Unicode Table Compression in FOX

First draft: 2005/03/30

Revised 2005/10/19

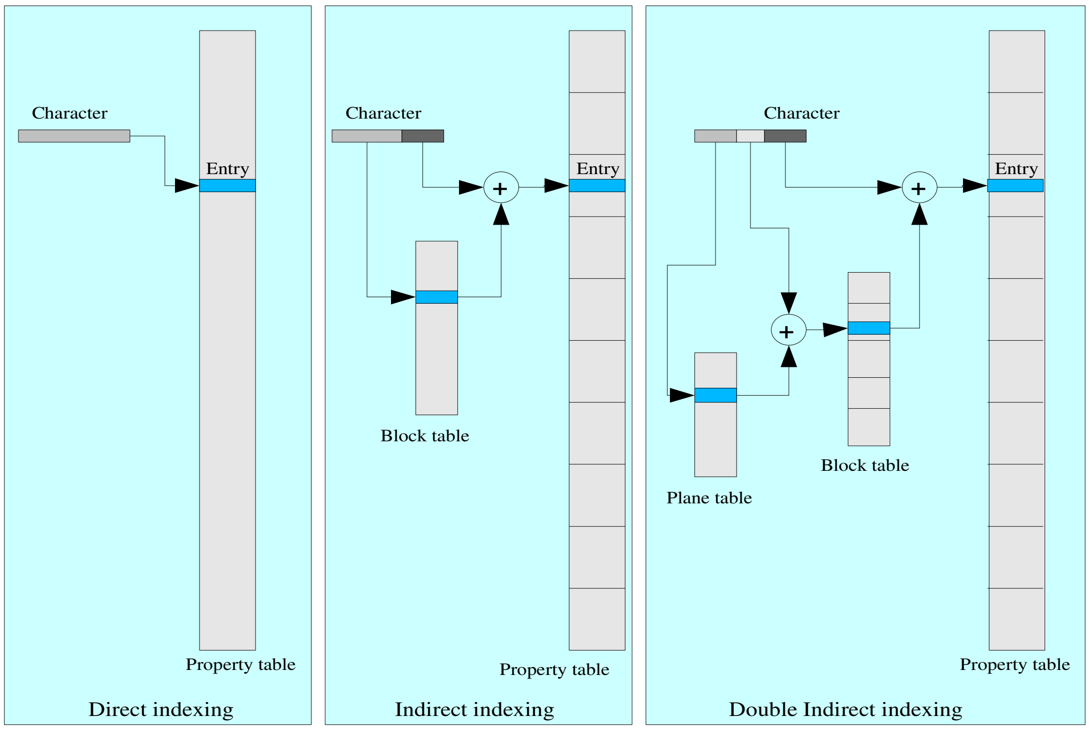
1. Unicode Tables

The unicode character set contains a huge number of code points, each of which has a number of properties. Examples of such properties are Lu (Letter/Uppercase), Mn (Mark/Non-Spacing), and so on. Recording all these attributes requires vast tables for all of the 1,114,112 code points in the unicode database.

However, these tables are very sparse, and this is something that can be exploited to reduce the table size.

2. Folded Tables

The Folded Tables algorithm is used to generate unicode character properties tables for use in FOX. The results of this algorithm are in ~src/fxunicode.cpp and corresponding header file ~include/fxunicode.h. Besides the tables themselves, this file also contains a number of small functions which access these tables.



The original table is accessed using direct indexing (left-most figure). Since there are 1,114,112 entries in the property table, this requires a very large amount of space.

However, the tables are very sparse, and large portions of it have identical information. The idea is not to store duplicate information. To this end, we cut the table into a number of blocks, and then overlapping those blocks which are identical.

We have chosen to cut the property table into 128-byte blocks, yielding a total of 8704 blocks for the entire unicode property table.

Since we now have to remember the start of each block, we need an extra table called the *block table* to remember where each block starts; thus the block table has 8704 entries.

The character is split into 7 bits property table index and 14 bits block table index.

This scheme achieves good compression of the property data, however it suffers from a pretty large block table overhead!

Thus, we implemented a double-indirect indexing scheme (rightmost figure). The property table is folded as before to merge blocks with identical information.

Blocks which contain identical information can be overlapped, and since many blocks do indeed contain identical information, a large compression factor can be achieved.

However, data is sparsely located over the tables and thus many blocks are mostly, but not completely, empty. To achieve at least some compression even in these cases, we shift such almost-empty blocks over the encoded blocks so as to benefit from a partial overlap.

We can obtain such a partial overlap by relaxing the requirement that blocks fall exactly on top of each other, and instead allow a block to start at any offset from the start of the table. As each block of the original table is compressed, we check every offset in the table to see if it fits there, and only place it at the end if no fit is found.

Finally, we notice that we can apply this algorithm recursively also for the intermediate level block table! Again this requires an extra table, the *plane table*. We split the original character into three parts now: the lower 7 bits which index into the property table, the middle 7 bits which index into the block table, and the upper 7 bits which index into the plane table.

With the introduction of the new plane table, we can fold both the property table as well as the block table.

In the current set of unicode tables, the property table compresses by a factor 100 to almost 2000. The block table can be compressed by a factor 8 to almost 40.

Thus, substantial amounts of space have been gained.

A typical code fragment used to access the table is shown below:

```
FXuint getProperty(FXwchar ucs){
  return property[block[plane[ucs>>14]+((ucs>>7)&127)]+(ucs&127)];
  }
```

This code contains only simple integer instructions and no branches or jumps and thus can be expected to pipeline very well.

Using the folded table developed above, character-class routines have been implemented which are needed for layout, numeric conversion, case folding, and so on. These functions are available in fxunicode.h and fxunicode.cpp.

The file fxunicode.cpp is automatically generated by a program **unicode.cpp** which reads and optimizes the Unicode Consortium's unicode tables for use in FOX. If the unicode standard is revised, generating updated tables should be pretty straightforward.