Table of Contents

PM and it philosophy

Nicholas Christopoulos - nereus@freemail.gr
Translated by Fotis Alexandrou – alfotis@yahoo.gr

Version 0.3, Athens, 28th April 2004

Abstract

PM is a system that I designed in order to solve specific problems such as a universal communication way of every device but also various libraries with the desktop environment of arm0nia project. Whilst, it has greater extensions. I think that it is very simple, but unfortunately it is impossible for most people to understand it, and here is the target of this document which is to explain this "weird" system, how is it working, and what are its possibilities. PM, except from a sub-system, can be used as a theoretical prototype for a universal interface, and also for sub-systems like OLE/CORBA and more.

1 The environment of the software

Us, programmers, know very well the complexity of things. We have to face a chaotic environment in which millions of lines of source code, millions of instructions, usually ordered in libraries, that interact and are combined in order to get the final project, either an application or even an Operating Systems with all its components.

There are enormous size of code, written by various programmers among the world, for an also enormous variety of subjects.[1] Visiting sourceforge, codeguru, simtel or anywhere else you will get convinced, although only a small part of code is "transfered" over the Internet.

Through this "space of source code"[2] we often find solutions that we need or "would be useful", or "would be nice", or "would be smart", or "would be convenient for the future" to add. Solutions ready and tested, solutions that are being developed and updated by others, solutions that increase software development as more as it gets via community work.

1.1 The new team development

In "The Cathedral and the Bazaar" [Bazaar], Raymond tried to explain the power of community work, viewing it from a crazy, typical american (>1950) view, which is propagandized or non-existing social and political philosophy. One way or another, this project is important in many ways and has many interesting opinions. The most

1 Let me remind you the program that wrote poems :)
2 My draft definition, related to dataspace, expressing code's total amount, public domain, open free/source but also legally copyrighted, that you can find over the net.

important is that it is complete and even the reader that has not got in touch with open/free source, can see the "revolution" in software development the past few decades[3], as well as the power of the community work that is known to the people as "Linux".

Depending on these, we see that there is a base, efforts but also great stuff built to work for the community, which is us. Most free source community members are ideologists and base these efforts in ethics, so they call these programmers as volunteers.

Let me disagree here, because I see source code in a different way. I see it as an exchange of opinions and/or work[4]. Exchanging opinions is the base of the development of science and philosophy[5]. Exchanging does not require the desire with no good, that volunteering requires, and that's why I expect the population of the community to grow.

Free/open source is more a scientific "collective" than a "helping organization". Helping a project that we use, we help ourselves, because we will also use this project in the future. The development of such a project is our business too, because we base a part of our work on it.

You can find more information for the hardcore ideas of free software, in GNU and OSI web site.

2 The problem of code that is already written

Each project's team of developers, happens to format the communication with other code depending on its needs, and this communication occurs to be different every time from project to project, even if they are about the same subject. This is natural to happen, because the community work remains in project level and not community, thing that is very difficult anyway.

This ends up forcing the programmers to avoid the use of such projects because:

1. It takes time to learn different interfaces
2. Usually, the interface, doesn't fit the structure of the code that exists
3. They are forced to write special code to support each interface. This is time consuming and tiring, but also unsafe, because the more complex the code is, the more possible is to have vulnerabilities.

2.1 Input/Output

Every program is characterized by input and output. We call input the system that enters data for manipulation, where this can be a device such as a mouse or the results of some kind of process, like a file or more. Same way, output are the results of a program, that can be a file, graph representations on screen and so on.

Today, input/output systems, demand an absolutely different way of manipulation from

3 Open source officialy starts with TEX, I think
4 Coding is a way of expression, just like math and anyone that disagrees with that is because they are not really programmers
5 We should expect this great amount of space of source code to multiply

the program. For example, when we design a diagram on screen, we also need to write new, different code in order to print it or get it as a file.

2.2Input of the output

Things get very complex when we want to manipulate the results of an application, and also when we want the results of our work, to be accessed by other applications.

Usually, the communication of the programs is done using files, while special manipulation is required for the communication of every application. For example in order to design a vector, we are forced to use different kinds of files that are used by CAD programs, and if we want to do a good job, we must support 2-3 types like these at least.

2.3 Dead Nature

Files, among others, are "dead nature". They don't calculate, they don't act for our needs, they are static, dead in some way. This constrains the abilities of input and output. For example, you can't write a file of vector graphics, ask it to convert in a different resolution (useful for bitmap editing), you must do it in your program. This is a typical procedure – routine, that I believe many of us copy and paste it from program to program, although it is reasonable to be in one place, because the structure of the elements differs, we are forced to copy and edit this routine. This wouldn't be necessary if we could ask from the input, the type of the elements.

2.4 Conclusions

These constrain in least the test (and use) of interesting projects, and as a result, they constrain the local and total software development in international level. It makes difficult for us to test and use ready solutions, and if these problems solve, I am sure that we are going to have mass increase of software development, commercial or not, in international level.

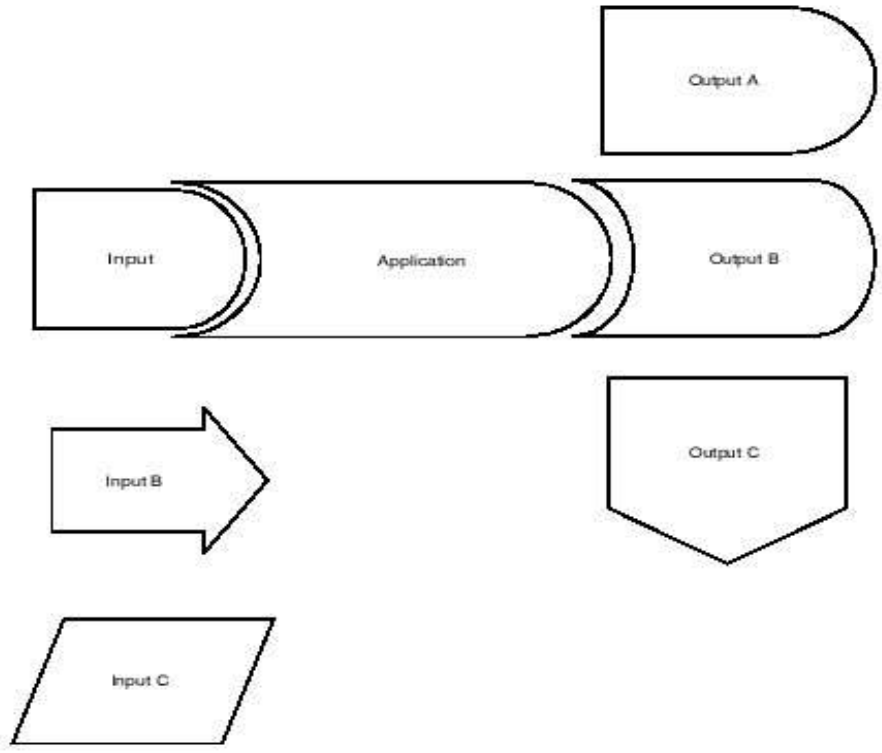
We will see how all these are solved and how they are forwarded in a totally different use and development sense. We are going to focus on so-called VFS, which is the Virtual File System, because it is the most important and the simplest part of PM.

3 A universal interface

I've felt these problems very intensively from the pre-DOS era. While the time was passing by and while the ugliness of Windows API [CrqAPI] was growing, I was forced many times to make C modules that did nothing else but provide a universal interface between libraries about the same subject or parts of the API!

In arm0nia project, I was in place that had to link thousands of libraries known or not, source code parts, known or not, and of course the interconnection of all those components. I insist that this is something that has to be done and PM did it quiet well. I wasn't that interested in arm0nia, because my time was limited, mainly because of my job and SB. The kind of the problem was that made me interested in arm0nia project.
The whole situation looked like the picture above. The programs where made just like the 'application' of the picture, I mean, they were made for a specific type of input or specific

type of linking with libraries. [6]



What was required, was that everything "spoke the same language", according to the sketch.

> *i. The universal interface for everything is the basic principle for PM. It is the principle that the whole system lies around*

4 In the beginning there was SB

Among others, I have made a new BASIC, SmallBasic (SB). SB is very important for PM, because SB is PM"s design predecessor.

SB is an unusual project. It has to provide maximum ease to users, because it is applicable – among others – to children's education, and also must support various Operating Systems like Palm OS, Linux, Windows, etc. This means that SB should provide a universal interface for a large amount of different things but also fast operation, especially in Palms, because their speed and their abilities are low.

In order to manage all these, I had to run back in the early Unix and DOS versions (also CP/M) and wandered why they didn't develop this ability. In both Operating Systems, serial port, is working in similar way with the console, and both of them work with file I/O. YES, they have a universal interface, even draft.

```
1   fp_console = fopen(“CON:”, “wt”); // Open console device
2   fp_serial = fopen(“COM1:”, “wt”); // Open serial port 1
```

6 In the image, there is shown only input and output, but same thing happens with various libraries

```
3
4   fprintf(fp_console, “Hello, world!”); // Print in monitor
5   fprintf(fp_serial, “Hello, world!”); // Send through serial port
```

Accepting that almost every data can be organized in a way that can be manipulated by file I/O commands, but also because I had already obtained experience in these subjects, instead of adding commands to SB for every driver, I changed every driver in order to act as file system.

So, today in SB, with four commands `OPEN, INPUT#, PRINT#` and `CLOSE` that are used in classic file I/O, users can do the following:

1. Operate their own files
2. Send and receive data through serial port
3. Write and read PDOC files (Palm OS database compressed documents)
4. Write and read memos from memo database
5. Write and read through telnet protocol, I mean to communicate with most Internet services (POP3, SMTP, HTTP, etc)

SB of course gave more things to PM, because the source for the first dynamic modules and more, is a copy of SB similar code.

4.1 Hierarchical file system

Hierarchical file system is a classic and very distributed structure that can express complex data structures. I don't think that there is anyone that can't describe data structure exactly like an RDBMS. The relationships of the elements and indexes, can be expressed in much more complex ways but also in file system framework and also operations with code. For example linked programs ("#!" in Unix and "extension" in Windows) and links (Unix).

Beyond that, the procedures that are necessary in order to read and write elements, either we have a file, or a socket, or a database, or a dynamic list, etc. is always the same. Only read and write commands are changed.

Typical pseudo code that explains a typical procedure with read and write.

```
1          Go to the start of elements
2          While elements are not over
3                read elements
4                processing
5                write elements in output
6                go to the next element
```

Either this is a dynamic list, or a database, or a text file, many times a socket etc. this pseudo code expresses something that we are bored to repeat. SB (PM also), doesn't promise that we will not write it, but promises that this is the way that it will be done and with the same commands for any subject. `INPUT#` for reading, `PRINT#` for writing.

- *SB users are happy using these instead of learning the properties and methods of every class*

4.2 How did that happen?

I have the disadvantage to love function-pointers, but that implementation would be very difficult in SB because of the multiple code segments that Palm OS required. So, this was implemented with the simplest way, which we'll see if we want us later to understand the really complex PM.

`OPEN` command uses as an argument the name of the file for opening. In BASIC, and here is another difference from C, this command uses as an argument file-handle too. When user, beyond the regular file-system, wants to use one from the default drivers that come along with SB, needs to declare that in file name. So, using the symbol ":" separates the name of the driver and its parameters. For example the file name "`myfile.txt`" means an ordinary text file, while "`COM1:9600`" tells to SB to use the serial port 1 driver (COM1) instead of the classic file system, in the speed of 9600 BAUDs

SB Source

```
1   handle = 1
2   OPEN "COM1:9600" AS #handle
```

So the example above, is translated from SB in the code shown below

Replacement of OPEN

```
1   fileName = “COM1:9600”;
2   driverName = fileName.leftOf (':');   //driverName <- “COM1”
3   driverParams = fileName.rightOf (':'); //driverParams <- “9600”
4   if (driverName == “COM1”) {
5         fileTable[handle].driverID = SERIAL; //use serial I/O library
6         fileTable[handle].serialHandle = serial_open(driverParams);
7    }
8    else {
9         fileTable[handle].driverID = DISK; //use file I/O library
10     fileTable[handle].fp = fopen(driverName + driverParams);
11  }
```

FileTable keeps the elements of every file-handle, and one of these elements is which library will use (in our example, for serial port or typical file system). This information is inserted in field named `driverID` as we saw.

Function serial_open along with `serial_write_string` which we'll see afterwards, is a part of the driver that SB uses to manage the serial port. What I mean is that this driver is nothing else but a typical library (or a simple C module) that contains at least these two functions.

SB Source

```
1   PRINT #handle; “Hello, world!”
```

Sending data (PRINT), is also done the same way

Replacement of PRINT

```
1   params = “Hello, world!”;
2   if ( fileTable[handle].driverID == SERIAL )
3       serial_write_string(fileTable[handle].serialHandle, params);
4   else
5       fprintf(fileTable[handle].fp, “%s”, params);
```

As we can see this procedure is very simple. A switch or an if implement this sub system.

5 The idea

As we explained, SB provides a universal interface to its users for a variety of abilities. What would have happened if everything around these two-three interfaces customized and what if the application used as a driver, as long as its source would be the same and the only thing that would change would be the argument of open() command and what if each driver loaded another? And what if the user (or the parent-program) chose which is the default driver for every operation?

Answer; crazy stuff, without changing a single line of source code, so crazy that not even our imagination can capture. The thing we dreamed, unity of a great amount of code.

ii. In PM, there is no more the meaning of file, image, database, TCP/IP etc. There only is unique code that acts in a default way.

The application opens a file as it does always, with the same commands, and of course with the same way of manipulation.

iii. Equally with the virtual files, the application operates more, real, or virtual input/output or processing systems that remain unknown in the application, but the application knows how to operate them

Some crazy examples

- A file manager that can send email without having code for it, because it doesn't need it, and doesn't know even what a socket is
- A file system that can add Unix attributes in FAT!
- An image that can be saved as a sound and the opposite! Have you ever draw lines in a sound?
- A Graphical User Interface that works in graphics terminals as well as in ASCII terminals
- An application that can be displayed selectively in a PDF file instead of a monitor, either because it was declared this way in the application, or because the user selected the PDF driver as a monitor driver!

Dimitris (koukos), had said that PM is the ultimate virtual, it is the virtual's virtual. He was right because this system offers a virtual environment, with virtual, or not data, from

virtual, or not devices, that each one can depend on each other.

For example it is possible a whole GUI to be built, whose input and output is controlled dynamically from user (executed in memory), and when we connect an output module such as a Graphics card, a printer, avi or pdf library, we can view the results on screen, printed, on another terminal's screen or a file. Same way, as input to this system, we could use another program, a keyboard, a mouse, or a combination of all these.

iv. *This is not about anything difficult, I repeat that we have to face a very simple system, with simple techniques, that it is designed in a way to unite with a satisfying way, irrelevant parts of software, but also to avoid the constraints of the nature of every code. The results are all that abilities.*

As long as the mouse (pointing device interface) can be replaced from code, screen (graphics device interface) can be replaced from code, and mail from code that can act as a file system (virtual file system interface). As long as all these can be done without changing a single line of code, then you shouldn't feel weird by the results, but criticize us for lack of imagination.

5.1 Modules independent from PM

Arm0nia project, at least for now, vulnerable. Today exists, tomorrow may be not, but the work on PM's principles should and must survive.

The reason is simple. We would and we are making a project. This project offers an operational model for different stuff that can be used from third parties, and this should be done, or at least should be demanded to be done.

So, PM except the whole system, also offers a way so that applications that are independent from PM, can be able to add PM modules in their code, without requiring PM itself or its library.

How this is being done, thing very simple and easy to use, will not be analyzed at this point, but I do think that is absolutely necessary to mention.

6 Techniques to develop PM

We will explain – with the most typical way – the techniques that were used in order to achieve PM development. All these are built-in the sub system and are invisible to the end-user (programmer).

6.1 Linking file and code

As we saw in SB, the selection of routines for open, read, write or send data was done with repeated if statements. This operation should be done automatically in PM, and how this is possible, will be explained now.

Instead of a block if, we linked the virtual file with the library that is able to manage it...

Replacement of OPEN

```
1   fileName = “COM1:9600”;
2   driverName = fileName.leftOf (':');   //driverName <- “COM1”
3   driverParams = fileName.rightOf (':'); //driverParams <- “9600”
4   if (driverName == “COM1”) {
5        fileTable[handle].openFuncPtr = serial_io_open;
6        fileTable[handle].writeFuncPtr = serial_io_write;
7    }
8    else {
9        fileTable[handle].openFuncPtr = file_io_open;
10       fileTable[handle].writeFuncPtr = file_io_write;
11       driverParams = driverName + driverParams;
12  }
13
14  // open 'file'
15  fileTable[handle].systemHandle =
16         fileTable[handle].openFuncPtr(driverParams);
```

The array `fileTable`, in this new table, keeps the addresses of the routines, instead of what library should use, while the functions for serial I/O and file I/O have been customized, in order to have the same syntax.

This has philosophical extensions because it adds a completely different logic and abilities to face the subject. While before we had an array of elements for every file, we now link it with the code for its manipulation (function pointers). This “code for its manipulation” can really refer in using a file but also in anything else (ex. Serial port, sockets, or some kind of processing).

v. With this technique, the rest of the code doesn't need to know what the file is to use it. Just calls the default functions, with default syntax, functions that are declared into fileTable[handle].

In the following replacement of “`PRINT`”, we clearly see that the program doesn't need to know that the hypothetical “`file`” is a serial port or a regular file.

Replacement of PRINT

```
1   params = “Hello, world!”;
2   fileTable[handle].writeFuncPtr(fileTable[handle].systemHandle, params);
```

6.2 Evolution in module

In the previous procedure, open filled `fileTable` with the addresses of the routines for methods open, close and read, write. We also saw that in order this to be done, we needed every library to act with the same way, which is to have syntax for the commands mentioned above.

Until now, all these were made into the same program-application. What we are interested in is, that the application loads selectively these libraries dynamically. This is possible to be done by using the dynamic link library mechanism, provided by current Operating Systems.

In order to achieve this, instead of having a variable of type `fileTable`, we create a structure, in which we keep the addresses of the routines.

Replacement of fileTable

```
1   struct module {
2   void *dlHandle; // Handle of dynamic link library
3   int (*open)(const char *fileName, const char *mode);
4   void (*close)(int handle);
5   int (*write)(int handle, char *buffer, int size);
6   int (*read)(int handle, char *buffer, int size);
7   };
```

So, we define the universal interface for all libraries, that from now on are called modules, that act as files. They are being operated with the classic file I/O commands[7].

Dynamic module loading

```
1   module serial_io_lib;
2   module file_io_lib;
3
4   serial_io_lib.dlHandle = dlopen("my_serial_io_lib.so", RTLD_LAZY);
5   serial_io_lib.open = dlsym(serial_io_lib.dlHandle, "open");
6   ...
7   file_io_lib.dlHandle = dlopen("my_file_io_lib.so", RTLD_LAZY);
8   file_io_lib.open = dlsym(file_io_lib.dlHandle, "open");
```

More for dynamically linked libraries in [LPLib]. In that way, it is possible to load lots of that modules, and keeping the same application code, to switch input/output by simply switching the pointer of the module as we can see in the following example.

Example with two modules

```
1   module *current_module;
2
3   //current_module = &serial_io_lib;
4   current_module = &file_io_lib;
5   int handle = current_module->open("myfile", "w");
```

7 open(), read(), write(), close(), eof(), seek(), fgets(), etc.

```
6   current_module->read(handle, buffer, 1024);
7   current_module->close(handle);
```

As we can see if we deactivate line 4 and activate line 3, then this code, instead of reading a file, would get data from the serial port.

At this point you must understand how easy would it be, by changing a single line of source code, the application to use compressed or not files, by using zlib.

6.3 VFS

VFS means Virtual File System. We named it that way because the application thinks that manages a file system, but this system can be reffered in totally different things. Typical examples are zipped files that contain compressed directories and files, Windows registry and more.

PM's sub system, VFS describes two states:

1. File Level, when we have to do with virtual files like serial port.
2. Directory level, when we have to do with a complete environment, with directories, files and file attributes.

I suppose that the word file still reminds you regular files. This is a very important mistake, it is not about files but code! The word file describes the way of communication with the module and not what the module does. File in VFS, could be a routine that sends back to the application the results of a process, that has been done the exact time that the application asked to “open” the file as well.

Let's see a simple example.

Suppose that you have a temperature monitor in your PC, that gives us the temperature in Kelvin degrees. We manage this device from the library offered by the manufacturer, and the library contains one single function, `curTherm`.

A typical program in order to return the temperature both in Kelvin and Celsius degrees is described below.

Thermometer

```
1   main()
2   {
3      double t = curTherm();
4
5      printf(“%f Kelvin\n”, t);
6      printf(“%f Celsius\n”, t-273);
7   }
```

We want us to be able to view these measurements in a file manager, so that when we

click a file called '/thermometer/Kelvin' and a file called '/thermometer/Celsius' to see the current measurement in each climax; we built the VFS module.

Thermometer VFS module

```
1   char strKelvin[32];
2   char strCelsius[32];
3   int therm_open(const char *fileName)
4   {
5       if ( strcmp(fileName, “Celsius”) == 0 ){
6           sprintf(strCelsius, “%f”, curTherm() - 273);
7           return 1;
8       }
9       else if (strcmp(fileName, “Kelvin”) == 0) {
10          sprintf(strKelvin, “%f”, curTherm());
11          return 2;
12      }
13      return -1; //error
14  int therm_read(int handle, char *buf, int size)
15  {
16      int len = 0;
17      switch ( handle ) {
18      case 1:  //Celsius-file data
19          len = strlen(strCelsius);
20          strncpy(buf, strCelsius, len);
21          break;
22      case 2: //Kelvin-file data
23          len = strlen(strKelvin);
24          strncpy(buf, strKelvin, len);
25      }
26      return len;
27  }
28  void module_init(module *mod);
29  {
30      mod->open = therm_open;
31      mod->read = therm_read;
32  }
```

This module is a virtual file system. Of course it is simplified, but it is not that far from reality. In `module_init` as we can see we define which function is responsible for open

and which is for read.

Let's see what is going to happen and how.

1. The application file manager asks from PM to load the module
2. PM executes `module_init` in a way that it knows what functions are implemented in the module and who is responsible for what
3. The application asks from PM to open the virtual file “Celsius”
4. PM executes `therm_open(“Celsius”)` and returns the virtual handle (returned from `therm_open()`) back in the application
5. The application asks from PM to take the contents of the file, with a typical `fgets()` call (module->fgets(...))
6. PM, with its own `fgets()`, calls `therm_read()` in order to take the data and return them to the application.

6.4 Besides VFS

...
... keys ...

6.5 PMFS

All these sound very nice but the application should know what modules are there in the system and for which interface. So, PM's daemon, is responsible to provide all these information through a virtual file system, called pmfs.

List all VFS modules

```
1   main()
2   {
3      pm_vfs_module_t *pmfs;
4      strlist_t *list;
5      int i;
6      pm_init(); //Initialize PM
7
8      //load PM virtual-file-system module
9      if ( (pmfs = pm_vfs_load(“vfs/pmfs”) ) == NULL)
10        panic (“pm_vfs_load(): failed”);
11
12     printf(“The following VFS modules found: \n\n”);
13     list = pmfs->list(pmfs->mid, “/vfs/*”);
14     for ( i = 0; i < list->count; i++ )
15         printf (“%d: %s\n”, i+1, list->str[i]);
16     strlist_destroy(list);
17     pm_vfs_release(pmfs);
18   }
```

Method list is a more advanced version of `opendir()/closedir()`, and returns a list with the file names, which is similar with the commands `ls` and/or `dir` that we know from the console.

6.6 One ready virtual file system

Suppose that we have a file manager that works with modules similar with the ones described above, but add some everyday tasks like `opendir()/closedir()` and `chdir()/rmdir()/mkdir()`.

> *We have already made a program-application like this, that works perfectly, vfssh which is a shell like bash and tcsh, only that instead of working with the Operating System's file system, works only with PM modules*

Think now, that we are going to build a vfs module for presenting the information of each user, as it is described in file `/etc/passwd`.

> *The module specified, and not exactly the same, can be found in the examples and you can see it work with `vfssh`.*

We have the following from the classic Unix files:
(todo: passwd, structure, groups)

Now we must imagine that we wanted them to exist in files. Usually we would need one directory for the users and one for the groups. Every single contained in the users directory, should contain the user's data and of course a name, which is the user's username.

Virtual file system hierarchy

```
/users/
   |--- root
   |--- nikos
   |--- dimitris
   +--- yannis
/groups/
   |--- root
   +--- users
```

So, this is how a typical file system should be represented

Contents of file /users/nikos

```
Nicholas Christopoulos
User-ID           :501
Group-ID          :100
Home directory    :/home/nikos
Shell             :/bin/tcsh
```

> *You can see these through vfssh by executing the commands "`ls`" and "`cat`" just as if they were real files.*

So, we've added a library in PM a library that builds a virtual file system in memory. The only thing that is necessary for our module to do is to operate the data contained in /

`etc/passwd` and `/etc/groups` and write them into the virtual file system, which is been done with typical file I/O commands, as if we were writing on disc. From now on, `vfslib`, the library mentioned above, is responsible for the module management, as long as the programmer decides to, of course.

7 PM Principles

7.1 Logical design

Logic in PM is more important than ability, because the programmer and every human being feels comfortable with typical logical solution and not with “juggling” abilities-drafts. When something is made to perform a specific operation, so that it remains, doesn't offer any irrelevant abilities.

In my opinion, the right design has to do with that, I mean that it shouldn't be necessary for the programmer to open the book, to find out how will do what, but this to be understood easily.

If ever there are any extra abilities needed, these can be done with an extra function; usually, these extras are only needed in exceptions.

The opposite of the “right design” can be studied in Windows API, or just read Spinelli's fine comments [CrqAPI].

7.2 A universal interface

When we write code for a bitmap image, it is certain to demand the code itself to function also for jpeg images. When we have a graphics library that writes/reads Video RAM it is certain that we demand that it operates memory bitmaps also. When we write and read in the default file system it is certain that we demand to operate memory sticks the same way.

These are some from the most typical examples that PM is responsible to solve. It doesn't stop here, but it is extended in this; while something can be built with the classic hierarchical structure of the file system, then it's certain that it must function the same way, and with the same commands, just like the classic system. When we have an image, it is certain that this image can be manipulated by the graphics library that PM offers, and there is no argue in that, while this is image it should be saved/read in every type with the same way, as long as it appears on screen, it can also be printed and written in a file.

7.2.1 Grouping in PM

But what parts of software match in such an interface? You see, file I/O can enhance almost everything, but neither can't satisfy all the requirements, nor can it make the system more flexible. For example, how would we draw lines using read/write commands? Sure we can, and we can have one more level, that anyway exists in pml, in order to use line instead of write but we could not take advantage of the graphics card graphics acceleration.

1. VFS (virtual file system interface). PM's module should support file I/O commands. Interface has two levels; a) as a file and b) as file system

2. GFX (bitmap and vector graphics interface). PM's module should support reading and writing pixels
3. SND (sound interface). PM's module should support reading and writing wave
4. PTD (pointing devices interface). PM's module should support keys' state and position
5. KBD (keyboard interface). PM's module should support keys-characters and combinations of them

7.3 Universal data transfer way

Universal interface though, doesn't solve every problem. If we suppose that we have a png image and we want it to convert to jpeg, these two must co-operate, which means that we must have universal data type.

For example, every image that are returned or sent in modules is necessary to have a specific type, and this is defined to be as in Video RAM but with default header that shows the resolution and bit depth. So, all graphics modules or libraries know exactly how to work and with what. Same way for the vector graphics (we know very well that every library has its own definition for vector). Same way for sound.

7.4 Modules management

First of all we had to build a system that loads and unloads drivers as fast as it gets and this was achieved by a daemon (“`pmd`”) and a library (“`pml`”) that is linked to the application. On the other hand, drivers were converted in shared libraries (dynamic linked libraries), that's why they are called modules.

For every modules there should follow specific rules, like:

1. All drivers should be loaded and unloaded dynamically, depending on the application's needs
2. All drivers should be also user-defined. I mean that the user should be able to select the the jpeg driver as output of an application instead of the screen
3. All images that are sent or received by a driver, should always be in a specific type that is also the simplest. This allows moving and processing data from common code.
4. All sound data that are sent or received by a driver, should always be in a specific type that is also the simplest.

PM has more things, as it supports operation for each version of each interface (where module refers which version and what interface supports) and also network communication etc. We will not go further in any of these, we want here to present the philosophy of the project and not its complete design.

7.5 Automatic modules' functions completition

Many things that have to be required from the interface can be replaced by common code, ex. Graphics. It isn't necessary for every GFX module to contain this algorithm, unless it supports an accelerator. Another example is `fgets()` in VFS, that is implemented by calling VFS `read()`, although it is necessary to be supported by VFS interface.

In these cases PM is responsible for completing these functions, if they are not declared in the module.

It is reasonable to expect, specially from graphics, in some cases that module contains only `getpixel/putpixel` functions, while in other cases the module needs to replace the greatest part of graphics library.

8 Ready examples

8.1 vfstest

The following example is a typical application that manages VFS modules. It does nothing more but load each module that takes as an argument and print the contents from one of the virtual files.

Module-key and file name are given by user from console.

```
1   //vfstest.c
2   #include <pm_app.h>
3
4   int main (int argc, char *argv[])
5   {
6       int handle;
7       pm_vfs_module *m;
8
9       pm_init();    //Initialize PM
10
11      if (argc != 3)   //arguments
12         panic(“usage: vfstest module-key open-file”);
13
14      //load the module
15      if ( ( m = pm_vfs_load(argv[1])) == NULL )
16         panic(“Can't load the %s module”, argv[1]);
17
18      //open the “file”
19      if ( (handle = m->open(m->mid, argv[2], 0)) == -1)
20         panic(“Can't open the %s file”, argv[2]);
21
22     //display their contents
23     while ( m->gets(m->mid, handle, buf, 256) )
24         printf(“%s”, buf);
25
26     m->close(m->mid, handle);   //close file
```

```
27      pm_vfs_release(m)           //unload the module
28      return 0;
29  }
```

We can now easily see some of PM's abilities. You'd rather be logged in as administrator (`root`) because some modules-examples have been built to work with full system privileges.

Example 1

Use vfstest to check time from a local or an Internet server

```
# vfstest vfs/telnet localhost:13
```

If connection fails, that means that timegen service is not activated in your system. Open `/etc/inetd.conf` with your favourite editor and remove "`#`" from the line that contains timegen. Reload inetd and try again.

Example 2

Use `vfstest` to connect through serial port with another device.

```
# vfstest vfs/serial /dev/ttyS1:57600
```

I tried it in my Palm, running sertest.bas, which is an SB example and... i sent characters in PC. If you try it, remember that vfstest reads data only, so the program that will be connected (ex. Minicom, Telix, Procomm, Hyperterminal) should send data (keys that you press).

Example 3

Use vfstest to get a user's information

```
# vfstest vfs/users users/root
```

This should return information such as the user's home directory and root's selected shell.

8.2 More examples – applications

You can download more examples through CVS. These examples can be found in directory apps.

8.3 More examples – modules

...

9 Theoretical examples

We temporarily name these examples theoretical simply because we haven't build them yet. We have no doubt that we can build them, but we haven't yet.

9.1 An email client that works as a file system

…

9.2 A fully powered tool with very few code

…

References

[Bazaar] “The Cathedral and the Bazaar”, Eric S. Raymond, 1998/08/11 20:27:29

[CrqAPI] “A critique of the Windows Application Programming Interface”, Diomidis Spinellis, University of the Aegean, Dec. 1997

[LPLib] “Programming Library HOWTO”, David A. Wheeler, LDP, ver. 1.07, 30th December 2002

Thanks

Dimitris Koukoravas and Yannis Vlahogiannis

Dimitri's reinforcement on PM – I intended and still intend not to work on Arm0nia project – but his annoying insisting, is the reason that this document exists.