Using XObject for Configuration Files

Joseph Shrago

Preface

This report covers certain aspects of working with configuration files (ini-files, profiles, registry, etc). Against the background of common ways of organizing such files, we'll review the language for informational (non-executable) objects description (as well as its OS/2 API), that has been developed for specific task (drilling training simulator for OS/2), but is pretty universal.

The most primitive description method "parameter"="value" is commonly used (also in *ix systems) and well known (author wouldn't even try to criticize the syntax of sendmail.cf — you probably understand why :-).

CONFIG.SYS file is a good example:

```
SET EPMPATH=C:\OS2\APPS;C:\opendoc\BIN;
PROTECTONLY=NO
SHELL=C:\OS2\MDOS\COMMAND.COM C:\OS2\MDOS
FCBS=16,8
RMSIZE=640
DEVICE=C:\OS2\MDOS\VEMM.SYS
DOS=LOW,NOUMB
DEVICE=C:\OS2\MDOS\VXMS.SYS /UMB
DEVICE=C:\OS2\MDOS\VDPMI.SYS
DEVICE=C:\OS2\MDOS\VDPX.SYS
DEVICE=C:\OS2\MDOS\VWIN.SYS
DEVICE=C:\OS2\MDOS\VW32S.SYS
```

This method has a drawback: there's no way to syntactically separate different objects' descriptions within one file. Still, it's easy to implement and in many cases suitable. Even if not always possible.

Windows ini-files (Windows 3.11) introduced a slight improvement: sections. Named sections allow structuring the configuration files and simplify querying.

Here's an example from SYSTEM.INI file for WIN-OS2 session:

```
[keyboard]
subtype=
type=4
keyboard.dll=kbdru.dll
oemansi.bin=xlat866.bin
typeofswitch=2
secondkeyb.dll=kbdusx.dll

[boot.description]
keyboard.typ=Enhanced 101 or 102 key US and Non US keyboards
mouse.drv=Microsoft, or IBM PS/2
network.drv=No Network Installed
language.dll=Russian
system.drv=MS-DOS System
codepage=866
woafont.fon=Russian (866)
```

The disadvantages of this method are: syntactical indifference when dealing with variable number of entries within one section and no nesting for section (which may be useful, since not all informational objects are linear). Of course, this may be implemented, but particular implementation will depend on developer's quick wit, rather than the result of syntactical structure.

The next steps in the evolution of description scripts are registry (Windows 9x or higher), and OS/2 ini-files. It is rather difficult to give any examples here, since they are not suitable for printing. Both allow tree structure (which is the development of nesting concept) and storing objects of any type by using binary objects (defined by those who use them).

To criticize existing implementations of such resource (realized WINPRF.DLL in OS/2), we have to specify requirements to the syntax of configuration files in general.

As mentioned before, the views expressed here are the result of working on a scenario maintenance subsystem for a drilling training simulator. A scenario for such training simulators contains all parameters that define the initial state of the drill and borehole, being, thus, not too different from a configuration file. It describes numerous independent parameters — hardware descriptors, rock and geological section descriptors as well as possible anomalies, scattered all over the borehole (to entertain the trainee) etc.

The scenarios are to be created by the end users of the product. Therefore the descriptions and tools to create them have to be intuitive enough for non-programmers to use. Since the same hardware descriptions may be used in many scenarios, they has to be defined separately and linked from within the scenario. A well thought-out scenario becomes a learning material, so it must be printable in a user-friendly format. These requirements resulted in the creation of informationally associated software tools:

- WPS-objects of each scenario object (hardware elements, geology etc);
- WPS-objects for each type of technological tasks scenarios (hardware is different, which means configuration dialogs, too);
- software module for extracting specific data from scenario and passing it as filled-in structure to the task-imitator.

Changes in the description of any object required correcting the software implementation of interconnected objects, which is very inconvenient. This problem became obvious when developers realized that simplifying a number of descriptions had been a mistake and they now need to be elaborated.

Problems described above serve as a good criticism when it comes to using Profile API for maintaining complex configurations.

So we've defined the following requirements to configuring concept:

- object oriented approach to save user time and effort;
- to make it more useable and for economy (of a possibility to make things obscure and cover one's tracks) text as well as binary form, regardless of how configurations are used;
- there must be a dialog for editing the configuration, or a possibility to edit it in the text editor;
- configuration must print itself;
- a possibility for configuration to operate the data stored in other files.

These requirements are fulfilled in our configuration files subsystem. It offers syntax, base objects, WPS-object for visualization and printing and an API for their software-based writing and reading.

Informal Description:

Any configuration file is rendered as a number of fields of certain types. Types are either simple or compound. Thus, configuration file itself is an object of a certain compound type (class).

A syntax we offer for describing such classes operates with keywords, literals and comments.

Literals are identical to ones in C++ : `\xx`, `\\`, `\n`, `\t` and special `%` literals.

Comments are defined by symbols `//`, `/*`, `*/` and used same way as in C++.

The system deals with the following objects:

classifier	transient	taken from class as a default value; can be changed in any representative
base data types	int double string group	integer (4 bytes) double (8 bytes) string of symbols (doesn't have to be limited with \0) arbitrary objects array: allows accessing by element's number
base data types for visualization and printing	IntField DoubleField StringField TextField RadioToggleField CheckBoxField ComboTogleField ReferenceField PasswordField	editable and printable int class editable and printable double class editable and printable string class editable and printable textarea class in-dialog selection class using RadioButton in-dialog selection class using CheckBox in-dialog from-list-selection class using ComboBox link-class, in the dialog edited by drag&drop class for inputing and storing a password, encoded with GOST 21847-89
list type	ObjectListField	class for storing reference elements of arbitraty type
	namedgroup	based upon group; used to describe dialog and printing

Objects' Description:

```
IntField = {
   string title = ""     // name, displayed in the edit dialog
```

```
                        // or when printing
   int value = 0        // field value
   int min = 0          // minimum available integer value,
                        // configurable in the dialog
   int max = 0          // maximum available integer value,
                        // configurable in the dialog
   int delta = 0        // auto-change pitch in the edit dialog
}

DoubleField = {
   string title = ""    // name, displayed in the edit dialog
                        // or when printing
   double value = 0     // field value
   double min = 0       // minimum available double value,
                        // configurable in the dialog
   double max = 0       // maximum available double value,
                        // configurable in the dialog
   double coef = 0      // conversion factor (useful when editing certain
                        // units of measuring, and using other)
   double delta = 0     // auto-change pitch in the edit dialog
   int decimals = 0     // number of symbols after decimal point,
                        // displayed when editing and printing
}

StringField = {
   int max = 0          // maximum number of symbols in line
   string title = ""    // name, displayed in the edit dialog
                        // or when printing
   string value = ""    // if the line is limited
                        // with single quotes '...', don't put
                        // \0 line delimiter
}

TextField = {           // same as StringField, but is textarea (MLE)
   int max = 0
   string title = ""
   string value = ""
}

RadioToggleField = {    // implementation of dialog element
   string title = ""
   int value = 0        // contains the number of selected line
   group elements = { } // array of "string" type strings
}

CheckBoxField = {       // implementation of dialog element
   string title = ""
   int value = 0        // contains 1 if the element is selected
}

 ComboToggleField = {   // same as above for ComboBox object
   string title = ""
   int value = 0
   group elements = { }
}
```

ReferenceField class allows using multiple settings in one if, for example, software product consists of multiple tasks, and each has own configuration with individual and shared part. And vice versa, if the settings of one object are used by many others.

```
ReferenceField = { // object-reference to other object. drag&drop in the
                   //     dialog
string title = ""
string class = ""  // name of the class,
```

```
                    // objects of which may fill this field
                    // (multiple are possible - class0|ckass1|...|classN)
string value = ""   // name of the class that contains object
int value2 = 0      // WPS-objectid
}
```

ObjectListField class allows creating lists of reference objects if, for example, list of persons in the group or list of different network connections has specific settings.

```
ObjectListField list = { // example of object list
                         // ReferenceField of a certain class
   transient string title = ""
   transient group factory = {
      ReferenceField rs = {
         transient string title = ""
         transient group menu = {
            "Открыть"
            "Очистить"
         }
        transient string class = ""
        string value = ''
        int value2 = 0
     }
   }
   transient group buttons = {
      "Добавить"
      "Удалить"
   }
   group elements = {
   }
}
```

When displaying dialog ObjectListField in the list there are values of element "string title" objects' descriptions, or value of element "name|value", is there's StringField name element in the description of reference object.

Access to a password, set in PasswordField, is provided by exported function:

```
rc=getPasswordFieldValue(*XObject,password,sizeof(password));
```

To describe rock images, we've implemented filling mask class:

```
ColorMaskField = {
   string title = ""
   int fcolor = 0        // foreground color
   int bcolor = 0        // background color
   int width = 16        // width, points
   int height = 16       // height, points
   string value = ''     // filling mask 8*8
}
```

Some data is predefined:

- title — by default, contains string-name of the element, which is used in the dialog and when printing;
- status — for writing SUBTITLE string of the dialog page;
- value — by default, contains element value;
- elements — used to describe dialog and when printing;
- pages — describes dialog page;

- sections — describes script printing section.

Using these types it is easy to create own description classes.

For example, that's how bottom-hole engine description class looks like:

```
DEngine drill={
   string title="Забойный двигатель"

   DoubleField diam_zd    ={ title="Диаметр заб.двигателя" coef=0.001
decimals=1 delta=0.1 min=30 max=400}
   DoubleField leng_zd    ={ title="Длина заб.двигателя" coef=1 decimals=2
delta=0.01 max=50}
   DoubleField q_zd       ={ title="Расход заб.двигателя" coef=1000
decimals=1 delta=0.1 min=30 max=400}
   DoubleField dens_zd    ={ title="Плотность жидкости" coef=1000
decimals=2 delta=0.01  min=0.8 max=2.5}
   DoubleField loss_px_zd ={ title="Потери давления на холостом ходу"
coef=98100 min=1 max=200}
   DoubleField freq_nx_zd ={ title="Частота вращения на холостом ходу"
coef=0.01666 min=10 max=999}
   DoubleField loss_pt_zd ={ title="Потери давления при торможении"
coef=98100 min=1 max=200}
   DoubleField moment_t_zd={ title="Тормозной момент" coef=10 min=1
max=1500}
   ComboToggleField drop   ={
       string title ="Тип двигателя"
       group elements = {
           string = "турбобур с постоянной линией"
           string = "турбобур с падающей линией"
           string = "объемный двигатель"
           }
       }
}
```

And that's the description for specific bottom-hole engine (/* comment */ in the first line is needed to interpret text as XObject):

```
/* XObject */
DEngine = {
diam_zd = {value = 30.0000}
leng_zd = {value = 0.0000}
q_zd = {value = 30.0000}
dens_zd = {value = 0.8000}
loss_px_zd = {value = 1.0000}
freq_nx_zd = {value = 10.0000}
loss_pt_zd = {value = 1.0000}
moment_t_zd = {value = 1.0000}
drop = {value = 0}
}
```

This syntax demonstrates the possibility to make changes to each object and additional fields: min, max etc.

As Lewis Carroll's Alice exclaimed: "and what is the use of a book without pictures or conversation?".

Let's take a look at the visualization and printing we've mentioned before.

We've created WPS-object that renders specifically organized parts of class descriptions as its property pages. To do so, we add special descriptors to class description: "group" with names "pages" (for dialog) and "sections" for printing.

Here's a part of the class that describes the dialog:

```
group pages = {
namedgroup = {           // dialog page description
title = "%MajorTab"      // symbol % means MajorTab
elements = {
string = "name 1"        // element of the class
string = "name 2"
...
string = "name N"
}
}
namedgroup = {           // second page of the dialog
title = "MinorTab pages"

         elements = {
            string = "name 1"
            string = "name 2"
                   ...
            string = "name N"
            }
         }
  ...                    // other pages of the dialog
   }
```

This is the part of description of bottom-hole engine (DEngine) class that refers to dialog:

```
DEngine drill={
  ...
    group pages = {
       namedgroup = {
          string title = "%Забойный двигатель"
          group elements = {
             string string = "drop"
             string string = "diam_zd"
             string string = "leng_zd"
             string string = "q_zd"
             string string = "dens_zd"
             string string = "loss_px_zd"
             string string = "freq_nx_zd"
             string string = "loss_pt_zd"
             string string = "moment_t_zd"
             }
          }
     }
 }
```

Note the second tab — "Type". It is always created and allows choosing way (text/binary) of saving. The last one allows easily edit any field of object using text editor. And vice versa, any object may be created in the text editor and then saved as binary. When saving in binary, object saves only differences from base description. The other elements of description are inherited, that's why the size of binary XObject file is much less than the text one.

To describe object printing we use similarly organized group sections, and that's what we get when printing:

```
DEngine drill={
  ...
   group sections={
    namedgroup={
       title = "Забойный двигатель"
       elements={
          string ="drop"
          string ="diam_zd"
```

```
            string ="leng_zd"
            string ="q_zd"
            string ="dens_zd"
            string ="loss_px_zd"
            string ="freq_nx_zd"
            string ="loss_pt_zd"
            string ="moment_t_zd"
            }
        }
    }
}
```

To print the object use the menu, or simply drag its icon onto printer's icon (folder). Here's the example output after printing one object. The same object, as ReferenceField inside the scenario, looks as follows:

Usage:

To use this configuring system, in the root or in the AMT_PATH environment variable directory one has to create file config.ini of the following content (we use our training simulator configuration as an example here):

```
/* AMT */
group Config = {
        string customer = "3AO AMT"
        int loglevel = 0       // -1 full debug output to \xobject.log
                               // 0 output error messages only
                               // 1 output warnings and error messages
        group classes = {
                string = "classes.ini"     // base classes descriptions
                                           // (required)
                string = "common.ini"      // description of the other group
                                           // of common classes
                string = "DSTclass.ini"    // drilling training simulator
                                           // classes description
                string = "KRSclass.ini"    // borehole extensive repairs
                                           // training simulator
                                           // classes description
        }
}
```

This file is used by WPS-object AMTWPSXObject that implements dialogs and configuration objects printing. This object is associated with AMT extension.

Copy all XObject support dynamic libraries into LIBPATH available in environment variable. And using this Rexx script, perform the registration:

```
/*****************************************************************/
/* XObject registration utility                                  */
/*****************************************************************/
Call RxFuncAdd 'SysLoadFuncs', 'REXXUTIL', 'SysLoadFuncs'
Call SysLoadFuncs
Say 'AMT XObjects to be registered'
rc=SysRegisterObjectClass('AMTWpsObject', 'amtwpso')
if rc then say 'AMTWpsObject registered'
else say 'AMTWpsObject cannot be registered'
```

Programming using API XObject:

When programming, one should read descriptions of used classes - for example, all config.ini and all files mentioned there, or only required for specific task. The following text gives and example of reading all classes libraries, described in config.ini:

```
#include "common.h"

XObject  *o_config;     // for the script
// function initializes XObject predefined classes subsystem
// and returns number of classes in the library
int InitXClasses(void)
{
char buf[256]="", *str;
char *amtPath = getenv(AMT_PATH);
int num=0;

   XOBJECT_INITIALIZE

// loading classes libraries
   buf[0] = 0;
   if (amtPath) strcat(buf, amtPath);
   strcat(buf, "\\");
   strcat(buf, AMT_CONFIG);
   o_config = XObject::loadObject(buf);
   if (!o_config)
      return 1;

   o_config = o_config->getGroupElement("classes");
   for (int i = 0; igetGroupSize(); i++ )
       {
       str = o_config->getGroupElement(i)->getStringValue();
       if (str)
          {
          buf[0] = 0;
          if (amtPath)
             strcat(buf, amtPath);
          strcat(buf, "\\");
          strcat(buf, str);
          num += XObject::loadClasses(buf);
          }
       }
   return num;
}
```

The following lines represent the idea of how to obtain certain class elements of training simulator scenario:

```
{
...
 o_main=XObject::loadObject(file_name);  // loading required scenario file
   if (!o_main)
      return 1;   // error reading scenario - running in the test mode
...
 o_sub = o_main->getGroupElement("title");
 Name=strdup(o_sub->getStringValue());

 o_sub = o_main->getGroupElement("model_type");
 model_type=o_sub->getIntValue();  // training simulator model type (number)

 o_sub = o_main->getGroupElement2("mode|value");     // using compound name here
 mode=o_sub->getIntValue();  // training simulator model type (number)

// if the element is link, using method GetValue(name),
// it will load link script
```

```
 o_ref=o_main->getValue("derrik"); // load drill description
// or, in such a way
//  o_ref=XObject::loadObject(o_main->getGroupElement2("derrik|value")-
>getStringValue();

// calculate g_kv parameter value using CI units of measurement
 g_kv=o_ref->getGroupElement2("g_kv|value")->getDoubleValue() * o_ref-
>getGroupElement2("g_kv|coef")->getDoubleValue();
}
```

This was a configuration reading example.

The configuration elements (parameters) can be accessed by class fields names (name), compound names ("name|subname"), and also by their sequence number in the group. In the above description of the bottom-hole engine, the value can be taken from the name <%1|value>, since diam_zd has sequence number 1 in the class.

XObject methods also allow saving information, obtained by software. This simplifies visualization, printing and maintenance.

The code below is the description of XObject class public objects:

```
class _Export XObject {
public:
        static void setLogLevel(int);
        static int log(char*, ...);
        static int log(int, char*, ...);
        static int logObject(XObject*, char* = NULL);
        static void logClasses();

        static void     initClass();
        static void     unInitClass();
        static int      loadFile(char*, char**, int*);
        static int      saveFile(char*, char*, int);
        static XObject* getInstance(char*, char*, int = FLAGS_NORMAL, int =-1);
        static XObject* getInstanceText(char*, int*, int =-1);
        static XObject* getInstanceBin(char*, int*, int = -1);
        static int      loadClasses(char*);
        static XObject* loadObject(char*);
        static int      saveObject(char*, XObject*);
...
public:
        static int      getClassNum();
        static char*    getClassName(int);
...
public:
        virtual ~XObject();
        int     isVoid();
        int     isInteger();
        int     isDouble();
        int     isString();
        int     isComplex();
        int     isGroup();
        int     isObject();
        int     isObject(char*);
        int     isBasic();

        virtual XObject*        clone();
        int     initFrom(XObject*);

        char*   getName();
        int     setName(char* n);
        char*   getClassName();
        int     getClassVersion();
```

```
        int     getFlags();
        int     setFlags(int);
        void    flagSet(int);
        void    flagClear(int);
        int     flagIsSet(int);

        int     getSizeBin();
        int     getSizeText();

        int     getIntValue();
        int     setIntValue(int);

        double  getDoubleValue();
        int     setDoubleValue(double);

        char*   getStringValue();
        int     getStringSize();
        int     getStringLength();
        int     setStringSize(int);
        int     setStringValue(int, char*);
        int     setStringValue(char*);

        int     getGroupSize();
        XObject*        getGroupElement(int);
        XObject*        getGroupElement(char*);
        XObject*        getGroupElement2(char*);
        int     addGroupElement(XObject*);
        int     removeGroupElement(int);
        int     removeGroup();

        int     loadBinFile(char*);
        int     readText(char*, int);
        int     readBin(char*, int);
        int     writeBin(char*, int);
        int     writeText(char*, int, int = 0);

        virtual XObject* getValue();
        virtual int show(int);
        virtual int print(int);
        virtual int html(int);
// misc
        static char*    strPrintInteger(char*, int, int);
        static char*    strPrintDouble(char*, double, int, int);
        static char*    strPrintString(char*, char*, int, int*);
        static char*    strPrintStringEsc(char*, char*, int, int*);
...
};
```

If you like the idea described, download SDK here.
If you would like to debate or just have a conversation with the author, email to:
LFer at rambler dot ru