TAP2ASCII – Developing custom extensions

Prerequisites

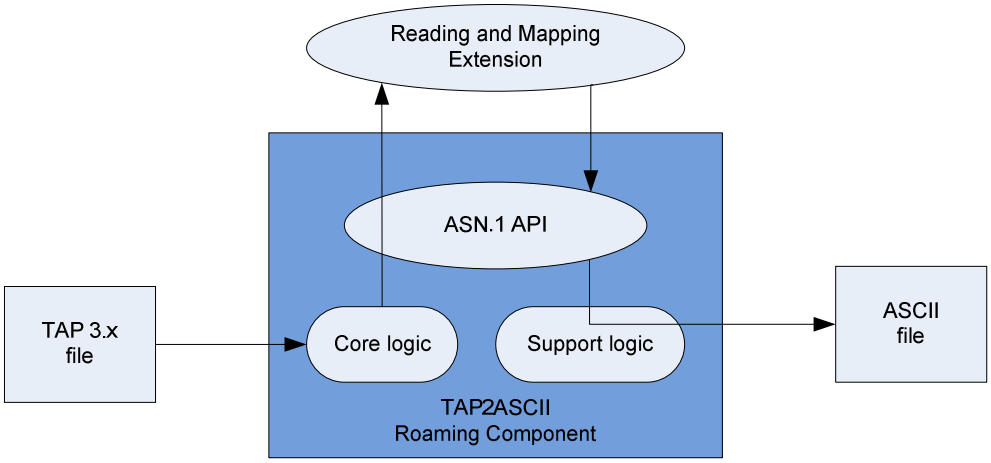
Programming experience with C, and familiarity with TAP grammars

Introduction

Organizations faced with the generation and parsing of TAP files are usually also faced with the interworking to/from specific ASCII formats. For example, it is usually an ASCII format that a mediation device uses to pass on call-related details. Most billing systems can also be configured to output call data in a configurable ASCII format (even allowing the advanced operator to select database columns to use when exporting). It is therefore evident that any roaming suite has to deal with this interworking.

Extending the platform

TAP2ASCII allows advanced users to create an extension (a DLL under Windows, a shared library under UNIX OSes) responsible for mapping and writing ASCII data from TAP structures. This method offers the greatest amount of freedom in handling the data inside TAP files. The platform supplies a set of programming primitives (an API) that greatly simplifies access to the TAP tree, allowing easy navigation and iteration over all nodes.



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 – use for example whatever C library enriches its functionality or accelerates its development. C code can be written to perform anything on the output data, e.g. send them to another machine over sockets, store them in a database, etc.

Writing an extension manually

Extensions work through bidirectional communication with TAP2ASCII. Contact is initiated by TAP2ASCII, via a call to the extension's “getAndSetCallbacks” function. This “bridge” function plays a twofold role; it is used by TAP2ASCII to pass function pointers to all the platform's API functions, and it is used by the extension to report two things back to TAP2ASCII:

- the function pointer that must be called for each desired type
- the desired types that interest the extension (e.g. MobileOriginatedCall, etc)

```
Callback *getAndSetCallbacks(Utils *pUtils, char *pszOutputFileName, char *pszDummy);
```

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);
    ...
    void (*pfnThrowException) (const char *);
} Utils;
```

The executive summary of the interface is the following:

TAP2ASCII initiates contact with the extension, calling “getAndSetCallbacks”, and passing in a pointer to a `Utils` struct and a pointer to the output filename. 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) – which is` the name of a TAP ASN.1 type (e.g. GprsCall) - and a function pointer that points to the extension's function that is responsible for handling the specified type.

Developing an extension therefore breaks down to the following steps:

1. The extension's developer decides on the set of TAP ASN.1 types that interest him/her (e.g. MobileOriginatedCall, GprsCall, etc). If access to the whole tree is necessary, then the type is simply the root type – “DataInterChange”.
2. One callback function (of type PFNMODULECALLBACK) is written for each of the desired ASN.1 types
3. A static array containing the callbacks' information is allocated in static (global) space:

```
Callback myfunctions[] = {
    {"RC-Version", ModuleVersion},      // see below
    {"DataInterChange", OnDataInterChangeCallback},
    {"RC-Shutdown", ModuleShutdown},    // see below
    {NULL,NULL}                         // Terminates the list
};
```

4. 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 *pszOutputASCIIfilename, char *dummy)
{
    g_pUtils = pUtils;    // store API access pointer to global variable
    g_pszOutputFilename = strdup(pszOutputASCIIfilename); // ditto
    return myfunctions;   // returns pointer to all callback information
}
```

The callbacks reported by the extension in getAndSetCallbacks will be called once for each instance of their respective type in the tree. 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 of the tree (DataInterChange).
2. The second argument is a handle to the visited node (always of the type requested)
3. The third argument is the 0-based index of the node, if the node is a member of a SEQUENCE_OF, otherwise it is -1.

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”:

- Exactly one entry in the callback list must exist with “RC-Version” as a typename. The corresponding callback function must report the TAP release it can handle:

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

- “RC-Shutdown” is an optional keyword; if it exists, the platform will call the corresponding callback after finishing the tree processing (after all the other callbacks have completed). It can be used for cleanup purposes (deallocating memory, closing files/sockets/pipes, etc)

Let's look at an example, that counts the number of MobileOriginatedCalls in a TAP 3.10 file:

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

#include “AsciiAPI.h”

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

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

unsigned g_totalMOCs = 0;


SIGNATURE void OnMOC(RCHANDLE treeRoot, RCHANDLE dummy, RCINT64 idxdummy)
{
  g_totalMOCs++;
  printf(“Met another MOC, totals: %d\n”, g_totalMOCs);
}

Callback myfunctions[] = {
  {"RC-Version", ModuleVersion},
  {"MobileOriginatedCall", OnMOC},
  {NULL,NULL}
};

SIGNATURE Callback *getAndSetCallbacks(
  Utils *pUtils, char *pszOutputASCIIfilename, char *dummy)
{
    return myfunctions;
}
```

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 TAP2ASCII Roaming Component:

```
TAP2ASCII.exe -ims SimpleTest.dll -i CDDEUD2GRCPF13110 -o dummy
Met another MOC, totals: 1
Met another MOC, totals: 2
...
Met another MOC, totals: 3459
```

The previous example didn't do anything with the output filename parameter, so obviously, no output file was generated. This can be easily rectified by writing code that creates a file with the given filename, utilizes the API to get around the tree and writes the data in whatever format desired.

Writing mapping code using the API

From the TAP2ASCII point of view, these are the API functions required:

```
/* Get the value of an INTEGER */
int GetInteger( RCHANDLE rcHandle, char *path, RCINT64* retVal);
/* Get the value of an OCTET_STRING (BCDStrings are auto-converted) */
int GetString( RCHANDLE rcHandle, char *path, char* destBuffer, int sizeOfDestBuffer);
/* Follow a path to a specific node */
int NavigateToAChildVariable(RCHANDLE parent, char *variableName, RCHANDLE *child);
/* Ask a SEQUENCE_OF how many entities it contains */
int GetCount( RCHANDLE rcHandle, char *path, RCINT64* retVal);
/* Navigate to a specific child of a SEQUENCE_OF (index based) */
int NavigateToAMemberOfASequenceOf(RCHANDLE parent, int idx, RCHANDLE *child);

/* Setup a recursive traversal of the tree from the parent, searching for all nodes of
Type 'type' */
RC_VISIT_HANDLE InitializeVisit( RCHANDLE parent, char *type );
/* Execute the traversal until you find the next node, or until you exhaust them - return
NULL if no
 * other node of type 'type' can be found (see InitializeVisit) */
RCHANDLE GetNext( RC_VISIT_HANDLE );
/* Abort a recursive traversal midway (before NULL is returned from GetNext() */
void DestroySearch(RC_VISIT_HANDLE);
```

There are three C types used by these functions:

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. RC_VISIT_HANDLE is a handle to an adaptive visit over the tree (more on this later)

Since OutputTree gets an RCHANDLE to the root of the TAP tree (the DataInterChange node), it would be very easy to get a handle to the TransferBatch node:

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

The check against 0 is done for two reasons. The requested ASN.1 node might be missing (the TAP tree could be a notification, not a transferBatch), and, unfortunately, the developer might mistype the variable name.

To navigate one more level down, for example to the batchControlInfo underneath transferBatch?

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

That pattern covers both CHOICEs and SEQUENCEs. What about SEQUENCE_OFs? Assuming a handle to a MOC node is accessible, a navigation to its first basicServiceUsed can be done like this:

```
RCHANDLE moc, bsu, bsu0;
RCINT64 totalBSUs;
... /* assign the moc handle */
if (0 == g_pUtils->pfnGetCount(moc, “basicServiceUsedList”, &totalBSUs)) {
  if (totalBSUs>0 &&
     (0==g_pUtils->pfnNavigateToAChildVariable(moc, “basicServiceUsedList”, &bsu)) &&
     (0==g_pUtils->pfnNavigateToAMemberOfASequenceOf(bsu, 0, &bsu0)))
  {
    /* Do whatever on the bsu0 handle */
  }
```

There are easier ways to get to the values, though. An OCTET_STRING of this MOC can be accessed directly, like this:

```
MOC mocData; // from automatically generated structure declarations
char tmp[20]; //  MAX_CALLED_NUMBER_SIZE
if (0 == g_pUtils->pfnGetString(
  moc,
  "basicCallInformation.destination.calledNumber",
  tmp,
  sizeof(tmp)) )
{
  /* Do whatever with the tmp data – usually assign it to the MOC structure */
  /* memcpy(&mocData.CALLED_NUMBER, tmp, strlen(tmp)); */
  /* mocData.bCALLED_NUMBER_Exists = 1; */
}
```

An INTEGER is even easier:

```
MOC mocData; // from automatically generated structure declarations
RCINT64 modInd;
if (0 == g_pUtils->pfnGetInteger(
  moc,
  "basicServiceUsedList.[0].chargeInformationList.[0].callTypeGroup.callTypeLevel1",
  &modInd))
{
  /* Do whatever with the modInd data – usually assign it to the MOC structure */
  /* mocData.MOD_IND = (int) modInd; */
  /* mocData.bMOD_IND_Exists = 1; */
}
```

The functions described so far constitute a minimal, but complete interface – that is, utilizing only these functions allows one to access all nodes in the TAP tree. The API however, contains a very helpful addition: a set of adaptive traversal functions. Assuming for example that all the `ExchangeRateDefinition` nodes underneath `transferBatch.accountingInfo` must be visited, instead of reading the size of the SEQUENCE_OF list and writing tedious enumerating code, this is far better:

```
RCHANDLE accountingInfo;
.../* get handle to accountingInfo */
RC_VISIT_HANDLE visit = g_pUtils->pfnInitializeVisit(
    accountingInfo, "ExchangeRateDefinition");
RCHANDLE cc;

while (cc = g_pUtils->pfnGetNext(visit)) {
    RCINT64 code;
    if (0 == g_pUtils->pfnGetInteger(cc, "exchangeRateCode", &code)) {
       /* whatever */
    }
}
```

The InitializeVisit and GetNext primitives perform an automated traversal over the subtree, starting

at the node passed in as the first argument to InitializeVisit, and stopping at every node of the requested type (second argument).

Aborting halfway through the search is also possible (if for example the desired node was found), by way of DestroySearch:

```
RC_VISIT_HANDLE tapDecimalPlacesVisit;
RCINT64 decimals;
RCHANDLE tdp;
tapDecimalPlacesVisit = g_pUtils->pfnInitializeVisit( treeRoot, "TapDecimalPlaces" );
while((tdp = g_pUtils->pfnGetNext(tapDecimalPlacesVisit))) {
  g_pUtils->pfnGetInteger(tdp, "", &decimals);
  g_pUtils->pfnDestroySearch(tapDecimalPlacesVisit);
  break;
}
```

When all fields in the C structures generated by the code generator are populated, we can request an output of a line to our ASCII file:

```
OutputMOC(fp, &mocData);
```

A complete example

The TAP2ASCII Roaming component contains the complete source code for an extension that outputs Mobile Originated Call information in 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 navigate the TAP tree using the API. This is not a naïve example; it can be extended to create any complicated format.

The example contains a GNU gcc based Makefile for UNIX platforms as well as a Visual Studio 8 project file to create an extension DLL for the Win32 version of the TAP2ASCII Roaming Component. You can follow the specific instructions in these packages to compile the extensions and integrate them in the appropriate platform.

Extensibility

By coding your extension in C, the user of TAP2ASCII has significant freedom in what actions to take on the TAP data. Instead of writing to the output file, there might be a need to send the TAP data over a socket to another machine, or store them in a database. All of that is, since the extension is written in C code – and the power from leveraging a full featured language is far beyond any scripted mapping approach.