# Linux MAPI programming over ExchangeRPC

Julien Kerihuel[1]
Dan Shearer[2]

*1:* OpenChange SARL, <j.kerihuel@openchange.org>
*2.* OpenChange Project, <dan@openchange.org>

**Abstract**

The OpenChange MAPI library aims to provide an Open Source implementation of Microsoft Exchange protocols under unix/linux. Firstly we provide a general overview of MAPI internals and the OpenChange MAPI library architecture. Then we focus on real-world programming examples and detail the internals of two sample applications based on libmapi, fetching Exchange e-mails and calendar items from a Linux terminal. Finally we provide an overview of graphical and console Linux desktop applications using OpenChange.

# Contents

# List of Figures

# 1 Opening Exchange to a wider world

## 1.1 OpenChange Project Goals

The OpenChange Project aims to provide a portable Open Source implementation of Microsoft Exchange Server and Exchange protocols. Exchange is a groupware server designed to work with Microsoft Outlook, and providing features such as a messaging server, shared calendars, contact databases, public folders, notes and tasks.

The OpenChange project has three goals:

- To provide a library for interoperability with Exchange protocols, and to assist implementors to use this to create groupware that interoperates with both Exchange and other OpenChange-based software.
- To provide an alternative to Microsoft Exchange Server which uses native Exchange protocols and provides exactly equivalent functionality when viewed from Microsoft Outlook clients.
- To develop a body of knowledge about the most popular groupware protocols in use commercially today in order to promote development of a documented and unencumbered standard, with all the benefits that standards bring.

## 1.2 Linux Users in a Microsoft Exchange World

In heterogeneous IT infrastructure containing both Unix/Linux and Windows operating systems, groupware decisions gravitate to Microsoft. There are three reasons for this:

- requirements specification is driven by IT decision-makers who are completely immersed in the Microsoft environment. There is little reason for them to think in terms of requirements Exchange cannot deliver, because they have never seen alternative approaches to the problem.
- the nature of monopoly in extinguishing recognition of other brands. Monopoly has created today's management layers who are rarely aware there could be non-Microsoft ways of delivering core functionality. So even if someone does see an unfulfilled need or conceive of a better way groupware might function for their organisation there is nothing obvious they can do about it. A quick call to the IT department will quickly elicit indignant squawks about compatibility!
- the desire for companies to roll out one size-fits-all infrastructure solutions in the interests of efficiency.

Once Exchange is in one part of the company, the universal requirement for Exchange happens in most companies by the IT Director taking the decision to deploy Microsoft Exchange client licenses to all employees. Even if there are thousands of Unix users, engineers perhaps, who have no need for or interest in Microsoft software.

The justification for this is that Microsoft-style global shared groupware can work quite well if sufficiently well-funded, and everyone needs shared calendaring, so a problem is being efficiently addressed. And so the global Microsoft Exchange deployment decision justifies itself. The company is now locked into the well-documented Exchange fix and upgrade cycle, and no alternative solutions can be introduced to the company without great upheaval.

In smaller companies there is even less of a debate. The Unix-using employee (*"You!"*) is required to configure their Outlook client. From Monday, the employee is asked to schedule all his business meetings in a Microsoft shared calendar and use the groupware messaging system for internal communications. If you ask why, the story is:

*Users are already familiar with Microsoft Office and regularly use Microsoft Outlook for business messaging. Microsoft Exchange Server provides us vital functions (messaging, calendars, contact databases, tasks, notes and integration with mobile devices and voicemail), and what could work better with Microsoft Outlook? It will improve information sharing efficiency no end.*

Since there is nothing to compare against ("Does it support Outlook natively?") there is no choice. If you are lucky you might be able to use Microsoft Exchange through its Web Access or the existing Evolution connector. But many IT departments restrict Exchange server access to *MAPI clients only* meaning you have to run a Microsoft Windows operating system with Microsoft Outlook, or perhaps some other solution such as CodeWeavers Crossover Office if allowed.

OpenChange addresses this issue, allowing you to say "yes!" to the native Outlook question. The OpenChange MAPI library offers users access to Microsoft Exchange Server from any supported operating system **using native Microsoft Exchange protocols**.

## 1.3 MAPI Overview

MAPI is the glue between Exchange and Outlook, but a common misconception is to consider it as a network protocol. MAPI, an acronym for *M*essaging *A*pplication *P*rogramming *I*nterface, refers to a proprietary set of function call interfaces developed by Microsoft before Microsoft Exchange existed. By purchasing licenses to Microsoft's proprietary and Windows-only MAPI libraries, anyone can create message services that communicate using these functions. A mail server implemented in this way is what Microsoft calls a MAPI Service Provider. Any protocol could be used as a transport for these MAPI communications.

When Microsoft Exchange 5.5 was developed in 1997, the decision was taken to create a proprietary transport protocol for MAPI which closely matches the MAPI calling interface. This protocol is called ExchangeRPC and used in Outlook-Exchange communications. ExchangeRPC is the only transport OpenChange supports, and in practice is the only transport of interest today. Most of the world is being forced to use Microsoft Exchange servers, so that defines the transport that matters.

When OpenChange team members first looked at the network network traffic these generated by calling MAPI functions on Windows operating systems, we noticed blobs of data first either compressed or obfuscated, then encapsulated by an RPC transport protocol function (EMSMDB [1] [6]) and finally pushed on the wire (figure 1). Transporting memory-image blobs on the wire is not good protocol design, however for a number of reasons the result now works quite reliably today in 2007.

[1] Exchange Transport

### 1.3.1 Describing a MAPI Conversation

A high-level view of a MAPI conversation follows. We also introduce important terminology:

1. MAPI applications call MAPI providers, using the API to pass data (eg a mail message body) or MAPI conversation requests and responses (eg 'search for this address').
2. MAPI providers pack the client or server MAPI information in a blob. There are only two really important providers, one for data destined for what is somewhat strangely termed the Message Store (although it handles more than just messages), and the other for data to be sent to the Address Book. These are called the EMSMDB and EMSABP providers, respectively.
3. ExchangeRPC protocol is used to transport the MAPI information over the wire, encapsulating inside it one of two MAPI-specific protocols: the EMSMDB Message Store Protocol, or the NSPI Addressbook Protocol.
4. The store provider on the server side associated with the protocol used (either EMSMDB or EMSABP) extracts the MAPI blob from RPC protocol functions, analyzes its content and performs operations embedded within it.

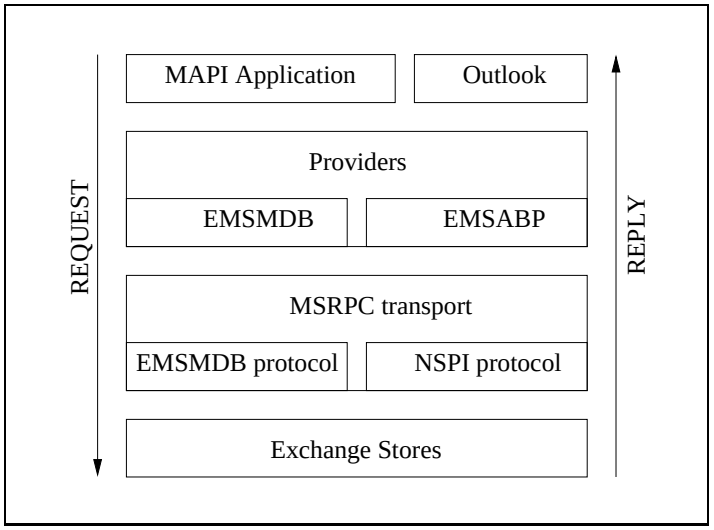


Figure 1: General MAPI architecture overview

### 1.3.2 How The Layers Interoperate

Depending on the kind of data MAPI applications need to retrieve, lower levels will either rely on EMSABP provider and NSPI [2] protocol for Address book operations or EMSMDB provider and EMSMDB protocol for message store operations. For example, when users compose emails with Outlook, internal organization usernames are autocompleted and magically turn to underline, and full names are supplied instead of the initial user entries. Outlook will use the `ResolveNames`

[2]Name Service Provider Interface over an RPC protocol, providing access to the global addressbook stored either in Exchange or in LDAP

MAPI function, which passes the data to an EMSABP provider for names resolution. In contrast, to send a message Outlook will call `SubmitMessage` MAPI function which sends the data to an EMSMDB provider. Although different providers get used in *MAPI communications*, according to Ronnie Sahlberg [7] the single EMSMDB RPC protocol function **EcDoRpc** (function 0x2) represents nearly all MAPI traffic.

## 1.4 OpenChange, Samba and the OpenChange MAPI library

This section documents the close and symbiotic relationship between the OpenChange and Samba[3] projects. OpenChange depends on Samba and the two projects are highly compatible, but they remain separate projects. The coupling is kept as loose as possible.

### 1.4.1 Architectural Relationship

The OpenChange MAPI library (libmapi) is developed in C, released under the GNU GPLv2 licence (with the *or later version* option. OpenChange will upgrade to GPLv3.)

Samba is a suite of programs and libraries also developed in C and released under the GPLv3 licence. Samba addresses the entire *Common Internet Filesystem* (CIFS) protocol. Version 4 of Samba introduces a modular design, and OpenChange uses this to choose from the very large Samba codebase the minimum amount of functionality to link against:

- MSRPC transport
- talloc [9, 11] memory allocation system
- ldb LDAP database.

talloc and ldb have recently been separated out into standalone projects, both of them still core to Samba4 and still also in the Samba source tree.

OpenChange provides the **full framework** needed to create effective MAPI clients for many operating systems (figure 2):

- **EMSMDB and NSPI protocols** support
- **MAPI API** that exposes these protocols

### 1.4.2 Practical Relationship

The Samba project has a great deal of experience handling Interface Definition Language (IDL) descriptions of RPC traffic, and encapsulation of the Network Data Representation (NDR) format used with RPC, and debugging these. This work has been extremely valuable to OpenChange, and probably made OpenChange possible in the first place by reducing the problem to manageable proportions. In 2007, Samba is the result of fifteen years of continuous implementation of CIFS protocols, and hundreds of man-years of effort.

[3] `http://www.samba.org`

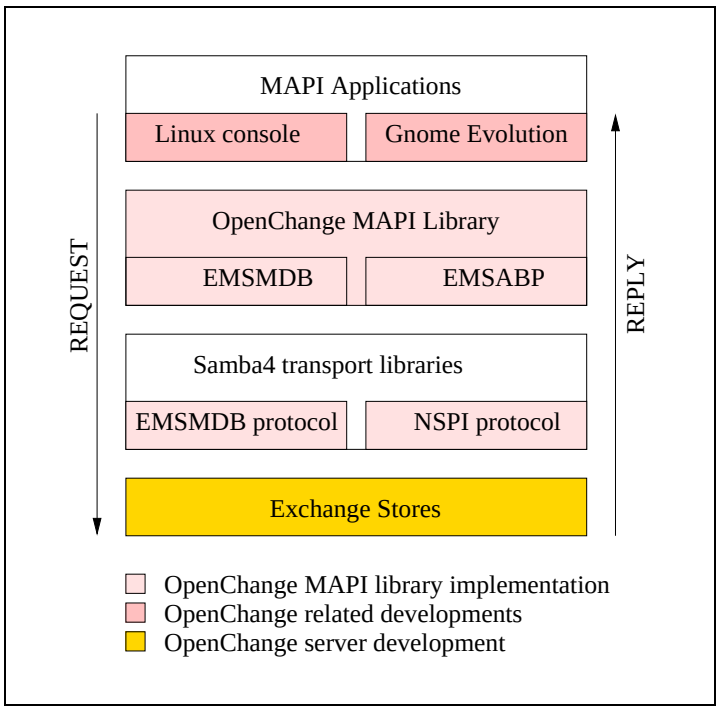


Figure 2: General OpenChange MAPI architecture overview

Then there is the special case of the OpenChange server implementation. A main goal of the OpenChange project is to produce a set of libraries that allow people to build ExchangeRPC-compatible clients and servers. In the case of the OpenChange server we are very much our own customer! The OpenChange ExchangeRPC server implements the facilities available with Microsoft Exchange and expresses them through EMSMDB and EMSABP server providers.

Since Microsoft Exchange is tightly integrated with Active Directory and other Windows services, the OpenChange server needs access to an Active Directory service, and what better Active Directory server could there be than Samba? So although libmapi only links against certain Samba libraries, the OpenChange server requires a full Samba installation.

Samba implements a plugin architecture for third-party implementors, and the OpenChange server is the first such plugin. Architecturally the OpenChange server can be thought of as running inside Samba! Being the first plugin, the OpenChange team has worked closely with the Samba team in terms of testing and refining the plugin system.

## 1.5 MAPI library design

The libmapi design goals were developed after carefully studying the public Microsoft MAPI programming documentation and considering Exchange protocols on the wire. The goals are:

- To provide a set of functions with similar semantics to the Microsoft C++ API, maximising the benefit of shared information across the two implementations. libmapi needs to be easily usable by Linux developers and Windows developers alike, including Windows developers who already know MAPI.
- To improve on the state of the art in MAPI implementations, maximising the efficiency of programmers who use libmapi. Wherever possible libmapi hides implementation details from the user, so that all they have to know is that they are passing in a groupware-related object and getting back a response in order to have a groupware conversation.

The practical result in terms of nomenclature is that the similar API function names help in comparing programming resources, from the Microsoft MSDN website, the online OpenChange resources and in discussion groups.

The implementation was a more difficult issue. Exchange is implemented by instantiating MAPI objects on the server and then operating on these objects by means of unique handles. Knowledge of this process is built into the EMSMDB protocol. libmapi has retained this design, but the implementation is significantly different in two ways:

- libmapi has a global context, rather than separately instantiated objects with private members. This gives less work for libmapi users and reduced risk of making common programming errors. It also allows libmapi to have substantial internal reworking without impacting public function prototypes.
- libmapi hides internal mechanisms. MAPI objects are opaque, and passed by reference between MAPI functions. Although the API is very similar to Microsoft MAPI, the Microsoft MAPI C++ object hierarchy is not implemented.

This architecture has served OpenChange well so far. The development team concentrates on implementing core MAPI functionality rather than building an object hierarchy that may later turn out to be suboptimal.

## 1.6 Overview of MAPI library capabilities

### 1.6.1 Wide compatibility

Ever since the `libmapi-0.2` release, OpenChange has received regular successful interoperability reports from testers running libmapi against various versions of Exchange server. Despite several areas where we know our knowledge is incomplete, applications build on libmapi work with the following Microsoft servers:

- Exchange 5.5
- Exchange 2000
- Exchange 2003
- Exchange 2007 (Once Public Folder store is created)
- Small Business Server 2003

### 1.6.2 MAPI profiles

OpenChange implements MAPI profiles with an objective similar to Microsoft one. Profiles are administrated through the `IProfAdmin` interface supplied in the MAPI library and stored in a LDB database [4]. They are used to set up an Exchange account on your workstation and hold information needed in further Outlook-Exchange communications:

- **Connection information**: IP address, binding strings, Exchange NETBIOS name, Windows domain name.
- **Credentials**: Exchange account username and password. Note that the password can be supplied to `IProfAdmin` functions and can be stored outside the profile database.
- **User information**: Email address, HomeMDB.

### 1.6.3 Common MAPI objects

- **Messages**: Fetching, sending, deleting and moving operations are available. We can delete emails from the Inbox or move them to another folder. We can set flags on the message such as read/unread or importance. We can fetch email attachments. We can send emails to other Exchange users, or to external users over the Internet. We can specify recipients (To, Cc, Bcc), set plain text or HTML content and add attachments.
- **Calendars**: We can fetch, create or delete appointments. We can set almost any appointments values such as start and end date, reminders, subject, content, location, categories or contacts.
- **Tasks**: We can fetch, create or delete tasks and set their attributes
- **Contacts**: We can fetch, create or delete contacts and set their attributes.
- **Notes**: We can fetch, create or delete notes.

### 1.6.4 MAPI containers

MAPI consists of objects and containers. Containers can hold *leaf* objects or other containers. OpenChange provides API functions to create, or delete folders, or delete folder contents.

### 1.6.5 MAPI tables and searching operations

Search operations are performed on the server and results are sent back to the client. These operations are realized on MAPI tables which can be defined as a temporary or a fixed view of a given container. These tables looks like SQL with columns and rows. OpenChange implements MAPI tables and provides methods to search, filter or restrict results given a specific rule.

### 1.6.6 MAPI Notifications

In addition to ExchangeRPC, Microsoft Exchange can communicate with clients using UDP packets for short notifications. A client must request this form of notification, and Outlook routinely does so. UDP communications are always from Exchange to the client, and to a random port number negotiated with the client. The most common notification is *NEWMAIL*, but clients can also request

[4] LDAP-like embedded database: `http://ldb.samba.org`

alert events triggered by a particular object being modified or by a critical error encountered by the server. OpenChange supports supports NEWMAIL notifications over UDP, and the OpenChange team anticipates a general implementation of the notification system to be very achievable.

### 1.6.7 MAPI Permissions

OpenChange also provides methods to Add, Modify or Remove permissions on a given folder using pre-defined Roles and Rights matching the pool of ACLs listed in Outlook. While we provide a public method to set credentials, we recommend to use one of the OpenChange `AddUserPermission`, `ModifyUserPermission` or `DeleteUserPermission` functions. These methods were designed to help developers in setting Access Control List easily and avoid wasting time setting structures required by the `ModifyTable` call.

### 1.6.8 Public Folders

Public Folders (PF) are collaborative area where users can collaboratively share information. Public Folders appear to the end user to be a specialised type of folder and the openchangeclient application can list the PF hierarchy. However Public Folders are normal folders in the mailbox store with special sets of permissions, and the special handling required is all on the client side.

# 2 Getting Started with OpenChange

## 2.1 OpenChange requirements

The most up-to-date instructions for installing and configuring OpenChange are in the `howto.txt` in the `docs` directory of the OpenChange source tree. You can always see the latest version via the websvn [8]. `howto.txt` will always supersede this section.

### 2.1.1 Installing Samba4

First of all, install Samba 4 correctly (see `howto.txt` in the Samba4 package). This should give you the libraries, headers and tools you need to compile OpenChange. As a hint, as well as your base compiler (`apt-get` install build-essential on Debian and Ubuntu) you will need `automake` and `pkg-config`:

- Get the latest SAMBA_4 trunk revision or one of the latest technical previews (**TP5** or later):

```
svn co svn://svnanon.samba.org/samba/branches/SAMBA_4_0 samba4
http://us1.samba.org/samba/ftp/samba4/
```

- Install SAMBA_4

```
$ cd samba4/source
$ ./autogen.sh
$ ./configure.developer --prefix=/usr/local/samba
$ make
\$ su
# make install
```

- Since the libraries will be installed in `/usr/local/samba/lib`, make sure this is listed in /etc/ld.so.conf and run `ldconfig`. On Linux, check with `ldconfig -v` afterwards.

- Install `PIDL` from Samba4 source directory:

```
# cd pidl && perl Makefile.PL && make && make install
```

- Adjust your `PKG_CONFIG_PATH` environment variable:

```
$export PKG_CONFIG_PATH=/usr/local/samba/lib/pkgconfig
```

### 2.1.2 Installing other libraries

In order to compile OpenChange, you will also need to install `libmagic` (`file-libs` and `file-devel` rpm packages on Fedora). This library is used by the `exchange2mbox` tool to detect MIME types.

## 2.2 Installing OpenChange

Once the environment is configured properly, compiling OpenChange is trivial:

- Get the latest OpenChange trunk revision:

```
$ svn co https://svn.openchange.org/openchange/trunk openchange
```

- Install OpenChange

```
$ cd openchange
$ ./autogen.sh
$ ./configure --prefix=/usr/local/samba
$ make
# make install
```

- Since the libraries will be installed in `/usr/local/samba/lib`, make sure this is listed in /etc/ld.so.conf and run `ldconfig`. On Linux, check with `ldconfig-v` afterwards.

### 2.2.1 Setting a MAPI profile database

Prior we write our first MAPI application, we need to create a profiles database and configure a profile. MAPI profiles hold common information used in various MAPI library operations and are required when connecting to Exchange (see 1.6.2). This task can be done with the `mapiprofile` tool.

- First we need to create the profile database:

```
$ mapiprofile -n
$ ls ~/.openchange
profiles.ldb
```

- We next create a profile.

```
$ mapiprofile -P linuxconf -I 192.168.194.22 -u jkerihuel -p openchange \
-D openchange -M laptop --create
Profile linuxconf completed and added to database /home/jkerihuel/.openchan\
ge/profiles.ldb
```

  In the example above, we create a profile named `linuxconf`. We query the Exchange server located at the address `192.168.194.22` with the username `jkerihuel` and password `openchange`. We specify the Windows domain `openchange`, the workstation hostname (here `laptop`) and finally specify the `--create` command.

- We can check if the profile was created correctly with the `--list` command and dump parts its contents with the `--dump` command:

```
$ mapiprofile --list
We have 1 profiles in the database:
        Profile = linuxconf
$ mapiprofile --dump -P linuxconf
Profile: linuxconf
        username       == jkerihuel
        password       == openchange
        mailbox        == /o=First Organization/ou=First Administrative Gro\
up/cn=Recipients/cn=jkerihuel
```

```
        workstation    == laptop
        realm          == (null)
        server         == 192.168.194.22
```

- We will end MAPI profiles operation by setting `linuxconf` as the default profile. Since we executed mapiprofile using the default database location, this latest operation will mark `linuxconf` as the default one in the database and avoid users from specifying the profile name on command line:

```
$ mapiprofile -S -P linuxconf
Profile linuxconf is now set the default one
$ mapiprofile --list
We have 1 profiles in the database:
        Profile = linuxconf [default]
```

## 2.3 Building a sample application

### 2.3.1 Writing our Hello Exchange example

```
#include <libmapi/libmapi.h>

#define   DEFAULT_PROFDB_PATH     "%s/.openchange/profiles.ldb"

int main(int argc, char *argv[])
{
        TALLOC_CTX              *mem_ctx;
        enum MAPISTATUS         retval;
        struct mapi_session     *session = NULL;
        char                    *profdb;
        const char              *profname;

        mem_ctx = talloc_init("mapi_sample1");
        profdb = talloc_asprintf(mem_ctx, DEFAULT_PROFDB_PATH, getenv("HOME"));

        retval = MAPIInitialize(profdb);
        mapi_errstr("MAPIInitialize", GetLastError());
        if (retval != MAPI_E_SUCCESS) return -1;

        retval = GetDefaultProfile(&profname, 0);
        mapi_errstr("GetDefaultProfile", GetLastError());
        if (retval != MAPI_E_SUCCESS) return -1;

        retval = MapiLogonEx(&session, profname, NULL);
        mapi_errstr("MapiLogonEx", GetLastError());
        if (retval != MAPI_E_SUCCESS) return -1;

        MAPIUninitialize();
        talloc_free(mem_ctx);
        return 0;
}
```

The listing above, doesn't sound like a *Hello World* as it calls more than a single function. It is anyway the most minimal code we can write that actually does talk to an Exchange Server and pro-

vides a good overview on how libmapi can be used, thus. Discussing `talloc` related functions (`talloc_init`, `talloc_asprintf`, `talloc_free`) would be outside the scope of this document, but `talloc` [9] can be summarized as *the core memory allocator used in Samba4*.

The first MAPI library function called is `MAPIInitialize`. As its name denotes, this function initializes MAPI library: It creates the MAPI global context, open the profile database store (database path passed as function parameter) and initialize Samba4 transport layer. **This function must be called prior any other MAPI library operations**.

Once MAPI is initialized, we need to create connection to Exchange and open MAPI sessions with the user credentials. This is the purpose of `MapiLogonEx` which needs to be executed prior doing any effective code. This function takes a pointer on a `mapi_session` structure, the profile username and an optional password in case you decided to store it outside the profile. In the example above, we retrieve the default profile name from the database using `GetDefaultProfile`. Note that `MapiLogonEx` opens connections both on EMSMDB and EMSABP store provider. If you intend to interact with a single provider, use `MapiLogonProvider` instead.

Finally we call `MAPIUninitialize` prior leaving the function. This opaque function will clean up the memory allocated during the session and stored within the global MAPI context.

Below is a brief reminder of essential steps developers should implement when writing Linux MAPI program:

1. **Initialize the MAPI library**.
2. **Open a MAPI session**
3. **Uninitialize MAPI library**

Last but not least, a few words on MAPI errors related functions. Public MAPI functions return values deal with `enum MAPISTATUS` values, but only return `MAPI_E_SUCCESS` (0x0) on success otherwise -1. The actual MAPI error code (when errors occur) is set in the global context structure to one of the value listed in `trunk/libmapi/conf/mapi-codes` and accessed through the `GetLastError` function.

### 2.3.2 Compiling and Running the application

Once compiled (remind to set `PKG_CONFIG_PATH` properly as described in 2.1.1), `mapi_sample1` should produce the output below.

```
$ gcc mapi_sample1.c -o mapi_sample1 ‘pkg-config libmapi --cflags --libs‘
$ ./mapi_sample1
    MAPIInitialize           : MAPI_E_SUCCESS (0x0)
    GetDefaultProfile        : MAPI_E_SUCCESS (0x0)
    MapiLogonEx              : MAPI_E_SUCCESS (0x0)
```

If your `mapi_sample1` output is similar to the example above, then congratulations, you have successfully created your first Linux MAPI application. In case you encounter execution errors at this stage, you may have a look to your profile database (2.2.1).

# 3 MAPI Concepts

## 3.1 MAPI objects

Almost any MAPI data you access, read or edit is associated to objects. No matter you intend to browse mailbox hierarchy, open folders, create tables or access items (messages, appointments, contacts, tasks, notes), you will have to initialize and use MAPI objects: **object understanding and manipulation is fundamental**.

- When developing MAPI clients with Microsoft framework, instantiated objects inherit from parent classes. As a matter of fact, developers know which methods they can apply to objects and we suppose it makes their life easier.
- In OpenChange, objects are opaque. They are generic data structures which content is set and accessed through MAPI public functions. Thereof, Linux MAPI developers must know what they are doing.

### 3.1.1 MAPI object related functions

Prior developers use MAPI objects, they are required to initialize them with `mapi_object_init`. This function will reset opaque structure fields and associates the object to the current MAPI session. Once objects are initialized, they can safely be used by MAPI functions.

When an object reaches its end of life, developers are asked to release it using the `mapi_object_release` method. This function will clean any memory allocated for the object and free the object on the Exchange server.

1. **`mapi_object_init`**
2. object manipulation with MAPI functions
3. **`mapi_object_release`**

A example of MAPI object manipulation is provided in the figure below:

```
mapi_object     obj_store;

[...]

mapi_object_init(&obj_store);
retval = OpenMsgStore(&obj_store);
if (retval != MAPI_E_SUCCESS) {
        mapi_errstr("OpenMsgStore", GetLastError());
        exit (1);
}
mapi_object_release(&obj_store);
```

Figure 3: MAPI object usage example

### 3.1.2 MAPI Handles

Beyond memory management considerations, understanding MAPI handles role in object manipulation provides a better understanding why `mapi_object_release` matters.

Handles are temporary identifiers returned by Exchange when you access or create objects on the server. They are used to make reference to a particular object all along its session lifetime. They are stored in unsigned integers, are unique for each object but temporary along MAPI session. Handles are the only links between objects accessed on the client side and efficiently stored on the server side.

Although OpenChange MAPI makes handles manipulation transparent for developers, `mapi_object_release` free both the allocated memory for the object on client side, but also release the object on the server.

## 3.2 MAPI properties

Properties are the attributes of a MAPI object and are used to describe something associated with the object. They are composed of a property tag and a property value. Property tags are unsigned integers constants wherein two distinct elements can be identified: the property identifier and the property type. The property identifier defines the purpose of the property and the range of properties it belongs to while the property type describe the kind of data associated with the property ([5] [3][2] [4]). See figure 4 below for a general overview of properties.

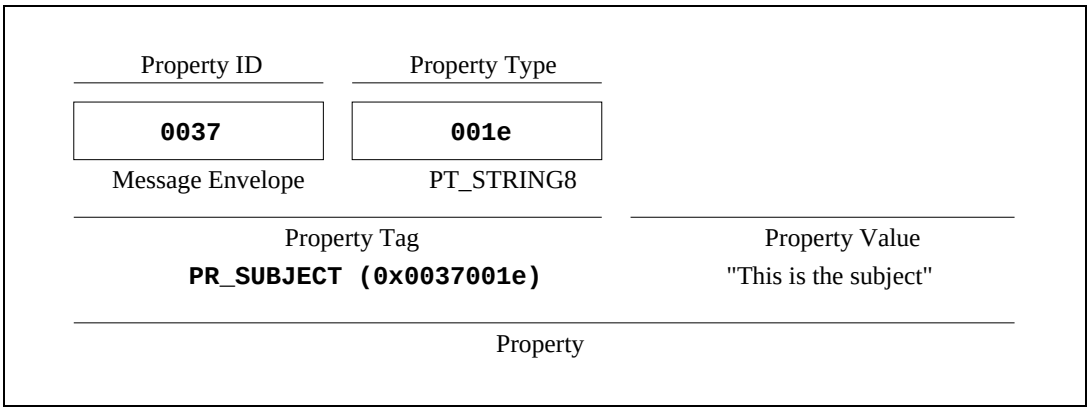


Figure 4: MAPI property overview

### 3.2.1 Property Type

MAPI provides two kind of property types: single-valued and multi-valued. Multi-valued types differentiate from single ones with a value resulting from the addition of a single-valued type and 0x1000. For example:

- **0x001e** = `PT_STRING8` **single UTF8 string**
- **0x101e** = `PT_MV_STRING8` **array of UTF8 strings**

### 3.2.2 Accessing MAPI properties in OpenChange MAPI

When programming with MAPI, `SPropValue` structures are used to represent MAPI properties (See Appendix B for an overview of its structure) and amongst public MAPI functions manipulating them, we often refer to `GetProps` and `SetProps` (available in `libmapi/IMAPIProp.c`).

**GetProps:** *Returns values of one or more properties for a given object*. The property tags array used by this function is stored in a `SPropTagArray` structure and set through the convenient `set_SPropTagArray` function from `libmapi/property.c`. When available, `GetProps` returns values associated to requested MAPI tags for the given object, otherwise a MAPI error code is set (generally `MAPI_E_NOT_FOUND`, `MAPI_E_NO_ACCESS` or `MAPI_E_NO_SUPPORT`) and is prefixed with a `PT_ERROR` (`0xa`) layout byte.

OpenChange has also introduced a range of functions designed to extract MAPI properties values from `SPropValue`, `SRowSet` or `SRow` structures and avoid common beginners mistakes assuming properties values are always available. Sample example of their use is available in section 4.

**SetProps:** *Set one or more properties on a given object*. This function takes an array of `SPropValue` structures and add or change MAPI properties associated with the object.

## 3.3 MAPI tables

MAPI tables are used to view MAPI objects as a set of rows and columns; where objects are rows and MAPI properties columns of the table. (see figure 5 below). So far OpenChange has identified three possible use of MAPI tables: hierarchy, contents and permissions tables.

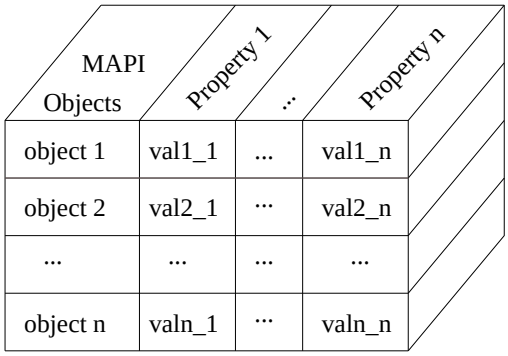


Figure 5: Sample MAPI table overview

### 3.3.1 Creating tables

**GetHierarchyTable** is used on containers to collect information about child containers. For example we will use this function to retrieve a pointer on a MAPI table listing Inbox sub-directories (children).

**GetContentsTable** is used to collect information about objects within a container. If we refer to the example described in the paragraph above, `GetContentsTable` would be used to retrieve a MAPI table pointer on objects. We would next be able to fetch object properties such as message envelope data (subject, body, recipients etc.).

**GetTables** is used to retrieve information about permissions for a given container or object.

### 3.3.2 Browsing tables

**SetColumns:** Once we retrieve a pointer on a table object, we need to customize the view and define the columns we want. This task is performed through the `SetColumns` call. It takes a pointer on the MAPI table object and a pointer on a `SPropTagArray` structure describing the property tags we want to view.

**GetRowCount:** We can use this function to retrieve the number of rows in the table.

**QueryRows:** When the table object is created and columns are defined, we can call this function to browse the table and retrieve rows with their associated data.

1. **Get a pointer on a MAPI table**
2. **Customize the view** with `SetColumns`
3. **Access rows and records** with `QueryRows`

# 4 Writing MAPI applications

## 4.1 MAPI fetchmail example

The complete MAPI fetchmail listing is available in Appendix D.

### 4.1.1 Initializing OpenChange MAPI

```
 retval = MAPIInitialize(profdb);
 MAPI_RETVAL_IF(retval, retval, mem_ctx);

 retval = GetDefaultProfile(&profname, 0);
 MAPI_RETVAL_IF(retval, retval, mem_ctx);

retval = MapiLogonEx(&session, profname, NULL);
MAPI_RETVAL_IF(retval, retval, mem_ctx);
```

As described in 2.3.1, we initialize MAPI library with the profiles database path, retrieve the default profile name (assumption is made you followed mapi profiles steps exposed in 2.2.1) and open connections both to Exchange message store provider (EMSMDB) and Exchange Address Book provider (EMSABP).

### 4.1.2 Open the message store

```
mapi_object_init(&obj_store);
retval = OpenMsgStore(&obj_store);
MAPI_RETVAL_IF(retval, retval, mem_ctx);
```

Now we have opened a connection to the Exchange message store provider, we can open the user mailbox store with `OpenMsgStore`. This function will return a set of pre-defined folder unique IDs (stored on double values) and a *pointer* to the upper object we can access in MAPI hierarchy.

### 4.1.3 Opening Inbox folder

```
retval = GetDefaultFolder(&obj_store, &id_inbox, olFolderInbox);
MAPI_RETVAL_IF(retval, retval, mem_ctx);

mapi_object_init(&obj_inbox);
retval = OpenFolder(&obj_store, id_inbox, &obj_inbox);
MAPI_RETVAL_IF(retval, retval, mem_ctx);
```

We now open the Inbox folder. Since `OpenMsgStore` returns a set of *common* folders identifiers we store in the message store object (`obj_store`), we can retrieve them using the convenient `GetDefaultFolder` function. This function doesn't generate any network traffic, but returns the folder identifier associated with the constant passed as argument (here `olFolderInbox`). See Appendix C for a list of supported folders.

According to section 3.3.1, we could have used MAPI tables and `GetHierarchyTable` function

to find Inbox folder identifier. We would have had to retrieve the Top Information Store hierarchy table, customize the view with `PR_FID` (Folder ID MAPI property) and `PR_DISPLAY_NAME`, find the `IPM_SUBTREE` folder identifier, open it, retrieve the Hierarchy Table, call `SetColumns` with `PR_FID` and finally browse table rows until we find the Inbox folder.folders to store emails within `IPM_SUBTREE` folder hierarchy (see figure 6 below).

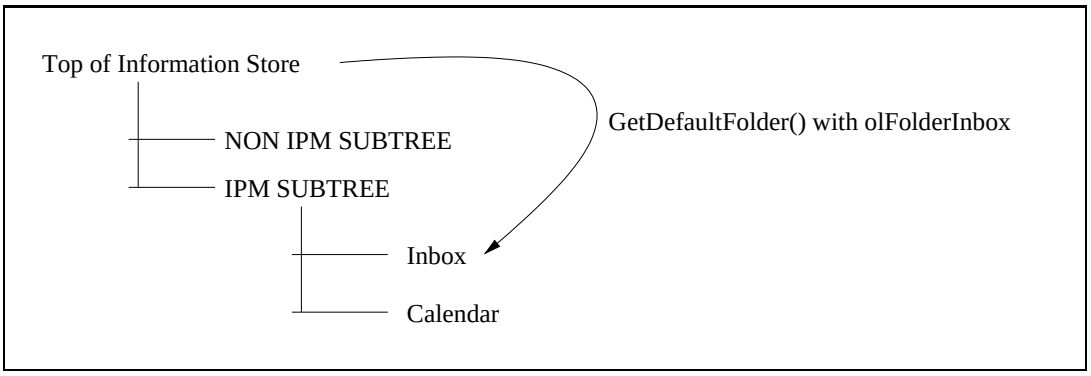


Figure 6: Truncated Mailbox hierarchy overview

Furthermore its complexity, this method outlines a major drawback. If we intend to produce a generic application, we can neither rely on folder names as they change across languages, nor on folders container class such as IPM.Post as we may create several message folders within `IPM_SUBTREE` folder hierarchy. For these reasons, developers must use `GetDefaultFolder` when accessing any *standard* mailbox folders.

#### 4.1.4 Retrieve contents table

```
mapi_object_init(&obj_table);
retval = GetContentsTable(&obj_inbox, &obj_table);
MAPI_RETVAL_IF(retval, retval, mem_ctx);
```

Once the Inbox folder is opened, we can call `GetContentsTable` to create the view needed to list all the children objects. In the current example we suppose we will only retrieve IPM.Post objects (emails).

#### 4.1.5 Customizing the MAPI view

```
mem_ctx = talloc_init("MAPI Table");
SPropTagArray = set_SPropTagArray(mem_ctx, 0x2, PR_FID, PR_MID);
retval = SetColumns(&obj_table, SPropTagArray);
MAPIFreeBuffer(SPropTagArray);
MAPI_RETVAL_IF(retval, retval, mem_ctx);
talloc_free(mem_ctx);
```

We now customize the MAPI view and set the columns with the property tags we want to access: `PR_FID` (Folder Identifier) and `PR_MID` (Message identifier). MAPI uses unique and permanent identifiers to classify objects. These identifiers are double values (8 bytes) and never change until you move the object to another location.

#### 4.1.6 Browsing the table

```
retval = GetRowCount(&obj_table, &count);
MAPI_RETVAL_IF(retval, retval, NULL);

while ((retval = QueryRows(&obj_table, count, TBL_ADVANCE, &rowset) != -1)
        && rowset.cRows) {
        for (i = 0; i < rowset.cRows; i++) {
             fid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_FID);
             mid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_MID);
             mapi_object_init(&obj_message);
             retval = OpenMessage(&obj_store, *fid, *mid, &obj_message);
                        if (retval != MAPI_E_NOT_FOUND) {
                                retval = GetPropsAll(&obj_message, &props_all);
                                mapidump_message(&props_all);
                                mapi_object_release(&obj_message);
                        }
       }
}
```

We now enter the last step of the fetching process:

- Call `GetRowCount` to retrieve the number of rows in the contents table
- Recursively call `QueryRows` (see 3.3.2) with `TBL_ADVANCE` flag to fetch table rows.
- Iterate through QueryRows results
- Retrieve columns values for each row with the convenient `find_SPropValue_data` (from `libmapi/property.c`)
- Open the message given its folder and message ids.
- Call `GetPropsAll` rather than `GetProps` to retrieve all properties associated with a given object
- Call one of OpenChange mapidump API function to display nice messages dump on standard output.

#### 4.1.7 Cleaning and Exiting MAPI

```
 mapi_object_release(&obj_table);
 mapi_object_release(&obj_inbox);
 mapi_object_release(&obj_store);

 MAPIUninitialize();
 return (0);
```

We finally release mapi objects and MAPI library properly and returns.

## 4.2 MAPI fetchappointment variant

The complete MAPI fetchappointment listing is available in Appendix E but is similar to fetchmail listing except two minor changes:

- We change the default folder constant from `olFolderInbox` to `olFolderContact` so any further operations are performed on a child of the calendar folder.
- We use `mapidump_appointment` rather than `mapidump_message` to dump appointments on standard output

## 4.3 Beyond MAPI examples

`mapidump` functions only provides a limited dump and object overview. They had initially been designed for debugging purpose but OpenChange kept using them in tools as they provided a convenient and easy way to display information. Getting beyond this example now suppose you inspect MAPI properties closely and figure out which information you want to fetch.

Developers interested in any of the MAPI capabilities described in 1.6 - but not cover in this paper - are invited to get a closer look to existing OpenChange tools such as openchangeclient or exchange2mbox exposed in next section.

# 5 OpenChange applications Tour

## 5.1 Console applications

The console applications described below are available and shipped within libmapi releases.

### 5.1.1 mapiprofile

`mapiprofile` is a command line tool designed to provide administrative support for OpenChange MAPI profiles. It is designed so it provides sample code for developers interested in adding OpenChange MAPI profile support to their applications. See section 2.2.1 for practical use of this tool.

### 5.1.2 openchangeclient

`openchangeclient` is a MAPI command line tool designed to facilitate mail send, receive and delete operations using the *MAPI protocol* [5]. It also provides operations on tasks, contacts (address book) and calendar operations, MAPI permissions, MAPI notifications or simply mailbox hierarchy display. This tool is improved along new MAPI calls implementation and provides with the MAPI torture suite the most up to date resource developers can check when providing additional support to their applications.

### 5.1.3 exchange2mbox

`exchange2mbox` provides a way to synchronize an Exchange mailbox with a mbox file. The tool is developed so it only retrieve mails not already stored in the message ID index database and reflect changes back to the Exchange server if local message copy are deleted. It relies on libmagic, supports MIME types and is able to store multiple attachment files for a specific message.

## 5.2 Graphical applications

### 5.2.1 Evolution Plugin

Along MAPI library development, we always attempted to stay close to possible developers expectations and provide a public API matching their potential application needs. One of most obvious result of this process is the OpenChange Evolution plugin we started a couple of months ago. While this plugin is still experimental, it provides several features such as Exchange account setup, fetch emails, send emails, delete emails, fetch and send attachments, handle Message flags (attachments and importance) or display light mailbox folder hierarchy.

### 5.2.2 KDE Akonadi Resource

Finally OpenChange is pleased KDE developers got interested into bringing Exchange support to their messaging application using our framework. This collaboration results in the development of a Akonadi resource for KDEPIM maintained and mainly developed by Brad Hards.

[5] incorrect but convenient naming

# A Single value property types

```
#define        PT_UNSPECIFIED  0x0000
#define        PT_NULL         0x0001
#define        PT_I2           0x0002
#define        PT_LONG         0x0003
#define        PT_R4           0x0004
#define        PT_DOUBLE       0x0005
#define        PT_CURRENCY     0x0006
#define        PT_APPTIME      0x0007
#define        PT_ERROR        0x000a
#define        PT_BOOLEAN      0x000b
#define        PT_OBJECT       0x000d
#define        PT_I8           0x0014
#define        PT_STRING8      0x001e
#define        PT_UNICODE      0x001f
#define        PT_SYSTIME      0x0040
#define        PT_CLSID        0x0048
#define        PT_BINARY       0x0102
```

Figure 7: Single value property types

# B Common MAPI data structures

## B.1 SPropValue union

```
union SPropValue_CTR {
        uint16_t i;                     /* [case(0x0002)] */
        uint32_t l;                     /* [case(0x0003)] */
        int64_t dbl;                    /* [case(0x0005)] */
        uint16_t b;                     /* [case(0x000b)] */
        int64_t d;                      /* [case(0x0014)] */
        const char *lpszA;              /* [case(0x001e)] */
        struct SBinary bin;             /* [case(0x0102)] */
        const char *lpszW;              /* [case(0x001f)] */
        struct MAPIUID *lpguid;         /* [case(0x0048)] */
        struct FILETIME ft;             /* [case(0x0040)] */
        enum MAPISTATUS err;            /* [case(0x000a)] */
        struct SShortArray MVi;         /* [case(0x1002)] */
        struct MV_LONG_STRUCT MVl;      /* [case(0x1003)] */
        struct SLPSTRArray MVszA;       /* [case(0x101e)] */
        struct SBinaryArray MVbin;      /* [case(0x1102)] */
        struct SGuidArray MVguid;       /* [case(0x1048)] */
        struct MV_UNICODE_STRUCT MVszW; /* [case(0x101f)] */
        struct SDateTimeArray MVft;     /* [case(0x1040)] */
        uint32_t null;                  /* [case(0x0001)] */
        uint32_t object;                /* [case(0x000d)] */
};
```

### B.2 SPropValue structure

```
struct SPropValue {
        enum MAPITAGS ulPropTag;
        uint32_t dwAlignPad;
        union SPropValue_CTR value; /* [switch_is(ulPropTag&0xFFFF)] */
};
```

### B.3 SRowSet structure

```
struct SRow {
        uint32_t ulAdrEntryPad;
        uint32_t cValues;
        struct SPropValue *lpProps;
};

struct SRowSet {
        uint32_t cRows;
        struct SRow *aRow;
};
```

## C Default folders defines

```
#define        olFolderTopInformationStore     1
#define        olFolderDeletedItems            3
#define        olFolderOutbox                  4
#define        olFolderSentMail                5
#define        olFolderInbox                   6
#define        olFolderCalendar                9
#define        olFolderContacts                10
#define        olFolderJournal                 11
#define        olFolderNotes                   12
#define        olFolderTasks                   13
#define        olFolderDrafts                  16
#define        olPublicFoldersAllPublicFolders 18
#define        olFolderConflicts               19
#define        olFolderSyncIssues              20
#define        olFolderLocalFailures           21
#define        olFolderServerFailures          22
#define        olFolderJunk                    23
```

# D MAPI Fetchmail listing

```
#include <libmapi/libmapi.h>

#define DEFAULT_PROFDB        "%s/.openchange/profiles.ldb"

int main(int argc, char *argv[])
{
        enum MAPISTATUS                 retval;
        TALLOC_CTX                        *mem_ctx;
        struct mapi_session             *session = NULL;
        mapi_object_t                   obj_store;
        mapi_object_t                   obj_folder;
        mapi_object_t                   obj_table;
        mapi_object_t                   obj_message;
        struct mapi_SPropValue_array        props_all;
        struct SRowSet                  rowset;
        struct SPropTagArray            *SPropTagArray;
        mapi_id_t                       id_inbox;
        mapi_id_t                       *fid, *mid;
        const char                      *profname;
        char                                *profdb;
        uint32_t                        i, count;

        mem_ctx = talloc_init("fetchappointment");

        /* Initialize MAPI */
        profdb = talloc_asprintf(mem_ctx, DEFAULT_PROFDB, getenv("HOME"));
        retval = MAPIInitialize(profdb);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Find Default Profile */
        retval = GetDefaultProfile(&profname, 0);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Log on EMSMDB and NSPI */
        retval = MapiLogonEx(&session, profname, NULL);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Open Message Store */
        mapi_object_init(&obj_store);
        retval = OpenMsgStore(&obj_store);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Find Inbox default folder */
        retval = GetDefaultFolder(&obj_store, &id_inbox, olFolderInbox);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Open Inbox folder */
        mapi_object_init(&obj_folder);
        retval = OpenFolder(&obj_store, id_inbox, &obj_folder);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Retrieve Inbox content table */
        mapi_object_init(&obj_table);
        retval = GetContentsTable(&obj_folder, &obj_table);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Create the MAPI table view */
        mem_ctx = talloc_init("MAPI Table");
        SPropTagArray = set_SPropTagArray(mem_ctx, 0x2, PR_FID, PR_MID);
        retval = SetColumns(&obj_table, SPropTagArray);
        MAPIFreeBuffer(SPropTagArray);
        MAPI_RETVAL_IF(retval, retval, mem_ctx);
        talloc_free(mem_ctx);

        /* Get MAPI table rows count */
        retval = GetRowCount(&obj_table, &count);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Iterate through rows */
        while ((retval = QueryRows(&obj_table, count, TBL_ADVANCE, &rowset))
               != -1 && rowset.cRows) {
                for (i = 0; i < rowset.cRows; i++) {
                        fid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_FID);
                        mid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_MID);
                        mapi_object_init(&obj_message);
                        retval = OpenMessage(&obj_store, *fid, *mid, &obj_message);
                        if (retval != MAPI_E_NOT_FOUND) {
                                retval = GetPropsAll(&obj_message, &props_all);
                                mapidump_message(&props_all);
                                mapi_object_release(&obj_message);
                        }
                }

        }

        /* Release MAPI objects */
        mapi_object_release(&obj_table);
        mapi_object_release(&obj_folder);
        mapi_object_release(&obj_store);

        /* Uninitialize MAPI */
        MAPIUninitialize();
        return (0);
}
```

# E MAPI Fetchappointment listing

```
#include <libmapi/libmapi.h>

#define DEFAULT_PROFDB        "%s/.openchange/profiles.ldb"

int main(int argc, char *argv[])
{
        enum MAPISTATUS                 retval;
        TALLOC_CTX                        *mem_ctx;
        struct mapi_session             *session = NULL;
        mapi_object_t                   obj_store;
        mapi_object_t                   obj_folder;
        mapi_object_t                   obj_table;
        mapi_object_t                   obj_message;
        struct mapi_SPropValue_array        props_all;
        struct SRowSet                  rowset;
        struct SPropTagArray            *SPropTagArray;
        mapi_id_t                       id_inbox;
        mapi_id_t                       *fid, *mid;
        const char                      *profname;
        char                                *profdb;
        uint32_t                        i, count;

        mem_ctx = talloc_init("fetchappointment");

        /* Initialize MAPI */
        profdb = talloc_asprintf(mem_ctx, DEFAULT_PROFDB, getenv("HOME"));
        retval = MAPIInitialize(profdb);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Find Default Profile */
        retval = GetDefaultProfile(&profname, 0);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Log on EMSMDB and NSPI */
        retval = MapiLogonEx(&session, profname, NULL);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Open Message Store */
        mapi_object_init(&obj_store);
        retval = OpenMsgStore(&obj_store);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Find Inbox default folder */
        retval = GetDefaultFolder(&obj_store, &id_inbox, olFolderCalendar);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Open Inbox folder */
        mapi_object_init(&obj_folder);
        retval = OpenFolder(&obj_store, id_inbox, &obj_folder);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Retrieve Inbox content table */
        mapi_object_init(&obj_table);
        retval = GetContentsTable(&obj_folder, &obj_table);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Create the MAPI table view */
        mem_ctx = talloc_init("MAPI Table");
        SPropTagArray = set_SPropTagArray(mem_ctx, 0x2, PR_FID, PR_MID);
        retval = SetColumns(&obj_table, SPropTagArray);
        MAPIFreeBuffer(SPropTagArray);
        MAPI_RETVAL_IF(retval, retval, mem_ctx);
        talloc_free(mem_ctx);

        /* Get MAPI table rows count */
        retval = GetRowCount(&obj_table, &count);
        MAPI_RETVAL_IF(retval, retval, NULL);

        /* Iterate through rows */
        while ((retval = QueryRows(&obj_table, count, TBL_ADVANCE, &rowset))
               != -1 && rowset.cRows) {
                for (i = 0; i < rowset.cRows; i++) {
                        fid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_FID);
                        mid = (mapi_id_t *)find_SPropValue_data(&(rowset.aRow[i]), PR_MID);
                        mapi_object_init(&obj_message);
                        retval = OpenMessage(&obj_store, *fid, *mid, &obj_message);
                        if (retval != MAPI_E_NOT_FOUND) {
                                retval = GetPropsAll(&obj_message, &props_all);
                                mapidump_appointment(&props_all);
                                mapi_object_release(&obj_message);
                        }
                }

        }

        /* Release MAPI objects */
        mapi_object_release(&obj_table);
        mapi_object_release(&obj_folder);
        mapi_object_release(&obj_store);

        /* Uninitialize MAPI */
        MAPIUninitialize();
        return (0);
}
```

# References

[1] CDOLive. Property tags and types. http://www.cdolive.com/cdo10.htm. [Online; accessed 6-Aug-2006. Third-party MAPI Documentation].

[2] Microsoft Corporation. About property identifiers. http://msdn2.microsoft.com/en-us/library/ms527066.aspx, 2004. [Online; accessed 6-Aug-2006. Microsoft MAPI Documentation].

[3] Microsoft Corporation. About property tags. http://msdn2.microsoft.com/en-us/library/ms531530.aspx, 2004. [Online; accessed 6-Aug-2006. Microsoft MAPI Documentation].

[4] Microsoft Corporation. About property types. http://msdn2.microsoft.com/en-us/library/ms529429.aspx, 2004. [Online; accessed 6-Aug-2006. Microsoft MAPI Documentation].

[5] Microsoft Corporation. Properties. http://msdn2.microsoft.com/en-us/library/ms528634.aspx, 2004.

[6] Jason Nelson. How outlook, cdo, mapi, and providers work together. http://technet.microsoft.com/en-us/library/aa996249.aspx, 2005. [Online; accessed 6-Aug-2006].

[7] Ronnie Sahlberg. Re: [ethereal-dev] mapi and exchange. http://www.ethereal.com/lists/ethereal-dev/200406/msg00362.htm, 2004. [Online; accessed 6-Aug-2006].

[8] OpenChange Team. Documentation - howto.txt. http://websvn.openchange.org/filedetails.php?repname=openchange&path=%2%Ftrunk%2Fdoc%2Fhowto.txt, 2005-2007.

[9] Andrew Tridgell. talloc library homepage. http://talloc.samba.org, 2003.

[10] Andrew Tridgell. ldb library homepage. http://talloc.samba.org, 2004.

[11] Andrew Tridgell. talloc reference guide. http://samba.org/ftp/unpacked/samba4/source/lib/talloc/talloc_guide.txt%, 2004.