Supporting languages in applications

This article is aimed at anyone who wishes to program applications that can support more than one language.
There are a large number of older applications that could benefit from multi-lingual support. Developers often create simple applets that grow out of an immediate need, later, when the applet actually become useful, a lot of work is required to support other languages.
Just such a scenario led me to develop MSGC, a simple method of tokenising strings that can then be used to provide multilingual support. This article hopes to explain some of the many ways in which multiple language strings can be supported by various applications, and where MSGC fits into this set of choices. The reasons you might use a particular method are not obvious, so let's examine the options.

Simple Applications

The simplest applet everyone has a go at when they first learn to program is the "Hello World" example. Here it is again:-

```
#include    <stdio.h>

int     main(void);

int main(void)
    {
        printf("Hello world\n");
        exit(0);
    }
```

The printf() function

The printf function is a C language standard IO library function that is really great at squirting out some text without any fuss. In fact, there are several variants of printf.
The variant fprintf() prints to a file stream, very handy for creating error logs. By the way, if the error log is designed to help you provide support of your application, perhaps the error messages should be in your own language. I mention this because it might save you the effort of translating error messages.
Probably the most useful variant is the sprintf() function. Sprintf writes the string to a memory buffer. This buffer can then be used to create an error message box, for example.

Displaying simple messages

All versions of printf() support variables, so you can build some fairly intelligent looking messages...

```
int main(void)
    {
        char    *message = "Hello world";
        char    buffer[100];
        sprintf(buffer, "The string %s is %d chars long.\n", message, strlen
(message));
        printf("%s", buffer);
        exit(0);
    }
```

This creates the message:-

```
The string Hello world is 11 chars long.
```

Printf() supports a number of extremely useful variables:-

- %s for strings
- %c for characters

- %d or %i for signed integers, %u for unsigned integers
- %o for octal integers
- %x for hex integers
- %f for floating point values
- %e for floating point values with exponents

All the above also accept a number of additional parameters to control how the variable is printed. I don't want to repeat the entire panoply of options, but I do want to mention one, and that is the length limiting options with %s.
%.10s limits the string to 10 characters,
and %.*s limits the string to a variable number of characters.
use it like this:-

```
printf("%.*s", length, string);
```

When messages become more complex

Below is a code snippet from rcp.exe, part of an applet I created to show OS/2 codepage information...

```
int main(void)
    {
        PCHAR           pathvar = getenv("ULSPATH");
        char            tmpbuff[CCHMAXPATH];

        printf("#    (c) 2002 Peter Koller, Maison Anglais. All Rights
Reserved\n");
        printf("#\n");
        printf("#    rcp.exe automatically produces codepage information\n");
        printf("#\n");
        if(!pathvar)
            {
                printf("# No ULSPATH found in config.sys\n");
            }
        else printf("# Searching \"%s\\codepage\"...\n", pathvar);
```

...as you can see, this applet is resolutely english. Lets examine the options for turning this into a multilingual applet.

Overview of alternative message systems

The way that an application becomes multilingual is to use a method that splits the text from the code. The text should reside in a separate file to determine the language of the application.
You could of course just recompile several versions of the code, but this doesn't help to maintain either text or code. Text buried in code is hard to find, and code buried in text is hard to debug.
The following is a simple overview of what is available on OS/2.

OS/2 error message files

One option open to us is OS/2 message files. Error message files are used by the system to create error messages. Error message files have two big advantages, first, they accept string substitution variables, and second, those substitutions can be out of order. Out of order substitutions are important when the grammar of a language expects the substitutions to be in a different order:-

```
"In %d minuten kommt die Datei %s an"
"The file %s will arrive in %d minutes"
```

Unfortunately, OS/2 message files also have some disadvantages. Firstly, the substitutions must always be strings, and secondly, not every operating system supports the same messaging format.
OS/2 message files use very specific tokes to identify the component where the error occured. The message files are compiled using MKMSGF, and read using DosGetMessage. See the OS/2 toolkit for further information

Gencat message files

The OS/2 toolkit also provides a catalog file system called GENCAT. This method uses numeric identifiers for messages, and does not directly support string substitutions. You can use a header file with #define directives to to give the token numbers a more meaningful name.

GENCAT has the advantage that is is supported on other operating systems. Messages need to be supplied in ascending numerical order. This is a workable option, but it seems like a lot of trouble to me.

Introducing MSGC

MSGC ovecomes many of these difficulties by using sprintf. The function sprintmsg() works just like sprintf() but uses a token in place of the string. The token is used to retrieve the string from a message file. The addition of a number after the % character allows out of order substitutions.

First you create the messages in a text file that associates an 8 character token to the message.

One file for German...

```
TOKEN001 = "In %1d minuten kommt die Datei %2s an";
```

...and one for english.

```
TOKEN001 = "The file %2s will arrive in %1d minutes";
```

A supplied compiler is used to create the binary message files.

The applet code can use sprintmsg() in place of sprintf():-

```
sprintf(buffer, "In %1d minuten kommt die Datei %2s an", minutes, filename);
//becomes
sprintmsg(buffer, msgfile, "TOKEN001", minutes, filename);
```

To change the language, you change the message file.

MSGC really covers most options.

- ANSII C open source means that the code can be made to work on all operating systems.
- Supports 8 character tokens, just long enough to be meaningful.
- Supports all variable substitution that are supported by the printf() functions.
- Supports out of order variable substitution.
- Fast token retrieval format binary files.
- **but**
- Limited to 9 variable substitutions.
- Variable strings are limited to 1024 bytes, but there are ways to overcome this limit.

Where MSGC wins hands down is that you can usually convert old code into multilingual code with little more than some cutting and pasting.

Code snippet revisited

The code snippet shown earlier can easily be handled by MSGC.

First, our message file source (lets call it mymsg.txt) could look like this:-

```
INTRO_01 = "#    (c) 2002 Peter Koller, Maison Anglais. All Rights Reserved\n";
INTRO_02 = "#    rcp.exe automatically produces codepage information\n";
SEARCHIN = "# Searching \"%1s\\codepage\"...\n"; //notice the %1s
FAILULSP = "# No ULSPATH found in config.sys\n";
```

Now you can translate this file into other languages...

Run "msgc.exe mymsg.txt", that creates the file "mymsg.msg".

Then modify the code to load the message file and print the messages

```
#include  "msgx.h"
int main(void)
    {
        FILE*           hmsgfile; //new
        PCHAR           pathvar = getenv("ULSPATH");
```

```
    char            tmpbuff[CCHMAXPATH];

    hmsgfile = fopenMessageFile("mymsg.msg"); //new
    if(!hmsgfile) exit(1);

    sprintmsg(tmpbuff, hmsgfile, "INTRO_01");
    printf(tmpbuff);
    printf("#\n");
    sprintmsg(tmpbuff, hmsgfile, "INTRO_02");
    printf(tmpbuff);
    printf("#\n");
    if(!pathvar)
        {
            sprintmsg(tmpbuff, hmsgfile, "FAILULSP");
        }
    else sprintmsg(tmpbuff, hmsgfile, "SEARCHIN", pathvar);
    printf(tmpbuff);
    ...
    fclose(hmsgfile);
    exit(0);
}
```

Windowed Applications

If your application puts up some windows then you have the choice of using **resource files**. With a little coding effort, resource files can be put into empty dll's and loaded by the application on startup.
Resource files are not only used for strings of course, they also contain dialog and menu templates, icons, bitmaps, and various other bits and pieces. To take maximum advantage of resource files, you should split language dependant items from common items. You can then compile the common items into the executable, and put the language dependant items into a dll.

Make some space

Here I must make a plea to all developers:-

- Please leave enough room for items in your dialogs at all resolutions.
- Please test dialogs at 640x480 vga as well as the minimum resolution for which the application is designed.
- Please allow the user to select the default font as well as 8.helv. At 1600x1200 this font becomes almost unreadable.
- Please test to see if the dialogs work as well at 96dpi as 120dpi. On Windows you can try a large range of different dpi resolutions. OS/2 will probably do the same in the future, don't allow your app to be dumped because no one can read it anymore!
- Please allow enough space in a dropdown combobox to see more that just a couple of items. I think 4 should be the minimum, otherwise the scroll bar becomes too squashed.

You will also have to redesign the size of some dialogs when using different languages. Text in one language can become much longer in another. ("End" in English becomes "Beenden" in German, you will need a bigger button for that!)

Strings in resource files

If you have a resource file, then this becomes the obvious place to put your string resources. This is particularly true where you don't need out of order variable substitution or other clever features.
In fact, MSGC is slightly more efficient at variable substitution because it all happens in just one function. To see what I mean, lets compare two examples of a message box.

Message box example

Here is a code snippet from Maul Publisher. This code uses messages loaded from the resource file, and creates a simple messagebox window. The two string buffers "tmpstr" and "statmsg" are defined elsewhere.

The messages:-

```
        IDS_WORDCOUNTMSG        "This article contains %d words."
        IDS_WORDCOUNTTITLE      "Word Count"
```

The code:-

```
        WinLoadString(pmh->hab, myModule, IDS_WORDCOUNTMSG, CCHMAXPATH,
statmsg);
        sprintf(tmpstr, statmsg, WordCount((acc.ptc), filesz));
        WinLoadString(pmh->hab, myModule, IDS_WORDCOUNTTITLE, CCHMAXPATH,
statmsg);
        WinMessageBox(HWND_DESKTOP, pmh->hWndMain, tmpstr, statmsg, 0, MB_OK |
MB_INFORMATION | MB_APPLMODAL);
```

The alternative would be to create and load a message file:-

```
WORDCNT1 = "This article contains %d words.";
WORDCNT2 = "Word Count";
```

Then the code could look like this:-

```
        sprintmsg(tmpstr, hmsgfile, "WORDCNT1", WordCount((acc.ptc), filesz));
        sprintmsg(statmsg, hmsgfile, "WORDCNT2");
        WinMessageBox(HWND_DESKTOP, pmh->hWndMain, tmpstr, statmsg, 0, MB_OK |
MB_INFORMATION | MB_APPLMODAL);
```

System message files, resource files, or MSGC?

The answer is that it depends on what you are doing (as usual). I think it is best to reserve system error message files for system errors. All the Dosxxx() functions return an apiret value that can be used with the system error file "OSO001.MSG".

If you are using dialogs and menus, you are best served by having resources loaded in a language specific resource dll. This is important when the size of the dialog boxes alters according to the language.

Where you have to build and output strings based on a process, for example a table of descriptions and numbers output into an editor window, MSGC will provide the best mix of performance and ease of use. You should also consider MSGC when creating console applications.

For complex applications, you could probably do with all three methods.

The trouble with coding for multiple operating systems

To conclude, MSGC provides a free piece of code that should run in any operating system. That allows you to compile the same code for systems other than OS/2.

The trouble with coding for multiple operating systems is that this does not solve the annoying differences in the way that resources are specified and sized, nor does it solve the perennial problem of a help system common to all. I do not know of any method of converting resources from one system to another at present, but there is some hope on the horizon with xchm, an open source helpfile viewer for Windows htmlhelp files.

Now all we need is for someone to make this work on OS/2...

Reprinted with permission of the author Peter Koller.
Published also on www.os2world.com 5/05/04