

## software localization

by [sascha leib](#)

if you want to sell software beyond your home market (and to more than the few freaks who prefer english software), you'll need to do some mysterious ritual called "localization". this tutorial will give you some help with this process. 'some help' means that it's really just an introduction as the topic is a little bigger as it could be covered here.

## translating software

the process of translating software is essentially different from that of translating books. literature translation has to keep a certain amount of the local colour of the original: you wouldn't want to have a translation of **don quichote** in which he fights the windmills of california instead of la mancha, or a dutch translation of james joyce's **ulysses** in which a leopold bloom would roam the streets of amsterdam: not quite genuine, indeed.

this is different with software: you wouldn't want to buy software that can only use some obscure chinese number format. so why should you expect people in china to buy software which can only use an american number format?

basically, a localized software should look and work as if it were written in the cultural context of the target market. this does not only include language, but also icons, "look & feel", and functionality. just recently, microsoft had to take it's german version of the financial software **money** from the market because it's **euro** support was not adequate. of course, european financial software usually has euro support, american software usually not (**quicken** had the same problem).

even icons can be culturally dependent: the icon to the right is taken from an email program. it might be a good choice for american users, as the picture obviously shows an american mail box. however, this kind of mailbox is not commonly used outside of north america. i have asked some people what they actually see there and the most common answer was a 'bread box'. nothing you would directly associate with email.



a good background reading for these issues is apple's [human interface guidelines](#). you should read it anyway because it provides essential information on good software design.

## terminology

you expect that a window on your computer screen is always called 'window' and not 'program area', or whatever else somebody might invent to describe it. and a button is always 'button', not 'action area' or whatever. while this is quite obvious in english, it happens that two different translators decide to choose two different translations, which would lead to quite confusing situations for the end users.

to avoid this problem you should use a terminology database which describes standard translations for common elements. apple provides this for various languages along with it's [apple glot](#) software. **apple glot** itself is quite useless for realbasic-built applications since it can only access the strings that are directly stored in the various resources, but the provided translations are stored in tab-text format and can therefore be read by any text editor.

### note

*localized software should look and work as if it were written in the cultural context of the target market.*

### note

*always respect the terminology lists the os manufacturer provides!*

here's a list of the standard items in the **edit** menu in a couple of languages:

english	french	german	spanish	finnish
<b>Edit</b>	<b>Édition</b>	<b>Bearbeiten</b>	<b>Edición</b>	<b>Muokkaus</b>
<b>Cut</b>	<b>Couper</b>	<b>Ausschneiden</b>	<b>Cortar</b>	<b>Leikkaa</b>
<b>Copy</b>	<b>Copier</b>	<b>Kopieren</b>	<b>Copiar</b>	<b>Kopioi</b>
<b>Paste</b>	<b>Coller</b>	<b>Einsetzen</b>	<b>Pegar</b>	<b>Sijoita</b>

unfortunately, the operating system manufacturers could not agree on a common terminology which works cross-platform. that's why we will not only find **exit** in windows applications to do the same as **quit** on the mac, but also completely different translations of some items. a 'button' is 'knopf' on german macs, but 'schaltfläche' on windows machines. other software manufacturers again use different translations: **star office**, for example, uses 'aktionsfeld'.

it is hard to say which of these translations is best, but it is definitely bad that there are so many of them.

### equivalence

the above may make you believe that it is actually much easier to translate software texts than a novel or so. as a matter of fact, it is not. it is even much more difficult.

the word of magic is **equivalence**. it refers to the fact that there is rarely any word that can always and under all circumstances be used as the best translation for a specific word in another language.

the german word **auto** (yes, from 'automobile') may be the usual translation for the english **car**, and in 99% of all cases it might be the best translation, but there are circumstances under which **wagen** ('wagon' or 'cart') or **gefährt** ('vehicle') may be better choices.

the problem in literature translation is that the translator has to decide **on every occurrence** of the word which translation is the best. the problem with technical translation is that he has to find a terminology that describes the car unambiguously and then use it consequently throughout all future translations. or in other words: the translator has to create terminology for your product.

this also means that much care has to be taken to choose the right word. a poor translation would not only make the program difficult to use, but would also have to remain there for some time, as those who already got used to it wouldn't want to learn a new terminology.

in their finnish version of the **internet explorer**, **microsoft** (again) uses the term **virkestä** as a translation for **refresh**. the translation is right, you will probably find it in every dictionary. what you won't find there is the connotations finns connect with this word: **virkestä** is pretty close to what americans mean if they say: "have a coke!"

as funny as it sounds, people may have thought that this is some kind of **pause** function, thus calling the support to ask where the 'refresh' button is. not very funny if **you** are the support person.

in the worst case, words can get completely new meanings: the terms 'lonesomeness' and it's german translation 'einsamkeit' both describe the state of being alone (they are valid translations). however, they are perceived in a completely different way in the cultural contexts of the united states and germany respectively. generally speaking, the state of being alone is understood as something positive in germany, but rather as negative in the u.s. (peter hofstätter, 1957: gruppodynamik, rowohlt, hamburg)

but more likely are cases where double-meanings change the whole statement to a pun. for your amusement, here some examples of how **not** to do it:

"**you can ride on your own ass!**"  
advertisement for donkey rides in greece

**"women are not allowed to have children in the bar"**  
sign in a hotel bar

if you are not a native speaker, or if you do not speak a language **almost** as good as a native speaker, you should seriously consider to ask a professional translator to do the job. there is a good reason why translators (should) have a long and intensive education (after which they earn good money).

in fact, you should even make sure that the translator is a native speaker of the target language. most translation offices will try to convince you that their translators can translate into both directions. even if they have a 'native' proof-reader, the quality will never reach that of a native speaker as translator.

<b>note</b>
<i>if you're not a native speaker, you should better ask a professional translator to do the job!</i>

there is even good reason to do it as the big software houses and make the localization in the country it is targeted for. the reason is that language and cultural concepts change quite fast. if you ever lived abroad for a year or more you may have noticed this when you come back. if a german in the u.s. does the translation for you, he may use terms and concepts that are 'out of fashion' back in germany since 2, 5 or more years.

---

**costs**

i agree that all of the above drastically increases the costs of localizations - and the money spent on localizations first has to come back by increased sales of your product. given the fact that many computer freaks understand at least enough english to work with most software, this may not always be the case.

on the other hand, there is usually relatively little text in most applications itself. if you look at the **realbasic** application: around 95% of the text to be translated is in the online help, which leaves basically the menu items and error messages as first priority.

here's a checklist for you to decide if localization pays off:

<b>note</b>
<i>smart software design can save you a lot of money in the localization process.</i>

- **does the target group use localized software?** this question is often forgotten. most programmers are used to english software and prefer it over localized versions; a finnish compiler would probably not sell at all. on the other hand, this also restricts the target group. localizing realbasic was, i think, a smart move as it allows non-english speakers to start programming.
- **is the market big enough?** if there are only 100 potential users in the target area, 20 of them use the english version and 60 others would use the english version if no localized existed, the 20 customers you'll lose might not be enough to give you reason to go through all this trouble.
- **do you have direct access to the market?** if you are located in the u.s. you may not be able to make full use of the market potential, simply because you do not know about the characteristics of the market, how to use multipliers, how to access to sales channels, etc. a localized version can help you to get access, but usually you should try to find a local distributor first who can help you with the localization.
- **can you provide a local infrastructure?** users may expect that they can call a help desk, or send emails in their native language.
- **does your software support easy localization?** you can save a lot of money if you make sure from the start that your application is easy to localize.

actually, the best this tutorial can do is to give you advice on how to design your software so it is easy to localize. if you do this smartly, all the language and culture-specific parts are collected in one place (usually one module). some of these things, however, can be left to the system, as the mac usually knows what formats to use:

---

### date and time

luckily, the mac os already takes most of the work with number formats, etc. away from you: the user can choose which date, time and currency format he prefers and all applications should respect this.

**realbasic** already does most of rest for you: if you use the **date** object, you have direct access to the date strings, etc. here's how the date function works:

	<b>.shortdate</b>	<b>.longdate</b>	<b>.abbreviateddate</b>
<b>u.s.</b>	01/02/92	Thursday, January 2, 1992	Thu, Jan 2, 1992
<b>spain</b>	2/1/92	jueves, 2 enero 1992	juev., 2 ener 1992
<b>germany</b>	02.01.1992	Donnerstag, 2. Januar 1992	Don, 2. Jan 1992
<b>france</b>	2/01/1992	Jeudi 2 Janivier 1992	Jeu 2 Jan 1992
<b>finland</b>	2.1.1992	torstai 2. tammikuu 1992	to 2. tammi 1992

plus, you can expect users to have customized formats. the author, for example, uses a modified british date format.

the above does not yet reflect the non-roman formats, which may not even use our gregorean calendar system, but as i said before, you just have to use the **date** object and the system takes care of the rest.

one thing you shouldn't even consider for a moment is to extract parts of the date strings for whatever purpose. for example: **nthfield(d.longdate, " ", 1)** may look smart to get the day of the week, but may as well return something completely different, depending on the user settings in the **date & time** control panel.

---

### addresses

you wouldn't believe how many address forms in the internet **require** that you select a **state** from some pop-up menu. or even worse: require you to enter the state abbreviation, and then complain that this is not a correct one. Of course, the **state** part of an address only applies to u.s. americans. we don't have states here in finland. of course we have some kind of states in germany, but they do not appear in the address (and they won't be recognized by these dumb web forms anyway)..

my country, of course, is **finland**, and i usually enter **California** for state. That's the only abbreviation i do know. The problem is that i occasionally get spam which is clearly targeted for a californian audience (or even a downtown san francisco target group it seems)

but this is only the start of the problems: addresses are formatted differently in different countries. In most european countries, the postal code is printed **in front** of the city, in america it's afterwards. in japan, the recipient's name is in the bottommost line and the city in the topmost.

if you want to print addresses, you should have a look at the great compilation of international postal address formats compiled by the [universal postal union](http://www.upu.int/addressing/an/). there are currently 189 countries listed, which is probably more than you'll need: <http://www.upu.int/addressing/an/>, also available as [pdf file](#)! read it carefully - it's a bible!

---

### word orders

you can not assume that the word order will be the same in all languages. this sounds obvious but is often forgotten. you can often see code like this:

```
MsgBox "The result is: " + str(result)
```

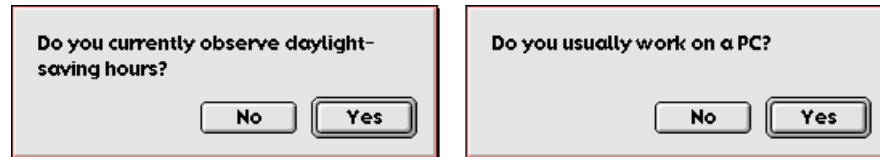
but in other languages the correct place for the result might be somewhere in the middle of the text.

a good example might be the **about..** item in the apple menu. it should contain the name of your application, and in most european languages the correct place for the name is in the end (but in front of the elipsis) - in **arabic**, however, the right place would be in the middle, in **chinese** and **japanese** it should be in the front.

this becomes interesting if you plan to use constants for localization. please see the second part of this tutorial for programming information, the "how to do it".

### foreign concepts

in some cases you may even have to cope with language concepts that are in all aspects foreign to english (or whatever your mother tongue is). take for example the following two dialog boxes:



ok, the examples are stupid. what we need to notice here is that in both of them the push buttons are labeled **yes** and **no**. most programmers would assume that it is enough to translate them once and then use these translations again. wrong! look at the finnish translations:



the first problem here is that in finnish the word **no** is a **verb**. that means that it is conjugated. here, **ei** means as much as "it is not" while **en** could be translated to "i am not". **kyllä** is a plain and straight "yes", but **on** comes closer to "it is that way".

i spare you the details why these words have to be used, just keep in mind that other languages have different concepts and translation is a lot more than just replacing one word with another.

### command keys

usually, the command keys should not change during the localization. that means that you can always use command-c to **copy**, no matter what language version you are using. this comes handy if you don't understand the language of the program. i recently had to work with some japanese software - a language i don't understand at all - still i could rely on **c** stands for **copy** and **q** for **quit**.

this becomes a problem, however, if the keyboard shortcuts are not available on foreign keyboards. this does not happen with the 'plain' characters (a-z) and numbers, but [ or ? for example are not available as command keys on my (finnish) keyboard (which makes debugging in realbasic really a pain!).

you should also be aware that users may have foreign keyboards but still use english software versions. to use [ and ] for important functions is an absolute no-no! here in finland many users have swedish software installed, while many estonians use finnish software. i can imagine the same for chinese who use taiwanese software or portuguese who use spanish programs. all of them may have problems to access certain keyboard shortcuts unless you stick to the ones that are the same for all keyboards: the letters a-z and the numbers. if you need more shortcuts, you can use the modifier keys (shift, option, etc.) with them. this makes the shortcuts also easier to remember.

**note**  
*avoid keyboard shortcuts that may not be available on foreign keyboards!*

## windows locales

if you've ever been unfaithful to your mac and started up a windows machine, you may have noticed that some terms are different on that other operating system. **exit**, for example, is what windows users select if they want to **quit** an application.

as a keyboard shortcut, you have to press a cryptic **alt-f4** combination to do the same as we would do with **command-q**. well, these are the people who complain that you have to drag a disk onto the trashcan to eject it.

anyway, if you plan to port to that other platform, you should be aware that the terms are slightly different.

but there's more to it than that: under windows, the programmer can define keyboard shortcuts that only come to effect when the user opened the menu using the keyboard (with **alt** and the arrow keys). these shortcuts are defined by adding a **&** to the menu text. the character directly following the ampersand will be displayed as underlined and can be used as shortcut key.

for example, the string **e&xit** would appear as **exit** in the menu, and the **x** key could be used as shortcut to select this item.

this has one clear disadvantage: since only letters can be used that appear in the menu item title, the shortcuts may have to change in localized versions, thus have to be taken care of in the localization process.

<b>note</b>
<i>windows terminology may be different from macintosh terminology!</i>

---

## things go wrong

there are about a million things that can go wrong with software localization, and there is no checklist for your application. the above only provides a few examples for things to look out for.

there's a great site in the net called [interface hall of shame](#) where you can find good examples of what goes wrong. there is even a great [globalization](#) section you should check out. very educating!

**ps:** why don't you write the [author](#) of that site a little note about the globalization problems involved with his **mailbox** icons. i'm sure he'd appreciate :-)

---

## software localization

while part 1 focused on background information on software translation, this part is on the technical side of the localization process, focusing on **realbasic**'s localization features. if you don't use realbasic, either download it from [www.realbasic.com](http://www.realbasic.com), or forget the rest of this page. goodbye then :-)

---

## constants

it is vital that you already take care of localization while you write the program. this basically means that you should keep all the texts in one place so the translator doesn't have to go through all your program code and change things there (you would also have to pay extra for a translator who can read sourcecode).

in a typical realbasic application you will have 3 types of text:

- **hard-coded texts**, like menu texts, button captions, window titles, etc;
- **soft-coded text**, that is all text that your program changes and therefore has to be represented somewhere in your sourcecode.
- **external text**, that is all text that is external to the actual application, i.e. texts that are loaded from a file (e.g. help texts) or the resource fork.

luckily, realbasic gives you a powerfull tool for software localization, that is the **constant** system. it means that you create constants for all the texts in your application. you can store all the constants in one module so the translator only has to take care of this one module.

one side-effect of this is that you don't have to disclose your sourcecode to anyone. be that because it is top-secret, or that it is such a mess :-)

so before we start, first create a new **module** and name it **locales** (it actually doesn't matter how you name it, just make sure you remember the name ;-)) then have a look at your sourcecode: most probably you will have a statement like this:

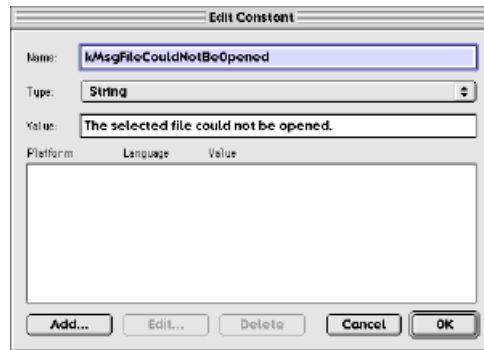
MsgBox "The selected file could not be opened."

this should be changed to:

MsgBox kMsgFileCouldNotBeOpened

and the constant should be added to the 'locales' module:

a few notes about constant names: since all constants are global (i.e. they can be accessed from all parts of your application), you should take care that they don't conflict with any of your local variable names. of course, you should also be able to remember them. to achieve this, you should use some kind of notation system. the system i use is quite popular with c-programmers: all constants start with a 'k' (for 'constant', of course), followed by a descriptor of what it is used for (e.g. **Msg** for message box, **Button** for button caption, or **Menu** for a menu text). then add a descriptive name of what is inside. this is, however, only a suggestion. you may use your own system if you like, but the examples here in this tutorial use this scheme.

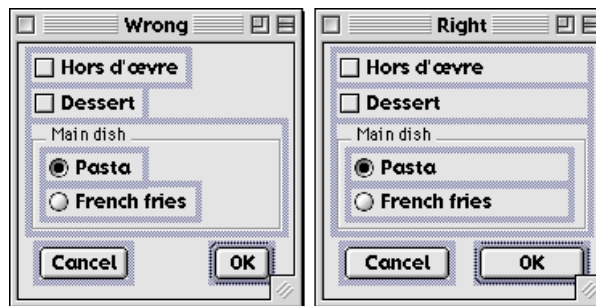


beginners often choose too short names for constants (and variables), but since most of them are only used at one place in the sourcecode, don't hesitate to use long names. the longer the name is, the smaller is the probability that you'll have some conflict with local variables. besides, longer names are usually more descriptive than short ones.

## controls

one place where you can find hard-coded texts are controls. the caption text of a button, or the text label of a check box usually does not change during runtime. if it does, you have to do that in your code and we spare that until later.

a common mistake is to make text labels exactly the size of the contained text. just like here:



there are basically three reasons why you shouldn't do this. first, of course, the user may use a different system font, which takes more space than the one you are using - and secondly (you guessed it), translated texts may take much, much more space. in fact, you may even have to use a completely different dialog layout.

but if you have space left in your window, there's no reason why you shouldn't use it. of course, you'll have to check with all the

translations to see if they still fit (for example, the **cancel** button is still too small for the german translation **abbrechen**, i've tried!)

of course, even in the foreign languages, there should still be enough space to make sure the text fits in even if the user has a different system font.

and there's more to it. the following picture shows how the above dialogs would look like on an arabic system:



you have probably learned in school that arabic and hebrew are written right to left! of course, apple has thought of that and spiced up the **text** control panel on such systems. one of the new options is called **system direction**. on other language systems (especially roman script systems like english) you may have wondered why there's a special control panel for **text** at all. now you know.

on right-to-left systems, it's not only the text that runs in the other direction, check boxes, radio buttons, etc. are also aligned to the right rather than the left. in the menus the command keys are on the left, as are the 'submenu' triangles, while menu icons are on the right. sounds strange, doesn't it?

ok, so now how do you get the constants into the button captions?

if you just enter the name of the constant, let's say "kButtonOKCaption", into the **caption** property field, you will get just that, i.e. the name of the constant as caption. that's not what we want, so there must be another way.

the symbol of magic bears the beautiful name 'octothorpe' but most people call it 'this strange cross symbol on the three key'. it looks like this: # . just put it in front of the constant name and everything will work as it should.

so with our example we should use "#kButtonOKCaption", and if this constant is defined the constant value (here 'ok') will automatically appear in the control. if the constant is not defined, the caption string will appear just as you have entered it.

whenever you have hard-coded texts, i.e. in menus, window titles, buttons, text fields, etc. you have to use the '#' in front of the constant name to tell realbasic that it should use a constant instead of a static text.

there are, however, a few places where this does not seem to work. in the latest version this seems to be:

- in the tab-panel, the text in the tabs

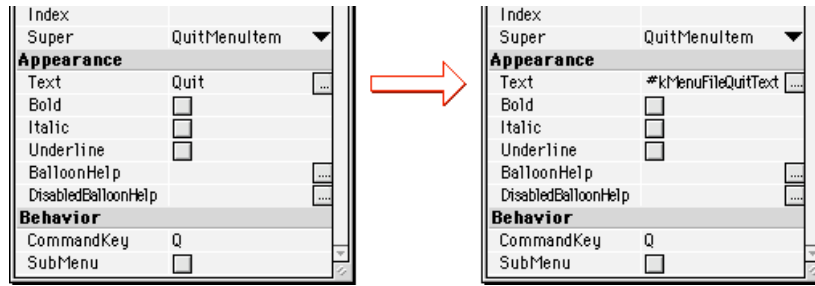
hopefully they will be supported in a future version.

#### menus

menu items on the other side are fully supported, and they are also where you will normally start your localization. in this tutorial we will use the **quit** menu item as an example for how to add locales to your application.

please open the **menu** in a new project in realbasic. then change the **text** property of the **quit** menu as follows:





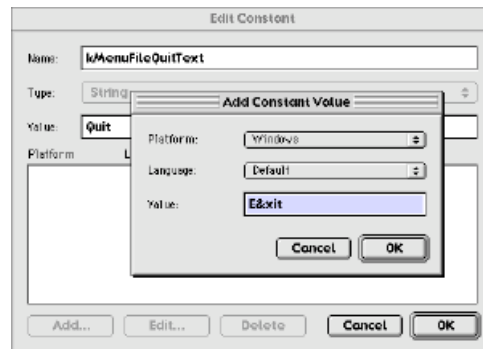
this is only the first part. now you need to set a default value for this menu. to do this, you need to add a **module** to your project, then add a new constant and name it **kMenuFileQuitText**. in the **value** field, write the default value you want to assign to it. that would be **quit**, of course.

#### default values

default values are used if no other, more specialized value can be found. what you write into the **value** field in the "edit constant" dialog is always the default value with the lowest priority. that means, it will only be used if there is no other value which matches the current configuration. in this case, however, there is no other value up to now, so this is not a question.

so let's add another value: press the **add** button and enter **e&xit** to the upcoming dialog. set the **platform** popup menu to **windows** but leave the **language** popup to **default**.

if you don't know what the ampersand symbol ('&') means in the string, please read the first part of this tutorial (or just try it!)



what we have now is a default value for the **windows** platform. that means that if we compile our application for windows, **exit** will be used for the menu item, while on the macintosh platform it is still **quit** (or more precise: for all platforms **except** windows **quit** will be used)

you can create default values for platforms or for languages (which are then used regardless of the language). the value in the **edit constant** window always has the lowest priority (i.e. it will only be used when no other value matches), but it is not quite clear whether language or platform defaults have higher priorities, so be careful when using both!

#### changing code

let's have another look at the message box example:

```
MsgBox "The selected file could not be opened."
```

that's actually not how you should really do such a message. better would be:

```
MsgBox "The file " + theFile.Name + " could not be opened."
```

this only leaves the problem how to make this localizable. one solution would be:

```
MsgBox kMsgTheFile + theFile.Name + kMsgCouldNotBeOpened
```

with the two constants then set to the appropriate values. this has two main disadvantages (plus a few minor):

- two constants for only one text

the constants are difficult to translate without knowing the sourcecode

let's have a look how this problem is solved in other programming languages. this might give us a hint on how we can handle it in realbasic.

in **c** we could make use of it's 'buffered' output feature. a sample **c** code could look like this:

```
printf ("The file %s could not be opened.", (*fh).name)
```

where **%s** would be replaced with the name of the file. don't worry if you don't understand these brackets - **c** is a bit more tricky than realbasic.

in realbasic, we could construct something similar (or even better) using the **replace** function. just change the **kMsgFileCouldNotBeOpened** constant to:

**The file %s could not be opened.**

and then change the code to:

```
MsgBox Replace(kMsgFileCouldNotBeOpened, "%s", theFile.Name)
```

if you want, you can, of course, use a different replace string than **%s**, and if you need to replace the same string at more than one place, you could also use **replaceall**, but that should rarely be necessary.

---

## translations

now the fun part starts. up to now we have only taken care that all texts are on one place and can be easily translated. now, of course you should translate them. i'm sorry, but i can't really help you with that.



once you have the translations, you simply add them as values to each constant. make sure to set the **language** (i always forget that!) and that's it.

in the picture on the right are some translations for the **quit** menu item. unfortunately there is no scroll bar next to the value list so you can only select and edit the few values that fit into the window. i hope very much that this bug

(it's definitely a bug!) will be fixed in the next update.

more fun is the debugging - at least on the mac platform. just set the language you want to test in the **project settings** dialog, and all constants will automatically represent the correct language string. you don't even have to compile the project, except if you want to test the windows locales. (a hint: **virtual pc** is a good help for testing windows compiles!)

however, once you think it's ready, you should still give a (compiled) version to some beta testers and ask them whether the translations are complete, correct, and make sense. translators and programmers usually only harmoniously cooperate in missing important details!

---

## that's it

so much for this little localization tutorial. if you found it helpful, have a look at my other articles, software and bits and pieces of useless information: