Use Strings to Internationalize C Programs

Nigel Jones

Products destined for use in multiple countries often require user interfaces that support several human languages. Sloppy string management in your programs could result in unintelligible babble.

A decade or two ago, most embedded systems had an extremely limited user interface. In most cases, the interface was either non-existent, or it consisted of a few LEDs and the odd jumper or push button. As the cost of display technology has plummeted, alphanumeric user interfaces have become increasingly common. Simultaneously, a variety of technological, economic, and political pressures have brought about the need for products to be sold in many countries. As a result, the need for an embedded system to support multiple languages has become apparent.

This problem is even more acute for the non-embedded computer world. Part of the solution for that marketplace was the introduction of Unicode, wide character types, and so on. Unfortunately, these techniques require storage capabilities and display resolutions rarely found in embedded systems. Instead, most embedded systems with displays typically use a low-resolution LCD or vacuum fluorescent display (VFD) with a built-in character generator. It's this type of display that I'll be concentrating on.

Lessons learned

A few years ago, I worked on an industrial measurement system. The product was to be sold in both North America and Europe. Consequently, one of the essential design requirements was to support multiple European languages. The product in question has a 240 x 128-pixel LCD panel with a built-in character generator. The character generator contains a subset of the ASCII 256-character set, including "specialized" characters such as c, u, and e.

Anyway, as I pondered possible approaches to the design, I looked back at my previous attempts to solve this problem. They weren't pretty! At the risk of ruining what little reputation I may have gained, I think it's instructive to look at these previous attempts.

Lesson 1

The first product that I designed that had an alphanumeric display was implemented with the typical arrogance of a native English speaker. Namely, it never occurred to me to even consider the rest of the world. As a result, my code was littered with text strings. That is, you'd see the assembly language (yes, it was that long ago) equivalent of:

```
WrStr("Jet Black");
```

To make matters worse, this construct would be found in many functions, split between several files.

The foolishness of this approach struck home when I was asked to produce a version of the product for the German market. I quickly realized that I had to edit source code files to implement the translation. This had several ramifications:

- A separate make environment was needed for each language.
- Every time a change had to be made to the code, the same modification had to be made to each language's version of the file. In short, it was a maintenance nightmare.
- The source code had to be given to the translator. I think you can imagine the problems this caused.

Lesson 2

The next product was a big improvement, because I did what should have been done in the first place, which was to place all the text strings into one file. That is, there were alternative string files called english.c, german.c, and so on for additional languages, each containing all of the strings for a particular language. Each string file contained a single array definition that looked something like this:

```
char const * const strings[] =
{
    "Jet Black",
    "Red",
    ......
};
```

Thus, to display a particular string, my code looked like this:

```
WrStr(strings[0]);
```

Now all I had to do to enable support for a new language was to hand a copy of english.c to the translator, and have him produce the equivalent strings for the new language. Unfortunately, it didn't work out that way. It turns out that the English language is extraordinarily terse when compared with certain other languages. For example, the German equivalent of Jet Black is Rabenschwarz.

Working from english.c, the translator assumed there were just nine characters into which to fit the translation. Thus, the translator was forced into abbreviating the German. However, in many cases, there was actually more space available on the display such that the abbreviation looked awkward in the product. The only way to find out was to execute the code and look at the results. This is a lot harder than it sounds, because many strings are only displayed in exceptional conditions. Thus one has to generate all the external stimuli such that those exceptions occur. In short, the translation effort remained a Herculean task.

Once it was complete, I still wasn't out of the woods, because the different length strings caused the memory allocation to change significantly. Although it did not happen, theoretically I could have run out of memory.

Lesson 3

By the time I was working on my third product requiring multiple language support, I was a lot wiser and memory capacities had increased dramatically. As a result, I now ensured that every string in my string file was padded with spaces out to the maximum allowed field width. Furthermore, I had also learned the intricacies of conditional compilation and passing command line arguments to make, such that I included every language into the same text file. Thus, strings.c looked something like Listing 1.

```
#if defined(ENGLISH)
   char const * const strings[] =
   {
      "Jet Black ",
      ...
      "Good Bye ",
      ...
      "Evening"
   };
#elif defined(GERMAN)
   char const * const strings[] =
   {
      "Rabenschwarz",
      ...
```

```
    "Auf Wiedersehen ",
    ...
    "Abend "
  };
#endif
```

Listing 1. Multiple languages in a single C module

This third solution worked well, except that at the same time, the size of the alphanumeric displays and the complexity of the user interface had increased. While my first product had just 30 or 40 strings, this latest product had around 500. Thus, the bulk of the user interface code ended up looking like this:

```
WrStr(strings[27]);
WrStr(strings[47]);
WrStr(strings[108]);
```

This code doesn't make clear what string I was actually displaying. So I was beginning to long for the original:

```
WrStr("Jet Black");
```

I ran into another major problem at this time. As the product evolved, so did the strings. I found myself wanting to delete certain strings that were no longer needed. But I couldn't do that without destroying my indexing scheme. Thus, I was forced into changing unwanted text into null strings, such that strings.c now contained sequences like this:

```
char const * const strings[] = {
    "Jet Black  ",
    ...
    "", /* deleted */
    ...
    "Evening"
};
```

Although this saved the space consumed by the string, I was still wasting ROM on the pointer. In addition, it looked ugly and had "kludge" written all over it. I also ran into a more serious problem. From a maintenance perspective, it would be very useful if related strings were in contiguous locations. Thus if a particular field could contain "Jet Black," "Red," or "Pale Grey," I would place these together in the string file. However, two years later, when marketing asked for "Yellow" to be added to the list of selections, I was forced to place "Yellow" at the end of the string list, well away from its partners. This pained me greatly.

There was one final problem with this implementation (and all the others) and that was the fact that the strings array was a global. I've become quite paranoid about globals in the last few years, such that when I look back at the code now, I have to confess that I cringe.

I also discovered a neat feature that was missing from all of the previous disasters. A few years ago, I saw a product demonstration in which the language was changeable on the fly. That is, without changing the operating mode or cycling power, the entire user interface could be changed to another language. The demonstration was incredibly slick. (Consider the following scenario. Your product is being introduced at a trade show. Some native French speakers come to the booth to look at the product. With the push of a button, you switch the user interface to French. You're already halfway to a sale.)

In addition to its value as a sales tool, the ability to change language on the fly is also a valuable tool for validating a new translation. It's particularly useful when the correct translation of a word depends heavily upon its context. When working through the string file, the translator can't see the context, so having the ability to operate the product and switch back and forth between languages is invaluable.

An international approach

Being quite a few years wiser than when Lesson #3 was learned, I was determined to come up with a scheme that would address as many of the aforementioned problems as possible and add the ability to switch languages on the fly. What follows is my solution. It's not perfect-but it is a lot better than any of the previous attempts.

The first decision I made was to separate the string retrieval mechanism and the string storage technique. There would be no more global strings array. Instead, strings would be accessed through a function call. This access function would take a string number as an argument and return a pointer to the desired string. Its provisional prototype is:

```
char const * strGet(int string_no);
```

This abstraction gave me freedom to implement the data storage part of the strings in whatever manner I saw fit. In particular, I realized that implementing the storage as an array of data structures would have considerable benefit. My data structure looked like this:

```
#define N_LANG4
typedef struct
{
    /*
     * Maximum length
     */
    int const len;
    /*
     * Array of pointers to language-specific string
     */
    char const * const text[N_LANG];
} STRING;
```

This arrangement offered some serious benefits. First, the maximum allowed string length for the field is stored with the text. Second, the original string and all of the various translations of it are together in one place. This makes life a lot easier for the translator. The downside is, of course, that this uses a lot more ROM than a preprocessor-based selection. In my case, I had the ROM to spare. With this data structure in hand, the strings array now looked like Listing 2.

```
static const STRING strings[] =
{
    {
        15,
        {
            "Jet Black ",           /* English */
            "Rabenschwarz    ",     /* German */
            ...
        }
    },
    {
         15,
        {
            "Red ",                 /* English */
            "Rot ",                 /* German */
            ...
        }
    },
    ...
};
```

Listing 2. A better string storage structure

The access function also had another valuable property. When the application requested a string, the access function could interrogate the relevant database to find out the current language, and return the requisite pointer. Voila! Language-independent code. The access function looks something like this:

```
char const * strGet(int str_no)
{    return strings[str_no].text[GetLanguage()];
}
```

Further improvements

The previous approach certainly solved a few of my problems. However, it did nothing for the problems of indecipherable string numbers or adding and deleting strings in the middle of the array. I needed a means of referring to the strings in a meaningful manner, together with a way of automatically re-indexing the string table. My solution was to use an enumerated type. The members of the enumeration are given the same name as the strings to which they refer. An example should help clarify this.

Assume the first four strings that appear in the strings array are "Jet Black," "Red," "Pale Grey," and "Yellow." To display "Red," I would have to call:

```
WrStr(strGet(1));
```

Instead, I define an enumerated list as follows:

```
typedef enum { JET_BLACK, RED, PALE_GREY, YELLOW, ... } STR;
```

I now change the prototype of `strGet()` to (with the changes in red):

```
char const * strGet(STR str_no);
```

Thus, to display the string "Red," the code becomes:

```
WrStr(strGet(RED));
```

Furthermore, by defining a macro, `Wrstr(X)` as follows,

```
#define Wrstr(X) WrStr(strGet((X)))
```

we can write:

```
Wrstr(RED);
```

This is just as readable as the original `WrStr("Red")`, but without any of the aforementioned problems. Furthermore, this technique allows one to insert or delete strings at will. For instance, I could insert "Pink" before "Red" in the strings array, do the same in the enumeration list, and recompile-and none of the code should be broken.

This was a major step forward, because I now had a system that met most of my goals

- No global data
- Easy to add additional languages
- Language selection on the fly
- Code is meaningful
- Allows strings to be inserted and deleted at will

Gotchas of enumerated types

However, a couple of "gotchas" arise when using enumerated types in this way. The first, and most important, is portability. ANSI C requires only that the first 31 identifiers in a name be significant. If you can guarantee that all of your strings are shorter than this, there is no problem. If you cannot make that guarantee, these are some of your options:

- Switch to a compiler that allows unlimited identifier lengths. Many compilers do have this feature.
- Ensure that all strings are unique within the first 31 characters. Note that if they aren't, the compiler should issue a re-declaration warning.

The next issue to watch out for occurs when you have a large number of strings. ANSI allows the compiler writer to implement enumerated types using an implementation-defined integer type. Thus, technically speaking, a compiler could limit the number of items in an enumeration to 127 (the largest positive number that can fit into an 8-bit integer). Thus, if you have rigid portability constraints, this technique may be problematic. However, practically speaking, most compilers appear to implement enumerations either as an int, or as the smallest integral type that can accommodate the enumerated list.

The third problem I ran into concerns the limited number of legal characters that can make up an identifier (that is, a-z, A-Z, 0-9, and _). For instance, it is impossible to exactly reproduce the string "3 Score & 10!". In situations like this, I used _ wherever I couldn't make the exact substitution. Thus, the enumerated value for "3 Score & 10!" would be `_3_SCORE___10_`, or possibly `_3_SCORE_AND_10_`. This isn't perfect, but it's still better than a meaningless identifier such as `STRING_49`.

The final issue was the absolute necessity of keeping the string file and the enumerated list synchronized. This proved to be quite difficult. To aid the process, I modified the string table slightly to include the enumerated type name in the comment field. Next, I ensured that the last entry in the enumerated list was always `LAST_STR`. This allowed the string array to be changed from being an incomplete declaration to a complete declaration. This means that the compiler will complain if the number of elements in the enumerated list does not exactly match the number of elements in the string array. This proved to be valuable in keeping the two files synchronized.

The winning design

The final enumerated list and string table declarations are as shown in Listing 3.

```
typedef enum
{
    JET_BLACK, RED, PALE_GREY, YELLOW, ..., LAST_STR
} STR;
static const STRING strings[LAST_STR] =
{
    { /* JET_BLACK */
        15,
        {
            "Jet Black ",          /* English */
            "Rabenschwarz ", /* German */
            ...
        }
    },
    { /* RED */
        15,
        {
            "Red ",                /* English */
            "Rot ",                /* German */
            ...
        }
    },
    ...
};
```

Listing 3: Final string storage structure

I did all of this manually, but you could certainly develop a script to automate the process of creating the enumerated list in the header file (from the contents of the string file).

Having gone through this exercise, I realized that I could make a bit more use of enumerated lists to make my code more readable and more maintainable. When the string data structure was introduced, a manifest constant `N_LANG` was used to specify the number of languages supported. A better approach is as follows (with the changes in red):

typedef enum
{ ENGLISH, FRENCH, GERMAN, SPANISH, LAST_LANGUAGE }
LANGUAGE;
Now, the STRING data structure is defined as:
typedef struct
{
/*
* Maximum length

```
    */
    int const len;

    /*
     * Array of pointers to language-specific string
     */
    char const * const text[LAST_LANGUAGE];

} STRING;
```

This change may look minor, but it makes adding another language more intuitive.

So far, I haven't mentioned the utility of storing the maximum string length in the STRING data structure. The use of this field arises when ROM is not plentiful, such that it is necessary to store strings without padding out to the maximum allowed field width. In cases like this, one has to be careful to clear the entire field before writing the string. This may be accomplished by using the string len parameter. If you can afford to pad all strings out to the allowed field width, it is permissible to drop the len parameter from the data structure.

Well, that's my fourth attempt at producing an international product. After three disasters, I'm reasonably confident that this latest attempt will at least make it into the "not bad" category. If you have any refinements that you would care to share, please e-mail me. In the meantime, I'm going to ponder how to elegantly and robustly support languages such as Chinese, Arabic and Russian in embedded systems. If I manage to find a reasonable solution, I'll let you know.

This article was published in the February 2001 issue of Embedded Systems Programming. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Support Multiple Languages" Embedded Systems Programming, February 2001.