MAPI functions tutorial

July 17, 2009

1 Introduction

This is a tutorial that shows how to implement new functions in MAPI. It starts by giving detailed description of all structures and interfaces that can be used by functions. It then shows several examples on how some of the functions in the standard library is implemented, starting with a simple function like PKT_COUNTER and gradually moving towards more advanced functions.

2 Structures

This is an overview of the structures that are used by MAPI functions. Figure 1 shows how the various structures are connected.

2.1 mapidflib_function_def

The mapidflib_function_def structure defines the static properties of a function and has the following entries:

char* libname name of library that the function is part of.

char* name name of the function

char* descr description of the function

char* argdescr description of the arguments that a function needs. Each character in the string specifies the type of an argument. The following types are supported:

s	string
i	integer
c	single character
l	unsigned long long
w	filename for writing
r	reference to a flow

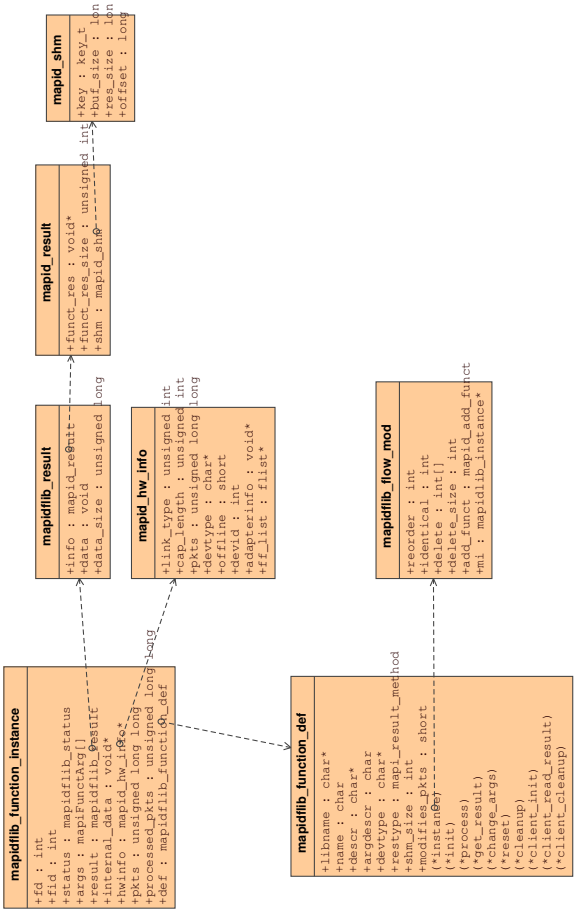

Figure 1: Structure relations. Warning: contains outdated info, not accurate.

f reference to a function

A function that takes two integers and a string as arguments, will then set argdescr to "iis"

char* devtype specifies the type of device that this function is compatible with. Valid device types are specified in the file *mapidevices.h*.

mapi_result_method_t restype specifies which method that should be used for returning results to the client application. Valid options are:

MAPIRES_NONE the function do not return any results

MAPIRES_IPC socket based IPC is used every time a result is sent from the MAPI daemon to the client application.

MAPIRES_SHM results are returned using shared memory.

MAPIRES_FUNCT function specific method. The function must implement its own method for returning results.

int shm_size size of the shared memory in bytes. Should be 0 if shm is not used.

short modifies_pkts if the function modifies packets this should be set to 1. Support for functions that modifies packets is optional. If a function that modifies packets is applied to a flow in a MAPI system where support for it is turned off, an error message will be returned to the user application.

short filters_pkts if the function filters packets this should be set to 1. This means that if the function lets all packets pass this should be set to 0.

mapidfilb_optimize_t optimize Method used for global optimization. Options are:

MAPIOPT_NONE No global optimization should be used.

MAPIOPT_AUTO Automatic global optimization.

MAPIOPT_MANUAL Manual global optimization. Functions using this method should use the *identical* field in the *mapidflib_flow_mod* structure for global optimization.

instance pointer to the instance interface of the function. See section 3.1 for details.

init pointer to the init interface of the function. See section 3.2 for details.

process pointer to the processing interface of the function. See section 3.3 for details.

get_result pointer to the get_result interface. See section 3.4 for details.

reset pointer to the reset interface. See section 3.5 for details.

cleanup pointer to the cleanup interface. See section 3.6 for details.

client_init pointer to the client_init interface. See section 3.7 for details.

client_read_result pointer to the client_read_result interface. See section 3.8 for details.

client_cleanup pointer to the client_cleanup interface. See section 3.9 for details.

2.2 mapidflib_function_t

The *mapidflib_function_t* structure defines an applied function and points to the function instance. Can be globally optimized to point to a previously applied function instance.

int fd flow descriptor of the flow that the function belongs to.

int fid unique ID of the function

int ref is 1 if the reference is to an instance in other flow.

mapidflib_function_instance_t *instance pointer to the function instance for this function. See section 2.3 for details.

2.3 mapidflib_function_instance

The *mapidflib_function_instance* structure defines the dynamic properties of a running function.

mapidflib_status_t status current status of the function. Valid values are:

MAPIFUNC_UNINIT the function is uninitialized

MAPIFUNC_INIT the function has been initialized

mapiFunctArg[] args arguments that were passed to the function

mapidflib_result_t result structure that contains the results of the function. See section 2.4 for more details.

void* internal_data pointer to internal data. The function is free to use this as it wants.

mapid_hw_info_t *hwinfo pointer to hardware specific information about the adapter the function is running on. See section 2.6 for details.

unsigned long long pkts number of packets that has been sent to the function.

unsigned long long processed_pkts number of packets that has been processed by the function and passed on to the next.

mapidflib_function_def_t *def pointer to a copy of the function definition. Since this is a copy, the definition can be changed by the function during the initialization.

int ret return value of the last time the function proceesed a packet.

int refcount number of flows that references this function isntance.

int apply_flags copy of the flags argument from mapid_apply_function()

2.4 mapidflib_result

The mapidflib_result structure contains information about the results of a function.

mapid_result_t info information about results that are sent to the client.

void *data pointer to memory that contains the results. Functions that needs access to results from other functions, uses this pointer to read the results.

unsigned long data_size size of the result data.

2.5 mapid_result

Contains information about MAPI function results that is sent back to the client.

void *funct_res pointer to function specific information

unsigned fuct_res_size size of the function specific information

mapid_shm_t shm contains information about shared memory. Functions should normally not need to access this structure themselves as it is handled automatically by code in mapidlib.

mapid_shm_t shm_spinlock read/write spinlock for shared memory.

2.6 mapid_hw_info

This structure contains information about a hardware adapter.

unsigned int link_type the link type of the adapter.

unsigned int cap_length the maximum length of a packet that can be captured by the adapter

unsigned long long pkts number of packets captured by the adapter

unsigned long pkt_drop numberof packets dropped by the hardware.

char *devtype type of device as defined in mapidevices.h

short offline a value of 1 or more indicates that the device is used for analyzing already captured trace files. 0 means proper online device.

int devid unique ID of the adapter

int devfd file descriptor for hardware device

global_function_list_t *gflist global list of flows and the functions applied to them. (gflist->fflist) There can be multiple flows pointing to the same function, see *optimize* field in section 2.1 for details.

void *adapterinfo adapter specific information

2.7 mapidflib_flow_mod

This structure contains various variables that can be used by a function to modify the behavior of a flow.

int reorder used for reordering the execution order of functions in the flow. A function can set this to indicate that this function should be processed before the function that has a function ID equal to the the value of reorder.

int identical a function can check already applied functions from the ff_list in the hw_info structure. If it finds an identical function it can set this variable to point to it. The internal functions in MAPId can then use this information to do global optimization.

int *delete pointer to a list of function IDs that can be deleted. This can for example be used by a function if one instance of this function can replace already applied functions.

int delete_size number of entries in the delete array.

mapid_add_function add_funct pointer to a function that can be used for adding new functions to the flow.

mapidlib_instance_t *mi pointer to the mapidlib instance. Used as an argument to mapid_add_function.

2.8 mapid_pkthdr

This structure contains information about a packet captured by a MAPI driver.

unsigned long long ts 64-bit timestamp of packet.

unsigned short ifindex interface index identifying which interface on the adapter that was used to capture the packet.

unsigned caplen the number of bytes that was captured

unsigned wlen the actual length of the packet including link layer header

int color colorization of the packet for optimizations

3 MAPI function interfaces

This is a description of the various interfaces that can be implemented by a MAPI function. Many of the interfaces are optional.

3.1 instance

```
int instance(mapidflib_function_instance_t *instance,
             int *fd,
             mapidflib_flow_mod_t *flow_mod);
```

The instance interface is called when an application uses *mapi_apply_function*. This interface should do a syntax check of arguments that are passed to the function. The function can also use this interface to indicate similar functions, apply new functions or delete functions. No resources should be allocated in this interface.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

int *fd flow descriptior of the flow.

mapidflib_flow_mod_t *flow_mod pointer to the structure used for modifying the flow. See section 2.7.

3.2 init

```
int init(mapidflib_function_instance_t* instance,
         int fd);
```

The init interface is called when an application uses *mapi_connect*. This interface allocates and initializes resources needed by the function.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

int fd flow descriptior of the flow.

3.3 process

```
int process(mapidflib_function_instance_t* instance,
            unsigned char* dev_pkt,
            unsigned char* link_pkt,
            mapid_pkthdr_t* pkt_head);
```

This is the interface where the actual processing of a packet takes place. If the function returns the value 1, then the next function applied to the MAPI flow will be called. The value 0 means that further processing of this flow should stop.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

unsigned char *dev_pkt pointer to the packet as captured by the device. This includes any device specific header.

unsigned char *link_pkt pointer to the packet starting at the link layer header.

mapid_pkthdr_t *pkt_head pointer to the packet header structure. See section 2.8.

3.4 get_result

```
int get_result(mapidflib_function_instance_t* instance,
               mapidflib_result_t **res);
```

This interface returns the results of this function to other functions or to the client application. MAPId has built in support for returning results through IPC or shared memory. This interface is only needed if these built in methods are not enough, for example if synchronization between MAPId and the client is needed or if results are read directly from the hardware adapter.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3.

mapidflib_result_t **res pointer to the result structure that is used for returning information needed by others to get hold of the results. See section 2.5.

3.5 reset

```
int reset(mapidflib_function_instance_t* instance);
```

This interface is called by other functions and is used for resetting the results of the function.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

3.6 cleanup

```
int cleanup(mapidflib_function_instance_t* instance);
```

This interface that is called when a flow is closed and should release all allocated resources.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

3.7 client_init

```
int client_init(mapidflib_function_instance_t *instance,
                void* data);
```

This interface runs on the client side and is called the first time *mapi_read_results* is called. It is used for initializing function specific code that runs on the client side.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

void *data pointer to the function specific data that was returned by the get_result interface.

3.8 client_read_result

```
int client_read_result(mapidflib_function_instance_t* instance,
                       mapi_result_t *res);
```

This interface is called each time *mapi_read_results* is used. It runs on the client side and is used for implementing function specific methods for returning results to the user application.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

mapi_result_t *res pointer to the result structure. See section 2.5.

3.9 client_cleanup

```
int client_cleanup(mapidflib_function_instance_t* instance);
```

This interface is called when the flow closes and should release all resources allocated by the function on the client side.

Arguments

mapidflib_function_instance_t *instance pointer to the instance of the function. See section 2.3

4 Tutorials

In this section we will demonstrate how to create new MAPI functions through a series of tutorials. All the functions are taken from the standard library, but during the tutorials we will create a new library, *tutlib*, to which we will add one function at the time.

When there exists two functions with the same name and for the same device type, the first one found will be used. To make sure that functions from the new library is chosen, one must specify the name of the library in the mapi_apply_function call like this:

```
mapi_apply_function(fd,"tutlib:<function name>")
```

4.1 Simple function: PKT_COUNTER

The first function we will create is a simple packet counter. All this function does is to increment a counter each time the process interface is called. Results are sent to the client using shared memory.

Implementation

To start off we first have to create a new directory for the new library. In the main mapi directory do:

```
mkdir tutlib
cd tutlib
```

To create the new function we can use the function template located in the mapi main directory:

```
cp ../funct_template.c pktcounter.c
```

In this template all the MAPI function interfaces are present. Most of these are not needed for the simple packet counter functions so the first thing to do is to remove them. For this function only process and reset are needed.

We will use the prefix *pktc* for each function in the source code, so we should replace the text <*funct_name*> with pktc. We are then left with only two interfaces like this:

```
static int pktc_process(mapidflib_function_instance_t *instance,
                        const unsigned char* dev_pkt,
                        const unsigned char* link_pkt,
                        mapid_pkthdr_t* pkt_head)
{
  return 1;
}
static int pktc_reset(mapidflib_function_instance_t *instance)
{
  return 0;
}
```

We can now move on to the mapidflib_function_def structure that defines the function. The entries in this structure should be modified like this:

libname — empty string. This is set by the library that includes the function

name — The function name can be set to any value but in this example we will call the function PKT_COUNTER

descr — free text describing the function.

argdescr — the function do not take any arguments so this should be an empty string.

devtype — we will implement a generic software function that runs on top of all adapters. The devtype should therefor be set to *MAPI_DEVICE_ALL*.

restype — results are returned by shared memory so this should be set to *MAPIRES_SHM*.

shm_size — the function returns an unsigned long long counter so this should be set to *sizeof(unsigned long long)*

modifies_pkts — the function do not modify any packets so this should be set to *0*

filters_pkts — the function do not filter any packets so this is also set to *0*

optimize — the function will not use global optimization. So we set this to *MAPIOPT_NONE*.

process — should be set to the address of the pktc_process interface

reset — should be set to the address of the pktc_reset interface

For the rest of the MAPI function interfaces in the structure, the value should be set to *NULL* since these functions do not exist.

With these modifications we get a structure looking like this:

```
static mapidflib_function_def_t finfo={
  "", //libname
  "PKT_COUNTER", //name
  "Counts number of packets\n\tReturn value: unsigned long long", //descr
  "", //argdescr
  MAPI_DEVICE_ALL, //devtype
  MAPIRES_SHM, //Method for returning results
  sizeof(unsigned long long), //shm size
  0, //modifies_pkts
  0, //filters_pkts
  MAPIOPT_NONE, //optimize
  NULL, //instance
  NULL, //init
  pktc_process, //process
  NULL, //get_result,
  pktc_reset, //reset
  NULL, //cleanup
  NULL, //client_init
  NULL, //client_read_result
  NULL  //client_cleanup
};
```

At the end of the file there is also a function called get_funct_info. This is used by the library to return information from the function. The only change necessary for this function is to change the <funct_name> to *pktc* so that the full name of the function becomes pktc_funct_info.

We can now start to look at the actual code for this packet counter MAPI function. Since we use shared memory, allocation and initializing the memory to 0 is already taken care of. In the processing interface all that is needed is to increment the unsigned long long counter in the shared memory. The location of this counter is pointed to by the data pointer in the result structure of the current instance. The code inside the pktc_process interface should then simply be:

```
(*(unsigned long long*)instance->result.data)++;
```

The code for the rest interface is similar and just reset the shared memory counter to zero:

```
(*(unsigned long long*)instance->result.data)=0;
```

The entire source code for the packet counter function should then be:

Function library

To be able to use this new MAPI function from an application, it must be included in a library. For this tutorial we will create a new library, tutlib, by using createlib.pl:

```
../createlib.pl tutlib tutlib.c
```

This will create a new library called *tutlib* in the file *tutlib.c*. All functions in the current directory will automatically be included in the library.

Makefile

We can now compile the new function and library. To do this it is best to create a Makefile. The contents of a makefile could look like this:

```
INCLUDE=-I. -I..
CFLAGS=-g -Wall -Wsign-compare -Wpointer-arith -Wnested-externs \
       -Wmissing-declarations -Wcast-align -D_GNU_SOURCE $(INCLUDE) \
      -DDEBUG=1
TARGETS=tutlib.so
all: $(TARGETS)
tutlib.o: tutlib.c ../mapidflib.h ../mapi.h
  gcc $(CFLAGS) -c $<
tutlib.so: tutlib.o pktcounter.o
  gcc $(CFLAGS) -shared  -o $@ $^ -lfl -lrt -L.. -L. $(LIB_DIR)
pktcounter.o: pktcounter.c
  gcc $(CFLAGS) -c $<
clean:
  @/bin/rm -f *.o *.so *~ $(TARGETS)
```

The new MAPI function and library can now be compiled by running make. The last step to be able to use the new function is to make sure that the tutlib library is loaded when MAPId is started. To do this modify mapi.conf so that the *libpath* entry includes the tutlib directory and the the *libs* entry includes the tutlib library. It should look something like this:

```
libpath=.:ipfixlib:tutlib
libs=mapidstdflib.so:ipfixlib.so:tutlib.so
```

4.2 Function with internal data: GAP

This is a slightly more advanced function that uses internal data to store information between each call to the process interface. The function calculates the gap or the time that has elapsed between to consecutive packets in a flow.

Implementation

Go to the tutlib directory created in the previous section. Copy the function template:

```
cp ../funct_template.c gap.c
```

Interfaces that are not needed should be removed. Since this function uses internal data, the init and cleanup interfaces are needed in addition to the process interface. The string <*funct_ name*> should be replaced with *gap* and the mapidflib_function_def structure needs modification so that it looks like:

```
static mapidflib_function_def_t finfo={
  "", //libname
  "GAP", //name
  "Returns the gap between to consecutive packets\n
   Return value: unsigned long long", //descr
```

```
  "", //argdescr
  MAPI_DEVICE_ALL, //devoid
  MAPIRES_SHM, //Method for returning results
  sizeof(unsigned long long), //shm size
  0, //modifies_pkts
  0, //filters_pkts
  MAPIOPT_NONE, //optimize
  NULL, //instance
  gap_init, //   gap_init,
  gap_process,
  NULL, //get_result,
  NULL, //reset
  gap_cleanup, //cleanup
  NULL, //client_init
  NULL, //client_read_result
  NULL  //client_cleanup
};
```

We are now ready to implement the code for the three interfaces this MAPI function needs. What this function do is to store the time stamp of a packet and when the next packet is captured it calculates the time difference between the new and old timestamp. This means that it needs to store the timestamp of a packet as internal data so that it can be accessed again when the next packet is processed.

Timestamps are unsigned long long values so the init interface should simply allocate memory to store such a value and let the internal_data pointer point to this memory. The function should also initialize the unsigned long long value to 0. The code for this is then:

```
instance->internal_data=malloc(sizeof(unsigned long long));
*((unsigned long long*)instance->internal_data)=0;
```

The process interface stores the difference between the timestamp of the current packet and the previous one and stores it in the result data:

```
unsigned long long *gap,*old;
old=instance->internal_data;
gap=instance->result.data;
if(*old!=0)
  *gap=pkt_head->ts-*old;
*old=pkt_head->ts;
```

When the flow closes, the internal data has to be freed. This is done by the cleanup interface:

```
free(instance->internal_data);
```

The entire source code for this function would then be:

Function library

The function library is recreated using the createlib.pl script as in the previous section:

```
../createlib.pl tutlib tutlib.c
```

Makefile

The Makefile created in the previous section must be modified so that it compiles the new function:

```
INCLUDE=-I. -I..
CFLAGS=-g -Wall -Wsign-compare -Wpointer-arith -Wnested-externs \
-Wmissing-declarations -Wcast-align -D_GNU_SOURCE $(INCLUDE)
TARGETS=tutlib.so
all: $(TARGETS)
tutlib.o: tutlib.c ../mapidflib.h ../mapi.h
  gcc $(CFLAGS) -c $<
tutlib.so: tutlib.o pktcounter.o gap.o
  gcc $(CFLAGS) -shared  -o $@ $^ -lfl -lrt -L.. -L. $(LIB_DIR)
  cp $@ ..
pktcounter.o: pktcounter.c
  gcc $(CFLAGS) -c $<
gap.o: gap.c
  gcc $(CFLAGS) -c $<
clean:
  @/bin/rm -f *.o *.so *~ $(TARGETS)
```

4.3 Function with multiple process interfaces: PKTINFO

We are now going to implement a function that returns some information about a packet. To keep things simple we will limit the information to either packet length or timestamp. Results from this function will typical be used by other functions for further processing.

Implementation

As in the previous examples, start by copying the function template and set the function name to pktinfo:

```
cp ../funct_template.c pktinfo.c
```

This function will need an argument to decide which information that should be returned. This means that we need to specify the type of argument in the mapidflib_function_def structure. We will also need to implement the instance interface to do a syntax check of the argument. The argument type will be an integer so the argdescr should be set to "i".

Since the information returned by the function is the same for the entire lifetime of the function it is a waste of clock cycles to do a check for every packet to decide which type of information should be returned. Instead we can implement two version of the process interface, one that returns packet length and one that returns the timestamp. We can decide in the instance interface which process interface to use.

The mapidflib_function_def structure should then look like this:

```
static mapidflib_function_def_t finfo={
  "", //libname
  "PKTINFO", //name
```

```
  "Returns information about a packet as unsigned long long",
  "i", //argdescr
  MAPI_DEVICE_ALL, //devoid
  MAPIRES_SHM, //Method for returning results
  sizeof(unsigned long long), //shm size
  0, //modifies_pkts
  0, //filters_pkts
  MAPIOPT_NONE, //optimize
  pktinfo_instance,
  NULL, //init
  NULL, //process
  NULL, //get_result,
  NULL, //reset
  NULL, //cleanup
  NULL, //client_init
  NULL, //client_read_result
  NULL  //client_cleanup
};
```

To make things more user friendly we will first create an enum that can be used for specifying the type of information that should be returned by the function. This is placed in the file pktinfo.h:

```
#ifndef _pktinfo_h
#define _pktinfo_h
enum pktinfo {   PKT_TS,PKT_SIZE };
#endif
```

The code in the instance interface should then verify that the value of the argument is either *PKT_ TS* or *PKT_ SIZE* . If the value is not one of these, an error message should be returned. Based on the argument the instance interface should then modify the function instance copy of the mapidflib_function_def structure so that the process pointer points to one of the two processing interfaces that are implemented.

To implement the instance interface we should start by defining to variables, one for storing the type of information that should be returned and one pointer for pointing to the arguments:

```
int type;
mapiFunctArg* fargs;
```

To parse function arguments, the getarg functions found in mapiipc.h can be used. To get hold of the integer that specifies the information type we use:

```
fargs=instance->args;
type = getargint(&fargs);
```

We can now check the value of *type* and set the pointer to the correct process interface accordingly:

```
if(type==PKT_SIZE)
  instance->def->process=pktinfo_process_size;
else if(type==PKT_TS)
  instance->def->process=pktinfo_process_ts;
else
  return MFUNCT_INVALID_ARGUMENT_2;
```

The last step is to implement the two processing interfaces, *pktinfo_process_size* and *pktinfo_process_ts*. The code for the interface returning packet size will be:

```
(*(unsigned long long*)instance->result.data)=
  (unsigned long long)pkt_head->wlen;
return 1;
```

For returning the packet timestamp we get:

```
(*(unsigned long long*)instance->result.data)=
  (unsigned long long)pkt_head->ts;
return 1;
```

The entire code for the pktinfo function will then be:

Function library and Makefile

The function library and Makefile must be updated in the same way as explained in the previous section.

4.4 Function that reads results from other functions: STAT

The statistical function reads unsigned long long values from other functions and calculates the total number of values read, the sum, square of sum, maximum and minimum values. It is a good example of how one function can read results from another function.

Implementation

Copy the function template from the mapi main directory:

```
cp ../funct_template.c stat.c
```

Replace the text "<funct_name>" with the text "stat". Delete interfaces that are not needed. This function needs arguments and uses internal data to store information so it need to implement the instance, init, process and cleanup interfaces.

The function will take three argument, one integer that specifies the flow ID of the flow and one integer that specifies the function ID of the function that the results should be read from and one char that specifies if the result from the first packet should be skipped. The last parameter is needed if we are reading results from a function like GAP. This function will always return a 0 when the very first packet is processed and if we did not skip this first result, the minimum value would always become 0 in the STAT function.

This function will count the number of values read from another function, the sum, square of sum, maximum and minimum value. All this information will be stored in a structure placed in a stats.h file:

```
#ifndef _STATS_H
#define _STATS_H 1
typedef struct stats {
  unsigned long long count; //Number of elements
  long double sum; //Sum
  long double sum2; //Sum of square
  double max; //Maximum value
  double min; //Minimum value
} stats_t;
#endif
```

It is also this structure that is returned to an application that calls mapi_read_result.

The function will then need an instance interface for verifying the arguments, an init interface to initialize internal resources that stores the statistics, a processing interface to do the processing of packets and a reset interface for resetting the results. The mapidflib_function_def interface should then be:

```
static mapidflib_function_def_t finfo={
  "", //libname
  "STATS", //name
  "Returns statistical information about
   unsigned long long values from other functions",
  "iic", //argdescr
  MAPI_DEVICE_ALL, //devtype
  MAPIRES_SHM, //Method for returning results
  sizeof(stats_t), //shm size
  0, //modifies_pkts
  0, //filters_pkts
  MAPIOPT_NONE, //optimize
  stats_instance, //instance
  stats_init, //init
  stats_process, //process
  NULL, //get_result,
  stats_reset, //reset
  stats_cleanup, //cleanup
  NULL, //client_init
  NULL, //client_read_result
  NULL  //client_cleanup
};
```

In the instance interface we must check that the first argument points to an already existing function applied to the flow and we should check that the third parameter is either a 0 or a 1. To get the values of the arguments that is passed to the function getargint and getargchar is used. To check the first two parameter we use a helper function that loops through the flow list and then the function list for the flow if it is found. The code would then be:

```
static int stats_instance(mapidflib_function_instance_t *instance,
                          int fd,
                          mapidflib_flow_mod_t *flow_mod)
{
  int fid,afd;
  mapiFunctArg* fargs;
  int skip;
```

```
  fargs=instance->args;
  afd = getargint(&fargs);
  fid = getargint(&fargs);
  skip=getargchar(&fargs);

  if(fhlp_get_function_instance(instance->hwinfo->gflist,afd,fid)==NULL)
    return MFUNCT_INVALID_ARGUMENT_1;
  if(skip>1 || skip<0)
    return MFUNCT_INVALID_ARGUMENT_2;

  return 0;
};
```

In the init interface we have to store a reference to the function which results are read from as well as the value for the skip argument so that these can be used when processing packets. To do this we define a structure that the internal_data pointer can point to:

```
typedef struct stats_inst {
  mapidflib_function_instance_t *res_instance;
  char skip;
} stats_inst_t;
```

The implementation of the init interface would then look like this:

```
static int stats_init(mapidflib_function_instance_t *instance,
                      int fd)
{
  int fid,afd;
  stats_inst_t *i;
  mapiFunctArg* fargs=instance->args;
  i=instance->internal_data=malloc(sizeof(stats_inst_t));
  afd=getargint(&fargs);
  fid=getargint(&fargs);
  i->skip=getargchar(&fargs);
  if((i->res_instance=fhlp_get_function_instance(
      instance->hwinfo->gflist,afd,fid))==NULL)
  {
    free(i);
    return MFUNCT_INVALID_ARGUMENT;
  }
  return 0;
}
```

The code for processing packets is relatively straight forward. The results are stored in *result.data* and information about which function to read values from is located in *internal_data*. The only problem is how to read the results from another function. The method used depends on how the other function returns results. If the function uses shared memory, the result should be read directly from memory. If however, the function implements its own read_result interface, this interface must be used. To ease the implementation a function in fhelp.[c/h] can be used:

```
fhlp_get_res(i->res_instance)
```

This function will always return a pointer to a mapidflib_result_t structure containing the result. The code for the processing interface will then be:

```
unsigned long long *res;
stats_inst_t *i=instance->internal_data;
stats_t *s=instance->result.data;
if(i->skip==1)
  i->skip=0; //Skip first packet
else {
  //Get result from other function
  res=((mapidflib_result_t*)fhlp_get_res(i->res_instance))->data;

  //Calculate statistics
  if(s->max<*res)
    s->max=*res;
  if(s->min>*res || s->min==0)
    s->min=*res;
  s->count++;
  s->sum+=*res;
  s->sum2+=((long double)*res*(long double)*res);
}
return 1;
```

The reset interface simply has to set the various variables in the result structure to 0:

```
stats_t *stats;
stats=instance->result.data;
stats->count=0;
stats->sum=0;
stats->sum2=0;
stats->max=0;
stats->min=0;
```

The last interface to implement is cleanup which releases the internal data:

```
free(instance->internal-data);
```

The entire code for this function can be found in *stdflib/stats.c*

4.5 Function with function specific get_result: TOBUFFER

The last function we will look at is the TOBUFFER function which store packets in a circular buffer so that clients can read them using mapi_get_next_pkt. The circular buffer uses shared memory, but since two way communication using semaphores are needed to control reading and writing to this buffer, the TOBUFFER function must implement its own get_result interface on the client side.

The description here is not a full description of how to implement the TOBUFFER function, but focus on explaining the challenges of using a function specific method for reading results.

Implementation

In the TOBUFFER function we use shared memory to transfer the packets to the client application. One problem is that we want to be able to store the same number of packets on different types of hardware, which means that the size of this buffer will depend on the maximum packet capture length of the adapter that the function is running on. Because of this it is not possible to put a fixed number in the mapidflib_function_def structure to define the size of shared memory that the function needs.

What we can do instead is to implement the instance function interface and modify the *shm_size* entry inside this interface based on the value of *hwinfo->cap_length*. In addition to the buffer we also need to share a structure that can be used for accessing the circular buffer in a controlled manner:

```
typedef struct to_buffer {
  unsigned long read_ptr; //Pointer to the next packet that can be read
  unsigned long write_ptr; //Pointer to where the next packet can be written
  int cap_length; //Maximum size of a captured packet
  unsigned bufsize;  //Size of buffer
  fhlp_sem_t sem; //Struct containing semaphore info
  char *buf; //Pointer to buffer
} to_buffer_t;
```

The code for deciding the amount of shared memory needed will then be:

```
 instance->def->shm_size=sizeof(to_buffer_t)+
                         NUM_PKTS*(sizeof(struct mapid_pkthdr)+
                         instance->hwinfo->cap_length);
```

Since there are no arguments to this function, there is no other work to be done in the instance interface.

In the init interface we now need to initialize the circular buffer and create semaphores for coordinating the access to the buffer. The to_buffer_t structure is placed in the beginning of the shared memory and the circular buffer occupies the rest. To create a semaphore a function from fhelp.c can be used. The code for the init interface will then be:

```
to_buffer_t *mbuf;
int ret;
mbuf=instance->result.data;
mbuf->buf=(char*)instance->result.data+sizeof(to_buffer_t);
//adding semaphore
if((ret=fhlp_create_semaphore(&mbuf->sem,2))!=0)
{
  DEBUG_CMD(printf("Error initializing semaphore: %d\n",ret));
  return ret;
}
mbuf->read_ptr=0;
mbuf->write_ptr=0;
mbuf->bufsize=NUM_PKTS*(sizeof(struct mapid_pkthdr)+
              instance->hwinfo->cap_length);
mbuf->cap_length=instance->hwinfo->cap_length;
return 0;
```

The processing interface is not described here, but to get hold of the *to_buffer_t* structure it simply uses the results.data pointer.

The cleanup interface needs to release the semaphore created in init. This can also be done by a function in fhelp.c:

```
to_buffer_t *mbuf;
mbuf=instance->result.data;
fhlp_free_semaphore(&mbuf->sem);
```

To be able to return packets to the user, the TOBUFFER function also has to implement a client_get_result interface. It is important to realize that while the code for this interface is in the same source code, it runs on the client side and not inside MAPId as the rest of the interfaces.

Since TOBUFFER uses shared memory the instance->result.data argument to the client_read_result interface will automatically point to the shared memoy. This means that instance->result.data points to the same to_buffer_t structure as seen by the processing interface and that the circular buffer is located right after this structure:

```
to_buffer_t *tb=instance->result.data;
char *buf=(char*)instance->result.data+sizeof(to_buffer_t);
```

The entire source code for this function can be found in the file stdlib/tobuffer_all.c