Dr. Unix (not so) Weekly: On character sets, the Euro symbol (and dead keys)

Dr. Unix 09-07-2000

Recently I received this question from **Expander**:

Q: Dear Dr. Unix, I would like to ask you two things that have been on my mind for some time. First, is there support for the Euro symbol in Unix/Linux? And secondly, I recently read that my keyboard might have *dead keys*. Should I replace these like I did with my dead goldfish?

A fair set of questions indeed. Below, I try to answer these questions in the unsurpassable Dr. Unix style: easy to digest and close enough to the truth to be **almost** indistinguishable from it.

Table of Contents

Use the following link table to jump directly to the information you're looking for.

The Euro

With your permission, I would like to start off with a short introduction in the Euro for our American readers: Motivated by three savage wars that rampaged the continent within the span of a hundred years, the leaders of a few important west European countries thought it might be a good idea to cooperate a little closer in order to increase the affluence of their citizens. This grand scheme was based on the (in my eyes correct) premise that people with a colour TV set, a VCR, two cars, a motor bike and two holidays abroad a year are much less likely to go out and pillage and plunder a neighbouring country (except when there is a major football/soccer tournament going on, but the Romans taught us that the plebejans should have their fun and games too). And so the European Union was formed.

Over the years, the union's countries integrated more and more closely, up to the point where most economic

legislation now comes from Brussels. A couple of odd years ago, some enlightened minds came up with the idea to replace the individual currencies of the member states with a single European currency: the Euro. The decision to undertake this astounding project was taken swiftly, and, true to form, the major disagreement between the member states was over where the new European Central Bank should be located (Frankfurt), and what the nationality of its president should be (the French wanted a Frenchman, but because nobody trusts the French with money, it became a Dutchman).

(Oh, and by the way, I ask forgiveness of the Europeans outside of the union for using the term "European" to denote the 15 (or EMU-11) member countries of the European Union. Please accept that I fully acknowledge your geographical locations within Europe, it's just too bad that you're not members yet....)

Eleven countries (of the fifteen, the so-called EU-11 or EMU-11) decided that they wanted to join the Euro project. The United Kingdom, Sweden, Greece and Denmark were either not allowed or did not want to join the Euro zone. At the time of writing (July 2000) the Euro has been in existence for some time. Its current implementation is through a fixed exchange rate between the original member currencies and the Euro, and a floating rate of the Euro against other international currencies. The exchange rate of the Dutch guilder against the Euro for instance has been set at 2.20371 guilders to the Euro. Through the fixed rates, the rate of the guilder against any other member currency is fixed as well. For instance the rate of the guilder against the Spanish peseta is determined by the officially set rates of these two currencies against the Euro. European law stipulates that to convert from one member currency to another you **have** to "go through the Euro" (meaning: convert from currency 1 to the Euro, and then from the Euro to currency 2). Here you will find a patched configuration file for the **GNU units** program that understands the Euro and the exchange rates against the member currencies.

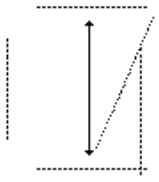
```
$ units -f ./units.dat
1382 units, 57 prefixes

You have: euro
You want: guilder
        * 2.20371
        / 0.45378022
You have: guilder
You want: italylira
        * 878.64102
        / 0.0011381212
You have: guilder
You want: mark
        * 0.88751696
        / 1.126739
You have:
$
```

However, we are not talking about an administrative exercise only! Starting in 2002, Euro coins and notes will gradually (over a six month period) replace the old coins and notes. So, no more yellow light houses on the Dutch fifty guilder note, no more gold and silver ten French franc coins, but instead a boring bunch of notes and coins, designed by some Austrian chap who won the competition on how to design the most non-offensive currency (imagine having to please both the Fins and the Italians, and everybody in between!).

The Euro symbol

The European "government" (ha ha) decided that the new currency should get its own currency symbol, and, in order to make life as difficult as possible for everyone they decided to design a completely new character, a sort of captical E on steroids (see picture).



This decision immediately sent a shiver through the spines of everyone with more than half a brain and some knowledge of computer systems. A new character, not present in any of the character sets in existence! Phew, why not go for the obvious instead? Now we would have to make new keyboards, modify/create/extend

character sets, design new fonts (or characters in fonts) and change **all** software that handles money in some meaningful way..... Fortunately, they also decided that instead of the new symbol we would also be allowed to use the trigraph **EUR** to denote amounts in Euros. So, I'll probably lose half my readers here, knowing that they can legally get away with not supporting the Euro symbol at all.

The inclusion of a new character into the world of computing is a depressingly non-trivial matter. The problem is mostly related to the fact that most software has been coded with a specific character set in mind, and can not easily be adapted to other character sets. However, before I delve into the specific solutions at hand for the new Euro symbol, I would like to spend some time discussing the concepts behind character sets, fonts and input methods since these are usually not well understood.

Character sets

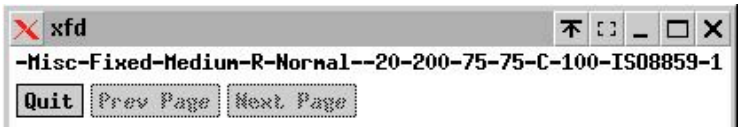
Each language has a set of characters that can be used to write text in that particular language. Most western languages derive their letters from the latin alphabet, but most tag on a few extra tildes, accents or cedilles left or right (e.g. þ, ý, ë). For instance the Dutch language has a 27th letter in its alphabet: the "ij" (written as a single character with the 'i' and the 'j' connected). Now, given the mindboggingly stupid necessity of computers to degrade every concept it wants to represent to a whole positive number, we've had to assign numbers to characters so that we could manipulate texts with computer programs. And so a bunch of people decided that it might be a good idea to associate the letter 'A' with the number 65, the figure '0' with 48, the space with 32 and so forth. However, other people thought it a much better idea to associate the 'A' with the number 193, the '0' with 240 and the space with 64! And thus the problem of different character sets was born.

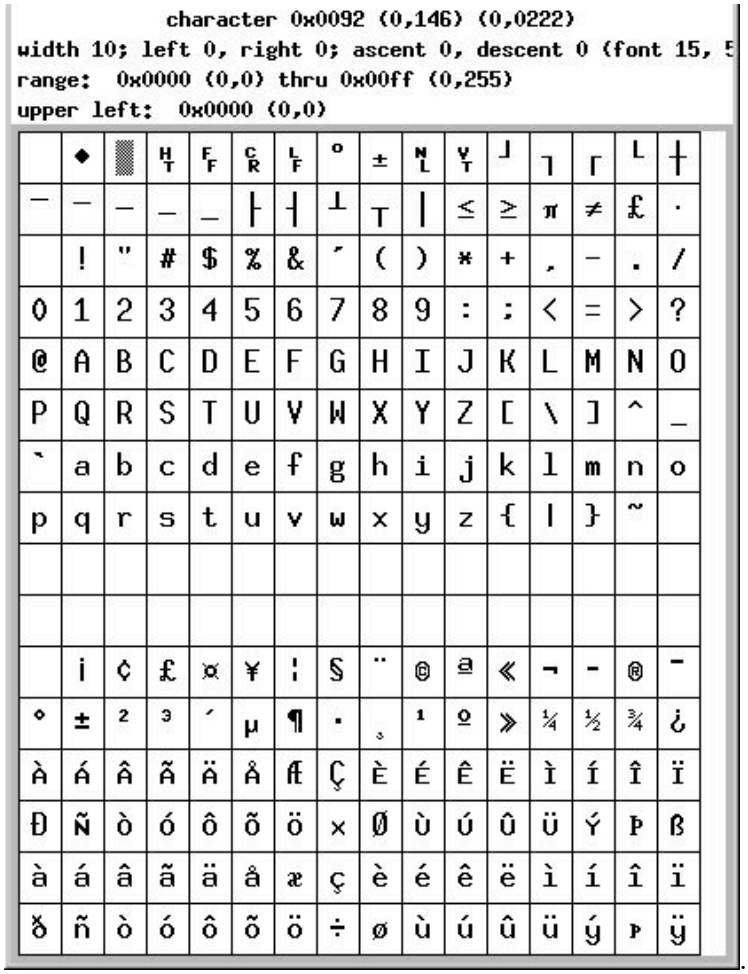
A **character set** is a table which defines the numerical codes to be used for characters. So if I give you a number and ask you what character that number corresponds to, the only correct answer you can possibly give me is: "In what character set?". Given that every file on disk just consists of a string of numbers, the interpretation of that file as a text file is thus highly dependent on the (implied) character set that underlies that file's content. Apart from human stubborness and the NIH (Not-Invented-Here) syndrome, there is also a real (technical) problem with defining character sets. Modern computers use the 8-bit byte (also called an octet) as the basis for information representation. With a single octet, you can only represent 256 different numbers (0 to 255), and there are simply more than 256 different characters in use in the combined western languages! So any single octet based character set that you create will leave some peoples on the planet very unhappy.

American cultural imperialism (and our willingness to succumb to that imperialism) left us with the omnipotent ASCII (American Standard Code for Information Interchange) character set: a 7-bit character set containing the 128 characters that are used in the English language (excluding some very British characters such as the Pound Sterling sign) plus some control characters. ASCII has become the lingua franca of computer text files: it is very well understood by almost all computers in the world. However, if you're not American, ASCII leaves a lot to be desired. For instance, suppose your name is Mariëlle (a completely random name, let me assure you of that :-), it would be pretty sad if you could not write your own name correctly using a computer! Obviously the ASCII character set does not suffice for her because it does not contain a code for the "ë" character. We thus definitely need another character set.

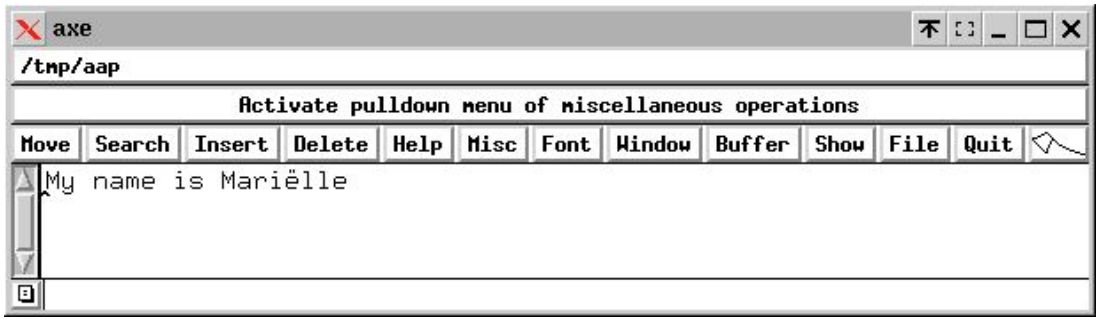
Many, many organizations have thrust themselves upon the problem of creating "standard" character sets that contain the various characters that these goodwilling people felt were necessary to do any meaningful work. And following their work, software developers started creating software that had some understanding of different character sets in the sense that they at least acknowledged that there was more than ASCII alone. Unfortunately, as has been pointed out before, it is simply impossible to create a single-octet based character set (containing a max of 256 characters) that suffices for all (even only European) languages. Compromises would have to be made! The International Standardization Organization defined a bunch of 1-byte character sets that embody these compromises. Wisely, they decided to keep the first 128 characters identical to ASCII, and used only the upper 128 positions in the table for so-called "international" (i.e. non-US) characters.

The most commonly used ISO character set is called ISO8859-1, and it contains the characters used for most western European languages. (see picture).





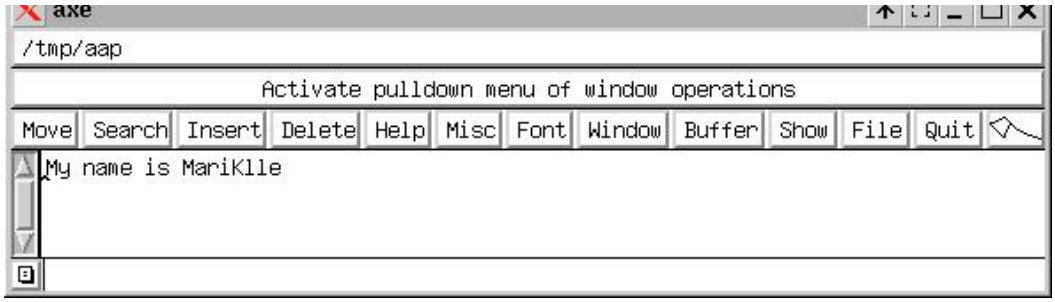
As you will see in the picture, Mariëlle will be pleased with the ISO8859-1 character set because it contains the "ë" in position 235 (or, in hexadecimal, 0xeb). So, that means that if she uses an editor that supports the ISO8859-1 character set, and she can somehow enter character code 235 from the keyboard, all is Hunky Dory! (we'll go over how you can enter these non-standard character codes later on). And, when using the X Window System and a suitable editor, this is not a problem, as the next image shows:



When we save this data to a text file using this particular editor (the "axe" editor), keycode 235 appears in the file:

```
$ od -c -t u1 /tmp/aap
0000000   M   y       n   a   m   e       i   s       M   a   r   i   ë
         77 121  32 110  97 109 101  32 105 115  32  77  97 114 105 235
0000020   l   l   e  \n
        108 108 101  10
```

However, when we open this file in an editor, print it on a terminal or on a printer, the way the character appears is completely dependent on the character set (and font) used by the program processing the file! If for instance we start the "axe" editor with a different character set and font setting (using the "-fn" option), and open the file, it could for instance look like this:



In the character set now employed by "axe", character 235 is evidently a sort of capital "K"! This can cause severe headaches! "axe" is obviously not aware of the character set intended to be used with this file. It just reads the file and displays the characters found using the font and character set that it is currently instructed to use (with the "-fn" command line option).

Fonts

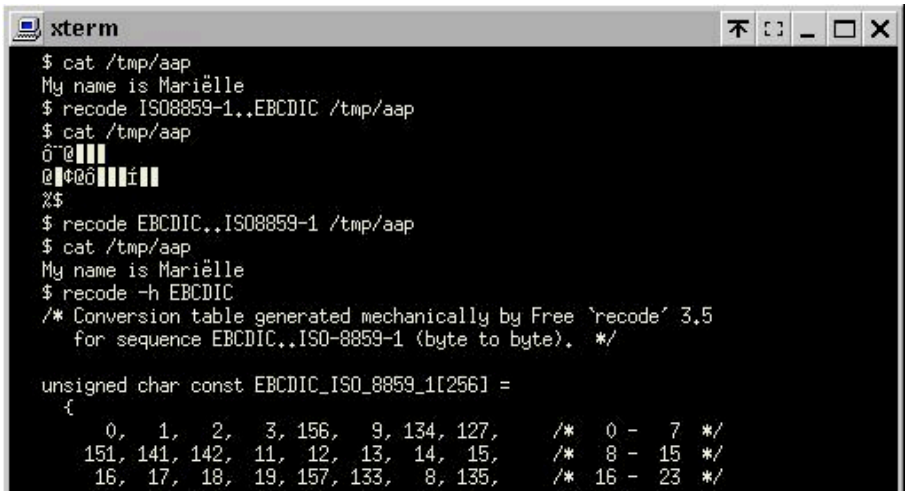
I've loosely coined the term "font" already a couple of times. Without going into too much detail I think we can safely say that a font is a set of instructions on how to draw characters in a certain character set. Take for instance the letter "A". There are at least five different ways to draw an "A" that are all still recognizable by me as an "A". In computers, a font file contains the instructions that are used by graphics software (such as the X Window System) to draw a particular character in a particular way. Fonts are usually tied into character sets because whatever instructions a font contains for drawing character 235, that is the way that we are going to perceive it. X couldn't care less whether we get to see an "ë" or a "K", as long as it knows how to draw character 235 it is perfectly happy. Whether or not it draws what we expect is something else alltogether.

Most programs (like the "axe" editor) don't give a flying f***ck (excusez le mot) about character sets. They are configured using a font, and that's how they are going to draw the characters received from files or through the keyboard. You enter character 235, it draws character 235. In whatever font you asked it to. Some programs however **have** to know about character sets. For instance because they have to know that "ë" sorts like an "e". These programs have to take care that they know the character set in which the data they process have been encoded (e.g. through a setting, or storing it in the file or whatever). These programs should then also not allow the user to select a font that is not designed for the character set the data has been coded in!

Most programs are not character set aware but still allow a user to specify a font to use (e.g. "axe", "xterm" and loads of others), even though that font might not be based on the character set that the user's data is encoded in. Talk about letting someone shoot him/herself in the foot!

Utilities

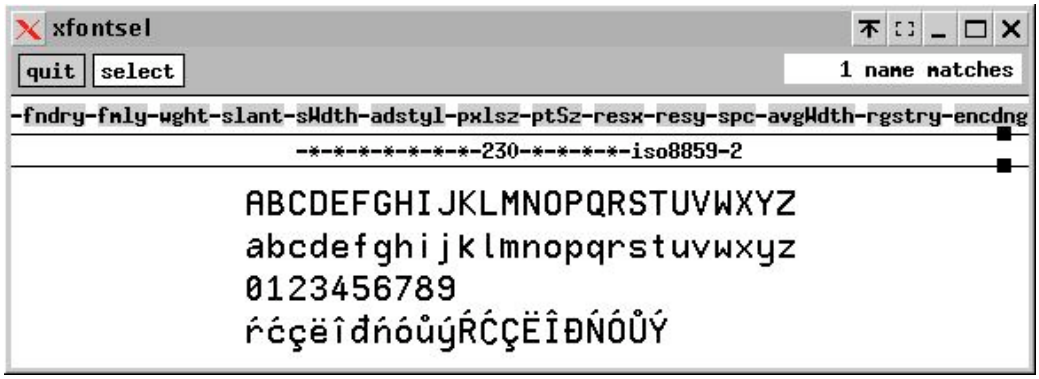
As a little sidebar, there are a number of interesting utilities that can help you sort out the mess surrounding character sets and fonts. First of all, there is the **GNU recode** utility. "recode" can be used to translate a text file from one character encoding to another. It knows the position of all characters in an astounding number of character sets and can perform translations between them. As an added bonus it can generate C arrays that can be used to perform translations in your own program. An example of "recode":



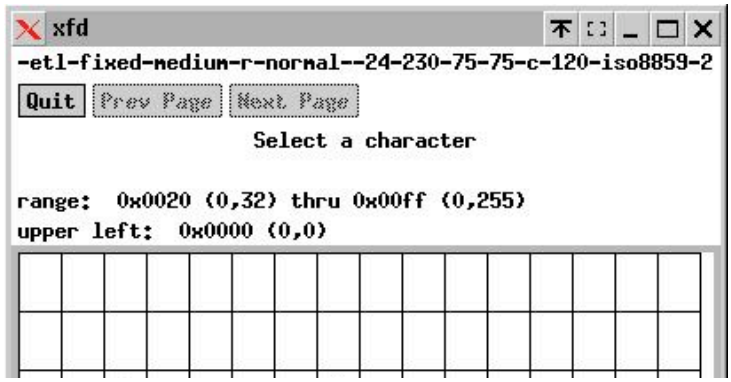
```
     24,  25, 146, 143,  28,  29,  30,  31,     /*  24 -  31  */
    128, 129, 130, 131, 132,  10,  23,  27,     /*  32 -  39  */
    136, 137, 138, 139, 140,   5,   6,   7,     /*  40 -  47  */
    144, 145,  22, 147, 148, 149, 150,   4,     /*  48 -  55  */
    152, 153, 154, 155,  20,  21, 158,  26,     /*  56 -  63  */
     32, 160, 161, 162, 163, 164, 165, 166,     /*  64 -  71  */
    167, 168,  91,  46,  60,  40,  43,  33,     /*  72 -  79  */
     38, 169, 170, 171, 172, 173, 174, 175,     /*  80 -  87  */
    176, 177,  93,  36,  42,  41,  59,  94,     /*  88 -  95  */
     45,  47, 178, 179, 180, 181, 182, 183,     /*  96 - 103  */
    184, 185, 124,  44,  37,  95,  62,  63,     /* 104 - 111  */
    186, 187, 188, 189, 190, 191, 192, 193,     /* 112 - 119  */
    194,  96,  58,  35,  64,  39,  61,  34,     /* 120 - 127  */
    195,  97,  98,  99, 100, 101, 102, 103,     /* 128 - 135  */
    104, 105, 196, 197, 198, 199, 200, 201,     /* 136 - 143  */
    202, 106, 107, 108, 109, 110, 111, 112,     /* 144 - 151  */
    113, 114, 203, 204, 205, 206, 207, 208,     /* 152 - 159  */
    209, 126, 115, 116, 117, 118, 119, 120,     /* 160 - 167  */
    121, 122, 210, 211, 212, 213, 214, 215,     /* 168 - 175  */
    216, 217, 218, 219, 220, 221, 222, 223,     /* 176 - 183  */
    224, 225, 226, 227, 228, 229, 230, 231,     /* 184 - 191  */
    123,  65,  66,  67,  68,  69,  70,  71,     /* 192 - 199  */
     72,  73, 232, 233, 234, 235, 236, 237,     /* 200 - 207  */
    125,  74,  75,  76,  77,  78,  79,  80,     /* 208 - 215  */
     81,  82, 238, 239, 240, 241, 242, 243,     /* 216 - 223  */
     92, 159,  83,  84,  85,  86,  87,  88,     /* 224 - 231  */
     89,  90, 244, 245, 246, 247, 248, 249,     /* 232 - 239  */
     48,  49,  50,  51,  52,  53,  54,  55,     /* 240 - 247  */
     56,  57, 250, 251, 252, 253, 254, 255,     /* 248 - 255  */
};
$
```

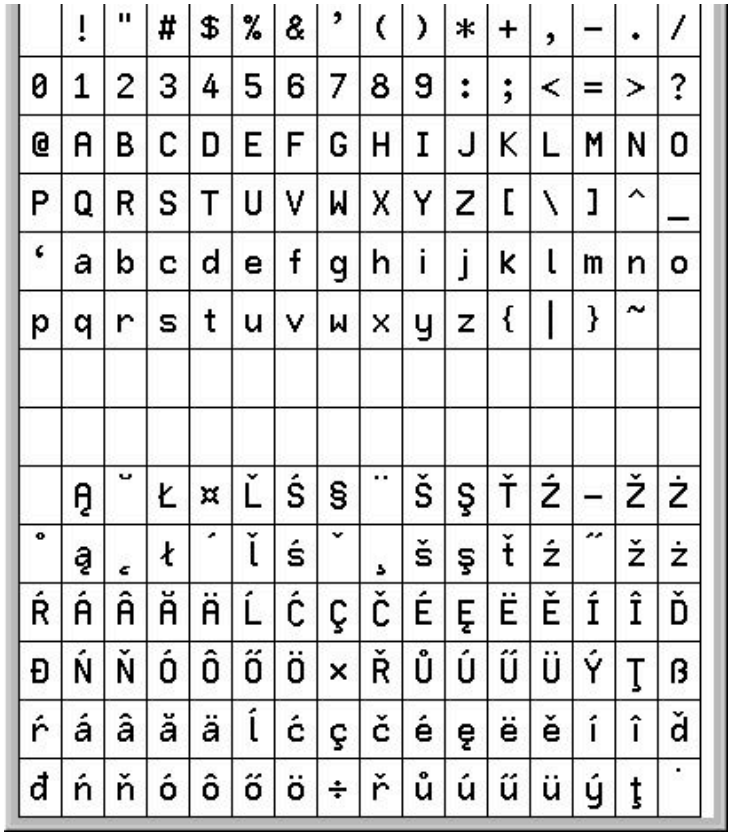
In this example I recode the file earlier from ISO8859-1 character set into an EBCDIC format. Now EBCDIC is a very uncommon character set that is mostly used on IBM mainframes. If I type the file to the terminal after the recode, "xterm" can not make heads or tails from it because it uses a font that is not based on the EBCDIC encoding.

Another interesting set of utilities are the X font selector **xfontsel** and the font dumper **xfd**. Through "xfontsel" one can create an X font name based on characteristics such as font weight (bold, demi, normal), slant (italic or not) *and* character set (through two values, the **registry** and the **encoding**).



The example above shows the selection of an ISO8859-2 font with point size 230. "xfontsel" contains a "select" option that copies the fontname selected so far to the paste buffer so that it can be pasted in another window (e.g. in the "-fn" parameter of an "xfd" command). "xfontsel" shows an example of the font, "xfd" gives a complete overview of all the characters in the font, and how they look when drawn to the screen.

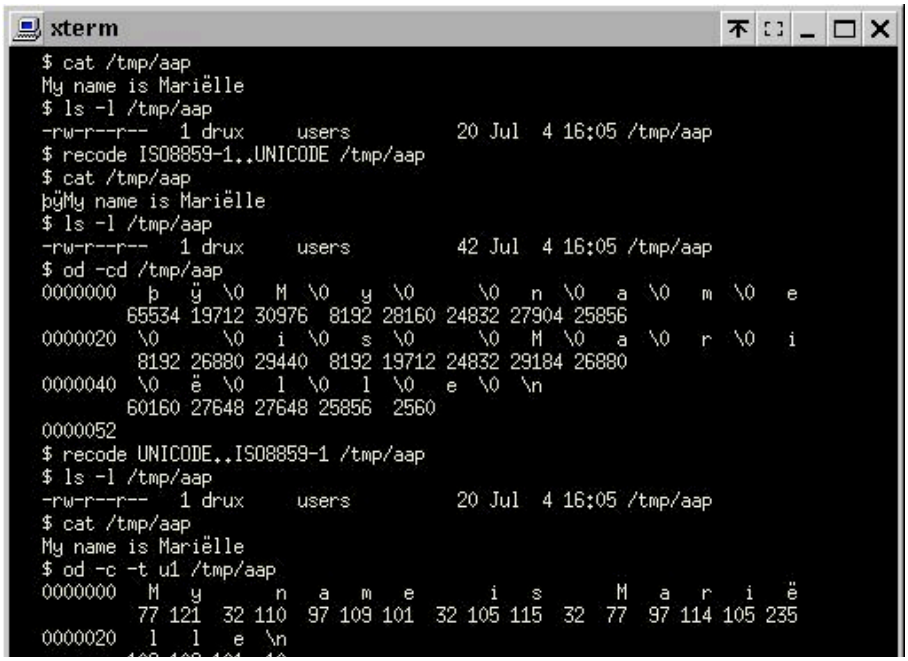




Please compare this example with the "xfd" dump of an ISO8859-1 font above, and notice the different characters that are present in this particular character set! However, Mariëlle would still be happy with an ISO8859-2 font because the "ë" is still there at location 235!

Unicode

The current "be it and end all" of character set is the Unicode character set. In contrast to the character sets we have seen so far, Unicode is a 16-bit character set. It can thus contain an astounding 65,536 different characters: enough to contain all western alphabet characters and a lot of characters of the most common asian languages. Unfortunately, because each character in a Unicode encoded text takes up 2 octets (16 bits), it brings with it a whole new range of problems for applications that live in an 8-bit character universe (like most applications). It also increases the file size of text files with a factor two (approximately). See the example below:

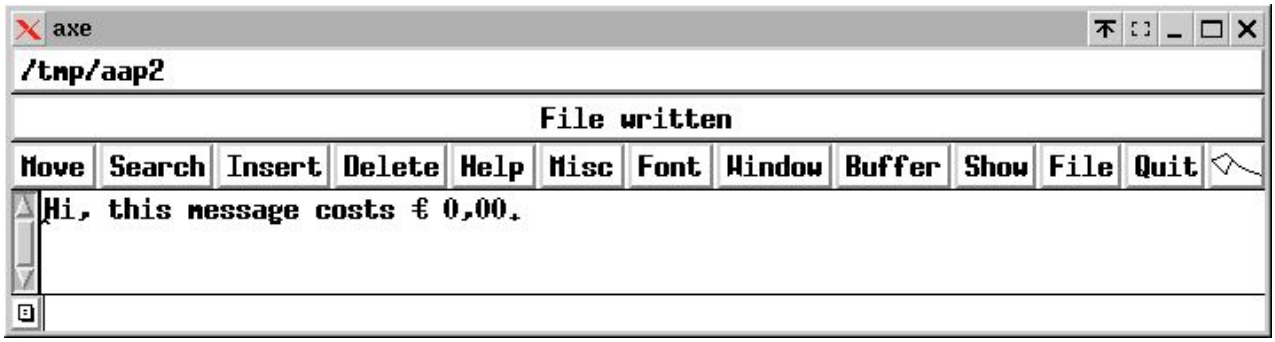


There are other solutions through special treatment character sets like UTF-8, but it would be beyond this article to delve into that.

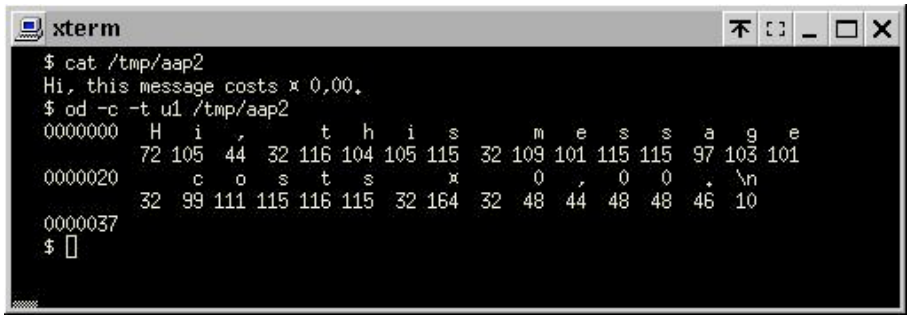
Displaying the Euro symbol

Now, when word was leaked that there was a new important character coming up, the world's character set standardizers had to figure out what to do with it. As you'll imagine, most character sets were more or less full, so they either had to change an existing character set, or create new ones. The character set that accomodated the new character the easiest was Unicode. Since there are numerous blank spots in the Unicode definition, the body responsible for Unicode simply allocated a blank cell to the new character and told everyone what the character code of the Euro symbol in Unicode 2.1 was (the exact number eludes me for the moment, but it is there somewhere, trust me, at U+20AC if I am not mistaken).

ISO did a long and painful navel staring session, and then decided to create a new character set (ISO8859-15) that contained the Euro symbol at location 164 (or, in hexadecimal, 0xa4). So, in order to use the Euro symbol in an editor, you must convince the editor to use an ISO8859-15 font and enter character 164 in one way or another. Look at the example below to assure yourself that it is possible:



The particular font used here is pretty ugly, but it happened to be the only ISO8859-15 font that I had lying around on my laptop. One can't have it all I guess... If I look at the actual content of the file edited here, I see that there is character 164:



However, please note that my xterm displays the character 164 not as the Euro symbol but as another one! That is because this xterm uses an ISO8859-1 based font so character 164 looks different in this font! However, if I start another xterm with an ISO8859-15 font, e.g. through:

```
$ xterm -fn -*-*-*-*-*-*-*-160-*-*-*-*-iso8859-15
```

then the display of the file looks like this:

Are you confused already?

Quite simply, because "xterm" is completely unaware of the character set of the data printed to the terminal, it just outputs everything in its selected font. The characters are drawn based on the instructions in the font, which happen to be based on a particular character set. "xterm" does not explicitly control that, so it just looks the way it looks in the font (and by implication character set) that "xterm" happens to be using.

So, the morale of the story so far is: If you want to display the Euro symbol in an application, be sure that that application uses a font that is based on a character set that contains the Euro symbol somewhere. Then make the application emit the character code of the Euro symbol, et voila! You might want to download a whole bunch of fonts that are based on a Euro symbol compatible character set! If you have an application (like an advanced word processor) that allows font switching **within** the application, you can select an ISO8859-15 (or other Euro compatible font) for one character and enter the Euro symbol character code (this is how Microsoft Word and other editors solve it. Adobe has a series of fonts that only contain Euro symbols exactly for this purpose).

Printing the Euro symbol

Now when we finally succeeded in convincing the X Window System and X client applications to display the Euro symbol, along comes our wish to print this symbol as well. As you'll imagine, you open a whole new can of worms here. Basically, inside the printer is a rendering engine not unlike an X server: it knows how to draw characters based on a numerical character code and a font definition that specifies character drawing instructions. In Unix it is up to the application to generate a print stream that instructs the printer to select an appropriate font (based on a character set containing the Euro symbol), and then send the printer the numerical character code of that Euro symbol. Although the mechanics of this are more or less the same, the actual procedure and support differs hugely from printer to printer. Adobe has created a set of PostScript^{tm} fonts that contain the Euro symbol (see the links below). As an end-user, you are more or less at the mercy of your applications here. When printing, they must be smart enough to remember the font that you selected for displaying non-ASCII characters and send the printer the correct information to select the same (or a similar) font (at least based on the same character set).

This can be a real pain in the you-know-where. Prepare for some tinkering!

Using the Euro symbol in HTML

The World Wibe Web consortium (or whoever is in charge nowadays) has decided to create a standard facility for displaying the Euro symbol in HTML pages. Special character code € can be used to render the Euro symbol in your web pages. It is up to the browser to either display this symbol as the trigraph "EUR", or as the official "capital E on steroids". Your browser displays it like this: €.

Euro and Unix URL's

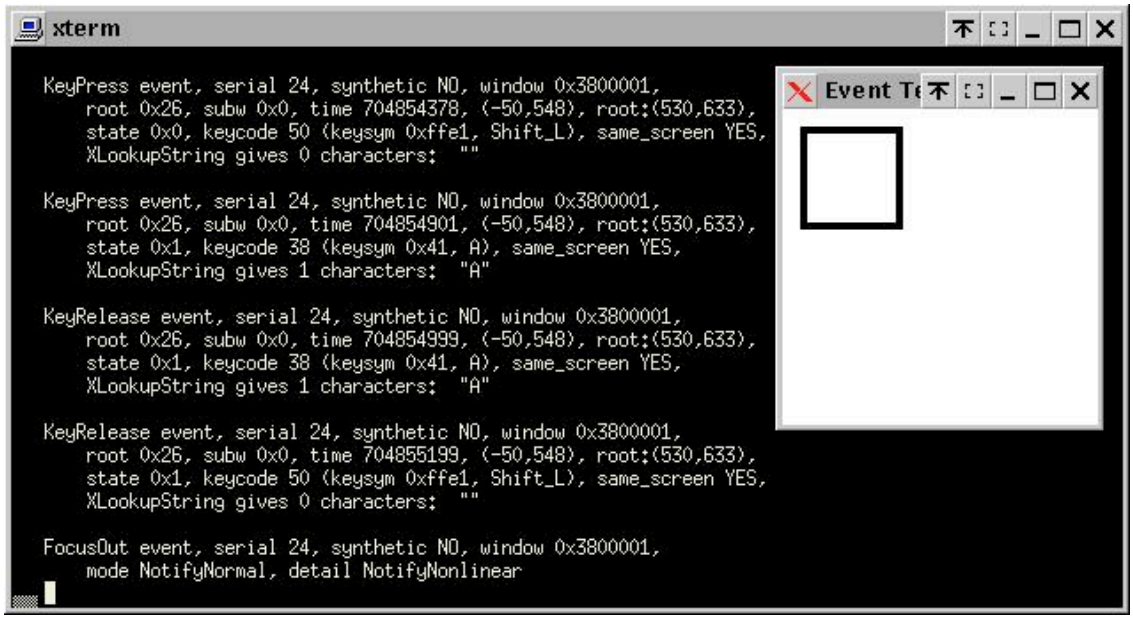
If you want even more information on character sets, Unix and the Euro symbol, the following list of URL's is a nice starting point:

- http://czyborra.com/charsets/iso8859.html
- http://www.adobe.com/type/eurofont.html
- http://www.ifconnection.de/~jsf/europunx_en.html
- http://www.muelly.ch/projects/kde/euro/generic/introduction.html
- http://www.ATComputing.nl/euro/
- http://www.ispo.cec.be/y2keuro/src/eupract.htm
- http://www.europa.eu.int/euro/html/entry.html

Inputting the Euro symbol

An interesting side problem connected to this whole Euro business is how we can input the Euro symbol's character code in a given application? Without delving into the dungeons of X keyboard abstractions and input methods too deeply, I want to make a few useful remarks here that can assist you.

In the X Window System, each keypress and release generates an "event" that is sent to the application that has the input focus. A good utility to visualise this is the "xev" command. "xev" opens a window and then dumps a textual representation of all events that it receives:



In the example above I've entered a capital "A" from the keyboard, which evidently generated the following events:

1. A keypress for "left shift".
2. A keypress for "A".
3. A keyrelease for "A".
4. A keyrelease for "left shift".

In the event listing you'll see two interesting pieces of information: the **keycode** and the **keysym**. Each physical key on the keyboard is internally associated with a keycode. These keycodes are mostly determined by the layout of the keyboard. In the "xev" event listing we can see that the keycode of the left shift key is "50", and that of the "a" key is "38". So each keypress generates an event, even if it's a key that normally does not generate a character directly (such as shift, control and alt). Because keyboard layouts differ from computer to computer, and from country to country, the logical keypress is not determined by a keycode, but by a **keysym**. Loosely speaking we may say that the keysym is the numerical value of the input character in the character set that we are using. So, in all ASCII-compatible character sets (like ISO8859-1), the keysym for "A" is 65 (or, in hexadecimal, 0x41). The X server contains a (keyboard and country dependent) translation table for keycode to keysym. In the "xev" sample we can see that the keypress for the "A" is accompanied by the information that the keysym value is "0x41" (65).

Now since the keyboard is physically too small to contain a separate key for all characters that we want to have at our disposal, it has been standard practice for ages to use key compositions to input less commonly used characters, like capital letters, certain punctuation et cetera. For instance on my keyboard, to generate an exclamation mark I must press "shift+1". Since there are multiple keypress events coming in, the X server must keep track of the fact that the shift itself does not generate data, and that for the second keypress a different keysym must be returned to take the fact into account that one of the shift keys is currently down. (Technically speaking, the shift key generates a ksysym as well (0xffe1 for left shift), but it is ignored because the auxilliary function "XLookupString()" returns 0 to indicate that there is currently no real character data to report.)

The keys on the keyboard that are only used for composition purposes (like shift, control and alt) are called **modifier keys** or **dead keys** (since you can press them, and nothing seems to happen. Try it, press shift five times in a row now. The shift key looks pretty dead to me :-).

The translation table that the X server uses is pretty simple. It can contain up to four different keysyms for each keycode:

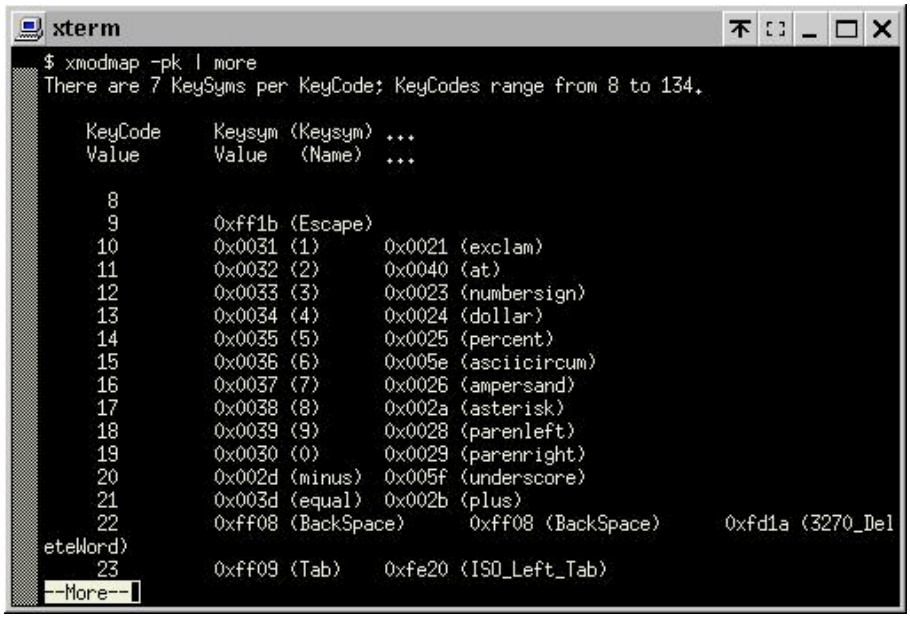
1. For the key.
2. For the key, when pressed in combination with shift.
3. For the key, when pressed in combination with another modifier.
4. For the key, when pressed in combination with shift and the other modifier.

So for instance on my laptop computer, the translation map for keycode 24 usually contains only two keysyms: "0x0071" (q) and "0x0051" (Q). So when I press key 24 the X server generates a keypress event for a keysym of "q", unless also the shift key is down, in which case it generates a keypress for a keysym of "Q".

However, to allow for more flexible input, the X server also allows one to use another modifier in combination with the keys to generate (non-standard) keysyms. This facility is somewhat limited, and for full internationalization of the input an application must conform to the X11 i18n (internationalization (which starts with an "i", then has 18 other letters, and ends with an "n", hence the name "i18n") standard which is **way** beyond the scope of this article.

xmodmap

The mapping tables that the X server uses to convert from keycodes to keysyms under various modifier conditions can be viewed and modified with the **xmodmap** utility. "xmodmap -pk" shows the current mappings from keycode to keysym:



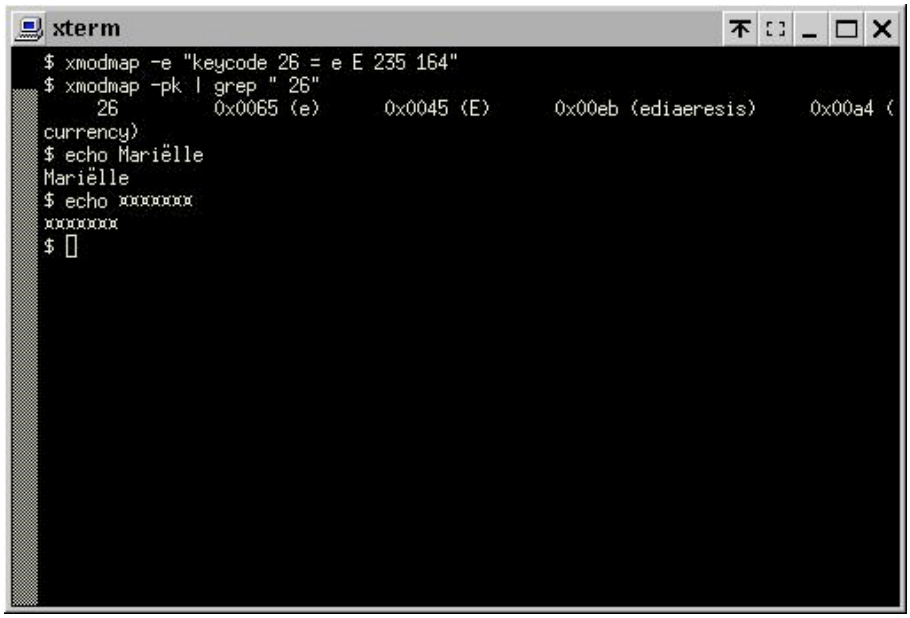
In this table you can see which keysyms are generated for the four cases outlined above. Mostly, only two keysyms are registered for a keycode: the normal and the shifted case. We can for instance see that for keycode 10 the X server generates keysym "0x0031" (1) in the normal case and keysym "0x0021" (!) in the shifted case. The third and the fourth entries in the tables are empty, meaning that we have not defined additional keysyms for use with another modifier key. (Another thing you might note is that most (if not all) keysyms also have a descriptive name. These names relate back to the X Window System include file "keysymdef.h" which is normally located in "/usr/include/X11".

Using the "xmodmap" utility we can also change the mapping tables, for instance to bind a keysym to another keycode, or to add a keysym to a keycode. In our case we want to bind keysyms that are not already bound (the "ë" and the Euro symbol) somewhere in the map so that we can enter them from the keyboard. By far the easiest way to accomplish that is to assign a new (and unused) modifier to the mapping table and then enter the characters we want in the third and fourth column in the mapping table. Now, it so happens that my laptop keyboard has two (until now) completely useless keys with the Microsoft Windows logo on them. Under the X Window System, these keys generate the keycodes 115 (left Windows key) and 116 (right Windows key). The accompanying keysyms are "Super_L" (0xffeb) and "Super_R" (0xffec). These codes and syms are easily verified with the "xev" utility described earlier.

To use these keys as the modifier key for the third and fourth column of the mapping table we must bind the special keysym "Mode_switch" (0xff7e) to their keycodes. This is easily accomplished with the "xmodmap" utility:

```
$ xmodmap -e "keycode 115 = Mode_switch"
$ xmodmap -e "keycode 116 = Super_R Mode_switch"
$ xmodmap -pk
    ...
    115         0xff7e (Mode_switch)
    116         0xffec (Super_R)        0xff7e (Mode_switch)
    ...
```

In the example above I bind the "Mode_switch" to keycode 115 (left Windows key) and to the shifted state of keycode 116 (right Windows key). This means that both the left Windows key and shift plus the right Windows key will enable the mode switch to the third and fourth column of the mapping table. I can then use "xmodmap" again to bind the special characters 235 (the "ë" in ISO8859-1 and other character sets) and 164 (the Euro symbol in ISO8859-15) to the "E" key (keycode 26):



In the example above you can see that the keysyms "e", "E", 235 (the "ë") and 164 (the Euro symbol's character code) are bound to keycode 26 (the "E" key on my laptop keyboard). This means that when that key is pressed, the X server will generate one of these keysyms depending on the state of the shift and the "Mode_switch" keys. Once bound, I can now enter the "ë" and the Euro symbol from the keyboard through "left Windows key + E" and "shift + left Windows + E", or through "shift + right Windows key + E" (Euro symbol only!). In the example you can see that I can enter these characters from the keyboard.

The only quiz question that I leave to you is: "Why do I get the funny "o" instead of the Euro character when I press "shift + left Windows + E" in "xterm"? (Answers to Dr.Unix@e-zine.nluug.nl).

Other articles by Dr. Unix:

- Dr. Unix runs into corporate security
- Dr. Unix (pas très) hebdomadaire : des jeux de caractères, du symbole Euro (et des touches mortes)
- Dr. Unix Weekly: Why Linux will replace most proprietary Unices
- Dr. Unix runs DOS and Windows programs on Linux

E-mail: Dr. Unix

[Index]