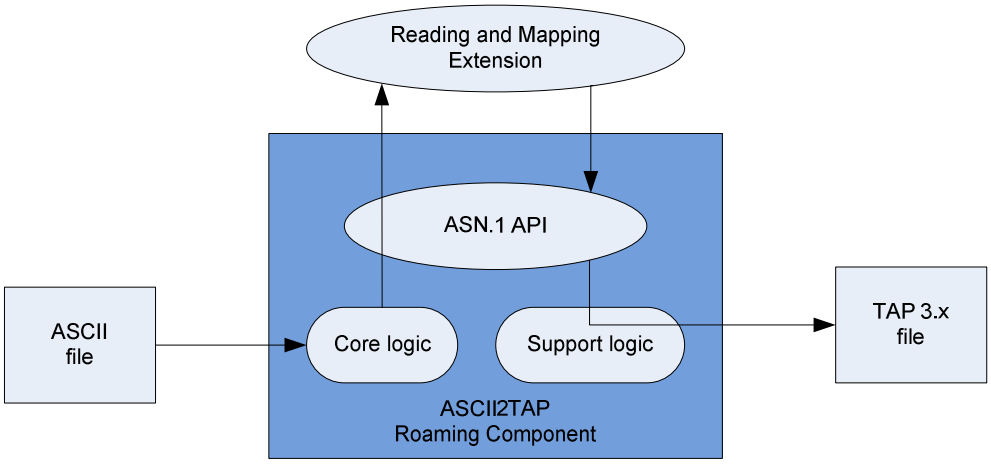
(a) converting your ASCII file to an XML format using scripting or UNIX text tools,, and then
(b) using XML2TAP to import it as a TAP file

…may be an option you should seriously consider.

Extending ASCII2TAP

The ASCII-related Roaming Components allow advanced users to create an extension (a DLL under Windows, a shared library under UNIX OSes) responsible for reading and mapping ASCII data into TAP structures. This task is necessary for the first approach as well (the one described in the previous section); this method, however, offers unlimited freedom in terms of data handling. ASCII2TAP supplies a set of programming primitives (an API) that greatly simplifies access to the TAP tree, allowing easy navigation, iteration and setting of values over all nodes.

Furthermore, a large degree of automation can be applied in this approach: most ASCII formats can be described in a simple “guide” file, and this guide file can be subsequently used to automatically generate the “reading” part of the code. Adaptive changes to the input format can then be performed with a minimum amount of code rewrites.



Since the API is a C API, the developer of the extension can use the full power of the C language to create the extension – e.g. use whatever C library for custom functionality or easier development. The afore-mentioned code generator is not a mandatory part of the process; generic C code can be written to process any kind of input data, e.g. read them from another machine over sockets, extract them from a database, etc.

Writing an extension manually

Extensions work through bidirectional communication with ASCII2TAP. Contact is initiated by the platform, through a call to the extension's "getAndSetCallbacks" function. This "bridge" function plays a twofold role; it is used by the platform to pass function pointers to the relevant API, and it is used by the extension to report back function pointers to callback functions implemented by the extension - for:

- Reporting the TAP release number generated by the extension (e.g. TAP 3.9)
- Performing the main mapping functionality
- Performing cleanup tasks

```
Callback *getAndSetCallbacks(
       Utils *pUtils, char *pszInputASCIIFileName, char *workingDirectory);
```

There are two data structures referenced by this gateway function, `Callback` and `Utils`. Both are defined inside AsciiAPI.h:

```
typedef void (*PFNMODULECALLBACK)(RCHANDLE, RCHANDLE, RCINT64);

typedef struct tagCallback {
    const char *_pszTypename;
    PFNMODULECALLBACK _pCallback;
} Callback;

typedef struct tagUtils {
    int (*pfnGetCount)   ( RCHANDLE rcHandle, char *path, RCINT64* retVal);
    int (*pfnGetInteger) ( RCHANDLE rcHandle, char *path, RCINT64* retVal);
    int (*pfnSetInteger) ( RCHANDLE rcHandle, char *path, RCINT64 val);
    int (*pfnSetString)  ( RCHANDLE rcHandle, char *path, char* srcNullTerminatedBuffer);
    ...
    void (*pfnThrowException) (const char *);
} Utils;
```

ASCII2TAP, while initiating contact with the extension, calls getAndSetCallbacks passing in a pointer to a `Utils` struct and pointers to the input filename and the desired working directory (which is the extension's directory). The Utils structure contains function pointers to all the API members, thus providing complete access to the API. In return, the extension answers with a set of Callback structs – getAndSetCallbacks returns a pointer to Callbacks. As seen in the declaration, each Callback has a name (`pszTypename`) and a function pointer. The name is an identifier of the functionality offered by the particular function pointer and can be one of:

- RC-Version:
  This callback is responsible for reporting the supported TAP release:

```
void ModuleVersion(RCHANDLE treeRoot, RCHANDLE dummy, RCINT64 idxdummy)
{
       *(int *)dummy = 9; /* minor version is 9, so TAP 3.9 is supported */
}                         /* by this particular extension */
```

- RC-Main:
  This callback contains the main mapping function
- RC-Shutdown
  This callback contains any cleanup code necessary

Developing an extension therefore breaks down to the following steps:

1. The extension's developer writes one callback function for each necessary functionality (RC-Main and RC-Version are mandatory).
2. A static array containing the callbacks' information is allocated in static (global) space:

```
Callback myfunctions[] = {
    {"RC-Version", ModuleVersion},
    {"RC-Main", ModuleMapping},
    {"RC-Shutdown", ModuleShutdown},
    {NULL,NULL}                         // Terminates the list
};
```

3. getAndSetCallbacks is written, storing the pointer to the API structure for further use by the callbacks, and returning a pointer to the callback array:

```
Callback *getAndSetCallbacks(
    Utils *pUtils, char *pszInputASCIIFileName, char *workingDirectory)
{
    g_pUtils = pUtils;    // store API access pointer to global variable
    g_pszInputFilename = strdup(pszInputASCIIfilename);    // ditto
    g_pszWorkingDirectory = strdup(workingDirectory);      // ditto
    return myfunctions;   // returns pointer to all callback information
}
```

The callbacks reported by the extension in getAndSetCallbacks will be called in the following order:

- RC-Version at the beginning, to verify that the extension generates the release the user requested
- RC-Main immediately afterwards, to read whatever input and perform the tree creation
- RC-Shutdown (optionally, if set) to cleanup whatever resources need cleanup

Since the callbacks will need to work on the tree through the API, the notion of “node handles” is introduced; each ASN.1 node is represented through an RCHANDLE, an opaque structure that provides access to a specific node. All the API functions work on these handles.

Three arguments will be passed each time a callback is invoked:

1. The first argument will always be a handle to the root node (DataInterChange).
2. The second argument is to be ignored (used only in TAP2ASCII)
3. The third argument is to be ignored (used only in TAP2ASCII)

Inside the callback array, besides the entries pertaining to normal ASN.1 nodes, two special keywords can be placed in the `_pszTypename` field: “RC-Version” and “RC-Shutdown”:

Let's look at an example that counts the number of input lines and creates an equal number of (empty) MobileOriginatedCalls inside a TAP 3.10 tree:

```
#include <stdio.h>
#include <stdlib.h>

#include “AsciiAPI.h”

#ifdef WIN32
#include <windows.h>
#define SIGNATURE __declspec(dllexport)
#else
#define SIGNATURE
#endif

FILE *g_fp = NULL;
Utils *g_pUtils = NULL;

SIGNATURE void ModuleVersion(RCHANDLE treeRoot, RCHANDLE ptrToMinorVer, RCINT64 idxdummy)
{
  *(int *)ptrToMinorVer = 10; // TAP 3.10 is supported by this extension
}

SIGNATURE void ModuleMapping(RCHANDLE treeRoot, RCHANDLE dummy, RCINT64 idxdummy)
```

```
{
  char tmp[200];
  while(!feof(g_fp)) {
    RCHANDLE cedl;
    if (!fgets(tmp, sizeof(tmp), g_fp))
       break;
    /* one line was read, create an empty MOC */
    if (0 == g_pUtils->pfnNavigateAndObtainHandle(
       treeRoot,
       “transferBatch.callEventDetails”,
       &cedl))
    {
       RCHANDLE ced;
       if (0 == g_pUtils->pfnAppendEntryInSequenceOf(cedl, &ced)) {
         RCHANDLE moc;
         g_pUtils->pfnNavigateAndObtainHandle(ced, “mobileOriginatedCall”, &moc);
       }
    }
  }
}

SIGNATURE void ModuleShutdown(RCHANDLE treeRoot, RCHANDLE dummy, RCINT64 idxdummy)
{
  fclose(g_fp);
  g_fp = NULL;
}

Callback myfunctions[] = {
  {"RC-Version", ModuleVersion},
  {"RC-Main", ModuleMapping},
  {"RC-Shutdown", ModuleShutdown},
  {NULL,NULL}
};

SIGNATURE Callback *getAndSetCallbacks(
    Utils *pUtils, char *pszInputASCIIfilename, char *workingDirectory)
{
  g_pUtils = pUtils;      // store API access pointer to global variable
  g_fp = fopen(pszInputASCIIfilename, “r”);
  return myfunctions;     // returns pointer to all callback information
}
```

Compiling this and creating a DLL (or a shared library, if you are doing this under any UNIX OS) will allow a simple integration test with the ASCII2TAP Roaming Component:

```
ASCII2TAP.exe -ims SimpleTest.dll -i anyASCIIinput -o dummy
```

The command will create as many empty MobileOriginatedCalls as there are lines in file 'anyASCIIinput'.

This is just a proof of concept, of course. Normally, parsing the input is a lot more work than just counting lines – splitting on delimiters or reading fixed width columns, converting data read to native C types, or even reading from a DB instead of the input file. This process however lends itself to a large degree of automation. In almost all extensions, files will always be read; sets of data will be input from these files and then the files will be closed. These files will almost always adhere to a simple ASCII format; “scanf” formating and the input process itself can become a source of bugs, when there really isn't any reason (most of the time) to let a human do this kind of coding.

This is the reason behind the introduction of a code generator (as mentioned in the previous section); to ease the burden of writing extensions for ASCII file input by leveraging a simple XML “guide” file.

Automation - The “guide” file

The “automated” extension writing process is broken down to these three steps:

1. Describe the desired input format in a “guide” file.
   This is a simple XML file, describing the available record types and their respective fields, along with their formating information.
2. Run the “guide” file through the code generator
   This will create the boilerplate code for the mapping code, as well as a complete implementation of the reading code
3. Fill-in the mapping code
   Utilizing the API, write the code that navigates the TAP tree and populates the TAP data structures.

Here is a simple guide file for input of Mobile Originated Calls data:

```
<?xml version="1.0" encoding="utf-8" ?>
<FileSpecification Name="SimpleMOC" Description="Simple MOC input specification"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="FileInputSpec.xsd">
  <FixedWidthFile>
   <Record Name="HEADER" Prefix="10">
     <StringField Name="SENDER" Length="5" Mandatory="true" DefaultValue=""/>
     <StringField Name="RECIPIENT" Length="5" Mandatory="true" DefaultValue=""/>
     <StringField Name="FSN" Length="5" Mandatory="true" DefaultValue="" />
     <DateField Name="FILE_CREATION_DATE" Length="6" Mandatory="true"
        DefaultValue="" Format="%y%m%d"/>
   </Record>
   <Record Name="MOC" Prefix="20">
     <StringField Name="IMSI" Length="15" Mandatory="true" DefaultValue="" />
     <StringField Name="CALLED_NUMBER" Length="21" Mandatory="true" DefaultValue="" />
     <NumberField Name="NATIONAL_IND" Length="1" Mandatory="true" DecimalPlaces="0"
       DefaultValue="" />
     <DateField Name="CALL_DATE_TIME" Length="14" Mandatory="true" DefaultValue=""
       Format="%Y%m%d%H%M%S" />
     <NumberField Name="DURATION" Length="7" Mandatory="true" DecimalPlaces="0"
       DefaultValue="" />
     <StringField Name="MSCID" Length="15" Mandatory="true" DefaultValue="" />
     <StringField Name="CALLEDCOUNTRYCODE" Length="5" Mandatory="true" DefaultValue="" />
     <NumberField Name="CHARGE" Length="9" DecimalPlaces="3" Mandatory="true"
       DefaultValue="" />
     <NumberField Name="CHARGEABLE_UNITS" Length="6" DecimalPlaces="0" Mandatory="true"
       DefaultValue="" />
     <NumberField Name="EXCHANGE_RATE" Length="8" DecimalPlaces="5" Mandatory="true"
       DefaultValue="" />
   </Record>
  </FixedWidthFile>
</FileSpecification>
```

As can be seen, this is a simple XML file. It describes two Record types, a HEADER and a MOC type. Each of these types is considered as input if one input line starts with the “Prefix” attribute (i.e. 10 for HEADER and 20 for MOC). When fed with this specification, the code generator writes a header file...

```
...
typedef struct tagHEADER {
    char SENDER[6];
    char RECIPIENT[6];
    char FSN[6];
    struct tm FILE_CREATION_DATE;
} HEADER;

typedef struct tagMOC {
    char IMSI[16];
```

```
    char CALLED_NUMBER[22];
    int NATIONAL_IND;
    struct tm CALL_DATE_TIME;
    int DURATION;
    char MSCID[16];
    char CALLEDCOUNTRYCODE[6];
    double CHARGE;
    int CHARGEABLE_UNITS;
    double EXCHANGE_RATE;
} MOC;

RCINT64 GetMinorVersionProduced();
void ProcessHEADER(RCHANDLE treeRoot, HEADER *pHEADER);
void ProcessMOC(RCHANDLE treeRoot, MOC *pMOC);
```

...as well as a “reading” implementation file that reads the input ASCII file, creates these structs in memory and passes them on to the `ProcessHEADER` and `ProcessMOC` functions. The code for these processing functions is written inside the mapping file by the developer. The code generator has already written the boilerplate:

```
RCINT64 GetMinorVersionProduced()
{
    return 10; /* e.g. version 3.10 */
}

void ProcessHEADER(RCHANDLE treeRoot, HEADER *pHEADER)
{
   /* write your mapping code here */
}

void ProcessMOC(RCHANDLE treeRoot, MOC *pMOC)
{
   /* write your mapping code here */
}
```

It is these functions that constitute the main part of the import module. ASCII2TAP will read the input data and depending on prefix fill the appropriate C structure and call these functions. An RCHANDLE (Roaming Component Handle) to the initially empty TAP tree is also passed (the DataInterChange node). This handle allows the code to perform any tree traversal as well as data insertion in the tree, using the API.

Writing mapping code using the API

From the ASCII2TAP point of view, these are the API functions required:

```
/* Set the value of an INTEGER */
int SetInteger( RCHANDLE rcHandle, char *path, RCINT64 val);
/* Set the value of an OCTET_STRING (BCDStrings are auto-converted) */
int SetString( RCHANDLE rcHandle, char *path, char* srcNullTerminatedBuffer);
/* Create a child node under a SEQUENCE or a CHOICE */
int CreateChildVariable(RCHANDLE parent, char *variableName, RCHANDLE *child);
/* Append a new entry in a SEQUENCE_OF */
int AppendEntryInSequenceOf(RCHANDLE sequenceOf, RCHANDLE *child);
/* Follow a path to a specific node */
int NavigateAndObtainHandle( RCHANDLE parent, const char *path, RCHANDLE *destHandle);
/* Read a template XML and store it for re-use */
int CompileSubTreeFromXmlTemplate(char* xmlFileName, XMLHANDLE *subtree);
/* Apply a stored XML template to a specific node, generating an offspring */
int AddSubTree(RCHANDLE parent, XMLHANDLE subtree, RCHANDLE *offspring);
/* get_taxCode()
 * Input : taxType, taxRate (for TAP3.11, chargeType as well - in other versions
 *         pass NULL for chargeType)
 * Output: taxCode (found or just created)
 */
int get_taxCode(
  RCHANDLE di, const char * mand_taxType, const char * mand_taxRate,
```

```
  const char * mand_chargeType, RCHANDLE *ptaxCode);
(more Code functions)
```

There are three C types appearing here that are new:

1. RCHANDLE is a handle to a node of the TAP tree
2. RCINT64 is a 64-bit integer, capable of holding the ASN.1 INTEGER types
3. XML_HANDLE is a handle to a template file (more on this later)

Since each ProcessXXX function gets an RCHANDLE to the root of the TAP tree (the DataInterChange node), it would be very easy to create a TransferBatch node (perhaps, in the ProcessHeader function):

```
RCHANDLE tb;
if (0 == g_pUtils->pfnCreateChildVariable(treeRoot, “transferBatch”, &tb)) {
   /* Do whatever on the tb handle */
}
```

The check against 0 is done because we might run out of memory (or the developer might pass an invalid string).
A handle to a SEQUENCE_OF is all that's necessary to add a new element:

```
RCHANDLE cedl, ced;
... /* assign a CallEventDetails node to cedl */
if (0 == g_pUtils->pfnAppendEntryInSequenceOf(celd, &ced)) {
  // New CallEventDetail Element appended and saved to ced
  ...
}
```

Assuming a handle to a MOC node is accessible, an OCTET_STRING can be set directly, like this:

```
/* pMOC points to a struct MOC, see header file made by code generator */
if (0 == g_pUtils->pfnSetString(
  mocNode,
"mobileOriginatedCall.basicCallInformation.chargeableSubscriber.simChargeableSubscriber.i
msi",
  pMOC->IMSI))
{
  // assigned correctly
}
```

An INTEGER is equally easy:

```
/* pMOC points to a struct MOC, see header file made by code generator */
RCINT64 charge = pMOC->CHARGE;
if (0 == g_pUtils->pfnSetInteger(
  mocNode,
"mobileOriginatedCall.basicServiceUsedList.[0].chargeInformationList.[0].chargeDetailList
.[0].charge",
  (RCINT64)(0.5 + 1000.0*charge)) )
{
  // assigned correctly
}
```

The functions described so far constitute a minimal, but complete interface. Utilizing only these functions all of the TAP tree is accessible. The API however, contains a helpful addition: a set of template functions. Instead of repeating code over and over, we can utilize a template to create a specific subtree of the TAP tree (e.g., a new MobileOriginatedCall) with our own default values. The template files can be exported from Roaming Studio, via the “Export to XML” button from the “Edit/Templates” submenu.

```
XMLHANDLE tbXml, mocXml;
RCHANDLE tbNode, cedl, mocNode;
g_pUtils->pfnCompileSubTreeFromXmlTemplate("MainBody.xml", &tbXml);
g_pUtils->pfnAddSubTree(treeRoot, tbXml, &tbNode);
```

```
g_pUtils->pfnCreateChildVariable(tbNode, "callEventDetails", &cedl);
g_pUtils->pfnCompileSubTreeFromXmlTemplate("SimpleMOCtemplate.xml", &mocXml);
g_pUtils->pfnAddSubTree(cedl, mocXml, &mocNode);
```

The MainBody.xml and SimpleMOCtemplate.xml files contain default values for subtrees under DataInterChange and CallEventDetail, respectively. The template files must be placed in the same directory the extension resides in. These templates contain lots of assignments, which would otherwise be needlessly cluttering the code. After the templates have been applied, we can navigate and change individual elements, as we did in the previous examples for INTEGERs and OCTET_STRINGs.

Additionally, when we reach nodes that relate to Codes in the ASN.1 grammar (e.g. recEntityCode, taxCode, etc) we can use the code helper functions:

```
RCHANDLE mocNode, utcCode;
... /* create a MOC and access it through the mocNode variable */
g_pUtils->pfnNavigateAndObtainHandle(
  mocNode,
"mobileOriginatedCall.basicCallInformation.callEventStartTimeStamp.utcTimeOffsetCode",
  &utcCode) )
g_pUtils->pfn_get_utcTimeOffsetCode(treeRoot, "+0000", utcCode);
```

The last statement is deceptively simple: it navigates to the appropriate lookup table in the TAP tree, checks to see if the provided UTC offset (“+0000”) already exists, and if it doesn't, it adds it up, assigning it a new code in the process. It then writes the offset code (that already existed in the lookup table or that was just now added) to the passed in node, utcCode. It does this regardless of the TAP release being generated.

There are such “accessor” code functions for all codes in the TAP releases. Further details exist inside the main header file “AsciiAPI.h”, especially the comments before each get_xxxCode function. A fair bit of engineering has been utilized to accommodate differences over assigned types, TAP releases and mandatory/optional arguments.

When all of the input data have been read, ASCII2TAP will attempt to create a valid auditControlInfo structure from the data of the tree, if invoked with argument “-fixup”. No aggregation and/or summarization is therefore required on the part of the developer; the complex and TAP-release specific logic is provided with no additional effort.

A complete example

Both Roaming Studio and the ASCII2TAP Roaming Component contain the complete source code for an extension that reads Mobile Originated Call information from a simple ASCII format. Parts of this extension have been used in the examples above to give an idea of how easy it is to populate the TAP tree using the API. This is not a naive example; it can be extended to create any complicated format.

The example contains a gcc Makefile for UNIX platforms as well as a Visual Studio 8 project file to create an extension DLL for the windows version of ASCII2TAP. You can follow the specific instructions in these packages to compile the extensions and integrate them in the appropriate platform.

Is the “guide” necessary?

In the process described so far, only using the API is absolutely necessary. As long as the callbacks get filled with code, the code will be executed. Using the code generator and the “guide” files is not mandatory – it is only provided as a convenience, which can be ignored if, for whatever reason, manual coding is preferred. For example, a complicated ASCII format might contain fields whose

format is not handled by the “guide” files – or contain nested records, something not currently supported. Or perhaps a more complex input architecture is in place, providing the input not from ASCII files but from data over a socket from another machine, or a database. All of that is feasible, since the extension is written in C code. The power from leveraging a full featured language is far beyond what any scripted mapping could approach.