

MULTILIZER[®]
THE SOFTWARE GLOBALIZATION COMPANY



Developer's Guide

Multilizer® 5.1 – Developer's Guide

April 2003

Copyright © 2003 Multilizer Inc. All rights reserved.

Multilizer is a registered trademark of Multilizer Inc. All other trademarks and registered trademarks are the property of their respective owners.

Table of Contents

Introduction.....	1
Multilizer naming conventions	1
Tutorials for different platforms	1
Conventions used in this book	1
Installation.....	2
Documentation.....	3
Registration.....	3
Technical Support.....	3
Part I: Getting Started.....	4
Overview	5
Internationalization	5
Localization	5
Localization Technologies	8
File Localization	8
Binary Localization	8
Resource Localization.....	10
Source Code Localization	10
Document Localization	11
Component Localization.....	12
Database Localization.....	12
Comparing Different Application Localization Technologies	12
Unique architecture	13
Translation memory	14
Platforms and editions.....	15
Supported tools, operating systems and documents.....	15
.NET and Visual Studio .NET	15
Databases	15
Delphi and C++Builder	15
J2ME	15
Java.....	15

Oracle Forms.....	15
Palm	16
StreamServe.....	16
Symbian.....	16
Visual Basic and Embedded Visual Basic.....	16
Visual C++ and Embedded Visual C++.....	16
Visual J++	16
WAP	16
XML	16
Multilizer Setups.....	16
Multilizer	17
Multilizer Translator Edition	17
Coping with different languages	18
Character sets.....	18
Single byte character sets	18
Multi or double byte character sets	19
Bi-directional character sets	19
Unicode.....	20
Single-byte and double-byte character set trouble.....	20
Universal character encoding system	20
Text Input	20
Western languages	20
Far Eastern languages	21
Middle Eastern languages.....	21
Country specific items	21
More about localization	21
Part II: Tutorials	22
.NET23	
Visual Studio .NET Project File Localization	23
.NET Resource File Localization	24
English Application.....	24
Internationalization	25
Internationalization of forms	25

Internationalization of code	26
Creating a Project	29
Translating a Project	32
Visual C++	34
Binary Localization	34
Resource File Localization	35
English Application.....	36
Internationalization	37
Creating a Project	43
Translating a Project	45
Visual Basic.....	47
How to use Multilizer	47
Localization process.....	48
Source localization	48
Windows	49
Windows CE.....	49
Binary localization	49
Component Localization.....	50
English Application.....	50
Internationalization	51
Creating a New Project	51
Translating a Project	54
Delphi and C++Builder	56
Binary Localization	56
Component Localization.....	57
English Application.....	58
Binary Internationalization.....	58
Creating a Binary Project	66
Translating a Project	69
Integrated Translation Environment.....	70
Controlling What Properties Are Localized.....	70
Component Internationalization	71

Creating a Component Project.....	77
Using Run-time Dictionary	78
Java 83	
Opening a Monolingual Application.....	83
Resource Bundle Localization.....	84
Resource Bundle Internationalization.....	84
Creating a Resource Bundle Project.....	86
Translating a Project	88
Localization with Multilizer Components	88
Making the Application Multilingual	88
Creating a Component Project.....	90
Component Internationalization	91
Changing Language at Run-Time	95
Writing Multilingual Applets.....	97
Writing Multilingual Swing Applications	97
Font Issue with Non-Western Languages	97
Java Micro Edition	98
J2ME Localization	98
Application With An English User Interface.....	99
Internationalization	100
Creating a New Project	102
Translating a Project	104
Visual J++.....	106
Open a Monolingual Application	106
Make Application Multilingual.....	107
Create a Project for the Application.....	108
Translating a Project	110
Internationalize Your Code.....	110
Change Language at Run-Time.....	114
Symbian	117
Symbian Localization	117

Application Using an English User Interface	119
Internationalization	119
Creating a New Project	120
Translating a Project	123
Deploying	124
Conditional Compiling	125
Palm 127	
Palm Localization	127
Application With An English User Interface	128
Internationalization	129
Creating a Project	130
Translating a Project	133
XML 135	
XML Localization	135
English File	136
Creating a New Project	136
Translating a Project	139
WAP 142	
WAP Localization	142
Application with an English User Interface	143
Internationalization	144
Creating a New Project	144
Translating a Project	147
Database	149
General Considerations on Database Contents Localization and Used Terminology	149
Field and Table Naming Conventions and Restrictions	151
Fields Localization	152
Tables Localization	153
Single Table Localization	155
Creating a New Project with Localized Fields	156
Creating a New Project with Localized Tables	159

Creating a New Project with a Single Table	160
Translating a Project	162
Oracle® Forms	163
Main differences between Oracle Translation Manager/Builder and Multilizer	163
Choosing the right native language.....	163
How Multilizer localizes fmb files.....	163
Multilizer project versus Oracle Translation Manager/Builder database	163
PL/SQL code localization	164
Oracle Forms Localization	164
Creating a New Project with Oracle Forms	165
Translating a Project	169
Importing Translations From OTM/OTB into Multilizer's Translation Memory	169
StreamServe	174
StreamServe Localization	174
English File	175
Creating a New Project	175
Translating a Project	177
Tagging.....	178
Translating a Project.....	179
Adding Languages.....	180
Part III: Using Multilizer.....	184
Globalization Process.....	185
Globalization team	185
Background	185
Team members	185
Tasks.....	185
Work-flows	186
Enabling concurrent work.....	186
Localization type.....	187
Internationalization	188

To do's	188
Creating a new project	188
Specifying the target type	188
Specifying a file target	189
Entering project information	190
Selecting languages used in the project	190
Adding Targets	191
Finishing the Wizard	192
I18N essentials	192
Source-code globalization	193
Component globalization	193
Binary globalization	193
Localization	194
To do's	194
Preparing a project to be outsourced	194
Adding visual context	195
Adding comments	196
Locking strings – preventing translation of strings	196
Adding project strings	197
Adjusting scanning options	198
Updating the project file	198
Pre-translation	199
Building a Localization Kit	199
Translation	203
Build localized software	204
To do's	204
Importing translation to a project	204
Starting the Import Wizard	204
Specifying import properties	206
Build localized software versions	208
Binary localization	208
Source localization	209
Component localization	209

Quality Assurance – QA	210
Test languages	210
Cell highlighting.....	211
Translator components	212
Translation Memory on database server	213
General considerations	213
Database related tasks	213
Create new database	213
Define Connection parameters	213
Connecting from Multilizer to MTM	214
Managing MTM rights	218
Builder Command Line Tool	220
Adding.....	221
Examples.....	221
Scanning	222
Examples.....	222
Removing.....	223
Examples.....	223
Importing	224
Examples.....	224
Exporting.....	226
Examples.....	227
Exchanging	228
Examples.....	228
Translating	229
Examples.....	229
Building	230
Examples.....	230
Creating a Sub Dictionary	231
Examples.....	231
Appendix A: Glossary	232
Code page.....	232
Dictionary.....	232

Globalization.....	232
Internationalization	232
Language IDs	232
Linguist	233
Locale	233
Localization.....	233
Module.....	233
Translator	233
Index.....	234

1

Introduction

The purpose of this Developer's Guide manual is to familiarize you with Multilizer and the concepts and techniques behind software and document localization. This manual gives a short overview of world's languages, which is a central issue in software localization. Also, the different scripts are discussed. Later these issues are described in the context of developing software. After the introduction to software localization related concepts, the Multilizer technology is introduced to the reader. This gives an overview of the technical solutions, which make it flexible and scalable even when working with multilingual – or multicultural – software.

This chapter gives important information on the following issues:

- Text style and symbol conventions used in this book
- Multilizer Installation notes
- Where to get support.

At the end of this manual, there is a glossary of terms and concepts used in this book.

Multilizer naming conventions



NOTE!

Multilizer 5.1 has different editions: **Multilizer Enterprise**, **Multilizer for Oracle**, **Multilizer for .NET**, **Multilizer for VCL** and **Multilizer for Java** are for developers or project managers. These editions have full functionality to scan, translate and build software and from here on in are referred to as **Multilizer**. The free-of-charge edition for the translator is called **Multilizer Translator Edition**. This edition is capable only for translating projects.

Tutorials for different platforms



NOTE!

This guide contains tutorials for every platform Multilizer 5.1 supports. Your particular product might not contain support for every platform represented in this manual.

Conventions used in this book

The following typographical conventions have been used in this book.

Names of windows, menu options and key buttons are printed in **bold Sans serif**.

Texts for figures and references to chapters and sections in this guide are shown in *italic Sans serif*.

Programming language related items are shown in the following manner.

- Names of components, component properties, procedures, and functions are shown in **bold monospaced font**.
- Code listings and URLs are shown in `monospaced font`.



MORE INFO

The More info symbol is used when there is additional information available either in the appendices of this document, in Multilizer online help or on Multilizer web-pages at: <http://www.multilizer.com>.

**NOTE!**

The note symbol is used in order to give emphasis for certain tasks or issues of big importance in the current topic.

**TIP!**

The text marked with the Tip symbol gives useful hints, which may simplify tasks described in the current chapter.

**WARNING!**

The Warning symbol is used whenever there may be a possibility to lose data or experience other kinds of damage. The normal context for this symbol indicates that you may lose your translation data if you proceed.

**CHAR.SET**

This symbol is used in issues describing different character sets. Character sets are one central issue to be taken into consideration when localizing software.

**NOTE!**

All screen-captured images have been taken when the active language of Multilizer is English.

Installation

Multilizer is installed from CD or directly from the Internet. Refer to the instructions for installing the software.

**NOTE!**

When purchasing Multilizer, you simply enter the licence number to your evaluation version.

**TIP!**

Commercial version users will find minor updates on the Multilizer support pages:

<http://www.multilizer.com/support>

New builds or patches can at all times be downloaded from:

<http://www.multilizer.com/download>

Evaluation versions of the software are available at the Multilizer download page:

<http://www.multilizer.com/download>

**NOTE!**

If you install a setup version that supports component localization, such as Delphi, the setup will automatically install the components into the IDE. If this fails, you have to install the components manually. In this case, consult the software development tool documentation.

Multilizer setup creates the following program group:

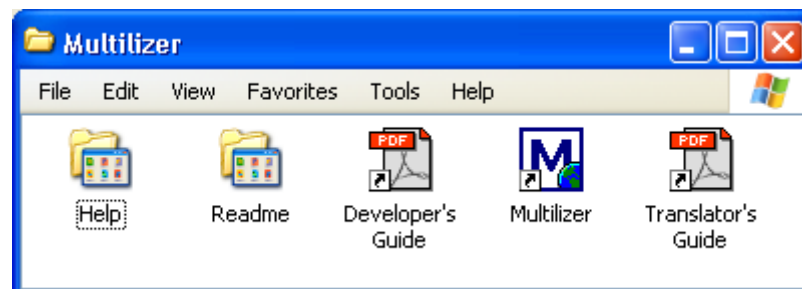


Figure 1 Multilizer program group.

Multilizer icon starts the Multilizer. The other icons are shortcuts to Multilizer documentation. Read the documentation in the following order:

1. *Readme* group contains the readme files.
2. *Developer's Guide* manual (this document)
3. *Help* group contains the platform specific online helps.

Documentation

The online documents included with Multilizer share the same content as the printed Multilizer manual. The following online documents are available after installing Multilizer:

- `multiliz.pdf` is Developer's Guide (this document).
- `translat.pdf` is Translator's Guide.

The printed manual includes both of the documents above.

The best source for technical information is the online help. It includes detailed information of Multilizer's features and usage as well as Multilizer component reference with code samples.

You can also get more information and tips on Multilizer website.

<http://www.multilizer.com/support>

Multilizer website includes latest versions of documentation, Multilizer Knowledge Base search functionality, technical documents and other support material.

Registration

By registering the product you will get technical support as defined in the Software License Agreement.

Although the Multilizer package contains the registration card, we strongly recommend that you register the software at our web site:

<http://www.multilizer.com/support>

This creates you a personal account to our extranet services. Using the account you can download new versions and bug fixes. You can also join a mailing list that keeps you updated about Multilizer and multilingual globalization technology.

You can download the latest Multilizer version at

<http://www.multilizer.com/support>

Technical Support

Support Center

If you have a question about Multilizer, look first at the online help. If you can't find the answer at the online help check it at the support center:

<http://www.multilizer.com/support>

Sometimes our technical support person will ask you to send a sample application. Try to create a simple program that demonstrates your case.

Sending your program source to technical support

If your development tool uses components or beans try not to use any 3rd party components. If they are needed to demonstrate the problem, include the 3rd party components, the application binary (resource based localization) or the source code files (component based localization) and the Multilizer project file (*.mpr) into a single zipped file and email it to our technical support.

Please include the version number of Multilizer, compiler version used and the operating system (version/language). This will greatly speed up solving the problem. Any source codes we obtain through bug reports are handled strictly confidentially.



NOTE!



Part I: Getting Started

This part explains some key tasks about software internationalization, localization and the Multilizer technology that introduces a fast and flexible way to localize your applications.

2

Overview

Internationalization and localization are the key part in the globalization process.

Internationalization

In software internationalization the hard-coded language or country dependent information is removed. In practice this means that no language specific information, currencies, dates, times etc. should be inside the program code. Software internationalization is the first thing to implement in order to make the software adapt to the target country. Depending on the software type and how it has been programmed, this phase may be very time-consuming. The programmer typically does the internationalization.

Localization

Localization means converting a program to a local market. The simplest way is to translate all the strings into the local language. In most cases you also have to make slight modifications to the program to meet the local standards and culture. In addition the software has to be prepared to support the target country's character set and language.

Localization needs both developers that do the engineering and linguists that do the translations.

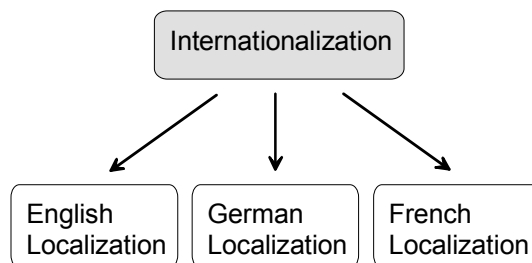


Figure 2 *Internationalization is the first step of localization*

The most common localization approach by far is the process where several localized applications are produced. The process starts with internationalization. The process continues with localization where linguists translate the user interface, manuals, etc. The result is several localized applications. Each localized program can only support one language.

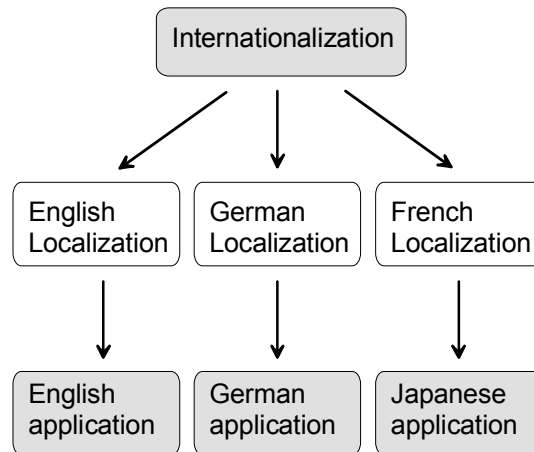


Figure 3 Multiple localized applications support one language each

Some platforms (e.g. Windows) can store resources in several languages. In such a platform it is possible to build a single binary file containing all languages.

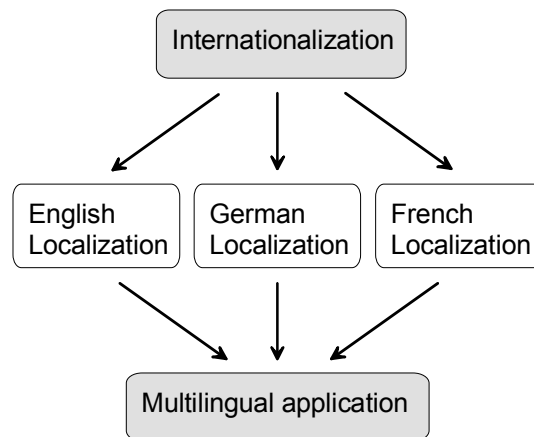


Figure 4 Single multilingual application contains all supported languages

Internationalization is used to produce single worldwide sources. Single worldwide source means that you have only one version of your source code. To localize the application to a new language you do not have to change the source code but the translation of the resource data is enough. You either compile or include new resources for each localized version. The methods for doing internationalization and especially localization can vary a lot. The time spent on these phases may become the most decisive factor in software delivery dates.

Traditional localization techniques may take several months to complete and in the meantime the source should be frozen. Multilizer provides a way that unifies the localization process. No matter what your application is, the localization process is always the same. This reduces the localization costs considerably. Furthermore, Multilizer makes it possible to start the localization process even before the software development process is completed. Software development can continue the same time as the localization takes place. This reduces the time localization process takes considerably.

Multilizer contains the following tools:

Tool	Description
Multilizer	Extracts the strings from the applications or content and provides methods to edit and translate the strings. Windows application (<code>multiliz.exe</code>).
Builder	A command line tool that creates the localized application files and/or the run-time dictionary for the application. Windows command-line application (<code>mlbuild.exe</code>).

	Read Chapter 27 to get more information about Builder.
Components	<p>Platform specific components (e.g. VCL, COM or JavaBeans) that attach the run-time dictionary to the application and make the application multilingual.</p> <p>Read the online helps to get information about Multilizer components.</p> <p>These are optional. They are used only when the component localization type is used.</p>

3

Localization Technologies

Multilizer supports four three of localization technologies. In Multilizer documentation these are referred to as *localization types*. The division is based on the way the project source-code and/or executable is accessed in the localization process:

Localization type	Description
File localization	Translates the data in the file to produce either localized file or a single multilingual file. This type can be divided into the following sub types: <ul style="list-style-type: none"> - Binary localization - Resource localization - Source code localization - Document localization
Component localization	Adds Multilizer components to the application to produce a single multilingual application.
Database localization	Localizes the content of the database. See Database Tutorial chapter



The localization type is based on target platform's localization support and the development tool. In some environments the developer can choose between multiple localization types.

Each localization type has its advantages and disadvantages. The following paragraphs describe these localization types in more detail.

File Localization

In file localization the files are translated. A file is either an application binary file, application resource file, application source code file, or a document file.

Binary Localization

In binary localization the application binary files are translated. An application binary file is either the application file (e.g. EXE), a library file (e.g. DLL) or a component file (e.g. OCX). Multilizer makes binary localization very easy. The following picture describes the binary localization process.

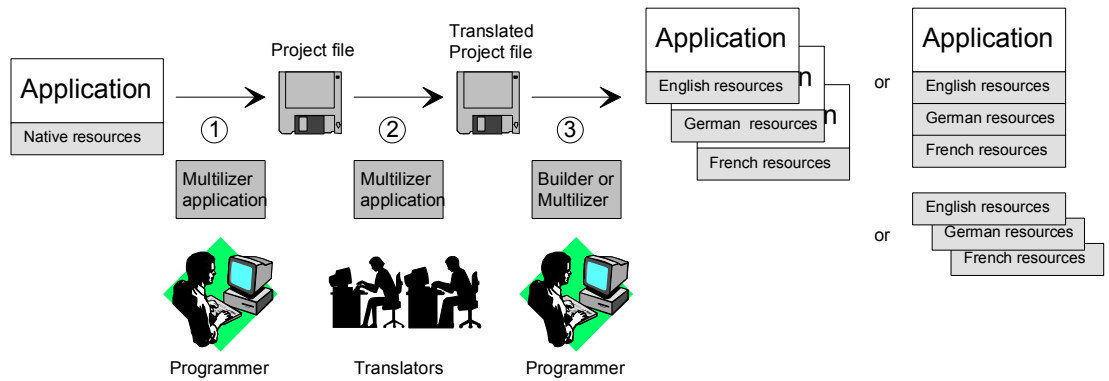


Figure 5 Binary localization process

The programmer uses Multilizer to extract strings from the application binary file (1). Multilizer saves these strings to a project file. The programmer sends the project file to the translators that use Multilizer to translate the project (2). The programmer uses Builder to create the localized versions of the binary file (3). As a result there will be one binary file for each language, one multilingual binary file containing all supported languages, or one resource file (e.g. DLL) for each language.

For example the native language might be English. If you decide to localize the program to German and French you will have three different versions of the program - each supporting one language. On some platforms you can also create a single application containing English, German and French resources in the same binary file.

The following table contains the description of each file that Multilizer generates in the example mentioned above.

File	Description	Resource language(s)
sample.exe	Original application file	English (in most cases)
en\sample.exe	English application file	English
de\sample.exe	German application file	German
fr\sample.exe	French application file	French
en\sample.ENU	English resource DLL	English
de\sample.DEU	German resource DLL	German
fr\sample.FRA	French resource DLL	French
all\sample.exe	Multilingual application file	English, German and French

When deploying the application you can either deploy the localized binary file (e.g. de\sample.exe), the multilingual binary file (all\sample.exe), or the original binary file (e.g. sample.exe) with the localized resource DLL(s) (e.g. de\sample.DEU).

Binary localization can be used with the following applications:

Application type	Multilizer localizes
C++Builder	Form and string resources of a C++Builder application.
Delphi	Form and string resources of a Delphi application.
Palm	Dialog strings, menu strings, and resource strings of a Palm application.
Visual Basic and Embedded Visual Basic	String resources of any Visual Basic or Embedded Visual Basic application.
Visual C++ and Embedded Visual C++	Dialog, menu, string, and accelerator resources of any Visual C++ or Embedded Visual C++ application. This is not Visual C++ specific but any Windows or Windows CE application having standard Windows resources can be localized using this method.

We recommend the use of binary localization. However not all platforms support binary localization. If your development tool is not included in the list above you have to use some other localization. type.

Resource Localization

In resource localization the resource files are translated. Multilizer makes resource localization very easy. The following picture describes the source localization process.

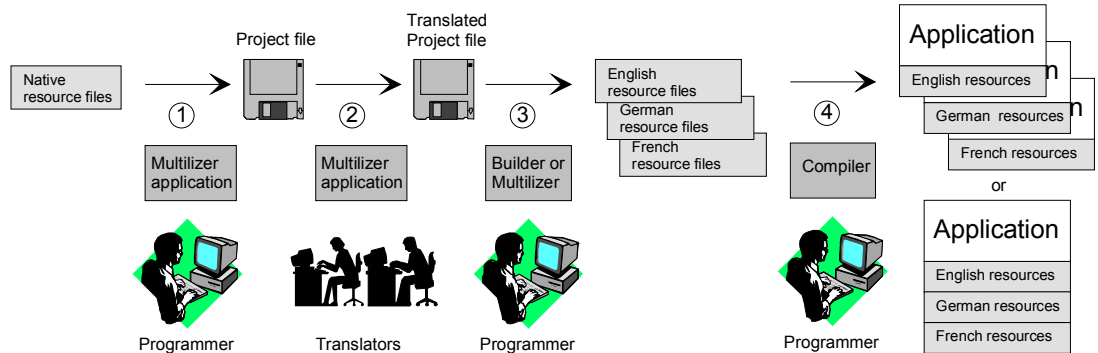


Figure 6 Resource localization process

The programmer uses Multilizer to extract strings from the resource files (1). Multilizer saves these strings to a project file. The programmer sends the project file to the translators that uses Multilizer to translate the project (2). The programmer uses Builder to create localized resource files (3). Finally the programmer compiles/links the application to add the localized resource files. As a result there will be one application for each language or one localized application file containing all supported languages.

We recommend the use of resource localization only if binary localization is not supported on your platform. Resource localization can be used with the following applications:

Application type	Multilizer localizes
Java	Property files of a Java application or applets.
Symbian	Dialog strings, menu strings and the resource strings of an Eikon application.
Visual C++ and Embedded Visual C++	Dialog strings, menu strings, string tables, and accelerator keys of a Visual C++ or Embedded Visual C++ application. This localization type is not Visual C++ specific but any Windows and Windows CE application having standard Windows resources file (.RC) can be localized with this method.
.NET	Dialog strings, menu strings and the resource strings of a .NET application.

Source Code Localization

In source code localization the source code files are translated. Multilizer makes source code localization very easy. The following picture describes the source localization process.

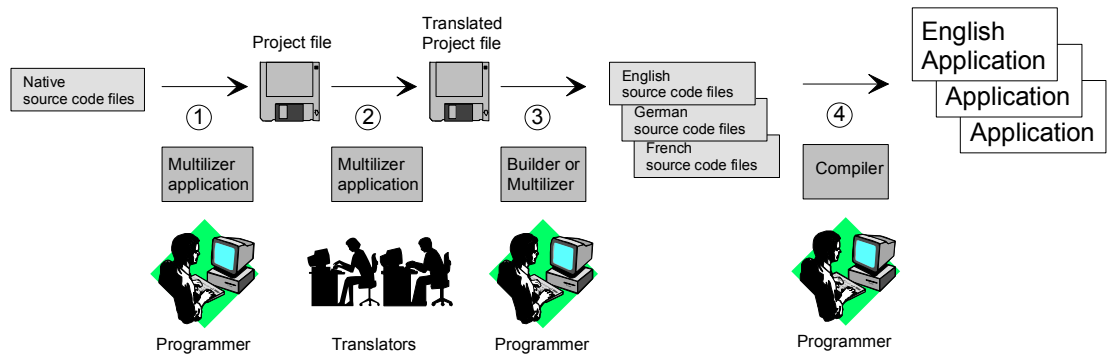


Figure 7 Source code localization process

The programmer uses Multilizer to extract strings from the source code files (1). Multilizer saves these strings to a project file. The programmer sends the project file to the translators that uses Multilizer to translate the project (2). The programmer uses Builder to create localized source code files (3). Finally the programmer compiles the application with the localized source code files. As a result there will be one application for each language.

We recommend the use of source code localization only if binary localization or resource localization is not supported on for your platform. Source code localization can be used with the following applications:

Application type	Multilizer localizes
Visual Basic and Embedded Visual Basic	Form strings and the resource strings of a Visual Basic or Embedded Visual Basic application.

Document Localization

In document localization the document are translated. A document file is any file containing data such as key file (.TXT), XML file (.XML), ini file (.INI), wap file (.WML), StreamServe file (.SLS), etc. Multilizer makes document localization very easy. The following picture describes the document localization process.

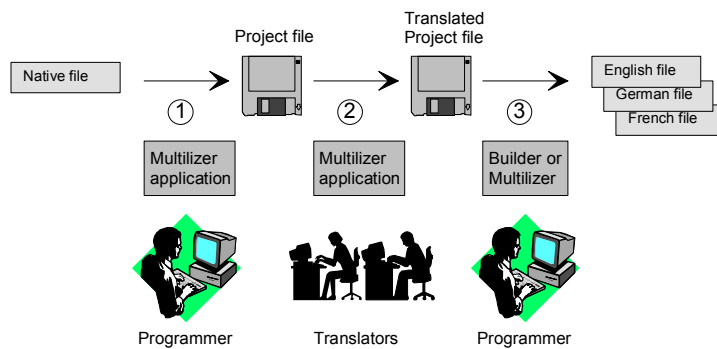


Figure 8 Document localization process

The programmer uses Multilizer to extract strings from the document files (1). Multilizer saves these strings to a project file. The programmer sends the project file to the translators that uses Multilizer to translate the project (2). The programmer uses Builder to create localized document files (3). As a result there will be one document file for each language.

Document localization can be used with the following document types:

Application type	Multilizer localizes
INI files	Selected keys
Key files	The value of each key
StreamServe SLS files	Strings

WAP files	WML tag values of WAP pages
XML files	Selected tags

Component Localization

Component based localization uses Multilizer components to make the application localized. This goes beyond traditional localization – it makes it possible to create multilingual applications. A single multilingual application supports multiple languages and the user can switch the language on the fly.

The programmer uses Multilizer to extract strings from the application (1). Multilizer saves these strings to a project file. The programmer sends the project file to translators that use Multilizer to translate the project (2). The programmer uses Multilizer components to add dynamic language support to the application (3). As a result there will be one single application supporting any number of languages.

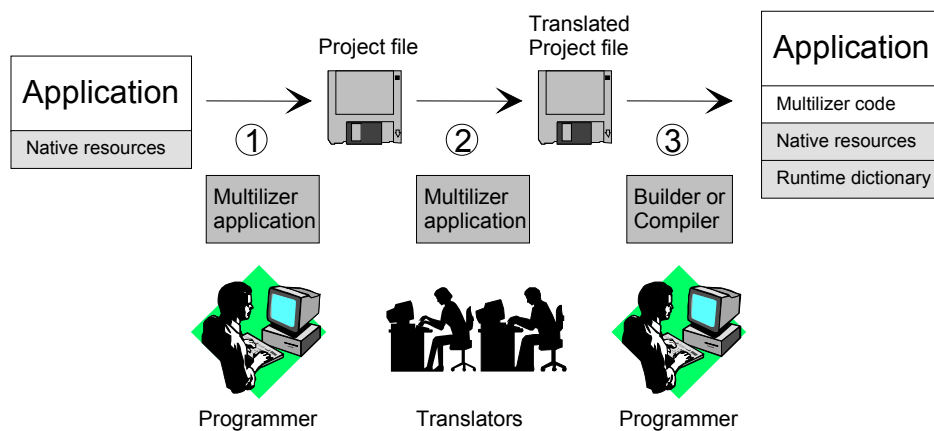


Figure 9 Component based localization process

Component- localization can only be used with a software development tool supporting components:

Development tool	Description
Delphi	Multilizer plus Multilizer components make forms and message strings of Delphi applications multilingual
C++Builder	Multilizer plus Multilizer components make forms and message strings of C++Builder applications multilingual
Java and Visual J++	Multilizer plus Multilizer beans make frames and message strings of Java applications and applets multilingual

Database Localization

See the *Database Tutorial* chapter.

Application type	Multilizer localizes
Database	Strings from selected database field(s).

Comparing Different Application Localization Technologies

The following table compares different application localization types.

	Binary localization	Source localization	Component localization
--	---------------------	---------------------	------------------------

Pros	+ Source codes are not needed	+ Source code doesn't need to be changed	+ Run-time language switch
	+ No size or speed overhead	+ No size or speed overhead	+ One application supports any number of languages
	+ No run-time code is added	+ No run-time code is added	+ Automatic bi-directional support
	+ No recompiling needed	+ No resourcing needed	+ No resourcing needed
Cons	- Resourcing needed	- Recompiling needed	- Source code need to be changed
	- No automatic bi-directional support except in Delphi and C++Builder	- No automatic bi-directional support	- Run-time code may cause troubles with 3 rd party components
	- No runtime language switch except in Delphi and C++Builder	- No runtime language switch	- Size and speed overhead: application gets a bit bigger and slower

In general we recommend using binary localization. If this is not available on your platform we recommend using source localization. Use component localization only if your application must have the run-time language switch. The following table contains supported localization types for each platform.

Application type	Binary	Resource	Source	Component
.NET	-	Yes	-	-
C++Builder	Yes	-	-	Yes
Delphi	Yes	-	-	Yes
Java	-	Yes	-	Yes
J2ME	-	Yes	-	-
Palm	Yes	-	-	-
Symbian	-	Yes	-	-
Visual Basic	Yes	-	Yes	(Yes) *
Visual C++	Yes	Yes	-	-
Visual J++	-	Yes	-	Yes

The recommended technology is highlighted with bold typeface.

*) Multilizer 4.x and 5.0 contained Multilizer components for Visual Basic. The current version does not contain those components. However Multilizer application can still create projects having Visual Basic targets using the Multilizer components.

Unique architecture

Multilizer separates the localized data from the source and stores it in a platform and compiler independent format. This *Multilizer project file* format is a Unicode based XML file. The separation makes it possible to localize all kinds of applications and documents in the same way. It is also possible to reshape the visual parts of the application without losing any translations, and add new languages without using development tools.

This type of architecture gives a very practical approach to localization:

- Human language tasks are executed with the Multilizer. The localization method is the same no matter what kind of development tool is used. Because the human

translator needs to learn only one tool, Multilizer, the localization process gets faster and less expensive.

- Software development tasks are done either by Multilizer alone (binary and source localization) or by including the Multilizer components in the project (component localization).

Multilizer and optional components are included in every Multilizer setup package.

Multilizer provides timesaving features for working with human languages: multiple wizards, re-use of translations, built-in quality assurance indicators such as statistical information, local and shared translation memory, and enterprise-wide leveraging capabilities.

Multilizer components provide the developers with an easy-to-use, yet powerful, interface for software development. Developing the RAD way, localized software's behavior is controlled through component properties, events and methods.

With Multilizer, you can continue developing the software even when localization takes place.

It is important to understand that Multilizer is not a tool that automatically translates words or phrases into another language. Instead it provides a mechanism that uses precompiled translator tables, dictionaries, to make the program multilingual. It is your (or your translator's) job to translate the strings in your program. Multilizer makes this translation as easy as possible with the usage of different kind of glossaries and a translation memory.

With Multilizer you can create a *single worldwide application*. Ultimately, the support for languages depends on the OS support for language specific features.

Translation memory

Translation memory stores every single translation you have done so that they can be used in future projects or in automatic translation. You can also import existing translation memories (e.g. TMX) and glossaries (e.g. Microsoft glossaries, Trados®MultiTerm™) to the translation memory. You can exchange translation memories between different computers or you can share the same translation memory between multiple users in the Intranet or using a database server, over the internet.

The translation memory is stored to a database. Multilizer can directly use most commercial databases including Oracle®8i/9i, Microsoft® SQL Server, Interbase, MySQL. (→ Translation Memory on database server, p. 213)

4

Platforms and editions

The first part of this chapter describes software development tools and operating systems Multilizer supports. The second part describes Multilizer setup files.

Supported tools, operating systems and documents

Multilizer supports all major desktop and mobile operating systems and software development tools.

.NET and Visual Studio .NET

.NET applications use the standard resource format of .NET. Multilizer supports both Visual Studio .NET project file localization and the .NET resource file localization.

See *.NET Tutorial* chapter for detailed information on .NET localization.

Databases

Most databases contain information that needs to be localized. For example a database can contain product information in English. This information needs to be translated and stored to a database.

See *Database Tutorial* chapter for detailed information on content(data) localization.

Delphi and C++Builder

Delphi and C++Builder are object and component oriented software development tools that use Object Pascal or C++ software development language and VCL component library. Multilizer supports both binary and component localization for Delphi.

See *VCL Tutorial* chapter for detailed information on Delphi/C++Builder localization.

J2ME

J2ME (Java Micro Edition) does not contain resource bundle classes. To support this localization, Multilizer contains a lightweight property file class for J2ME (`multilizer.microedition.Properties.java`).

See *Java Micro Edition Tutorial* chapter for detailed information on J2ME localization.

Java

Java contains resource bundle classes. Multilizer supports these. In addition Multilizer includes components that make localization even easier.

See *Java Tutorial* chapter for detailed information on Java localization.

Oracle Forms

Oracle Forms contains user interface elements. Multilizer localizes form files.

See *Oracle Forms Tutorial* chapter for detailed information on Oracle Forms localization.

Palm

Handheld computers and 3rd generation cell phones use Palm Operating System. Multilizer localizes compiled Palm applications (.prc).

See *Palm Tutorial* chapter for detailed information on Palm localization.

StreamServe

StreamServe's report string files (.sls) files contain string data that needs to be localized. Multilizer localizes the strings if the SLS files.

See *StreamServe Tutorial* chapter for detailed information on StreamServe localization.

Symbian

The handheld computers and 3rd generation cell phones use Symbian operating system. Multilizer localizes Symbian resource files.

See *Symbian Tutorial* chapter for detailed information on Symbian localization.

Visual Basic and Embedded Visual Basic

Visual Basic is a component oriented software development tool that use Basic software development language and COM components. Multilizer supports binary and source localization for Visual Basic (Windows) and Embedded Visual Basic (Windows CE).

See *Visual Basic Tutorial* chapter for detailed information on Visual Basic localization.

Visual C++ and Embedded Visual C++

Visual C++ applications use the standard resource format of Windows and Windows CE. Multilizer supports both binary and source localization of resources.

See *Visual C++ Tutorial* chapter for detailed information on Visual C++ localization.

Visual J++

Visual J++ is a component oriented software development tool that uses Java programming language and WFC components. Multilizer supports component localization for Visual J++.

See *Visual J++ Tutorial* chapter for detailed information on Visual J++ localization.

WAP

WAP applications are web applications targeted for mobile clients. A WAP application contains WML decks and WML scripts. Multilizer localizes deck files.

See *WAP Tutorial* chapter for detailed information on WAP localization.

XML

Most XML files contain information that needs to be localized. For example, an XML file can contain product information in English. This information needs to be translated. Multilizer also translates text and INI files.

See *XML Tutorial* chapter for detailed information on content(data) localization.

Multilizer Setups

Multilizer setups are available on CD or can be downloaded from the Internet from <http://www.multilizer.com/download>

All setups require a serial number if used for commercial purposes. You get the serial number by purchasing Multilizer.

If you install the setup without serial number, an evaluation version will be installed.



Setups install the following files:

- **Multilizer tool** (`multiliz.exe`). This is the Multilizer application.
- **Builder tool** (`mlbuild.exe`). This is the Multilizer command line tool.
- **Developer's Guide manual** (`multiliz.pdf`). This is this manual.
- **Translator's Guide manual** (`translat.pdf`).
- **Demo projects**. These tool-specific demo applications are included in Multilizer setups.
- **Components**. Components and component online documentation are included in Multilizer setups that target component-based software development tools (e.g. Delphi, C++Builder, VB, Java).

Multilizer

Run `ML51.exe` to install Multilizer.

Multilizer Translator Edition

Multilizer Translator Edition is installed in the following way:

- **Exchange Package**. A developer can use Multilizer to create an Exchange Package that includes Multilizer tool and the translatable data. This package is self-extracting and installs automatically in the translator's computer.
- **Internet**. The translator can download Multilizer Translator Edition setup file (`ML51TE.exe`) from Multilizer website and install it.
- **CD**. The translator can install Multilizer Translator Edition setup file (`ML51TE.exe`) from Multilizer CD and install it. The CD with accompanying printed documentation can be ordered from Multilizer website. This setup requires serial number for complete installation.

5

Coping with different languages

One main task in software localization is to make the software work in the language and character set of the target country. Originally, computers were designed to work with character sets including only those characters needed in English. When computers became commercially available everywhere in the world, there was a need to include the target country's characters into the character set as well.

To be able to handle a specific language in the computer, it must be possible to handle its character set. There must be methods for inputting, storing and outputting characters. Thus, localizing applications involves

- Processing character sets and
- Accommodation of the application's I/O methods for the current language.

The following chapters introduce different types of scripts. Later, the most important of these will be discussed in the context of today's information technology.

Character sets

All languages in the world can be divided into three basic groups, depending on the type of character set they use. The groups are:

- Single byte character sets.
- Multi or double byte character sets.
- Bi-directional character sets.

This division is a rather technical one, but as a matter of fact, it reflects three major cultures as well. The groups mentioned above could be rewritten in the following way:

- European character sets.
- Far Eastern character sets.
- Middle Eastern character sets.

Single byte character sets

Single byte character sets (SBCS), sometimes called single byte left to right, contain a maximum of 256 characters. Each character is stored in one byte. The text is written from left to right. Latin (e.g. English, German, French, Spanish, Finnish, Swedish), Greek and Cyrillic (e.g. Russian, Ukrainian) character sets are single byte.

These character sets evolved in Europe, and they started from the old Hellenistic culture. It is very possible that these scripts came into Europe from Mesopotamia via the Near East, though.

Greek

The ancient Greek character set was derived from Linear B, which was similar in structure to Japanese. Later, due to the Dorian invasions, a script based on the North Semitic model was adapted. As in modern Middle Eastern scripts, this was written from right to left.

Later, such inventions as the vowels a, e, i, o, u made it the most successful and the most practically useful of the world's scripts at that time.

Modern Greek script has undergone only a few changes since the classical Greek period. Most of the changes are phonological.

Latin

Old Greek script expanded over Southern Italy and came into Roman hands. The alphabet was simplified. Through the influence of the Roman Empire, the Latin alphabet came into use in the whole Western civilization. On the basis of Latin script, the modern European character sets developed.

Thus, these alphabets are very denominative for European culture(s) and for those countries where the Europeans established their colonies. The strong cultural expansion from the XVth century on has exported – especially the Latin alphabet – all over the world. In South and North America, Africa and Australia, the Latin alphabet is almost the only one in use.

All of these character sets have implemented accent marks to denote special sounds, i.e., the characters are based on a base character and the phonetic difference is marked with an accent. In addition, some additional characters have been taken in use.

For example, in German, 'ß' is used to denote 'ss' when it is in a certain place. In addition, in African languages characters like |, ||, ‡ and ≠ are used to mark non-pulmonic sounds (clicks) not used in European languages.

From the Middle Ages some ligatures survived: French œ, Danish æ. The best known is the ampersand '&', a ligature of the Latin word 'et' (*and*).

In addition, languages using different character sets and scripts are often transliterated in literature and schoolbooks into the Latin character set. For transliteration purposes, additional diacritics are attached to base characters.

Multi or double byte character sets

Multi byte character sets (MBCS), sometimes called double byte (DBCS) or Far East, contain more than 256 characters or idioms. Each character is stored either in one byte or two bytes. The text is written mainly from left to right top to bottom but sometimes from top to bottom left to right. Chinese, Japanese and Korean character sets are multi byte.

Bi-directional character sets

The bi-directional character set (BiDi), sometimes called single byte bi-directional, contains a maximum of 256 characters. Each character is stored in one byte. The text is written mainly from right to left but sometimes from left to right. Arabic and Hebrew character sets are bi-directional.

The ancient Arabic script was a derivative of the Nabataean consonantal script, which was used two thousand years ago. Later, Mesopotamian Kufic scripts influenced it. From the eleventh century onwards, a flowing cursive style was developed, and it became the Arabic script commonly used. This script underlies most contemporary type-fonts.

Arabic script is used for a number of important languages: Persian, Urdu, Pashto, Baluchi, Kurdish, Lahnda, Kashmiri, Sindhi and Uighur.

Since the phonological inventories between these languages may differ a lot from those of Arabic, the script has had to be augmented and adapted to meet the new demands made upon it. In some languages, such as Sindhi, certain Arabic letters are adapted to denote multiple sounds. On the other hand, in some languages there are a lot of redundant letters – in Persian there are four Arabic letters pronounced exactly in the same way.

Arabic script has been used in many other languages. However, many of them have abandoned Arabic script for Latin. Among them there are languages like Indonesian (Malay), Hausa, Somali, Sudanese, Swahili and Turkish. Partially this is explained by the last 400 years' aggressive expansion of the European cultures. In addition Turks wanted to modernize the country at the beginning of the 20th century and they adopted the Latin alphabet. Several Caucasian languages, e.g. Chechen, Kabardian, Lak, Avar, Lezgi used

the Latin alphabet for a while. Due to the expansion of Russian power, the Cyrillic alphabet was implemented.

Arabic script is used nowadays in a widespread area from the Atlantic coast in Morocco to India. It is very probable that these areas remain as users of this script. This area has been politically quite unstable, but economical growth is expected, thus bringing information technology to these countries. This will open a need for supporting Arabic script in software.

Unicode

Single-byte and double-byte character set trouble

Single-byte character sets are able to support a maximum of 256 different characters. Originally this was sufficient to cover English and some other European languages.

However, there are many languages using Latin characters with diacritics. Because there was no place for them, different code pages were implemented. Therefore, e.g., Romanian uses a different code page than French. To make the software run in both languages, the code page had to be changed, which often meant rebooting computer or installing a specific OS.

For Far Eastern languages Double-byte character sets were implemented. The scripts of these languages contained more than 256 ideographs. Thus the characters were encoded in two bytes.

These single or double-byte character sets had the problem that one code (such as Hex 67) could hold different characters, according to the code page used. Therefore Unicode was introduced.

Universal character encoding system

Unicode is a fixed-width, 16-bit worldwide character encoding system developed by the Unicode Consortium and endorsed by Window NT. It is a "universal" character set of approximately 35,000 characters encompassing all major character set standards.

In Unicode, each character has a unique code.

The Unicode Consortium, a nonprofit computer industry organization, continues to maintain and promote the system.

You can get more info on Unicode at: <http://www.unicode.org>



Text Input

Western languages

Western alphabets are based on characters. There are normally a very restricted number of characters with which all the words in a western language can be formed. Therefore most characters can be input with one character stroke on the keyboard.

Different Western languages use keyboard layouts that differ slightly from each other. This is due to the input of some language specific characters (Cf. *previous chapter*).

Multilizer is able to change the keyboard layout to match the current language being edited.

In addition Multilizer helps in entering Latin characters with diacritics: you can use existing characters or define your own compose character sequences for entering accented characters. This makes the working easier, e.g., with a US keyboard.

Far Eastern languages

Far Eastern languages need a specific way for inputting ideographs. There might be thousands of characters that should be input. For that reason, Far Eastern Windows language editions ship with Input Method Editors (IME).

Using the IME, users can compose each character in one of several ways: by radical, by phonetic representation or by the character's numeric code-page index.

Multilizer uses the IME provided by Far Eastern Windows language editions for inputting those languages.

Middle Eastern languages

Middle Eastern languages have a restricted number of characters in their alphabets, like Western languages. The problem of inputting characters in these languages lies in the following reasons.

- Text is written from right to left, numbers from left to right
- Arabic characters obtain different forms depending on where they are located in the word (initial character, in the middle of the word, final character or isolated character).

Arabic or Hebrew language editions of Windows are needed for inputting Middle Eastern characters in Multilizer.

For more information on editing a specific language in Multilizer, cf. Multilizer documentation.



MORE INFO

Country specific items

Besides making software work in different languages, software must also be fine-tuned to work with country specific standards. Country specific standards are later referred as locale data.

In fact locale information tells what language is spoken in the specified locale. Therefore the language can be considered as a subset of the locale.

In Multilizer a specific locale is referred to using the language and the country's name in parenthesis. Thus, selecting a locale like e.g., French (Canada), means that the country is Canada and the language is French.



NOTE!

More about localization

The following list contains various sources on where to get more information about localization:

- Developing International Software for Windows 95 and Windows NT, Nadine Kano, Microsoft Press, 1995. This is an excellent book about developing international software.

You can find this book on MSDN, too.

- A Practical Guide to Localization, Bert Esselink, John Benjamins Publishing Company, 2000.
- WIN32 API documentation
- <http://www.xerox-emea.com/globaldesign/>



Part II: Tutorials

This part contains the platform specific tutorials that go through the localization process of a sample application.

6

.NET

This tutorial discusses localization of software in .NET. Sample codes are written for C# and Visual Basic .NET.

Before localizing any software in .NET, you should be familiar with the Microsoft localization model in .NET environment. Because Multilizer supports this model, the most important parts of it are discussed in the Internationalization part of this tutorial.

Visual Studio .NET Project File Localization

Visual Studio .NET projects contain the resource data in the resource files (e.g. `.resx` or `.txt`). Multilizer creates the localized resource files from the original files. The following picture describes the localization process.

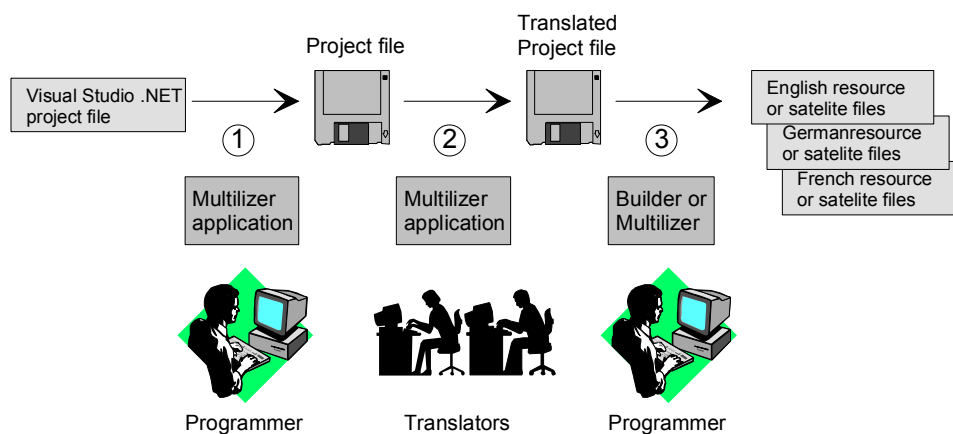


Figure 10 Visual Studio .NET project file localization process

The programmer uses Multilizer to extract strings from the original resource file(s) belonging to the project (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource files and/or the localized satellite assembly files (3). As a result there will be one resource file or one satellite assembly file for each localized language.

Multilizer creates subdirectories under original project file folder containing the localized satellite assembly files. I.e. there might be subfolders called

`..\en\MySample.resources.dll` (English) and

`..\fi\MySample.resources.dll` (Finnish).

When deploying the application you can either deploy the original application file with the localized satellite assembly (e.g. `de\MySample.resources.dll`), or you can link the localized resource file (e.g. `MySample.de.resources`) to the original application file.

The following example figure shows the files that Multilizer uses on the Visual Studio .NET project file localization process.



Figure 11 The files of the Visual Studio .NET project file localization process

Using this localization process Multilizer globalizes the form and strings resources of any .NET application written in C# or Visual Basic.

.NET Resource File Localization

If you do not use Visual Studio .NET but the command line tools of the .NET SDK or some other .NET development tool, you have to use the resource file localization process. The following picture describes the localization process.

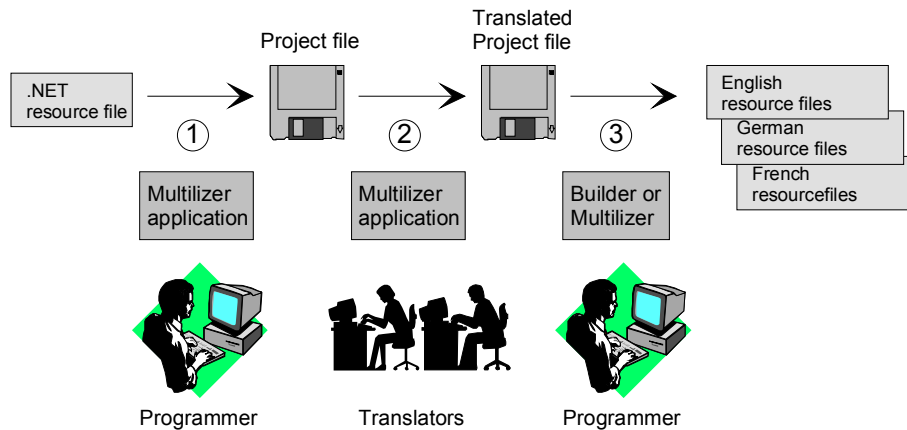


Figure 12 .NET resource file localization process

The programmer uses Multilizer to extract strings from the original resource file(s) (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource files (3). As a result there will be one resource file for each localized language.

When deploying the application you can either link the localized resource file (e.g. MySample.de.resources) to the original application file or create a satellite assembly file containing the localized resource file(s).

The following example figure shows the files that Multilizer uses on the .NET resource file localization process.



Figure 13 The files of the .NET resource file localization process

Using this localization process Multilizer globalizes all strings found from the text resource files (.txt) and from the .NET resource files (.resx).

English Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize.

<mldir>\NET\Samples\Tutorial\cs\Dcalc.csproj (C#) and

<mldir>\NET\Samples\Tutorial\vb\Dcalc.vbproj (Visual Basic) contain the project file of the Dcalc sample application. Compile and run the application. By default, Dcalc uses English language.

The application should look like this:

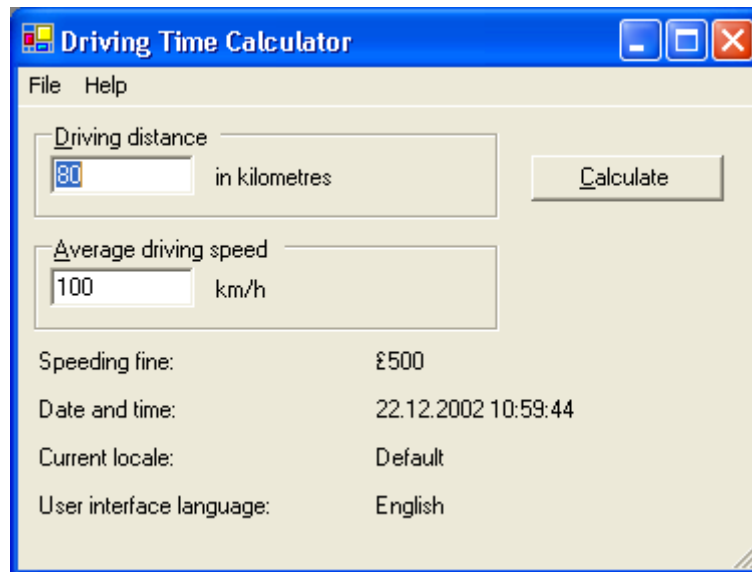


Figure 14 Dcalc application using an English user interface

This version of Dcalc is not internationalized and many of the locale-dependent features are hard-coded. For example the currency symbol is always pounds, distances are given in kilometers and speed in kilometers per hours.

Internationalization

Microsoft .NET localization model is based on localizing resources. This means that before localizing the software, all culture (locale) dependent data must be separated from the code and put in resource files. This is called internationalization. Internationalization tasks are discussed more in detail in chapter Overview.

Internationalization of forms

Internationalization of forms is easy, because .NET generates automatically the source code for them. To internationalize a form, you just need to set the *Localizable* property of the form to *true*. .NET will automatically create a resource file (*.resx*) for the forms elements and create the code that references to it.

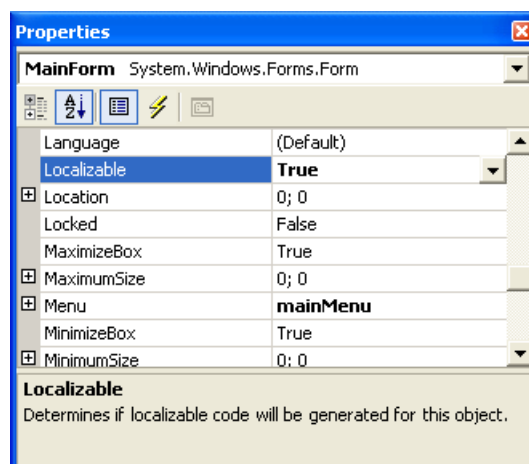


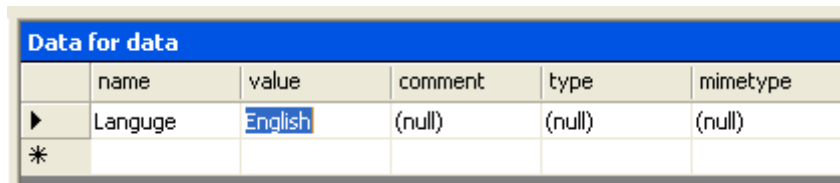
Figure 15 To localize the form set the *Localizable* property to *true*.

Internationalization of code

If there are hard-coded strings in the user-written part of code, you have to do the internationalization manually. You will need to create a resource file for strings, and call the internationalized strings from your code.

The first step is to create a resource file for the string. Choose Project | Add New Item from the Visual Studio .NET. From the Template list select Assembly Resource File. Rename the file to `Resource.resx`. Finally press the Open button to create the file. Visual Studio .NET create an empty resource file.

Let's add a string to the resource file. Double click `Resource.resx` from the Solution Explorer tree. This opens the resource table editor. Type "Language" to the first row of the name column and "English" to the value column. This adds one row the file where "Language" is the key and "English" is the translation.



Data for data					
	name	value	comment	type	mimetype
▶	Language	English	(null)	(null)	(null)
*					

Figure 16 Resource file after adding the first item.

During this internationalization process we are going to add several new items to the resource file.

To access the resource file during the run time we have to add a Resource Manager object to the application. Add the following lines to the MainForm class.

C#

```
public class MainForm : System.Windows.Forms.Form
{
    private ResourceManager rm;
    ...
}

...

private void MainForm_Load(object sender, System.EventArgs e)
{
    rm = new ResourceManager("Dcalc.Resource", this.GetType().Assembly);
    ...
}
```

Visual Basic

```
Public Class MainForm
    Inherits System.Windows.Forms.Form

    Dim rm As New Resources.ResourceManager("Dcalc.Resource",
    GetType(mainform).Assembly)
    ...
End Class
```

The `rm` object is used to get translation from the resource file.

Open the Click event of the `aboutMenu`. It contains two hard coded strings. To remove the hard coding we have to add the string to the resource file and replace the direct access to the string by the access to the resource file. Add both strings to the resource file and wrap them by the `rm.GetString` method.

C#

```
private void aboutMenu_Click(object sender, System.EventArgs e)
{
    MessageBox.Show(
        rm.GetString("Dcalc is a multilingual application that calculates the
average driving time"),
        rm.GetString("About Dcalc"),
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

Visual Basic

```
Private Sub AboutMenu_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles AboutMenu.Click
    MsgBox(rm.GetString("Dcalc is a multilingual application that
calculates the average driving time"), vbOKOnly, rm.GetString("About
Dcalc"))
End Sub
```

If you look at the calculate event you will see that the following lines are used to format the message string that show the driving time to the user.

C#

```
MessageBox.Show(
    "The average driving time is " + Convert.ToString(hours) + " hours and
" +
    Convert.ToString(minutes) + " minutes.",
    "Driving time",
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
```

Visual Basic

```
MsgBox("The average driving time is " + Convert.ToString(hours) + " hours
and " + Convert.ToString(minutes) + " minutes", vbOKOnly, "Driving time")
```

The above code contains both hard coded string and bad design. First of all the message is split into several string that do not contain a real sentence. That's why it is hard to translate them. Also the code assumes that the word order is text plus hours plus some other text plus minutes plus some other text. This works with English but may not work with some other language. That's why we have to use the Format method of the String class. It uses a message pattern and variables.

C#

```
MessageBox.Show(
    String.Format(
        rm.GetString("The average driving time is {0} hours and {1}
minutes."),
        Convert.ToString(hours),
        Convert.ToString(minutes)),
    rm.GetString("Driving time"),
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
```

Visual Basic

```
MsgBox(String.Format(rm.GetString("The average driving time is {0} hours
and {1} minutes"), hours, minutes), vbOKOnly, rm.GetString("Driving
time"))
```

The calculate event shows a message if the distance value is invalid. It also contains hard coded string and a dynamic message.

C#

```
MessageBox.Show(
    distanceText.Text + " is not a valid distance!",
    "Error",
    MessageBoxButtons.OK,
    MessageBoxIcon.Error);
```

Visual Basic

```
MsgBox(distanceText.Text + " is not a valid distance!", vbOKOnly,
"Error")
```

Use the same approach as with the result message.

C#

```
MessageBox.Show(
    String.Format(
        rm.GetString("{0} is not a valid distance!"),
        distanceText.Text),
    rm.GetString("Error"),
    MessageBoxButtons.OK,
    MessageBoxIcon.Error);
```

Visual Basic

```
MsgBox(String.Format(rm.GetString("{0} is not a valid distance!"),
DistanceText.Text), vbOKOnly, rm.GetString("Error"))
```

Resource the message box for the invalid speed in the same way.

The Load event initializes some user interface items such as the speeding file label and the current time label.

C#

```
private void MainForm_Load(object sender, System.EventArgs e)
{
    speedingFine.Text = "£500";
    currentTime.Text = DateTime.Now.ToString();
}
```

Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    speedingFine.Text = "£500"
    currentTime.Text = DateTime.Now.ToString()
End Sub
```

The current time is properly formatted. The ToString method converts the time to a string using the default formatting rules of the current culture. However the fine is hard code to £500. It can be fixed using the int.ToString method.

In US miles are used instead of kilometers. The code below checks if the current culture is English (US). If it is the labels and initial values are set to US system. Otherwise the metric system is used.

C#

```
private void MainForm_Load(object sender, System.EventArgs e)
{
    rm = new ResourceManager("Dcalc.Resource", this.GetType().Assembly);

    int fine = 500;
    speedingFine.Text = fine.ToString("C");
    currentTime.Text = DateTime.Now.ToString();

    currentLocale.Text = CultureInfo.CurrentCulture.NativeName;
    currentLanguage.Text = rm.GetString("Language");

    if (Application.CurrentCulture.LCID == 0x0409)
    {
        distanceLabel.Text = rm.GetString("in miles");
        speedLabel.Text = rm.GetString("mph");
        distanceText.Text = "100";
        speedText.Text = "55";
    }
    else
    {
        distanceLabel.Text = rm.GetString("in kilometres");
        speedLabel.Text = rm.GetString("km/h");
        distanceText.Text = "120";
        speedText.Text = "100";
    }
}
```

Visual Basic

```
Private Sub MainForm_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    SpeedingFine.Text = FormatCurrency(500)
    CurrentTime.Text = FormatDateTime(Now)

    CurrentLocale.Text = Application.CurrentCulture.NativeName()
    CurrentLanguage.Text = rm.GetString("Language")

    If Application.CurrentCulture.LCID = &H409 Then
        DistanceLabel.Text = rm.GetString("in miles")
        SpeedLabel.Text = rm.GetString("mph")

        DistanceText.Text = "100"
```



```

        SpeedText.Text = "55"
    Else
        DistanceLabel.Text = rm.GetString("in kilometres")
        SpeedLabel.Text = rm.GetString("km/h")

        DistanceText.Text = "120"
        SpeedText.Text = "100"
    End If
End Sub

```

Now we have fully internationalized the source code. Make sure that you added every single string inside the `rm.GetString` method to the resource file.

In order to test different locale we add one command line parameter to the application. The parameter is the culture id that specifies the culture that the application uses.

C#

```

static void Main(string[] args)
{
    if (args.Length > 0)
    {
        CultureInfo ci = new CultureInfo(args[0]);

        Thread.CurrentThread.CurrentUICulture = ci;

        if (!ci.IsNeutralCulture)
            Thread.CurrentThread.CurrentCulture = ci;
    }

    Application.Run(new MainForm());
}

```

Visual Basic

```

Public Sub New(ByVal culture As String)
    If culture <> "" Then
        Try
            Thread.CurrentThread.CurrentUICulture = New CultureInfo(culture)
        Catch e As ArgumentException
            MessageBox.Show(Me, e.Message, "Bad command-line argument")
        End Try
    End If

    InitializeComponent()
End Sub

<System.STAThreadAttribute()> _
Public Shared Sub Main()
    Dim args() As String = System.Environment.GetCommandLineArgs()
    Dim strCulture As String = ""
    If args.Length = 2 Then
        strCulture = args(1)
    End If

    Application.Run(New MainForm(strCulture))
End Sub

```

Although not related to internationalization, .NET support for Unicode® makes localization. There are no more code-page related issues, such as non-readable characters appearing in the software.

Creating a Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

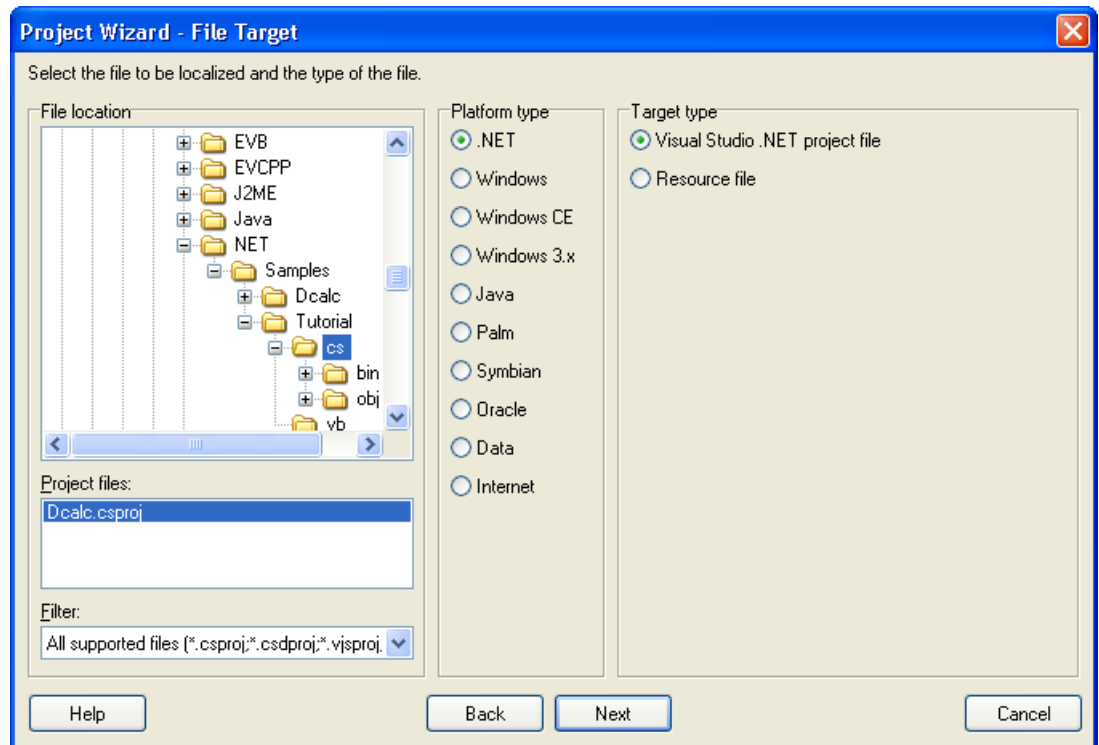


Figure 17 The Select Target sheet is used to specify the application file to be localized.

This sheet specifies the location of your application. Choose the <mldir>\net\Samples\Tutorial\cs (C#) or <mldir>\net\Samples\Tutorial\vb (Visual Basic) subfolder. Project Wizard detects the platform and target types. The Platform type should be *.NET* and the Target type should be *Visual Studio .NET project file*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the >> button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

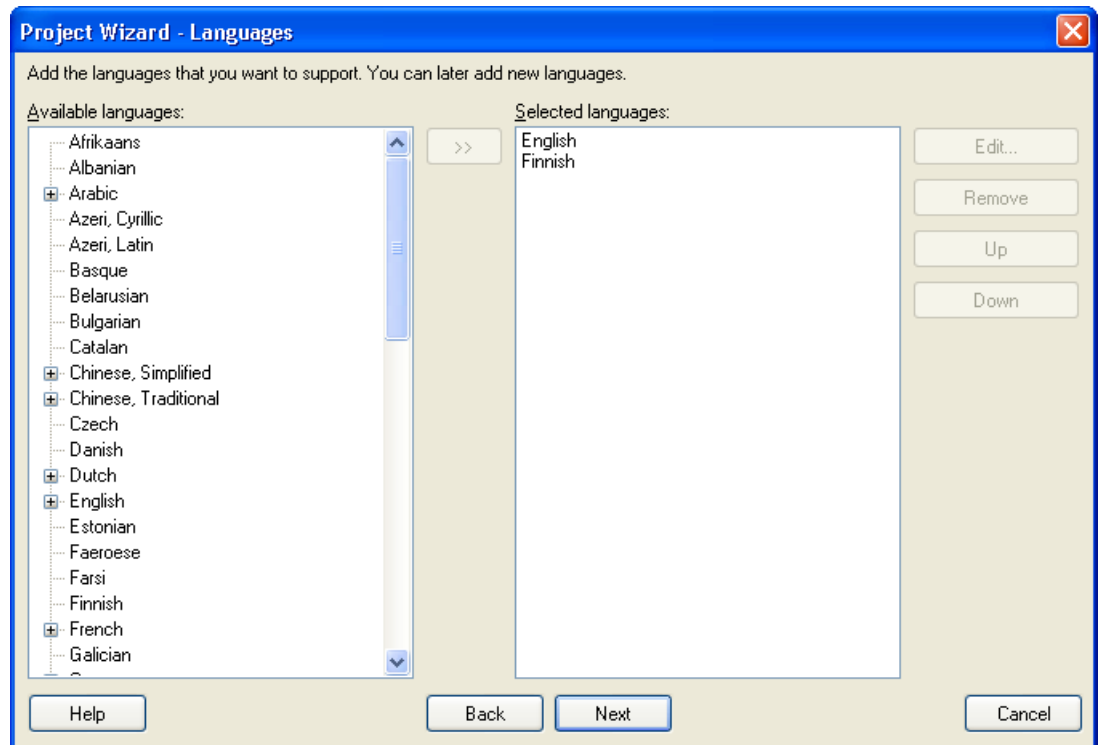


Figure 18 English and Finnish added to the project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

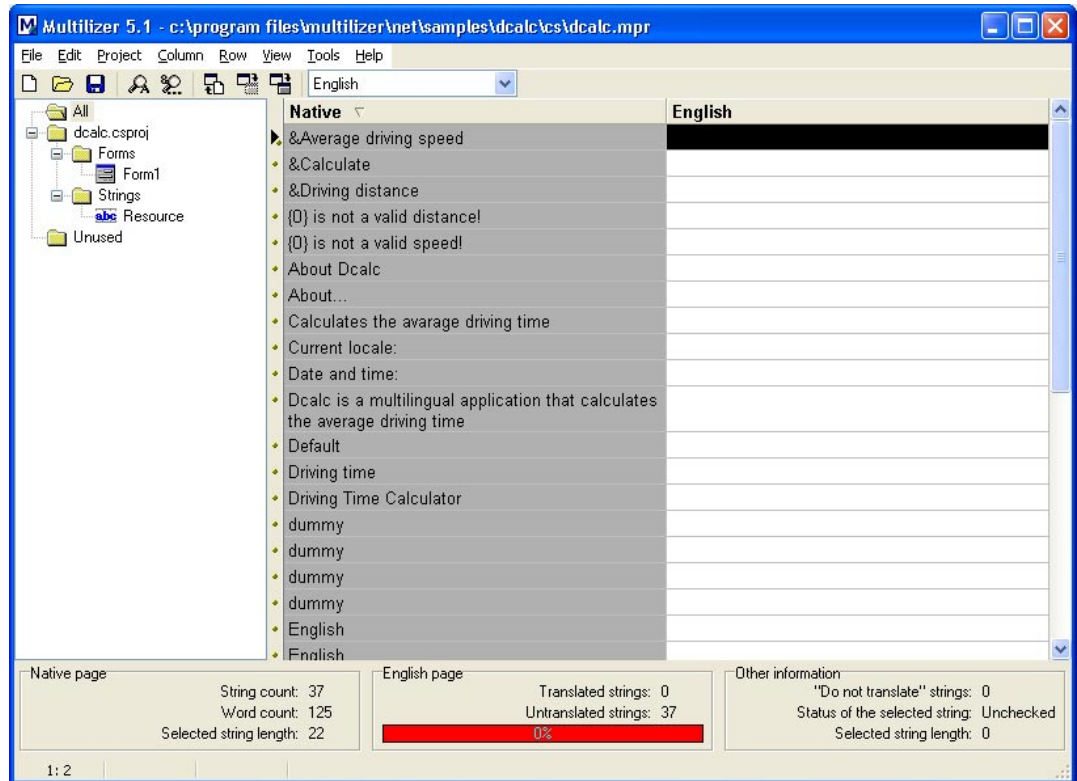


Figure 19 The project grid

If you cannot see any strings make sure that you set the Localizable property of the main form to true. Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Before we can create the localized files you have to set the .NET settings. Choose **Tools | Options | .NET**. The .NET Options dialog will appear. Select the Tools sheet and make sure that the both paths are properly set. If they are empty press the Default button. This tries to detect them. If the detection fails you have to manually set the file paths.

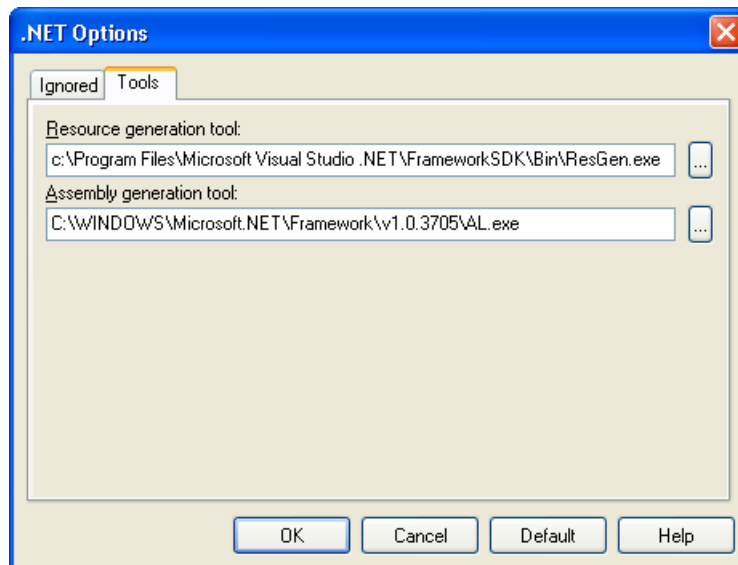


Figure 20 .NET Options dialog

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized satellite assembly files (.resources.dll) in the language specific sub-directories ('en' and 'fi'). In order to use the satellite assembly files they must locate on the sub directories of the main assembly file (Dcalc.exe). By default Visual Studio .NET place the EXE file to the bin or bin\Debug subdirectory. When running the application from Multilizer, Multilizer copies the application file to the project's main directory (e.g. bin\Debug\Dcalc.exe -> Dcalc.exe). Run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.

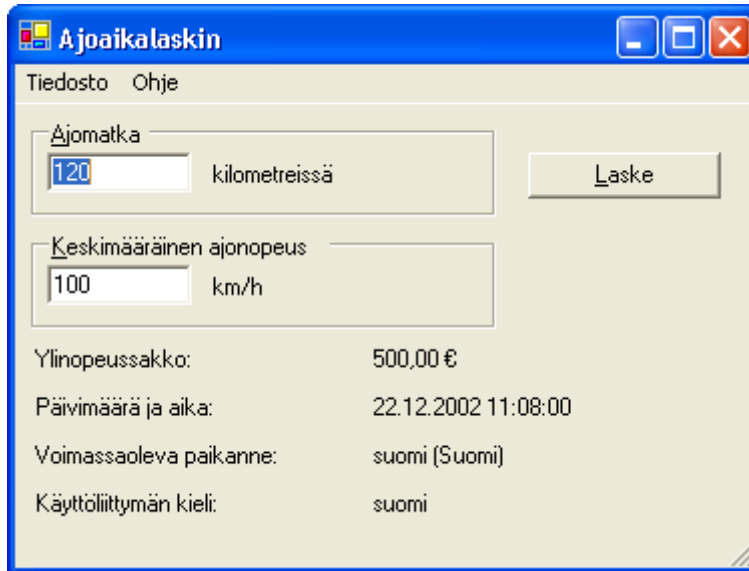


Figure 21 Localized Dcalc application

For more information about translating a project with Multilizer, see the Translator's Manual.



7

Visual C++

In this chapter we are going to create a localized Visual C++ application. Creating an application for Windows CE with Embedded Visual C++ is similar to creating an application for Windows with Visual C++ so this tutorial applies to Windows CE application localization as well. The application will be a simple driving-time calculator, Dcalc, which a user can use to calculate the average driving time for a given distance. You can actually use almost whatever application, library or component file and still follow the same steps mentioned in this tutorial to achieve a localized application.

The Dcalc application is very simple but still uses most features of Multilizer. The creation of the application is divided into several lessons, each covering one or more Multilizer functions.

Multilizer supports two kinds of localization of Windows C++ applications. They are binary localization and resource file localization. The following two chapters will describe these both.

Binary Localization

Binary applications, libraries or components contain the resource data in the application files (e.g. .exe), library files (e.g. .dll) or component files (e.g. .ocx). Multilizer creates the localized application files from the original file. The following picture describes the binary localization process.

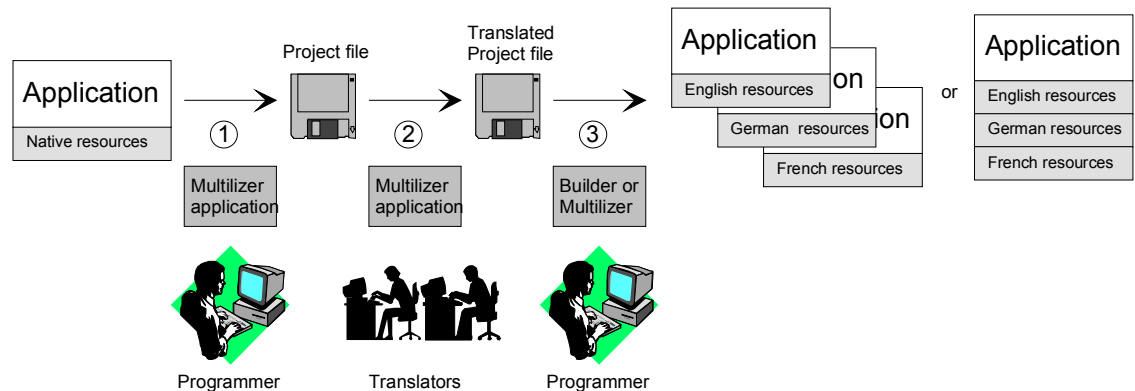


Figure 22 Binary localization process

The programmer uses Multilizer to extract strings from the original application file (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (1). The programmer uses Multilizer or Builder to create the localized application files (2). As a result there will be one application file for each localized language and/or a single binary file containing all languages.

Multilizer creates subdirectories under original file folder containing the localized file(s). I.e. there might be subfolders called `..\en\<localized file>` and `..\fi\<localized file>`. Replace the original file in the original program folder with the localized file from the localization folder and the next time you start the application the localized strings replace the native strings.

The following example figure shows the files that Multilizer uses on the C++ binary localization process in Windows.

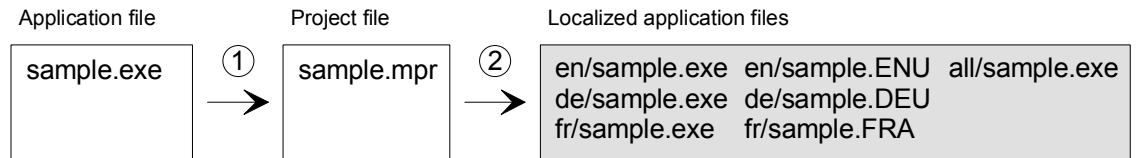


Figure 23 The files of the binary C++ localization process in Windows

When deploying the application you can either deploy the localized binary file (e.g. de\sample.exe), the multilingual binary file (all\sample.exe), or the original binary file (e.g. sample.exe) and the localized resource DLL(s) (e.g. de\sample.DEU).

Using binary localization Multilizer globalizes the dialog, menu, string and accelerator resources of any Windows or Windows CE application.



Besides applications built with C++, Multilizer binary localization type can be applied to applications compiled with other compilers. Multilizer automatically detects projects compiled with Delphi, C++Builder and Visual Basic. Refer to the corresponding tutorials, if you localize applications with any of the aforementioned compilers.

The binary localization process is described in more detail in chapter 5 and 6.

Resource File Localization

Application source code contains resource files (.rc). These files contain the data that needs to be localized. When linking the application the linker combines the compiled code and the resource data to create binary applications, libraries or components. Multilizer can create localized resource files from the original resource file. This secondary localization technology is called source localization. The following picture describes the source localization process.

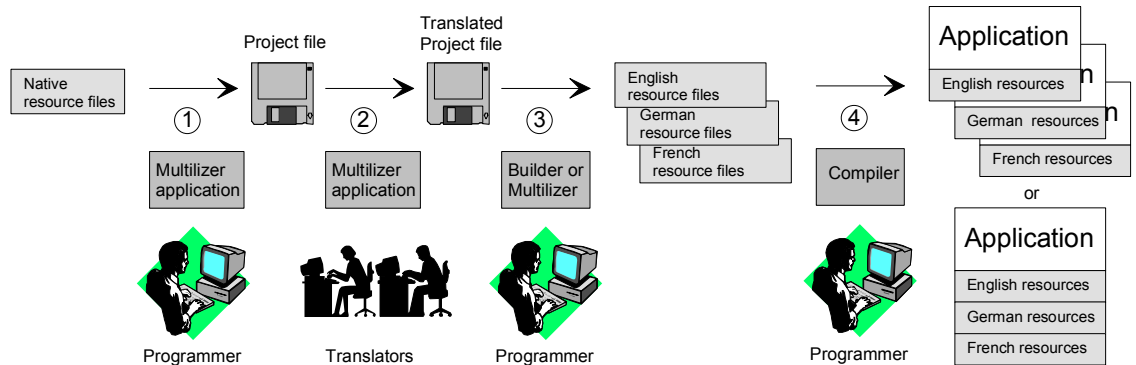


Figure 24 Source localization process

The programmer uses Multilizer to extract strings from the original resource file (1). Multilizer saves these strings to a project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (1). The programmer uses Multilizer or Builder to create the localized resource files (2). The programmer uses resource compiler to add the localized resource data to the application file (3). As a result there will be one application file for each localized language.

The following example figure shows the files that Multilizer uses on C++ source localization process in Windows.

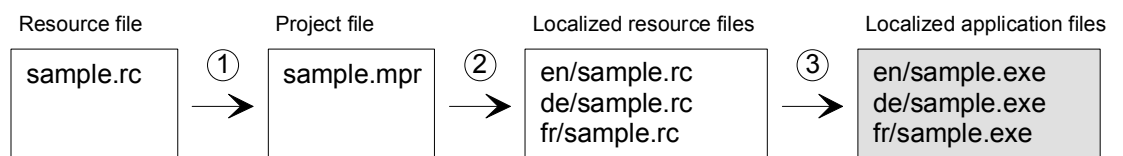


Figure 25 Different phases of C++ source localization process in Windows

When deploying the application you deploy the localized binary file (e.g. de\sample.exe).

Using source localization Multilizer globalizes dialog, menu, string and accelerator resources of any Windows or Windows CE application.

Source localization process is described in more detail in chapters 5 and 7.

English Application

We could start from scratch but in most cases it is a completed application or at least some specific application under construction that you want to globalize. This is what we are going to do. The `<mldir>\VCP\Samples\Tutorial\dcalc.dsw` contains the project file of Dcalc sample application for Visual C++. Compile and run the application.

The application should look like this:

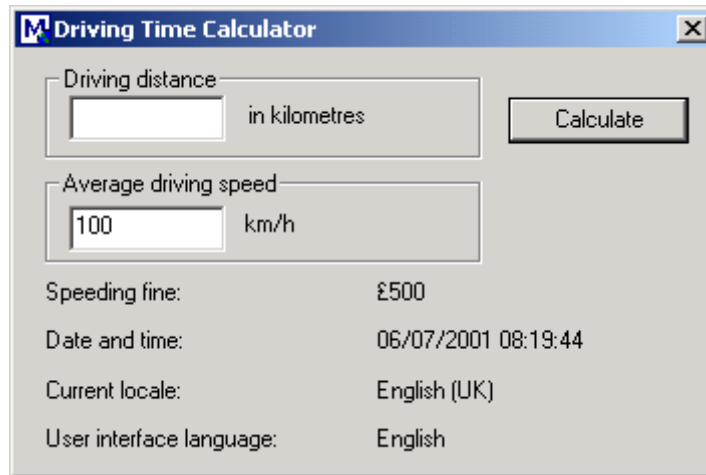


Figure 26 Driving time calculator with English user interface

The `<mldir>\EVCPP\Samples\Tutorial\dcalc.vcw` contains the project file of Dcalc sample application for Embedded Visual C++. Compile and run the application.

The application should look like this:

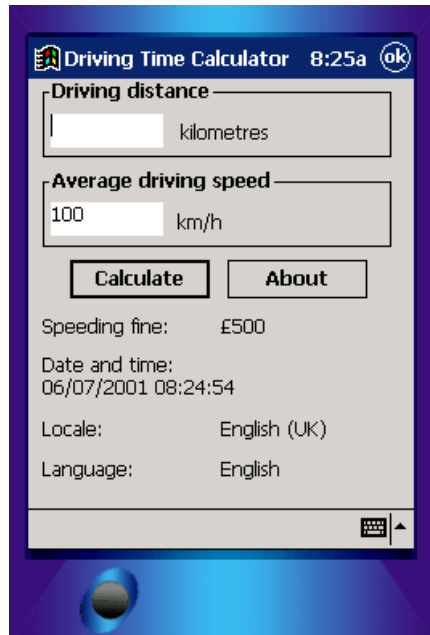


Figure 27 English Visual C++ Windows CE application

The user interface language is UK English and the applications use the UK format with currency, date and time. In the following chapters we will turn Dcalc into a truly multilingual application, step-by-step.

Internationalization

This chapter describes the binary internationalization process. Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development.

Open the Tutorial application, `<mldir>\VCP\Samples\Tutorial\dcalc.dsw`, or `<mldir>\EVCPP\Samples\Tutorial\dcalc.vcw`.

Study the source code of the application to familiarize yourself with it. It is not a complex application, so you should get the idea fairly quickly.

The most important part of the internationalization (i18N) is resourcing. This means removing all hard coded strings from the application's source code. Traditionally hard coded strings are turned into resources by moving the strings from the actual code into the resource strings.

Select the ResourceView sheet. Select the dcalc resource leaf from the tree and click the right mouse button. Choose Insert. The Insert Resource dialog appears.

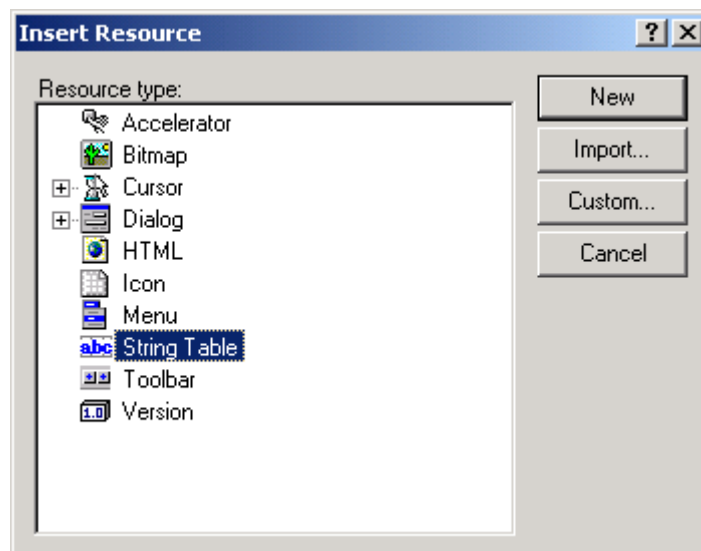


Figure 28 Insert Resource dialog box

Select String Table resource and press New. The String Table editor appears. Add the following resource strings to the table.

ID	Value	Caption
IDS_ABOUTBOX	101	&About DCalc...
IDS_INVALID_DISTANCE	102	"%s" is not a valid distance!
IDS_INVALID_SPEED	103	"%s" is not a valid speed!
IDS_RESULT	104	The average driving time is %d hours and %d minutes.
IDS_METRIC_DISTANCE	105	in kilometres
IDS_US_DISTANCE	106	in miles
IDS_METRIC_SPEED	107	km/h
IDS_US_SPEED	108	mph
IDS_LANGUAGE	109	English
IDS_INFORMATION	110	Information

Figure 29 String Table editor

The next step is to set the right value to the user interface labels. The original application shows the speeding fine in Pounds, the date and time in UK format, the locale and language labels have been hard coded to English (UK) and English. In addition the application requires the input in kilometers and in kilometers per hour.

An essential part of internationalization is to make the code locale independent. This means that the code is not hard coded to a single locale (e.g. English (UK)) but works with any locale.

Windows contains NLS API. It is a collection of locale functions that have access to the locale database. GetLocaleInfo function is used to get locale specific data such as measurement system, data format, etc.

To prepare your code to locale enabling we have to write some helper functions.

Visual C++

```
void CDcalcDlg::SetLabel(int control, int resourceId)
{
    CString str;

    str.LoadString(resourceId);
    SetDlgItemText(control, str);
}

void CDcalcDlg::SetLocaleLabel(int control, int localeItemId)
{
    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
    LPTSTR str = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
    SetDlgItemText(control, str);
    free(str);
}

int CDcalcDlg::GetLocaleInfoInt(int localeItemId)
{
    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
    LPTSTR str = (LPTSTR)malloc(len + 2);
    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
    int value = atoi(str);
    free(str);

    return value;
}
```

EVC

```
void CDcalcDlg::SetLabel(int control, int resourceId)
{
    CString str;

    str.LoadString(resourceId);
    SetDlgItemText(control, str);
}

void CDcalcDlg::SetLocaleLabel(int control, int localeItemId)
{
    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
    LPTSTR str = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
    SetDlgItemText(control, str);
    free(str);
}

int CDcalcDlg::GetLocaleInfoInt(int localeItemId)
{
    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
    LPTSTR str = (LPTSTR)malloc(len + 2);
    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
    int value = _wtoi(str);
    free(str);

    return value;
}
```

SetLabel function sets the label of a user interface element to a value found from the resource string. *SetLocaleLabel* sets the label of a user interface element to a value found from the locale database. *GetLocaleInfoInt* function returns an integer value from the locale database.

Now we can update the user interface items to match the current locale. Keep in mind that the system has a default locale. This locale is given to all applications currently running. You can change the default locale from the Control Panel.

OnInitDialog method is used to initialize the dialog box. Add the following code to the end of the *OnInitDialog* method.

setlocale function sets the formatting functions of the C run-time library to use the default locale. The original *Dcalc* uses kilometers and km/h. In United States miles and miler per hour are used. *LOCALE_IMEASURE* value of the locale database contains the measurement system of the locale. *GetLocaleInfoInt* get the measurement system. If the system is metric kilometers are used otherwise miles are used.

There are four different ways to show the currency value. They are 500 \$, 500\$, \$500 and \$ 500. You can but the currency label before or after the value and use a space between or not. *LOCALE_ICURRENCY* value if the locale database contains this information. The switch-case block formats the speeding fine according the current locale.

CTime class has the *Format* method that returns the date and time as a string that has been formatted according to the current locale.

The final step is to update the locale and language labels. *LOCALE_SLANGUAGE* returns the current locale as a string. *IDS_LANGUAGE* resource string contains the name of the language in its own language (e.g. English, Deutch, suomi).

Visual C++

```

BOOL CDcalcDlg::OnInitDialog()
{
    ...
    // Sets the locale depend format function to use the default locale

    setlocale(LC_ALL, "");

    // Gets the measurement system
    // Sets the Driving distance label: km or miles
    // and the Average driving speed label: km/h or mph

    if (GetLocaleInfoInt(LOCALE_IMEASURE) == 0)
    {
        // Metric

        SetLabel(IDC_DISTANCE, IDS_METRIC_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, "100");
        SetLabel(IDC_SPEED, IDS_METRIC_SPEED);
    }
    else
    {
        // US

        SetLabel(IDC_DISTANCE, IDS_US_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, "65");
        SetLabel(IDC_SPEED, IDS_US_SPEED);
    }

    // Set the fine value: $500, 500 mk, etc

    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, NULL,
0);
    LPTSTR currStr = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, currStr, len);

    LPTSTR buffer = (LPTSTR)malloc((strlen(currStr) + 5)*sizeof(TCHAR));

```

```

switch (GetLocaleInfoInt (LOCALE_ICURRENCY))
{
    case 0:
        sprintf(buffer, "%s500", currStr);
        break;

    case 1:
        sprintf(buffer, "500%s", currStr);
        break;

    case 2:
        sprintf(buffer, "%s 500", currStr);
        break;

    case 3:
        sprintf(buffer, "500 %s", currStr);
        break;
}

SetDlgItemText (IDC_FINE, buffer);
free(currStr);
free(buffer);

// Set the date and time

SetDlgItemText (IDC_DATETIME, CTime::GetCurrentTime().Format ("%c"));

// Sets the current locale and user interface language

SetLocaleLabel (IDC_LOCALE, LOCALE_SLANGUAGE);
SetLabel (IDC_LANGUAGE, IDS_LANGUAGE);

return TRUE;
}

```

OnInitDialog function in the Embedded Visual C++ is almost identical. We use COleDateTime instead of CTime.

EVC

```

BOOL CDialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, TRUE);

    // Gets the measurement system
    // Sets the Driving distance label: km or miles
    // and the Average driving speed label: km/h or mph

    if (GetLocaleInfoInt (LOCALE_IMEASURE) == 0)
    {
        // Metric

        SetLabel (IDC_DISTANCE, IDS_METRIC_DISTANCE);

        SetDlgItemText (IDC_SPEED_EDIT, L"100");
        SetLabel (IDC_SPEED, IDS_METRIC_SPEED);
    }
    else
    {
        // US

        SetLabel (IDC_DISTANCE, IDS_US_DISTANCE);

        SetDlgItemText (IDC_SPEED_EDIT, L"65");
        SetLabel (IDC_SPEED, IDS_US_SPEED);
    }

    // Set the fine value: $500, 500 mk, etc

```

```

    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, NULL,
0);
    LPTSTR currStr = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, currStr, len);

    wchar_t buffer[20];

    switch (GetLocaleInfoInt(LOCALE_ICURRENCY))
    {
        case 0:
            swprintf(buffer, L"%s500", currStr);
            break;

        case 1:
            swprintf(buffer, L"500%s", currStr);
            break;

        case 2:
            swprintf(buffer, L"%s 500", currStr);
            break;

        case 3:
            swprintf(buffer, L"500 %s", currStr);
            break;
    }

    SetDlgItemText(IDC_FINE, buffer);
    free(currStr);

    // Set the date and time

    SetDlgItemText(IDC_DATETIME,
COleDateTime::GetCurrentTime().Format());

    // Sets the current locale and user interface language

    SetLocaleLabel(IDC_LOCALE, LOCALE_SLANGUAGE);
    SetLabel(IDC_LANGUAGE, IDS_LANGUAGE);

    CenterWindow(GetDesktopWindow());

    return TRUE;
}

```

The CalculateButtonClick event needs a little bit more rewriting. Let's study the code that generates the driving distance message:

```

text = CString("The avarage driving time is ") +
    itoa(hours, buffer1, 10) +
    " hours and " +
    itoa(minutes, buffer2, 10) +
    " minutes.";

```

This seems to be just OK, but it will actually make the localization hard or even impossible. The reason is that the above logic assumes that the message always starts with the "The average driving time is " string, and then contains the hours, hour label, minutes and minute label. However, not all languages use the same order of words in a sentence. For example, the order might be: minute label, minutes, hour label, hours and text part. Reordering of the parts of the message is impossible if we use the code shown above.

Fortunately we can use CString's Format function. It uses message pattern that contains placeholders for the dynamic parameters. At run-time the function combines the pattern with the parameters to compose the message. Because the pattern is a single string it can be added to the resource strings, and it can then be translated as a single item. The following code contains the internationalized CalculateButtonClick event.

```

void CDcalcDlg::OnCalculate()

```

```

{
    // Calculates the driving time and shows it in a message box

    CString text;
    CString distances;
    CString speeds;

    GetDlgItemText(IDC_DISTANCE_EDIT, distances);
    GetDlgItemText(IDC_SPEED_EDIT, speeds);

    int distance = atoi(distances);
    int speed = atoi(speeds);

    if (distances == "" || distance < 0)
        text.LoadString(IDS_INVALID_DISTANCE);
    else if (speeds == "" || speed <= 0)
        text.LoadString(IDS_INVALID_SPEED);
    else
    {
        int hours = distance/speed;
        int minutes = (int)((double)distance/speed - hours)*60;

        text.Format(IDS_RESULT, hours, minutes);
    }

    CString str;

    str.LoadString(IDS_INFORMATION);

    MessageBox(text, str);
}

```

The dialog resource contains several strings that are obsolete because they all get set at run-time. A good practice is to replace these strings with “dummy” strings and then exclude these strings from the localization project.

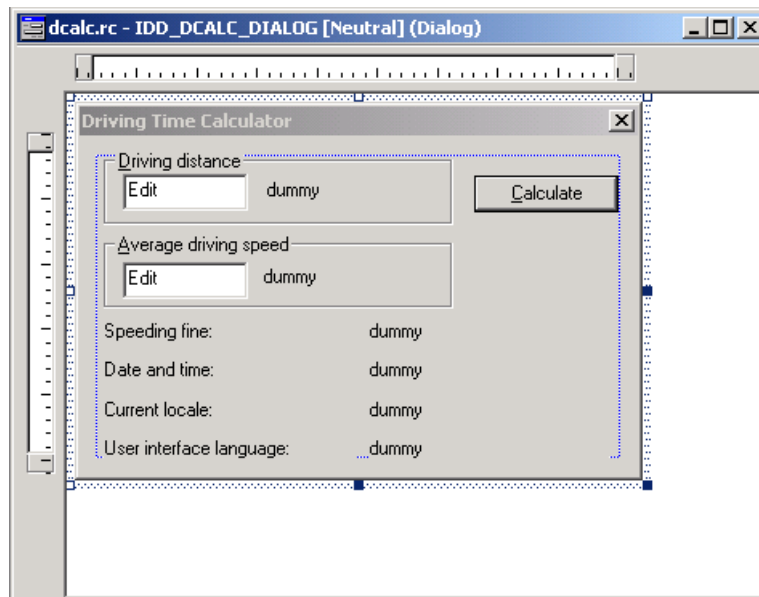


Figure 30 Replace dynamic items with dummy strings

Most translations get longer when translated from English to other European languages. The final internationalization step is to change the user interface such way that it can accommodate long translations. The easiest way is to set every user interface item as wide as possible. The following figure contains the reworked user interface.

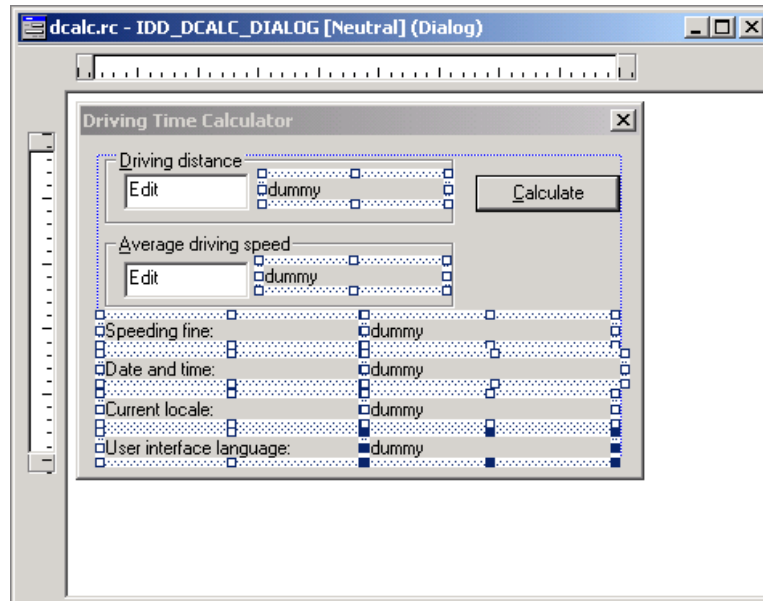


Figure 31 Resize dynamic items for long translations

Now we are ready to create a Multilizer project for Dcalc.

Creating a Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

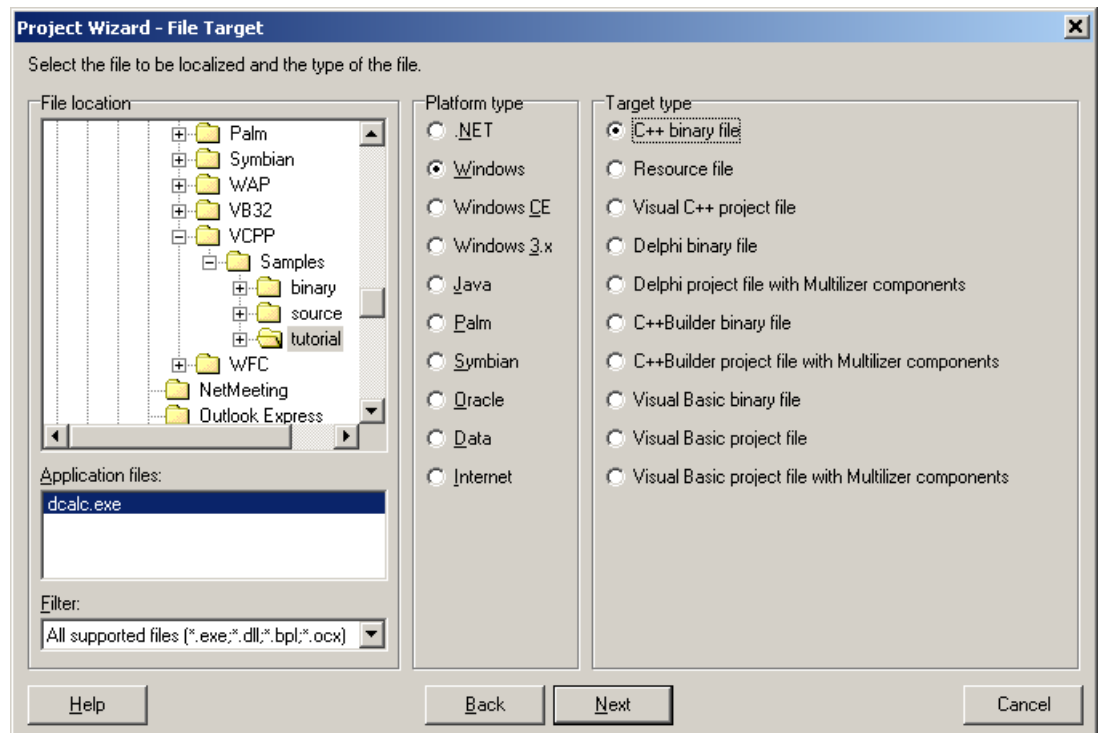


Figure 32 The Select Target sheet is used to enter the file to be localized

This sheet specifies the directory where your application is located. Choose the `<mdir>\VCpp\Samples\Tutorial` or `<mdir>\EVCpp\Samples\Tutorial` subfolder of your Multilizer setup. If your application is located in multiple directories you can later add more directories. Project Wizard detects the platform and target types. The

Platform type should be *Windows* and the Target type should be *C++ binary file*. If they are wrong, check the right types.

If you want to use resource localization set the Target type to *Visual C++ project file*.



NOTE!



NOTE!



NOTE!

Here you have different 'Target type' choices what to localize depending whether you are going to localize a Windows application that has been made with C++ compiler or other development tool. If some other tool is used, refer to the corresponding tutorial.

If software was built for Windows CE, Multilizer should detect it automatically and offer in the above dialog 'Windows CE' platform type.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the >> button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

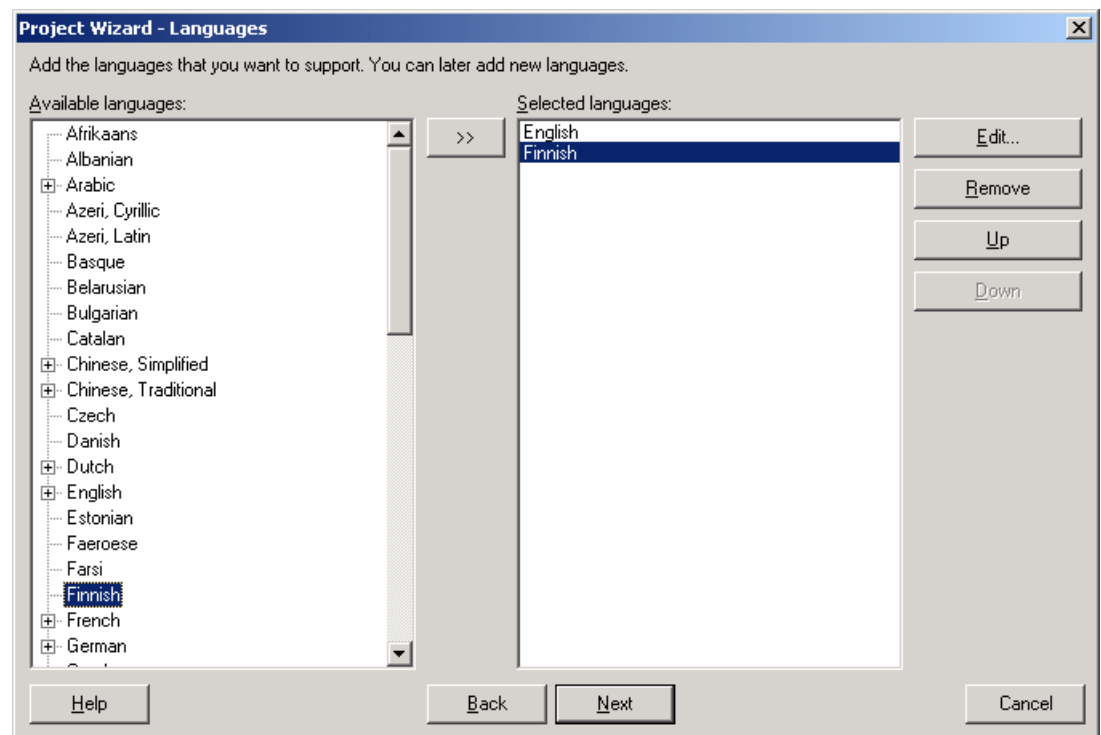


Figure 33 English and Finnish added to the project file

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

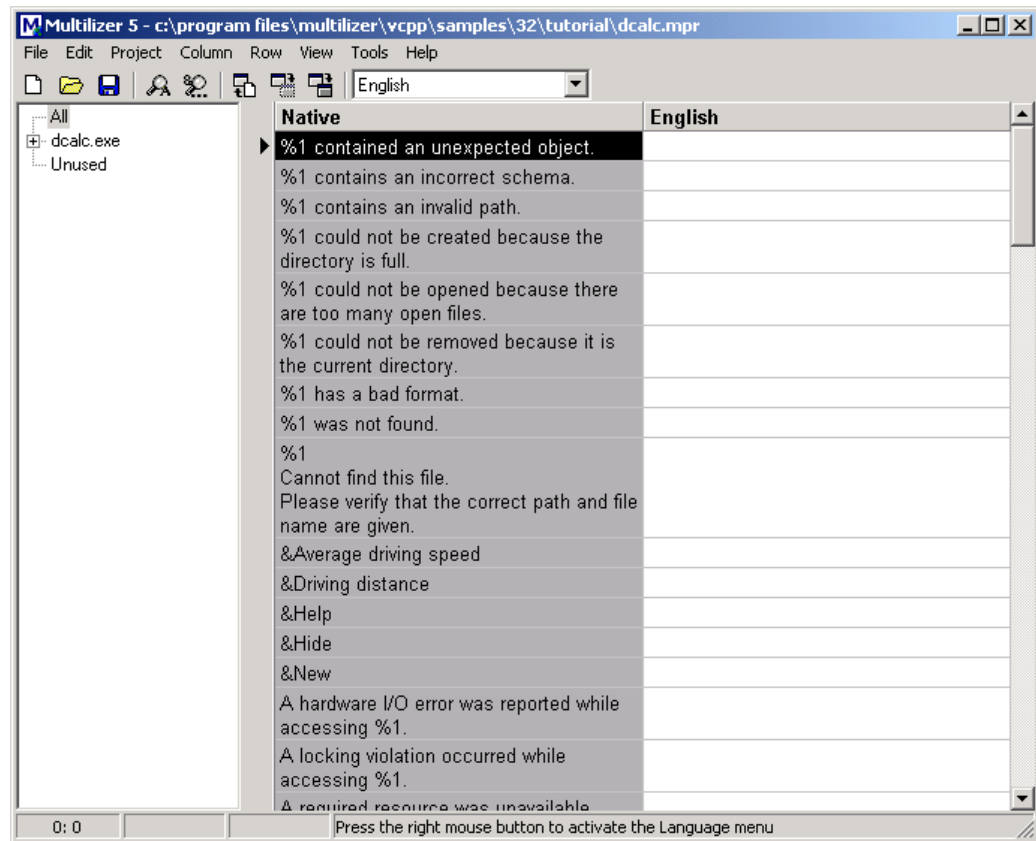


Figure 34 The project view grid.

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part.

When you have translated the project save it by choosing **File | Save**. Then create the localized application files by choosing **Project | Build Localized Files**. This creates the localized application files (.exe) in the language specific sub-directories ('en' and 'fi').

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.

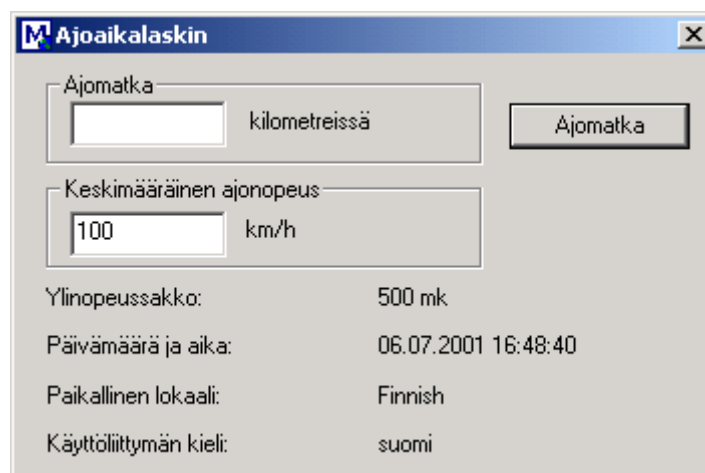


Figure 35 Localized Dcalc application (Windows)

Localized Windows CE version should look like this:

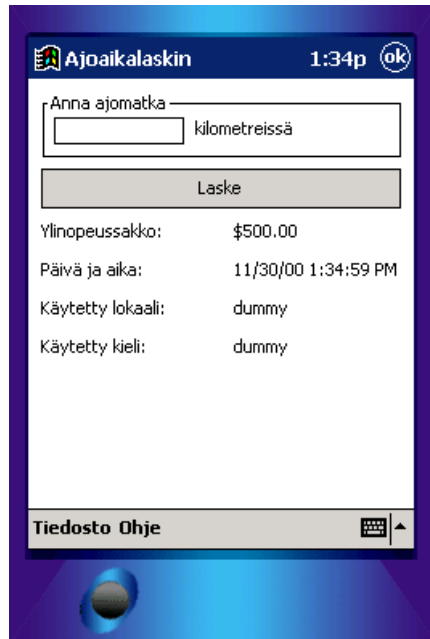


Figure 36 Localized Dcalc application (Windows CE)

By default Multilizer creates the localized application files. You can set it to create also the localized resource DLLs (e.g. .ENU) and the multilingual application file (e.g. all\ldcalc.exe) that contains resources of all languages of the project. Select `dcalc.exe` from the left-hand side of the tree view and right-click to open the popup menu. Choose **Edit target** to open the C/C++ Binary File Target dialog.

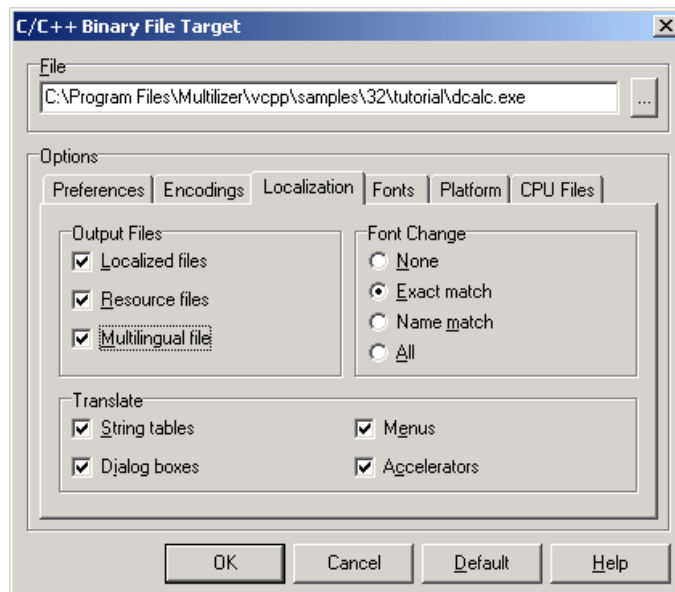


Figure 37 The C/C++ Binary File Target dialog

Select the Localization sheet and check the *Resource files* and the *Multilingual file* check boxes. Next time you build localized files Multilizer also creates resource DLLs and a multilingual application file.

For more information about translating a project with Multilizer, see the Translator's Manual.



8

Visual Basic

In this tutorial we are going to create a localized application. The application will be a simple driving-time calculator, Dcalc which a user can use to calculate the average driving time for a given distance.

The application is very simple but still it uses most features of Multilizer.

Visual Basic 6.0 and Embedded Visual Basic 3.0 are the environments for the examples of this chapter. The usage of some other versions of the aforementioned environments is almost identical. Some of the menu options may vary.

How to use Multilizer

When you create a new Visual Basic localization project with Multilizer, the Project Wizard will show the following screen after the welcome screen.

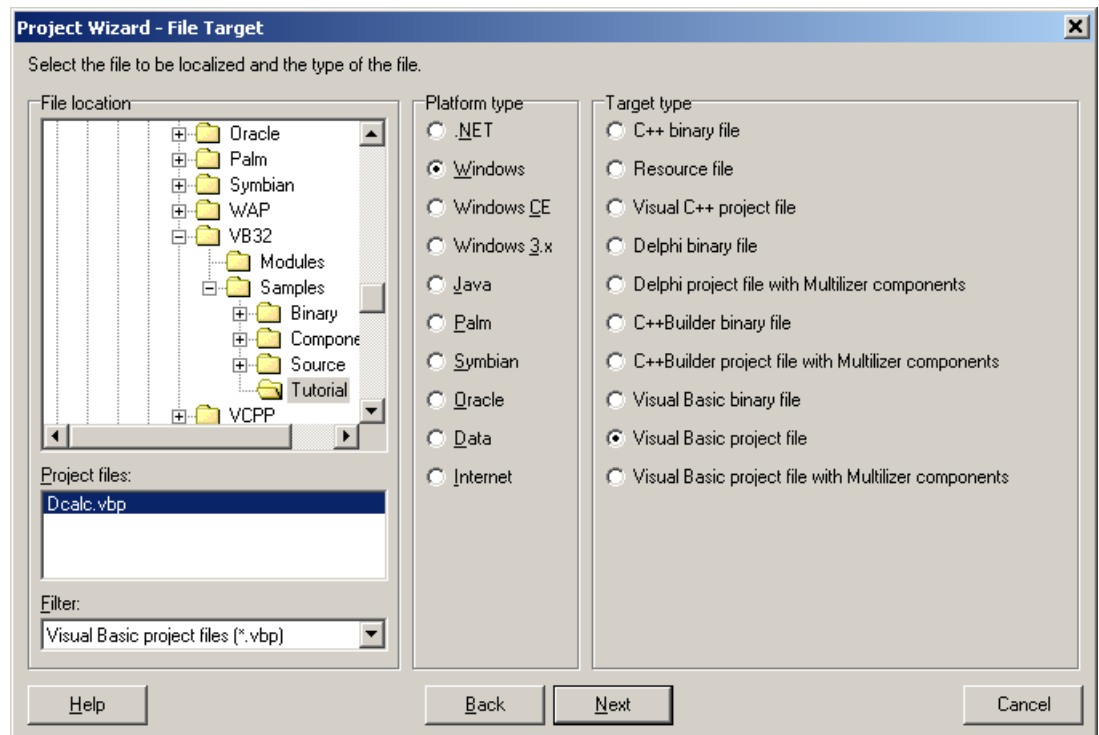


Figure 38 Windows localization project types

Before continuing with the Project Wizard, you have to know which of the different Project Types available is suitable for your Visual Basic project. This and the following chapter will explain the possibilities.

Three platform types contains Visual Basic targets types: Windows, Windows CE and Windows 3.x. When specifying the Visual Basic project directory (i.e. the directory where your Visual Basic project is located), you have to also specify the Target type. Depending on the target type, you will either do source localization or component localization (See the table below). The following chapter will explain the difference between source localization and component localization.

Following Target types are available:

Platform type	Target Type	Description
Windows	Visual Basic binary file	A <u>binary localization</u> project for 32-bit Visual Basic 4.0, 5.0 and 6.0 projects.
Windows	Visual Basic project file	A <u>source localization</u> project for 32-bit Visual Basic 4.0, 5.0 and 6.0 projects.
Windows	Visual Basic project file with Multilizer components	A <u>component localization</u> project for 32-bit Visual Basic 4.0, 5.0 and 6.0 projects. Multilizer OCX adds multi-language support to the software. Multilizer OCX was included to previous Multilizer version. It is not included to current version.
Windows CE	Embedded Visual Basic project file	A <u>source localization</u> project for Visual Basic software built with Embedded Visual Basic.
Windows 3.x	Visual Basic project file	A <u>source localization</u> project for 16-bit Visual Basic 4.0 projects.
Windows 3.x	Visual Basic project file with Multilizer components	A <u>component localization</u> project for 16-bit Visual Basic 4.0. Multilizer OCX adds multi-language support to the software. Multilizer OCX was included to previous Multilizer version. It is not included to current version.

Localization process

From the different localization types available, source localization is recommended. Resulting localized executables will not differ in size and the performance will be the same as of the native software.

If it's necessary to create multilingual software with dynamic language-change at run-time, then you must use component localization.

It is possible to write a Visual Basic application that gets all the user interface strings from string resources. However this is a rather time-consuming way to write Visual Basic applications. If you have done the work of isolating strings in string resources, you can use the binary localization method (described in the C++ tutorial) to localize the software.

Source localization

Visual Basic source code contains project, form and module files. Multilizer creates the localized source code files from the original source files. The following picture describes the Visual Basic localization process.

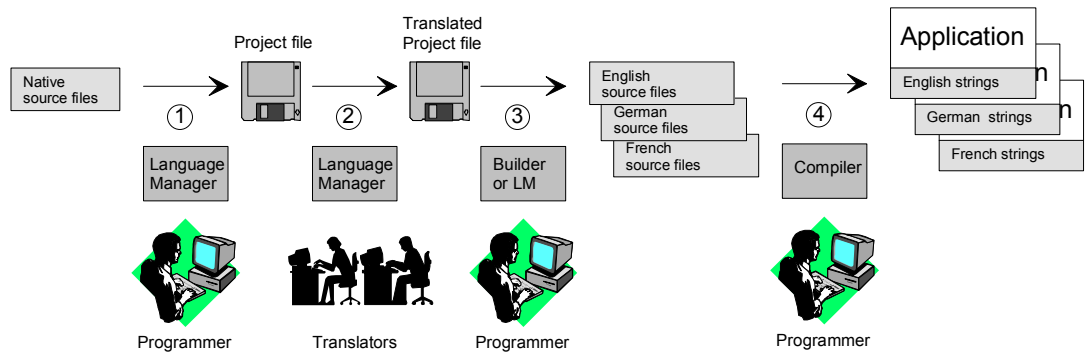


Figure 39 The Visual Basic localization process

The programmer uses Multilizer to extract strings from the resource source file (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource files (3). As a result there will be one application file for every language or one localized resource file for each localized language.

Windows

The following figure shows the files that Multilizer uses in the Visual Basic localization process in Windows.

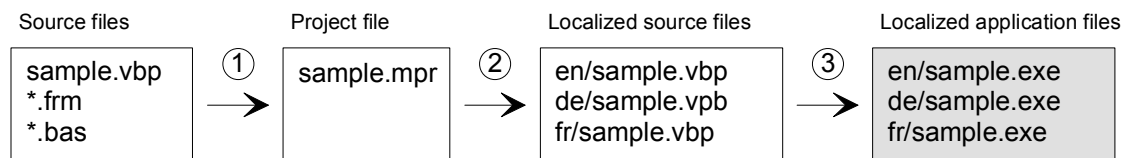


Figure 40 The files of the source code localization process of Visual Basic application

The programmer starts Multilizer's Project Wizard and selects a Visual Basic project file (.vbp) to be globalized. Multilizer extracts the string data from the form (.frm) and the source code (.bas) files to create the project file (1). The translators translate the projects. The programmer uses Multilizer or Builder to create the localized project and the source code files (2). The programmer uses Visual Basic to compile these projects file into binary files (3).

Windows CE

The following figure shows the files that Multilizer uses in the Visual Basic localization process for Windows CE platform.

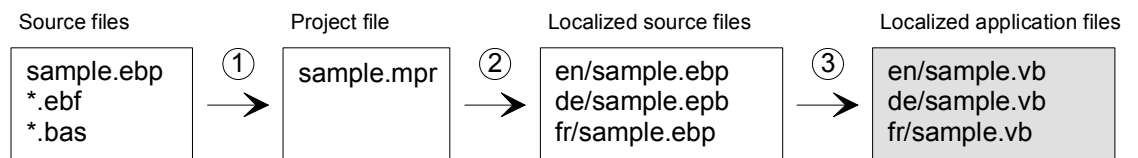


Figure 41 The files of the source code localization process of Embedded Visual Basic application

Embedded Visual Basic (Windows CE) uses the same localization process as desktop Visual Basic. However it uses a different project file (.ebp) and the form file (.ebf) extensions.

Binary localization

If you have isolated all strings, including the user interface strings in string resources, you can apply the binary localization type in your project. See binary localization described in the C++ tutorial for more information.

Component Localization

Use component localization only if your application must have the dynamic language switch feature.

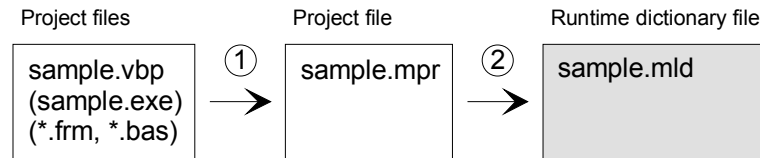


Figure 42 The files of the component localization process of a Visual Basic application

The programmer starts Multilizer's Project Wizard and selects a Visual Basic project file (.vbp) to be globalized. Multilizer extracts the string data from the form (.frm) and the source code (.bas) files, and the resource strings from the binary file (.exe) to create the project file (1). The programmer adds the dictionary components to the main form and the translator components to all forms. The translators translate the projects. The programmer uses Multilizer or Builder to create the run-time dictionary for the application (2).



Multilizer OCX was included to previous Multilizer version. It is not included to current version. If you have a previous Multilizer version installed you can use the component localized. If not you have to use either the source code or binary localization.

English Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. This is what we are going to do. The `<platform>\Samples\Tutorial` contains Dcalc with an English UI. `<platform>` is `VB16`, `VB32` or `EVB` depending on your target operating system. Open it, compile it, and finally run it.

The Windows application should look like this:

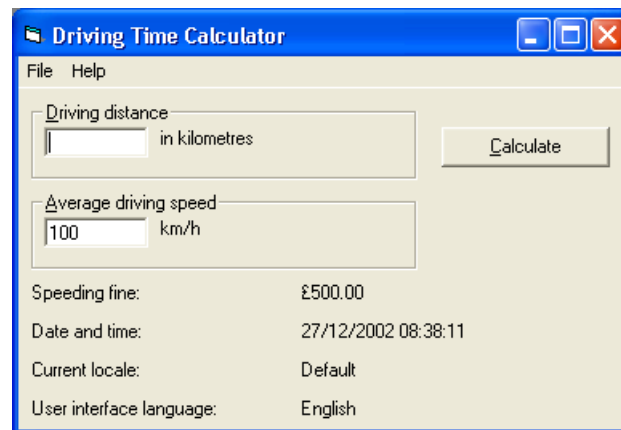


Figure 43 Visual Basic (Windows) application with an English UI

The Windows CE application should look like this:

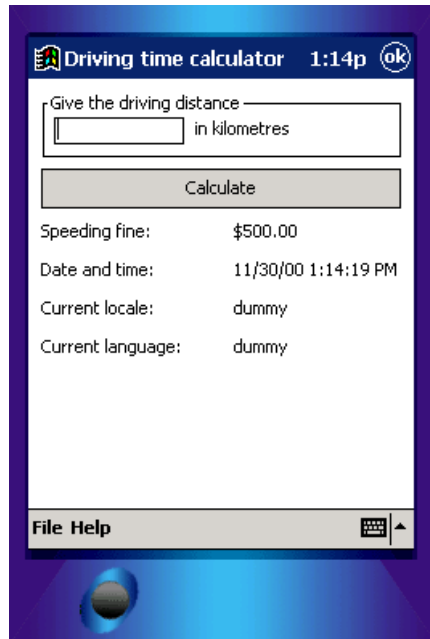


Figure 44 Visual Basic (Windows CE) application with an English UI

The user interface language is English. In the following chapters we will localize Dcalc step-by-step.

Internationalization

Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software.

Internationalization (I18N) takes place at the level of program design and document development. I18N is defined as the set of processes, tools, coding techniques and procedures used to write a software program that supports all of the language requirements and country conventions of all of the countries where the software will be used. For instance, writing an I18N ready application that supports the writing systems for Japan and English, including the special sorting for the different alphabets. The user interface of an I18N ready application is still in English, but the base code supports the language requirements for both languages.

When writing code that can be easily localized later on, i.e. doing internationalization, the following things should be taken into consideration:

- String lengths in general. English language for example has shorter strings than Finnish. So both the user interface and the source code containing the string should be able to contain relatively long strings in the actual code.
- The user interface can be designed by using different user interface elements so that string lengths don't pay that much importance in the localization. The developer can use for example radio buttons with descriptions instead of drop-downs and thus might not have to make the drop-down very wide just to make sure that most languages can fit in to the drop-down space.
- VB internationalization for Windows' desktop versions or for Windows CE follows exactly the same rules since the code is the same and the user interface issues are the same.

Creating a New Project

Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

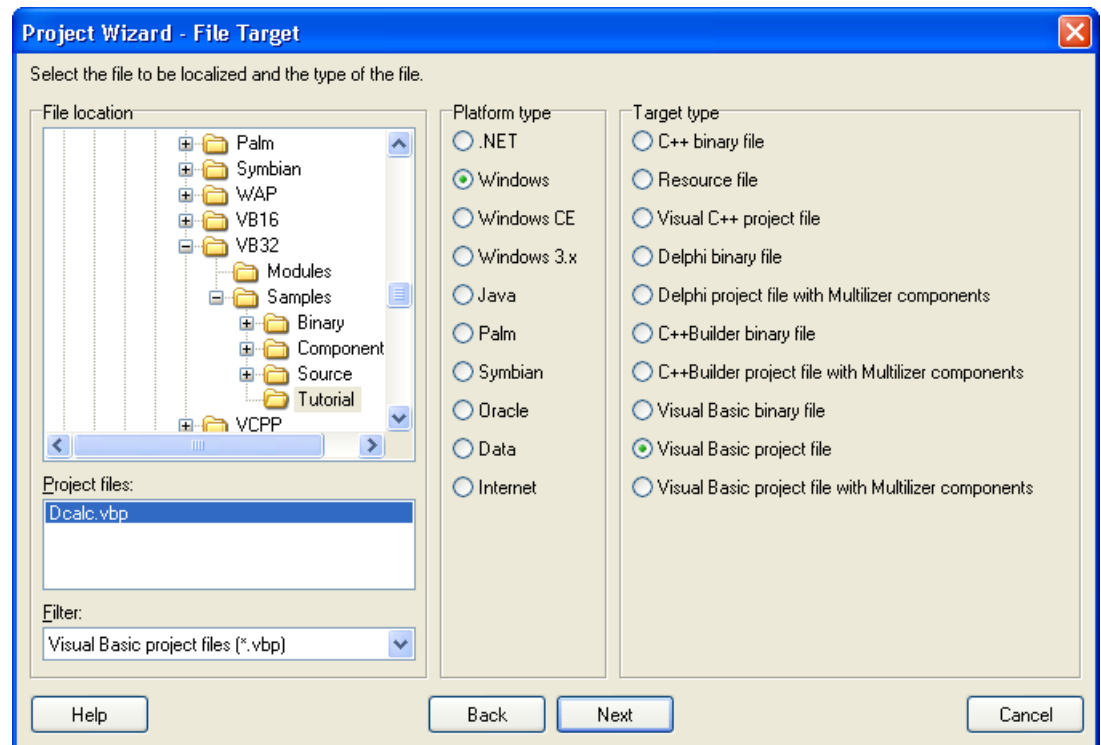


Figure 45 The Select Target sheet is used to specify the project to be localized

This sheet specifies the directory where your application is located. Choose the `<multilizer>\VB32\Samples\Tutorial` (Windows), `<multilizer>\VB16\Samples\Tutorial` (Windows 3.x) or `<multilizer>\EV3\Samples\Tutorial` (Windows CE) subfolder in your Multilizer directory. Project Wizard detects the application and project types. If they are wrong, check the right types matching on your project.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the `>>` button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

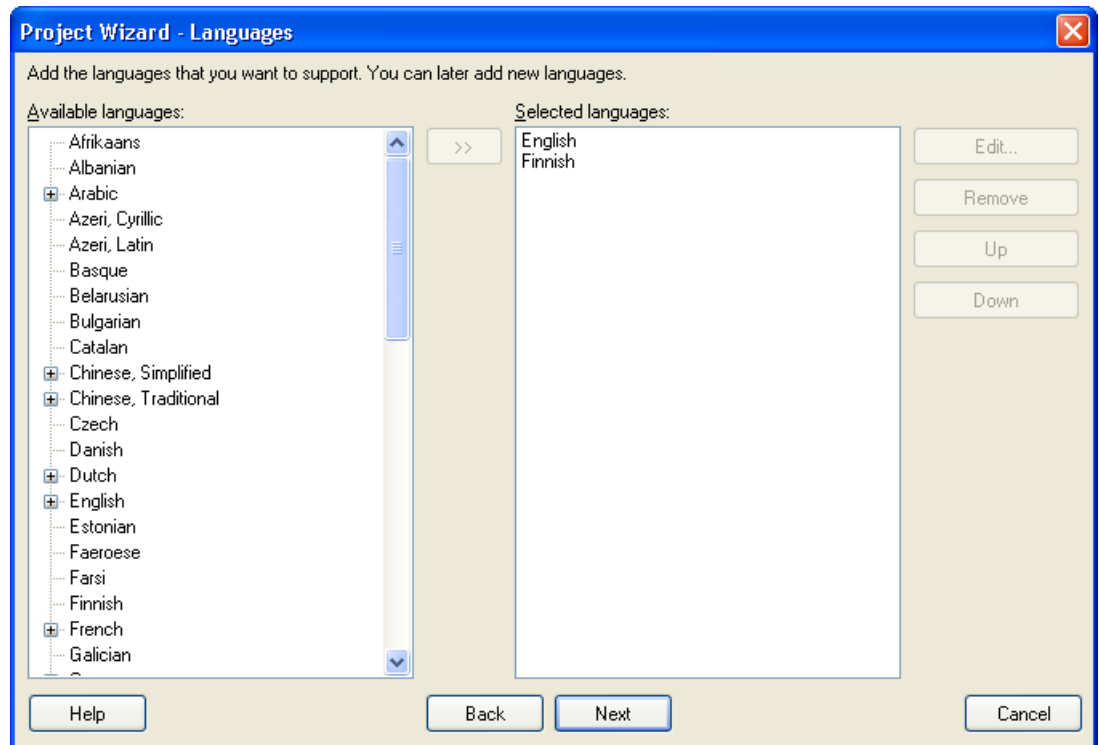


Figure 46 English and Finnish added to the project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

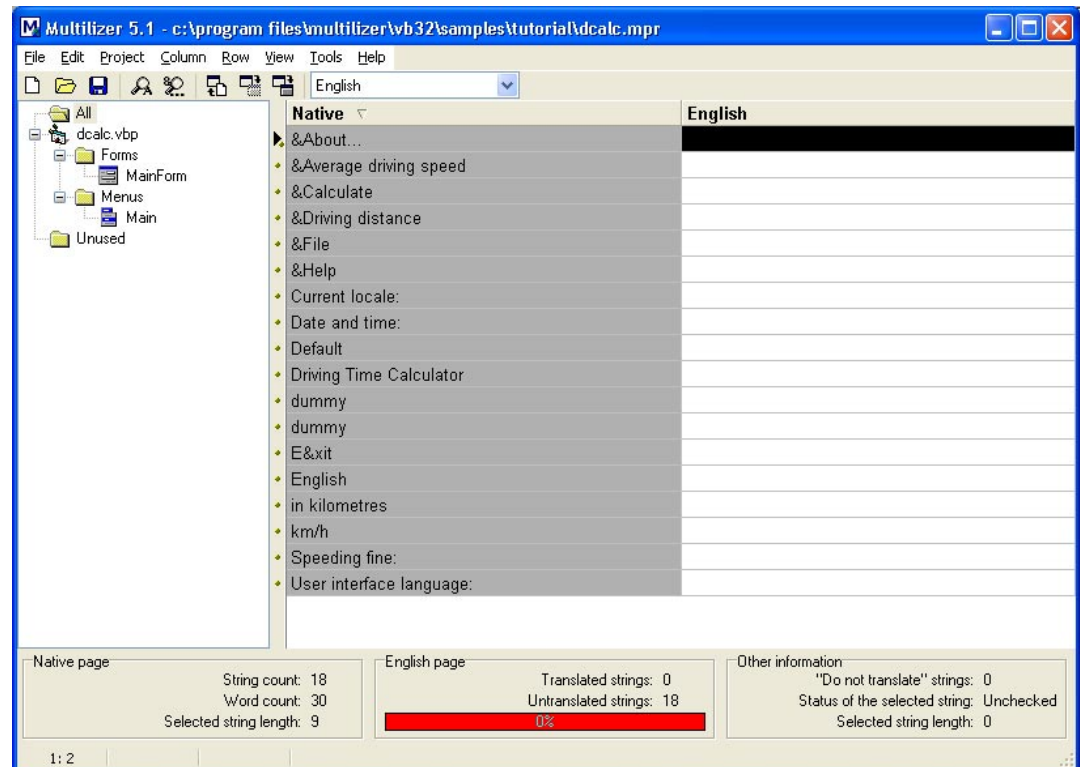


Figure 47 The project grid

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Create the localized source files by choosing **Project | Build Localized Items**. This creates the localized source files in language specific sub directories ('en' and 'fi').

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run. This will load the localized project into Visual Basic.

After compiler appears, run the project. The localized Dcalc appears.

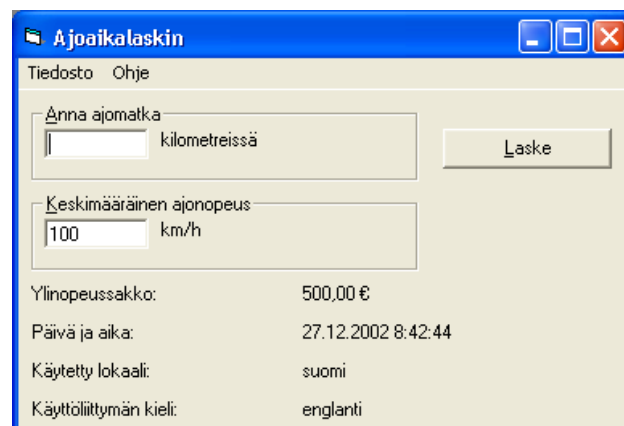


Figure 48 Localized Dcalc application (Windows)

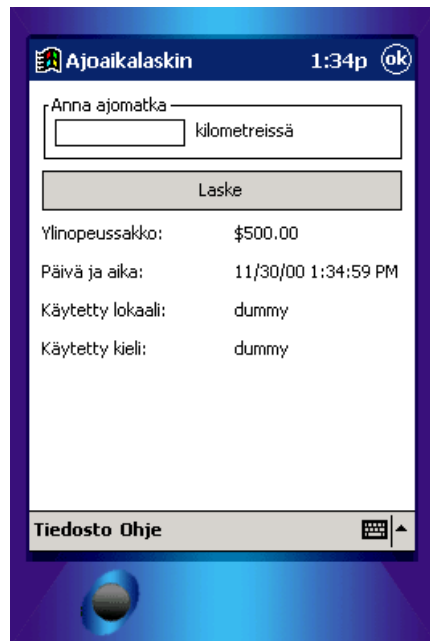


Figure 49 Localized Dcalc application (Windows CE)



For more information about translating a project with Multilizer, see the Translator's Manual.

9

Delphi and C++Builder

In this chapter we are going to create a localized Delphi/C++Builder application. The application will be a simple driving-time calculator, Dcalc, which a user can use to calculate the average driving time for a given distance. Don't let the fact that the Dcalc application is very simple fool you. It is a real application and uses most features of Multilizer.

This tutorial is written for Delphi 7, and C++Builder 6. Using an older version is almost identical. Some menu commands may vary.

Multilizer supports two different kinds of localization methods with VCL applications. They are binary localization and component localization. The following two chapters will describe both in detail.

Binary Localization

The compiled VCL applications (.exe or .dll) contain resource data. When doing binary localization, Multilizer scans the original binary files and creates localized binary files as copies of the original files. The following picture describes the binary localization process.

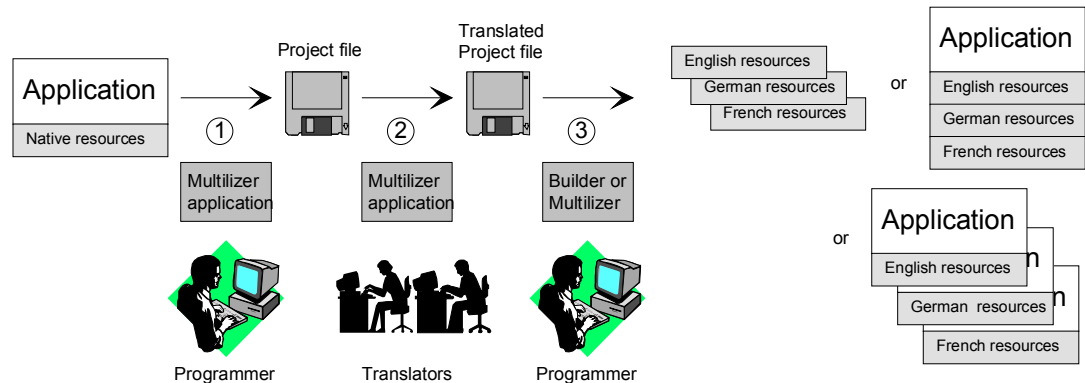


Figure 50 Binary localization process of a VCL application

The programmer uses Multilizer to extract strings from the original binary file(s) (1). Multilizer saves these strings to a project file. The programmer uses Multilizer to send the project file to a translator(s) who uses Multilizer to translate the project file, and then sends the translated project file back to the programmer (2). The programmer then uses Multilizer to create localized binary files (3). As a result there will be one resource file for each localized language. Multilizer can also produce a single multilingual binary file containing all the languages of the project, or one binary file for each language.

Multilizer creates subfolders in the original file folder containing the localized files. The subfolders are named after the language they contain by a two or five-letter identifier. For example `..\en\<localized file>` would contain English localized files, `..\en-US\<localized file>` would contain English (United States) localized files, and `..\fi\<localized file>` would contain Finnish localized files.

The following figure shows the files that Multilizer uses in the binary localization process of a VCL application.

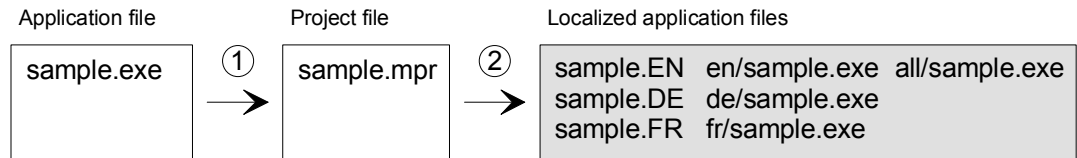


Figure 51 The files of the binary localization process of a VCL application

When deploying the application you can either deploy the original application file with the selected resource file(s), the localized application file(s), or the multilingual application file. When using the resource files it possible to change the language of the user interface on run-time.

The binary localization is the recommended localization method. The binary localization process is described in more detail in chapters 5 and 6.

Component Localization

The VCL applications source contains form files (.dfm) and source code files (.pas or .cpp). Multilizer scans these files and creates a project file that contains the strings of the application. The following picture describes the component VCL localization process.

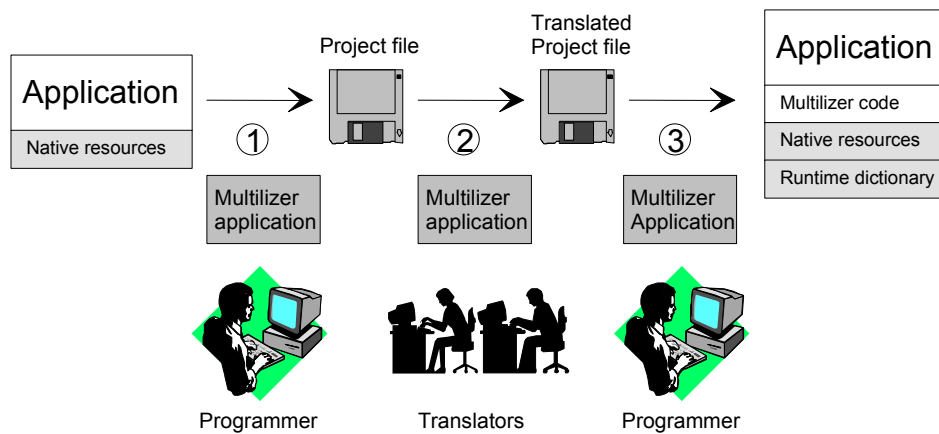


Figure 52 Component localization process of a VCL application

The programmer uses Multilizer to extract strings from source files (1). Multilizer saves these strings to a project file. The programmer uses Multilizer to send the project file to a translator(s) who uses Multilizer to translate the project file, and then sends the translated project file back to the programmer (2). The programmer then uses Multilizer to create a run-time dictionary file for the application (3). As a result there will be one version of the actual application that can use all the languages in the project file. Component localization makes it possible to change the language used in the application at run-time.

The following figure shows the files Multilizer uses in the component localization process of a VCL application.

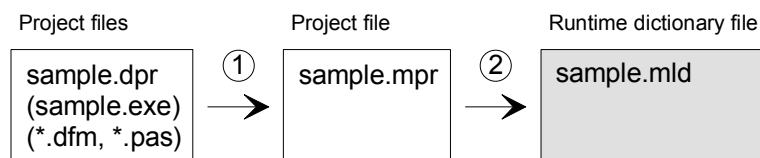


Figure 53 The files of the component localization process of a VCL application

When deploying the application you can either embed the dictionary inside the application binary file or leave it to an external file. If you leave it to an external file you can modify the translations or even add new languages to the application just by replacing the dictionary file.

The component localization process is described in more detail in chapters 7, 8 and 9.

English Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. The `<mldir>\<compiler>\Samples\Tutorial\dcalc.dpr` contains the project file of the Dcalc sample application. `<compiler>` is *Delphi1*, *Delphi2*, *Delphi3*, *Delphi4*, *Delphi5*, *Delphi6*, *Delphi7*, *CBuilder1*, *CBuilder3*, *CBuilder4*, *CBuilder5* or *CBuilder6* depending on your compiler version. Compile and run the application. By default, Dcalc uses English language.

The application should look like this:

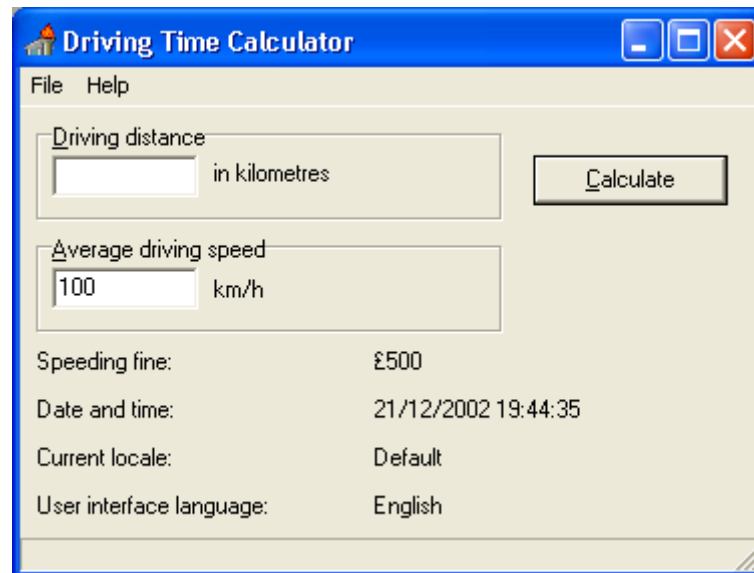


Figure 54 Dcalc application using an English user interface

The user interface language is English (UK) and the application formats currency, date and time according to English (UK) standards. In the following chapters we will turn Dcalc into a truly multilingual application, step-by-step.

Binary Internationalization

This chapter describes the binary internationalization process. Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development.

Open the Tutorial application,
`<mldir>\<compiler>\Samples\Tutorial\dcalc.dpr`.

Study the source code of the application to familiarize yourself of its behavior. It is not a complex application, so you should get the idea fairly quickly.

The main form contains some labels that are locale dependent. The label on the right side of the edit box contains the distance unit. Not every country uses kilometers. That's why we must update the label at run-time using a resource string, to make sure that a correct unit is used. Similarly we assign the screentip text of the edit control and the label containing the current language at run-time. We could add the initialization code into the OnCreate event of the main form but let's prepare to the runtime language switch and write a separator function for the initialization.

Delphi

```
procedure TMainForm.InitFrom;
resourcestring
  SLanguage = 'English';
  SDefault = 'Default';
```

```

    SMetricDistanceHint = 'Give the driving distance in kilometres';
    SMetricSpeedHint = 'Give the average driving speed in kilometres per
hour';
    SMetricDistanceLabel = 'in kilometres';
    SMetricSpeedLabel = 'km/h';

    SUsDistanceHint = 'Give the driving distance in miles';
    SUsSpeedHint = 'Give the average driving speed in miles per hour';
    SUsDistanceLabel = 'in miles';
    SUsSpeedLabel = 'mph';
begin
    Application.OnHint := DisplayHint;

    SpeedingFine.Caption := Format('%m', [500.0]);
    CurrentTime.Caption := DateTimeToStr(Now);

    CurrentLocale.Caption := GetLocaleStr(
        LOCALE_USER_DEFAULT,
        LOCALE_SNATIVELANGNAME,
        SDefault);

    CurrentLanguage.Caption := SLanguage;

    if GetMeasurementSystem = ivmsMetric then
    begin
        DistanceEdit.Hint := SMetricDistanceHint;
        DistanceLabel.Caption := SMetricDistanceLabel;

        SpeedEdit.Text := '100';
        SpeedEdit.Hint := SMetricSpeedHint;
        SpeedLabel.Caption := SMetricSpeedLabel;
    end
    else
    begin
        DistanceEdit.Hint := SUsDistanceHint;
        DistanceLabel.Caption := SUsDistanceLabel;

        SpeedEdit.Text := '65';
        SpeedEdit.Hint := SUsSpeedHint;
        SpeedLabel.Caption := SUsSpeedLabel;
    end;
end;

```

C++Builder

```

void __fastcall TMainForm::InitForm()
{
    Application->OnHint = DisplayHint;

    SpeedingFine->Caption = Format("%m", OPENARRAY(TVarRec, (500.0)));
    CurrentTime->Caption = DateTimeToStr(Now());

    CurrentLocale->Caption = GetLocaleStr(
        LOCALE_USER_DEFAULT,
        LOCALE_SNATIVELANGNAME,
        LoadStr(SDefault));

    CurrentLanguage->Caption = LoadStr(SLanguage);

    if (GetMeasurementSystem() == ivmsMetric)
    {
        DistanceEdit->Hint = LoadStr(SMetricDistanceHint);
        DistanceLabel->Caption = LoadStr(SMetricDistanceLabel);

        SpeedEdit->Text = "100";
        SpeedEdit->Hint = LoadStr(SMetricSpeedHint);
        SpeedLabel->Caption = LoadStr(SMetricSpeedLabel);
    }
    else
    {
        DistanceEdit->Hint = LoadStr(SUsDistanceHint);
    }
}

```

```

    DistanceLabel->Caption = LoadStr(SUsDistanceLabel);

    SpeedEdit->Text = "65";
    SpeedEdit->Hint = LoadStr(SUsSpeedHint);
    SpeedLabel->Caption = LoadStr(SUsSpeedLabel);
}
}

```

The most important part of internationalization (i18N) is resourcing. This means removing all hard coded strings from the application's source code. Traditionally hard coded strings are turned into resources by moving the strings from the actual code into resource strings.

Delphi makes this extremely easy because of its built-in support for resource strings, with the resourcestring clause. It defines one or more resource strings. The resourcestring block contains the resource strings used in the function. If you are not familiar with resource strings in Delphi, refer to the VCL documentation. To put it briefly, you use them almost exactly as you would use string constants.

With C++Builder things are little bit more complicated because you have to use the old-fashioned resource scripts. The following paragraph contains the resource script header file `dcalcres.h`. It specifies the id of each resource string.

C++Builder

```

#define SAboutMsg          0
#define SLanguage         1
#define SDefault          2

#define SMetricDistanceHint 3
#define SMetricSpeedHint   4
#define SMetricDistanceLabel 5
#define SMetricSpeedLabel  6

#define SUsDistanceHint   7
#define SUsSpeedHint      8
#define SUsDistanceLabel  9
#define SUsSpeedLabel     10

#define SInvalidDistance  11
#define SInvalidSpeed     12
#define SCalculateMsg0    13
#define SCalculateMsg1    14
#define SCalculateMsgN    15

```

The resource script file is shown below.

C++Builder

```

#include "dcalcres.h"

STRINGTABLE
BEGIN
    SAboutMsg "Dcalc is a multilingual application that calculates the
average driving time";
    SLanguage "English";
    SDefault "Default";

    SMetricDistanceHint "Give the driving distance in kilometres";
    SMetricSpeedHint "Give the average driving speed in kilometres per
hour";
    SMetricDistanceLabel "in kilometres";
    SMetricSpeedLabel "km/h";

    SUsDistanceHint "Give the driving distance in miles";
    SUsSpeedHint "Give the average driving speed in miles per hour";
    SUsDistanceLabel "in miles";
    SUsSpeedLabel "mph";

    SInvalidDistance "\"%s\" is not a valid distance!";
    SInvalidSpeed "\"%s\" is not a valid speed!";
    SCalculateMsg0 "The average driving time is %d minutes.";
    SCalculateMsg1 "The average driving time is one hour and %d minutes.";

```



```

    SCalculateMsgN "The average driving time is %0:d hours and %1:d
minutes.";
END

```

The second code section in the begin end section of the IniForm function formats the speed and time in locale independent ways. The `Format` and `DateTimeToStr` functions convert the value to a string value using the formatting rules of the current locale.

The next code section sets the caption of the current locale label to match the current locale. The name is given in the native language of the locale.

The next code section sets the caption of the current language label to match the current language. The `SLanguage` resource string contains the name of the language in its native language (e.g. English, Deutch, suomi, svenska, etc).

The last code section sets the initial values, labels and screentips for distance and speed. Metric system uses kilometers and km/h. US system uses miles and mph. Unit `lvl18N` contains the `GetMeasurementSystem` function that returns the measurement system of the current locale.

To do the first initialization we call the initialization function from the `OnCreate` event.

Delphi

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
    InitForm;
end;

```

C++Builder

```

void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    InitForm();
}

```

The `CalculateButtonClick` event needs a little bit more rewriting. Let's study the code that generates the driving distance message:

Delphi

```

'The avarage driving time is ' + IntToStr(hours) + ' hours and ' +
IntToStr(minutes) + ' minutes.',

```

C++Builder

```

"The avarage driving time is " + IntToStr(hours) + " hours and " +
IntToStr(minutes) + " minutes.",

```

This seems to be just ok, but it will actually make the localization hard or even impossible. The reason is that the above logic assumes that the message always starts with the "The average driving time is " string, and then contains the hours, hour label, minutes and the minute label. However, not all languages use the same order of words in a sentence. For example the order might be: minute label, minutes, hour label, hours and the text string. Reordering of the parts of the message is impossible if we use the code shown above.

Fortunately we can use VCL's `Format` function. It uses message pattern that contains placeholders for the dynamic parameters. At run-time the function combines the pattern with the parameters to compose the message. Because the pattern is a single string it can be added to the resource strings, and it can then be translated as a single item. The above code after the internationalization is:

Delphi

```

Format(SCalculateMsg, [hours, minutes]),

```

C++Builder

```

Format(LoadStr(SCalculateMsg), OPENARRAY(TVarRec, (hours, minutes))),

```

`SCalculateMsg` is the pattern and the hours and minutes are parameters.

The next step is to internationalize the calculate event. The following line of code contains the `Calculate` event.

Delphi

```

procedure TMainForm.CalculateButtonClick(Sender: TObject);
resourcestring
    SInvalidDistance = '"%s" is not a valid distance!';
    SInvalidSpeed = '"%s" is not a valid speed!';
    SCalculateMsg0 = 'The average driving time is %d minutes.';

```

```

    SCalculateMsg1 = 'The average driving time is one hour and %d
minutes.';
    SCalculateMsgN = 'The average driving time is %0:d hours and %1:d
minutes.';
var
    str: String;
    distance, speed, hours, minutes: Integer;
begin
    distance := StrToIntDef(DistanceEdit.Text, -1);
    if distance < 0 then
        begin
            MessageDlg(
                Format(SInvalidDistance, [DistanceEdit.Text]),
                mtError,
                [mbOK],
                0);
            DistanceEdit.SetFocus;
            Exit;
        end;

    speed := StrToIntDef(SpeedEdit.Text, -1);
    if speed <= 0 then
        begin
            MessageDlg(
                Format(SInvalidSpeed, [SpeedEdit.Text]),
                mtError,
                [mbOK],
                0);
            SpeedEdit.SetFocus;
            Exit;
        end;

    if GetMeasurementSystem = ivmsUS then
        begin
            distance := Trunc(MILE_IN_METERS*distance/1000);
            speed := Trunc(MILE_IN_METERS*speed/1000);
        end;

    hours := distance div speed;
    minutes := Round(60*(distance mod speed)/speed);

    case hours of
        0: str := Format(SCalculateMsg0, [minutes]);
        1: str := Format(SCalculateMsg1, [minutes]);
    else
        str := Format(SCalculateMsgN, [hours, minutes]);
    end;

    MessageDlg(str, mtInformation, [mbOK], 0);
end;

```

C++Builder

```

void __fastcall TMainForm::CalculateButtonClick(TObject *Sender)
{
    int distance = StrToInt(DistanceEdit->Text);
    if (distance < 0)
    {
        MessageDlg(
            Format(
                LoadStr(SInvalidDistance),
                OPENARRAY(TVarRec, (DistanceEdit->Text))),
            mtError,
            TMsgDlgButtons() << mbOK,
            0);
        DistanceEdit->SetFocus();
        return;
    }

    int speed = StrToInt(SpeedEdit->Text);
    if (speed <= 0)

```

```

{
    MessageDlg(
        Format(
            LoadStr(SInvalidSpeed),
            OPENARRAY(TVarRec, (SpeedEdit->Text))),
        mtError,
        TMsgDlgButtons() << mbOK,
        0);
    SpeedEdit->SetFocus();
    return;
}

if (GetMeasurementSystem() == ivmsUS)
{
    distance = MILE_IN_METERS*distance/1000;
    speed = MILE_IN_METERS*speed/1000;
}

int hours = distance/speed;
int minutes = float(60)*(distance%speed)/speed;

AnsiString str;

switch (hours)
{
    case 0:
        str = Format(LoadStr(SCalculateMsg0), OPENARRAY(TVarRec,
(minutes)));
        break;

    case 1:
        str = Format(LoadStr(SCalculateMsg1), OPENARRAY(TVarRec,
(minutes)));
        break;

    default:
        str = Format(LoadStr(SCalculateMsgN), OPENARRAY(TVarRec, (hours,
(minutes)));
}

MessageDlg(str, mtInformation, TMsgDlgButtons() << mbOK, 0);
}

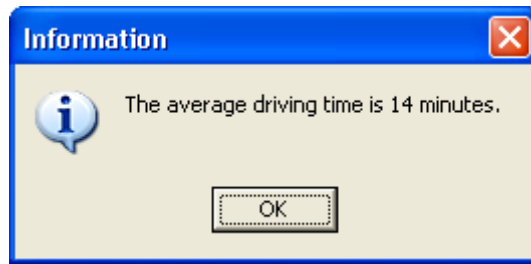
```

The hard coded error message has been replaced with the Format message.

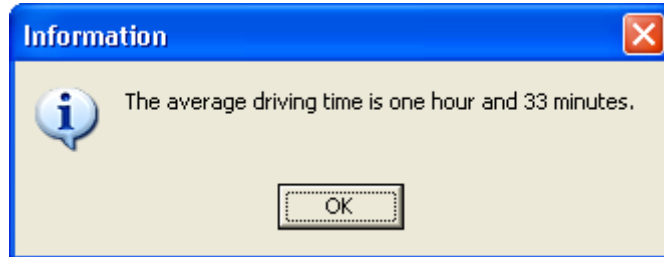
If the selected locale uses miles instead of kilometers we have to treat the value in the edit box as miles. Then we also have to convert distance from miles to kilometers before calculating the driving time. It is always a good idea to internally use the metric system and convert the input and output to the US system when application is run on a US locale, because that makes the calculations easier.

After we have calculated the average driving speed we have to show it to the user. As described previously we are going to use the Format function and message patterns. However we want to make the message grammatically correct. That's why we need three message patters. The first one is for the case when the time is less that an hour, another for the case when the time is between one and two hours, and last for the case when the time is two hours or more. This is because in most languages the single and plural forms are handled in different ways. For example "one hour" vs. "two hours".

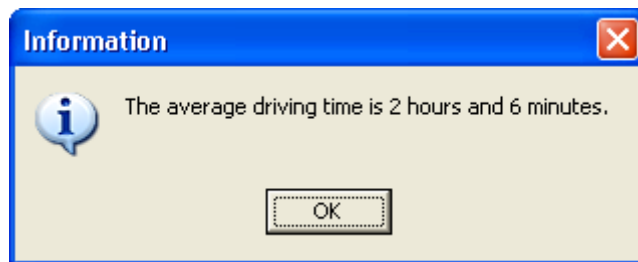
The following figure contains the message when the time is less that one hour. Note that there is no hour string present.



The following figure contains the message when the time is more than one hour but less than two hours. Note that the value for hour is not given as a number but written in letters.



The following figure contains the message when the time is more than two hours. Both hours and minutes are shown as numbers and in plural form.



Even this solution is not perfect because:

- There might be a language that has a specific word for two hours. The above logic assumes that only 0, 1 and 2 or more are handled each in different ways.
- We should use the same logic for minutes as well but this would require 3 by 3 equals 9 message patterns.

The final task left is to resource the message used in the about box:

Delphi

```

procedure TMainForm.AboutMenuClick(Sender: TObject);
resourcestring
    SAboutMsg = 'Dcalc is a multilingual application that calculates the
    average driving time';
begin
    MessageDlg(SAboutMsg, mtCustom, [mbOK], 0);
end;
  
```

C++Builder

```

void __fastcall TMainForm::AboutMenuClick(TObject *Sender)
{
    MessageDlg(
        LoadStr(SAboutMsg),
        mtCustom,
        TMsgDlgButtons() << mbOK,
        0);
}
  
```

Because we set many property values dynamically at run-time the original design time values in the form files become obsolete. They cause no harm but it makes translator's job easier if we remove them. We could set those values to empty but this would make it harder to edit the form files because the labels would no longer be visible. A good solution is to set all dynamic visible property values to "dummy".

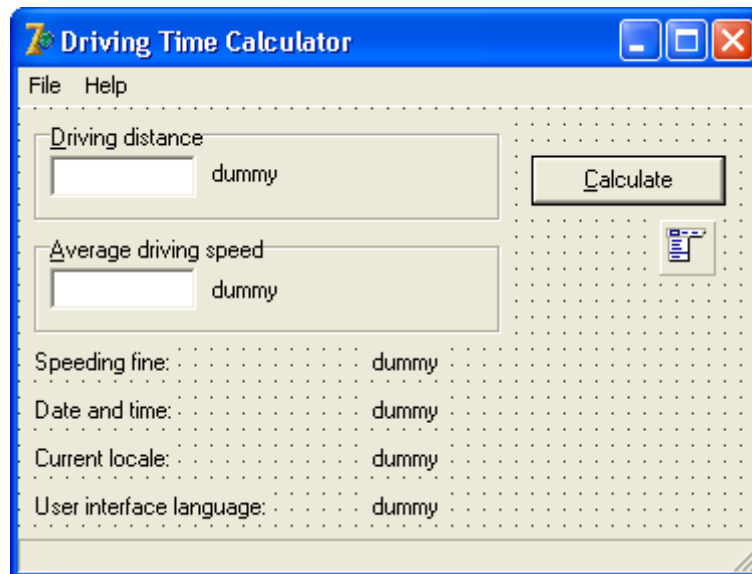


Figure 55 The internationalized Dcalc form on Delphi IDE

Now the Dcalc application has been internationalized. Compile and run it to see that it works before moving on.

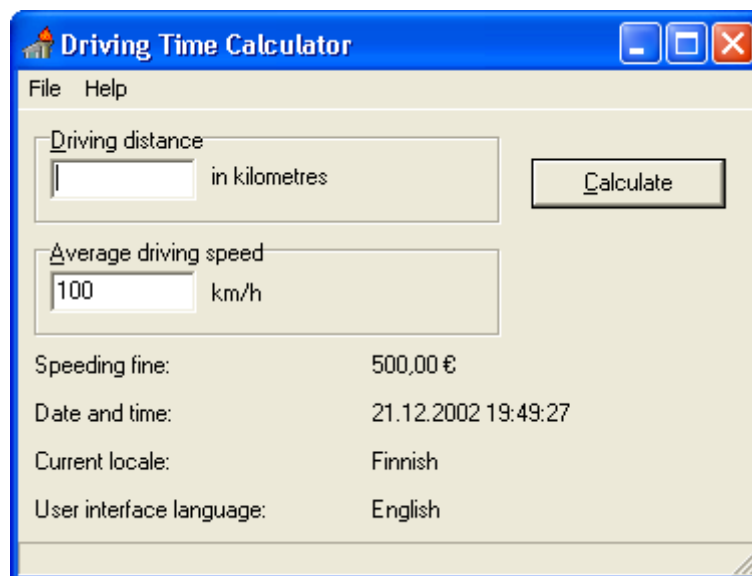


Figure 56 The internationalized Dcalc application running with Finnish locale

This simple internationalization demonstrates three of the most important issues to take into consideration in internationalization: resourcing, dynamic messages and unit conversions. There are quite many other things that you need to know about internationalization as well. Refer to Delphi's online help and/or an I18N book to get more information about internationalization.

The final task is to implement runtime language switch. This can be done if the resource DLLs are used. Add Language... menu to the File menu and write the following code.

Delphi

```
procedure TMainForm.LanguageMenuClick(Sender: TObject);
begin
    if SelectResourceLocale then
    begin
        InitForm;
        SetCurrentDefaultLocaleReg;
    end;
end;
```

C++Builder

```
void __fastcall TMainForm::LanguageMenuClick(TObject *Sender)
{
```

```

    if (SelectResourceLocale())
    {
        InitForm();
        SetCurrentDefaultLocaleReg();
    }
}

```

The `SelectResourceLocale` function shows a dialog box that shows the available resource language and loads the selected resource DLL. This will remove all runtime modifications of the forms. That's why we have to call the `InitForm` function again. Finally we save the selected language to the system registry.

When starting the application VCL is going to select the resource DLL matching to the current locale settings of the user. We want better control over the initial language. The first choice would be a command line parameter (e.g. `dcalc.exe en_US`). If that is not present then we would like use the previous language stored in the system registry under the `HKEY_CURRENT_USER\Software\Borland\Locales` key. This registry key is the build in feature of VCL. Only if that does not exist we would like to use the default language. Add the following code in the initialization part of the main form.

Delphi

```

var
    locale: Integer;
initialization
    if GetCommandLineLocale(locale) then
        SetNewResourceDll(locale);
end;

```

C++Builder

```

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        int locale;
        if (GetCommandLineLocale(locale))
            SetNewResourceDll(locale);

        Application->Initialize();
        Application->CreateForm(__classid(TMainForm), &MainForm);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}

```



NOTE!

We will perform one more task to make the localization easier. When using `resourcestring` clause, Delphi puts the strings to the string resources automatically. However it does not let you to choose what string ids will be used. In addition Delphi will most likely change those ids on next time you compile your application. What remains constant are the resource string names (e.g. `SInvalidDistance`). Unfortunately the compiled binary file (.exe or .dll) does not contain the resource string name but only the string ids. Fortunately it is possible to make Delphi to create a resource string file that contains all the resource strings name and ids used by the application. To create such a file open a Delphi project, choose **Project | Options**, select the Linker tab and check `Detailed` in the Map file radio group. Rebuild the application by choosing **Project | Build DCalc**. Delphi generates the resource string file called `dcalc.drc`.

Creating a Binary Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button.

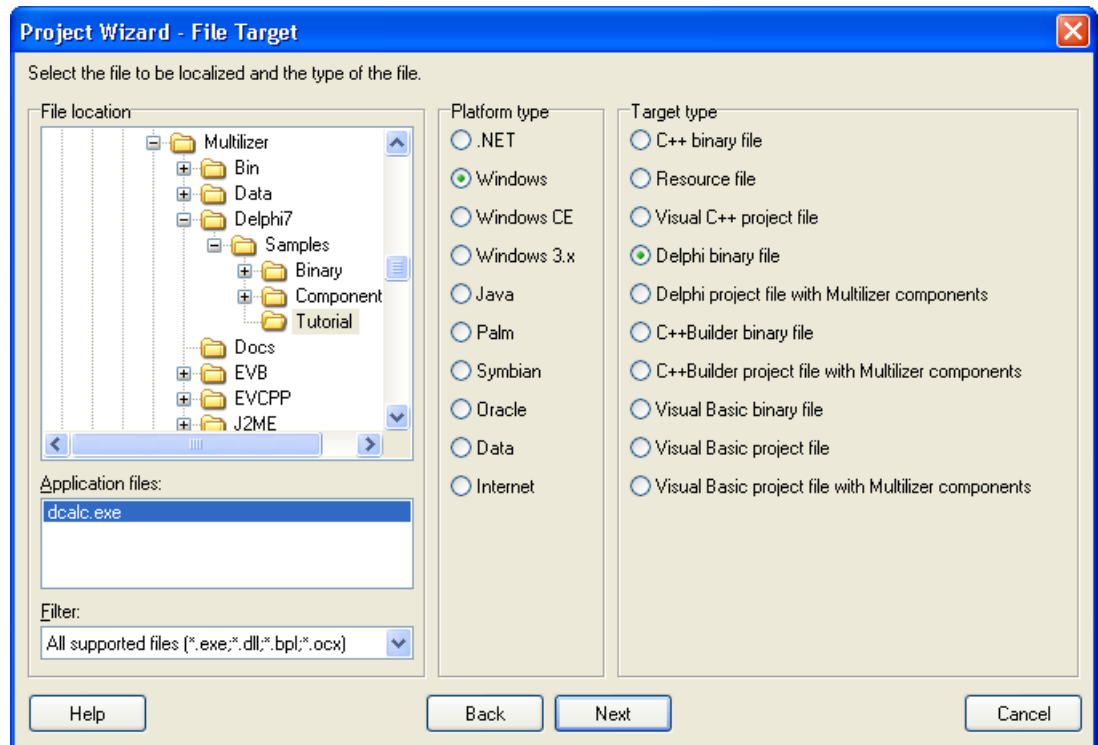


Figure 57 The File Target sheet is used to specify the application file to be localized.

This sheet specifies the location of your application. Choose the <mdir>\<compiler>\Samples\Tutorial subfolder. Project Wizard detects the platform and project types. The Platform type should be *Windows* and the Target type should be *Delphi binary file*. If they are wrong, check the right types.

Press the **Next** button. The VCL Binary sheet appears:

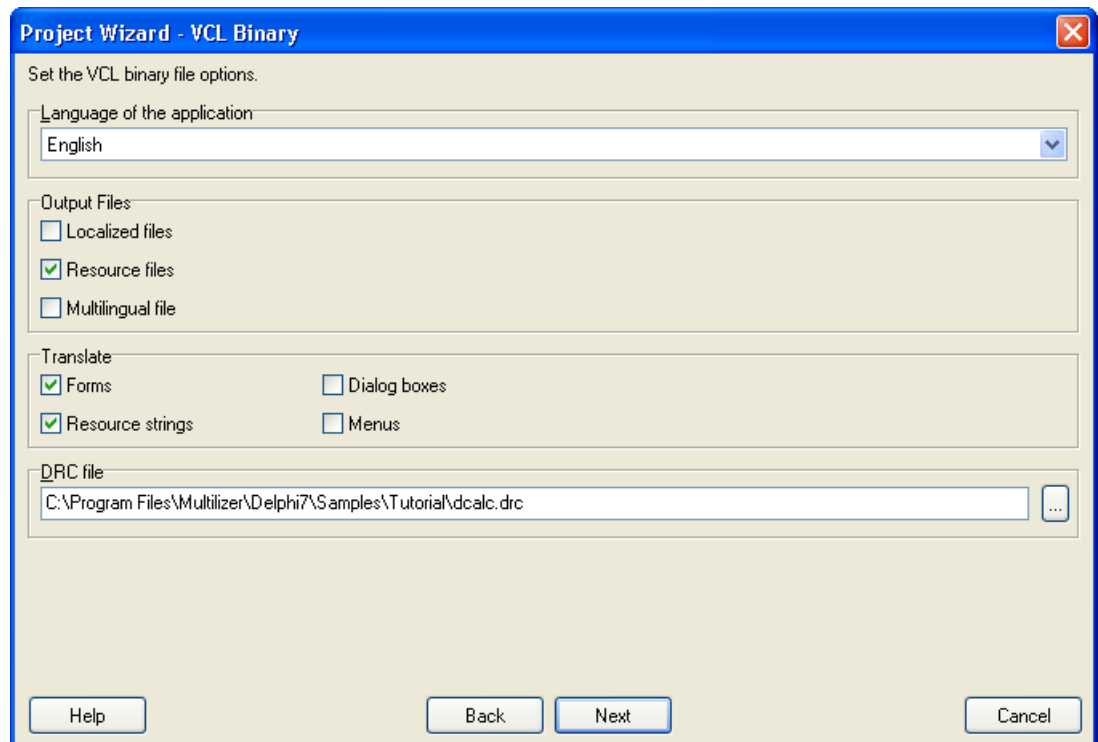


Figure 58 The VCL Binary sheet is used to set VCL specific options.

This sheet specifies the native languages, output types, translated items and the string string file (Delphi). Accept the default values by pressing the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related

information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the >> button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

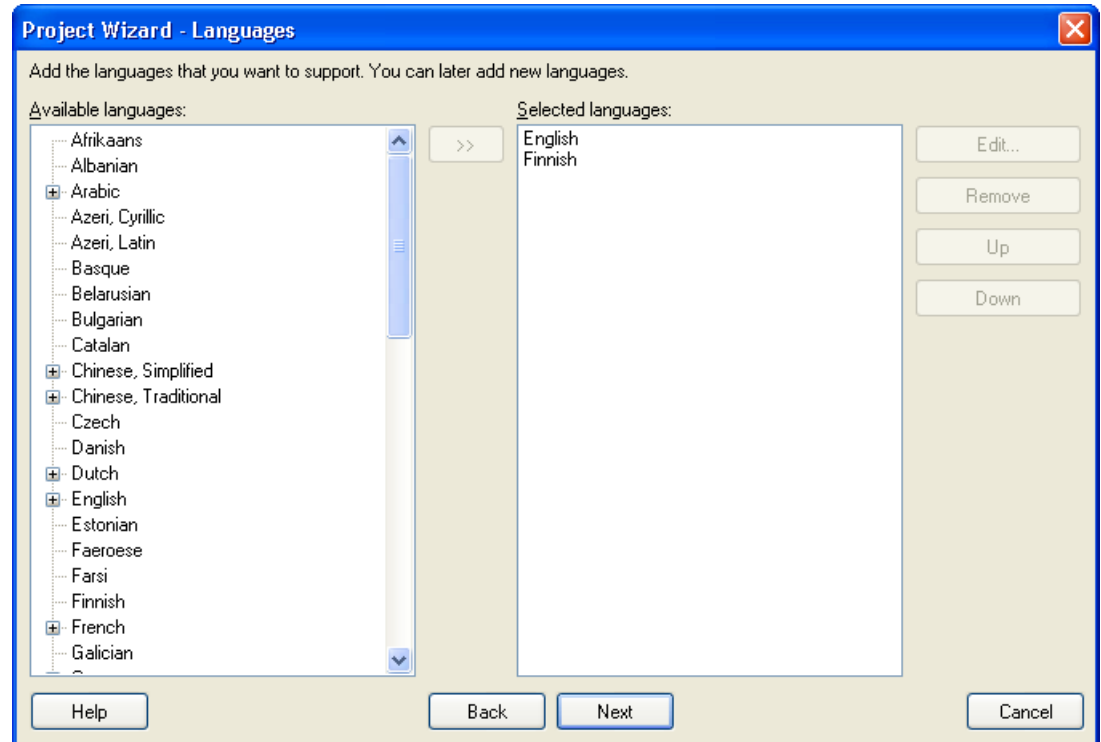


Figure 59 English and Finnish added to a project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

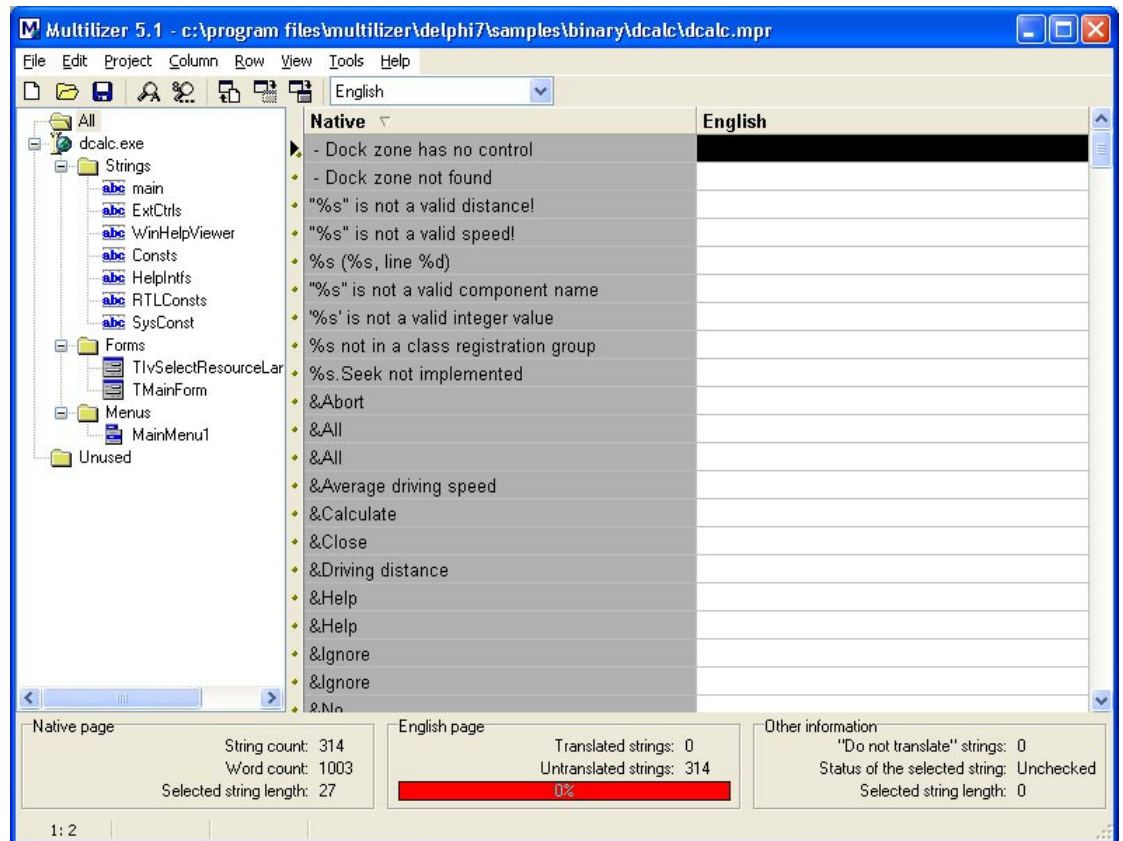


Figure 60 The project grid

On the left side there is a tree view that contains the targets and the items they contains. Our project contains one target, `dcalc.exe`, and it contains several string tables, two forms and one menu. To show only the string belonging to a specific item select the item in the tree. On the right side there is the editing grid. It contains the native column and the English column. To show another language select the language from the combo box above the grid.

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part.

Next step is to create localized application files. You do this by choosing **Project | Build Localized Files** from the menu. This creates the localized resource files (.EN and .FI) in the same folders where the application file is located.

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing **Run**. This is the easiest way to test the localized version.

By default Multilizer creates the localized resource files. You can set it to create also localized application files (e.g. `en\dcalc.exe`) and a multilingual application file (e.g. `all\dcalc.exe`) that contains resources in all languages of the project. Select `dcalc.exe` from the left-hand side tree view and right-click to open the popup menu. Choose **Edit target** to open the Delphi Binary File Target dialog.

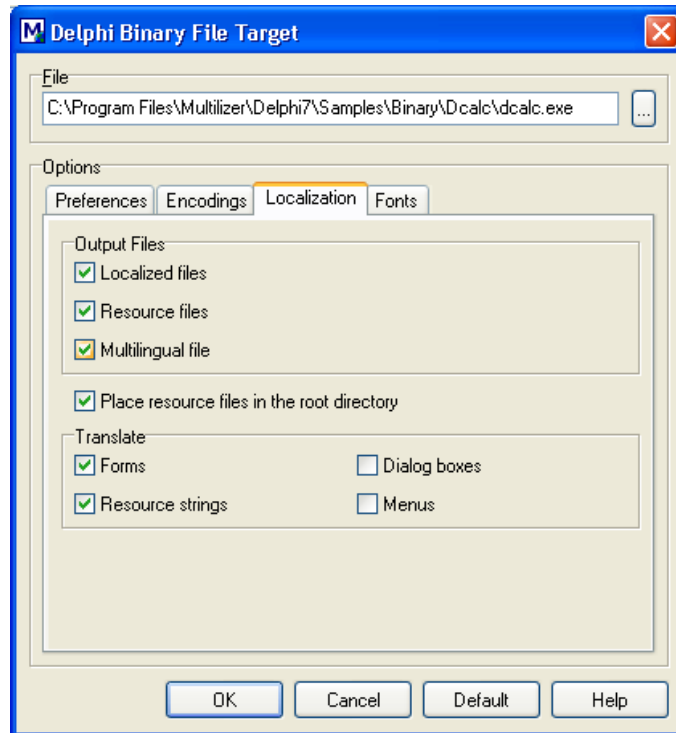


Figure 61 The Delphi Binary File Target dialog

Select the Localization sheet and check the *Localized files* and the *Multilingual file* check boxes. Next time you build localized files, Multilizer also creates the localized application files and the multilingual application file.

For more information about translating a project with Multilizer, see the Translator's Manual.



Integrated Translation Environment

If you have previously used the Integrated Translation Environment (ITE) to localize your application switching to Multilizer is very easy. All you have to do is to create a Multilizer project. When creating the project the following message box will appear.

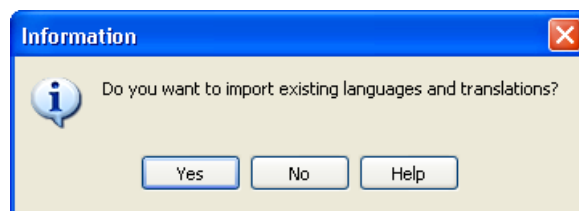


Figure 62 A message box that shows that there are existing ITE translations.

Press **Yes** to import the initial languages and translations from ITE generated resource DLLs. From now on you do not have to use ITE any more. You can delete ITE directories, projects files, and project groups.

Controlling What Properties Are Localized

Multilizer localizes the string properties. An application has string in two different places. The first place are the string resources. These contain the resource strings used by the application. Another place are the forms. These contain string data.

The first way to control the localization is to select what string types are localized. This is done using the target dialog shown in figure 61. By default the Forms and Resource strings check boxes are checked. To disable localization of either type uncheck the check box.

You can control the localization of form strings in more detail. Choose **Tools | Options | VCL** to open the VCL Options dialog. This dialog lets you specify the properties that are not localized (Ignored tab), specify the components that contain binary data (Components tab), and specify the fonts (Fonts tab). For more information about VCL options, see the Online Help by pressing the Help button.

Component Internationalization

This chapter and the following two chapters describe the component localization process. The first step to take is to internationalize the Dcalc application. Let's start by dropping two components on the main form. Select the Multilizer sheet from the Component Palette, and drop TivTranslator and TivTestDictionary components on the form.

The result should look like this:

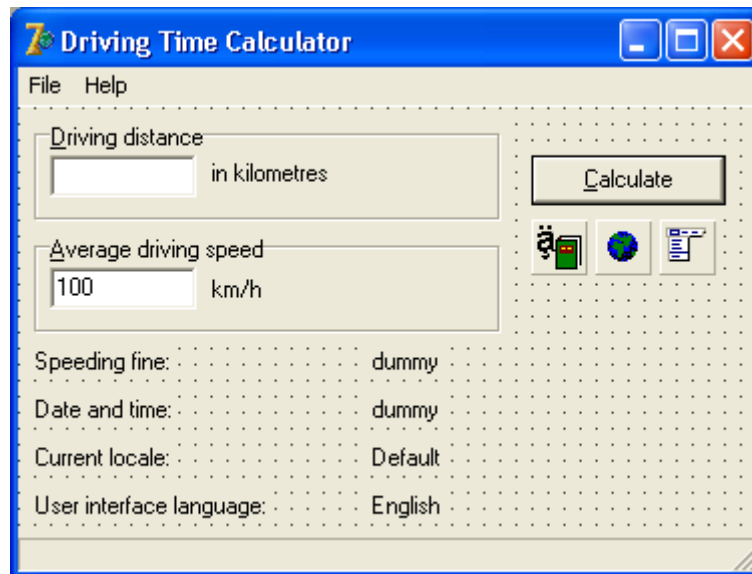


Figure 63 Translator and dictionary components have been added to the form

What are these two components for? TivTestDictionary is one of the dictionary components of Multilizer. A dictionary component provides string or phrase translation for the application. Normally each application contains only one dictionary component connected to one project file, which contains all the translation data of the application. Multilizer contains several different types of dictionary components: one for getting translation data from a text file, another for data from a database, etc.

TivTestDictionary component is a special case. It does not require any actual dictionary data, but it makes the translation on the fly by changing the original string to a test string. You cannot use the test dictionary in your final application because the translation is not a real language, but it is extremely useful in the early development phase.

TivTranslator component is the component that actually does all the work. It scans the form before the form becomes visible, and translates the user interface strings from the original value to the corresponding string of the currently selected language.

Now you can compile and run Dcalc. It should look like this:

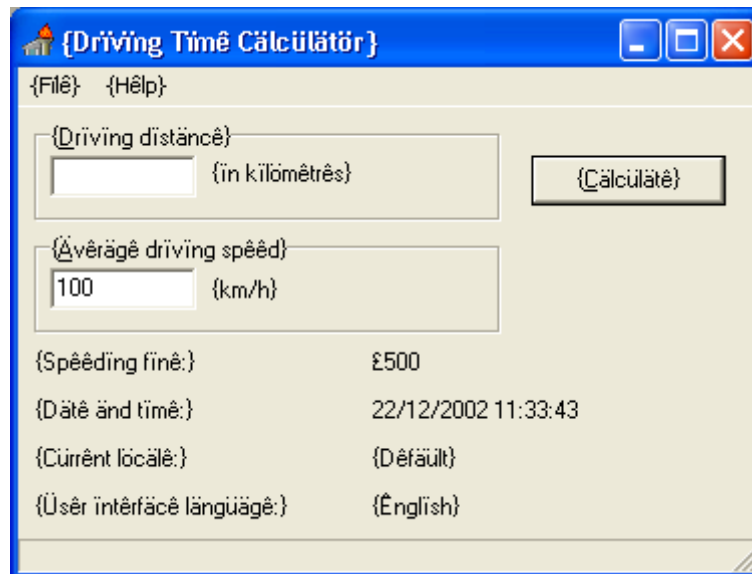


Figure 64 “Translated” applications. The test dictionary translated every string to an upper case string

As you can see, every user interface string has been changed using the pseudo translation algorithm. The translator changed every string type property after the form had been loaded from the resource. By default the test dictionary translates every string by using the pseudo translation algorithm, thus making it easy to see if some part of internationalization is not properly done – for example, strings hard-coded into the source code would not be translated.

Generally you need to add one Dictionary component on the application main form and one Translator component on every form in the application that requires translating. Check the on-line help topic “Translator Usage” for more detailed information on this.

For additional information on using the test dictionary, see the online help topic “TlvTestDictionary”.

This was a quick demonstration of the power of Multilizer. In the next chapter we will create a real dictionary that contains real languages. However, first we have to internationalize the code properly.

If the program contains items, which are country (locale) -specific or hard coded in the source code, they must be removed. This phase is called internationalization: it makes your software language/country independent. The next phase would then be to localize the program, i.e. add for each target country the locale-specific issues.

The Dcalc application calculates the average driving time at a given driving speed. Most countries in the world use the metric system, where the distance is expressed in kilometers. However, for example in the US miles are used instead, and Dcalc needs to be correctly internationalized to be compatible to use correct units.

When a user clicks on the Calculate button, Dcalc calls the following event:

```

procedure TMainForm.CalculateButtonClick(Sender: TObject);
var
    distance, speed, hours, minutes: Integer;
begin
    distance := StrToIntDef(DistanceEdit.Text, -1);
    if distance < 0 then
        begin
            MessageDlg(
                DistanceEdit.Text + ' is not a valid distance!',
                mtError,
                [mbOK],
                0);
            DistanceEdit.SetFocus;
            Exit;
        end;
end;

```

Delphi



NOTE!



MORE INFO

```

speed := StrToIntDef(SpeedEdit.Text, -1);
if speed <= 0 then
begin
  MessageDlg(
    SpeedEdit.Text + ' is not a valid speed!',
    mtError,
    [mbOK],
    0);
  SpeedEdit.SetFocus;
  Exit;
end;

hours := distance div speed;
minutes := Round(60*(distance mod speed)/speed);

MessageDlg(
  'The average driving time is ' + IntToStr(hours) + ' hours and ' +
  IntToStr(minutes) + ' minutes.',
  mtInformation,
  [mbOK],
  0);
end;

```

C++Builder

```

void __fastcall TMainForm::CalculateButtonClick(TObject *Sender)
{
  try
  {
    if (DistanceEdit->Text.Length() == 0)
    {
      DistanceEdit->Text = " ";
      throw Exception("");
    }
    int distance = StrToInt(DistanceEdit->Text);
    if (distance < 0)
      throw Exception("");

    int hours = distance/100;
    int minutes = 0.6*(distance%100);

    MessageDlg(
      Format(
        "The average driving time is %d hours and %d minutes.",
        OPENARRAY(TVarRec, (hours, minutes))),
      mtInformation,
      TMsgDlgButtons() << mbOK,
      0);
  }
  catch (...)
  {
    MessageDlg(
      Format(
        "\"%s\" is not a valid distance!",
        OPENARRAY(TVarRec, (DistanceEdit->Text))),
      mtError,
      TMsgDlgButtons() << mbOK,
      0);
    DistanceEdit->SetFocus();
  }
}

```

When the English (United States) locale is active the user gives the distance in miles. To convert miles to kilometers add the following just before line

```
hours := distance div 100;
```

Delphi

```

if IvDictionary1.LocaleData.MeasurementSystem = ivmsUS then
begin
  distance := Trunc(MILE_IN_METERS*distance/1000);

```

```

    speed := Trunc(MILE_IN_METERS*speed/1000);
end;

```

C++Builder

```

if (IvBinaryDictionary1->LocaleData->MeasurementSystem == ivmsUS)
{
    distance = MILE_IN_METERS*distance/1000;
    speed = MILE_IN_METERS*speed/1000;
}

```

This is enough for the system to convert miles to kilometers but not for the user. The user will most definitely be a bit confused if the user interface still prompts for kilometers. To make the user interface react to the locale change, add the OnLocaleChange event to the dictionary and write the following code:

Delphi

```

procedure TMainForm.IvTranslator1LanguageChange(Sender: TObject);
begin
    if IvDictionary1.LocaleData.MeasurementSystem = ivmsMetric then
        begin
            DistanceLabel.Caption := Translate('in kilometres');
            DistanceEdit.Hint := Translate('Give the driving distance in kilometres');

            SpeedEdit.Text := '100';
            SpeedEdit.Hint := Translate('Give the average driving speed in kilometres per hour');
            SpeedLabel.Caption := Translate('km/h');
        end
        else
            begin
                DistanceLabel.Caption := Translate('in miles');
                DistanceEdit.Hint := Translate('Give the driving distance in miles');

                SpeedEdit.Text := '65';
                SpeedEdit.Hint := Translate('Give the average driving speed in miles per hour');
                SpeedLabel.Caption := Translate('mph');
            end;

            SpeedingFine.Caption := Format('%m', [500.0]);
            CurrentTime.Caption := DateTimeToStr(Now);
            CurrentLocale.Caption :=
IvDictionary1.LocaleData.GetDisplayName(ivdnTranslated, IvDictionary1);
            CurrentLanguage.Caption :=
Translate(IvDictionary1.LanguageData.EnglishName);

            SpeedingFine.Font.Charset := IvLangIdToCharset(IvDictionary1.Locale);
            CurrentTime.Font.Charset := IvLangIdToCharset(IvDictionary1.Locale);

            IvTranslator1.UpdateControls;
end;

```

C++Builder

```

void __fastcall TMainForm::IvBinaryDictionary1LocaleChange (TObject
*Sender)
{
    if (IvDictionary1->LocaleData->MeasurementSystem == ivmsMetric)
    {
        DistanceLabel->Caption = Translate("in kilometres");
        DistanceEdit->Hint = Translate("Give the driving distance in kilometres");

        SpeedEdit->Text = "100";
        SpeedEdit->Hint = Translate("Give the average driving speed in kilometres per hour");
        SpeedLabel->Caption = Translate("km/h");
    }
    else
    {
        DistanceLabel->Caption = Translate("in miles");
        DistanceEdit->Hint = Translate("Give the driving distance in miles");
    }
}

```

```

        SpeedEdit->Text = "65";
        SpeedEdit->Hint = Translate("Give the average driving speed in miles
per hour");
        SpeedLabel->Caption = Translate("mph");
    }

    SpeedingFine->Caption = Format("%m", OPENARRAY(TVarRec, (500.0)));
    CurrentTime->Caption = DateTimeToStr(Now());
    CurrentLocale->Caption = IvDictionary1->LocaleData-
>GetDisplayName(ivdnTranslated, IvDictionary1);
    CurrentLanguage->Caption = Translate(IvDictionary1->LanguageData-
>EnglishName);

    SpeedingFine->Font->Charset = IvLangIdToCharset(IvDictionary1->Locale);
    CurrentTime->Font->Charset = IvLangIdToCharset(IvDictionary1->Locale);

    IvTranslator1->UpdateControls();
}

```

First the code checks the measurement system. Comparing the global variable `MeasurementSystem` does this. The code updates the label and screentip. Let's study the following code in more detail:

Delphi `DistanceLabel.Caption := Translate('in kilometres');`

C++Builder `DistanceLabel->Caption = Translate("in kilometres");`

In a monolingual application you would have used the following code:

Delphi `DistanceLabel.Caption := 'in kilometres';`

C++Builder `DistanceLabel->Caption = "in kilometres";`

This isn't a proper way in a multilingual application because the same EXE file must work correctly on every language and locale. That's why the native string is translated before assigned to the `Caption` property.

The lower part of the event updates the speeding fine, current time, active language name and active locale name.

The monolingual Dcalc contains the following event:

Delphi

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
    Application.OnHint := DisplayHint;

    SpeedingFine.Caption := '£500';
    CurrentTime.Caption := FormatDateTime('dd'/'mm'/'yyyy hh:nn:ss',
Now);
end;

```

C++Builder

```

void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    Application->OnHint = DisplayHint;

    SpeedingFine->Caption = "£500";
    CurrentTime->Caption = FormatDateTime("dd'/'mm'/'yyyy hh:nn:ss",
Now());
}

```

The event is not required any more because the `IvBinaryDictionary1LocaleChange` event updates the labels. You can remove it.

We need to make a few modifications to the `CalculateButtonClick` event to make the message boxes multilingual. Consider the following code:

Delphi

```

MessageDlg(
  Format(
    'The average driving time is %0:d hours and %1:d minutes.',
    [hours, minutes]),
  mtInformation,
  [mbOK],
  0);

```

C++Builder

```

MessageDlg(
  Format(
    "The average driving time is %d hours and %d minutes.",
    OPENARRAY(TVarRec, (hours, minutes))),
  mtInformation,
  TMsgDlgButtons() << mbOK,
  0);

```

Multilizer translates the message dialogs but not the message text. You must use the Translate method to translate it.

Delphi

```

MessageDlg(
  Format(
    Translate('The average driving time is %0:d hours and %1:d
minutes. '),
    [hours, minutes]),
  mtInformation,
  [mbOK],
  0);

```

C++Builder

```

MessageDlg(
  Format(
    Translate("The average driving time is %d hours and %d minutes."),
    OPENARRAY(TVarRec, (hours, minutes))),
  mtInformation,
  TMsgDlgButtons() << mbOK,
  0);

```

Remember that you have to translate the exception message as well.

Delphi

```

MessageDlg(
  Format(
    Translate('"%s" is not a valid distance!'),
    [DistanceEdit.Text]),
  mtError,
  [mbOK],
  0);

```

C++Builder

```

MessageDlg(
  Format(
    Translate("\%s\" is not a valid distance!"),
    OPENARRAY(TVarRec, (DistanceEdit->Text))),
  mtError,
  TMsgDlgButtons() << mbOK,
  0);

```

Translate the message box in the AboutMenuClick event.

Delphi

```

MessageDlg(
  Translate('Dcalc is a multilingual application that calculates the
average driving time'),
  mtCustom,
  [mbOK],
  0);

```


C++Builder

```

MessageDlg(
    Translate"Dcalc is a multilingual application that calculates the
    average driving time"),
    mtCustom,
    TMsgDlgButtons() << mbOK,
    0);

```

Assign the `IvBinaryDictionary1LocaleChange` event to the `OnLanguageChange` event as well. This is because the above messages need to be updated every time either the active language or active locale changes, as these can be changed independently.

Finally, you need to add the `IvMlUtils` unit to the `uses` clause of the implementation part:

Delphi

```

uses
    IvMlUtil;

```

C++Builder

```

#pragma package(smart_init)
#pragma link "IvMlUtil"

```

Creating a Component Project

Creating a project for an application using components is fairly similar to the binary project. The only differences are that you have to select a different target type and specify the dictionary.

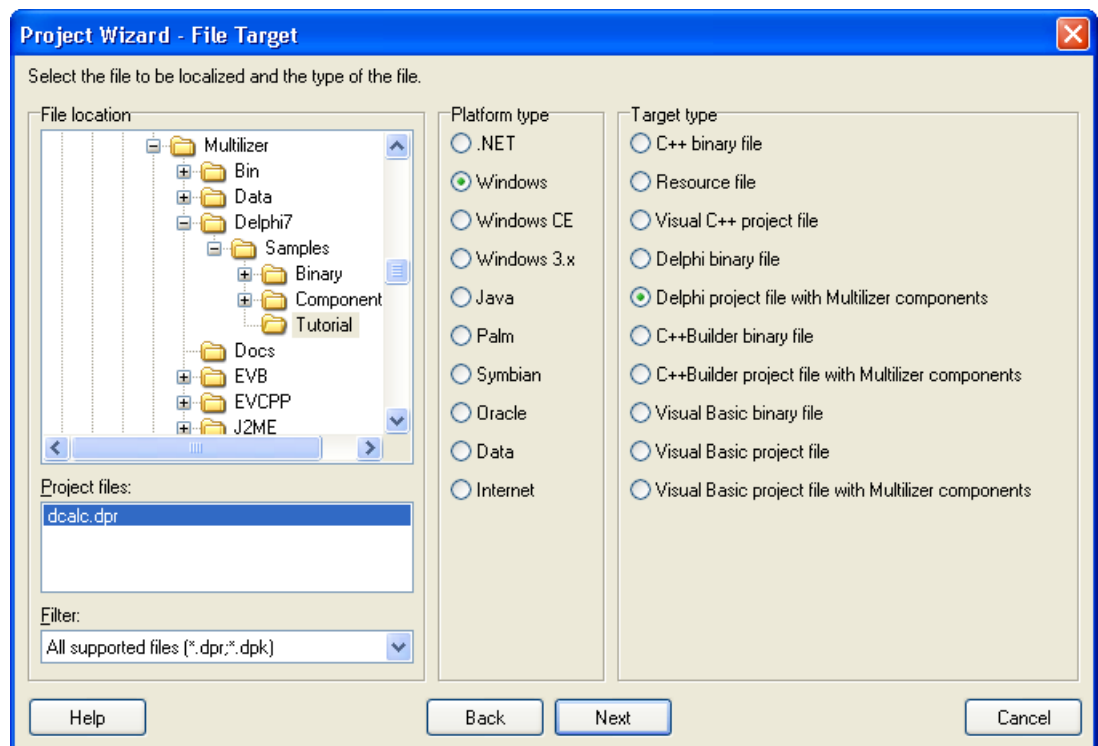


Figure 65 The Select Target sheet is used to specify the project file to be localized.

A few sheets later the wizard shows the Dictionary sheet that specifies the type of runtime dictionary to be used by the dictionary component:

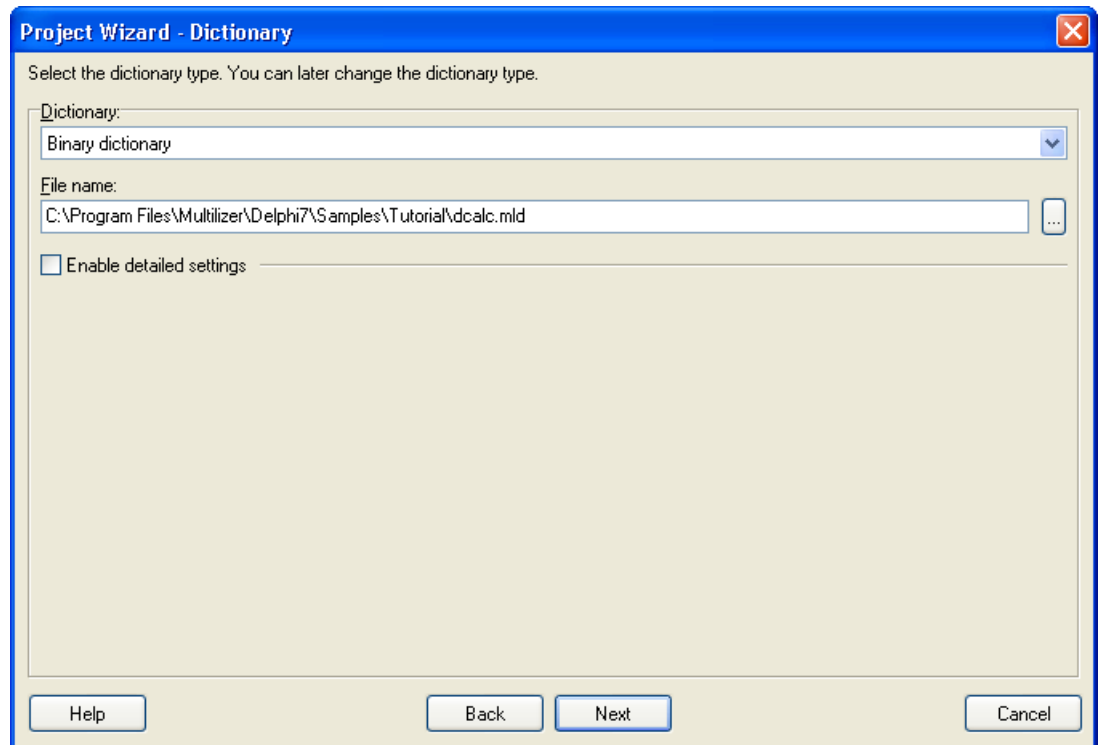


Figure 66 Dictionary sheet is used to specify the dictionary type

This sheet specifies the type of dictionary. The online help describes each dictionary type. Accept the default dictionary (Binary file) by pressing the **Next** button. The rest of the wizard is similar to binary localization.

We recommend that you use the Binary dictionary, unless you specifically need to use another type.

Now you can translate the project. When you are done translating, save the project and create the run-time dictionary by selecting **Project | Build Localized Items** from the menu.

Using Run-time Dictionary

Open the Tutorial application in Delphi or C++ Builder. Add the TlvBinaryDictionary component to the main form. If you already have the TivTestDictionary component on the form, remove it now. Choose the BinaryDictionary component and move to the Object Inspector. Set the FileName property to `dcalc.mld` which you created in previous lesson.



NOTE!

If you pressed the ... button and browsed the file name, the Object Inspector would add the full file name and path (e.g. `C:\Program Files\Multilizer\Delphi7\Samples\tutorial\dcalc.mld`). You should avoid using absolute paths, as they may cause problems when deploying your application, so remove the path part from the file name now.

Later you will learn how to include the dictionary file in the EXE file so that there is no need to deploy it as a separate file. The Object Inspector should look like this:

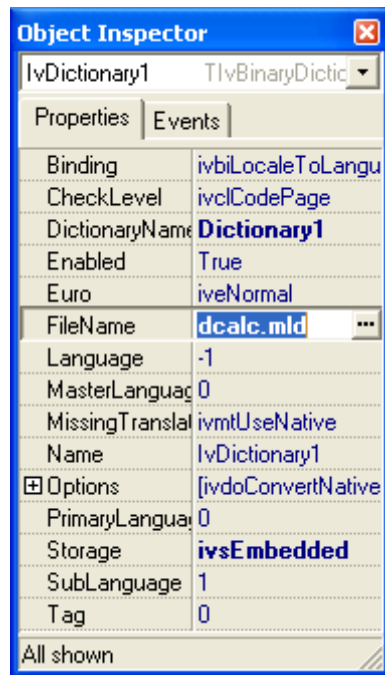


Figure 67 Object Inspector showing the properties of a binary dictionary component

Let's study some of the properties. The Language property specifies the active language. It is now -1. This makes Multilizer check the current locale of the user's operating system, and find the language that matches the locale. If the correct language is not found, the first (non-native) language is used. Therefore it is recommended that you would always have English language as the first language in your dictionary – it is supported by every Windows version, so your application can safely use it as the default language.

The PrimaryLanguage and the SubLanguage properties specify the active locale. The active language determines the language of the user interface. The active locale, however, determines the locale used by the application. The locale is a country and language specific object that controls how the date, time, currency, number, etc. are formatted.

The PrimaryLanguage property is 0 and the SubLanguage property is 1. These make Multilizer use the default locale of the user. See more in the online documentation under the topic TlvDictionary.

In our case the project file contains English and Finnish. If the locale setting of the user is Finnish (Finland) the user interface of Dcalc will be in Finnish and the locale will be Finnish (Finland).



NOTE!

One important property is Storage. It specifies how the dictionary is used at run-time. In this example, the dictionary will be in an external file also at run-time. The dictionary could also be embedded in the EXE file, or stored in resource data.

Now you can run the application. If you have translated it completely to Finnish, it should look like this:

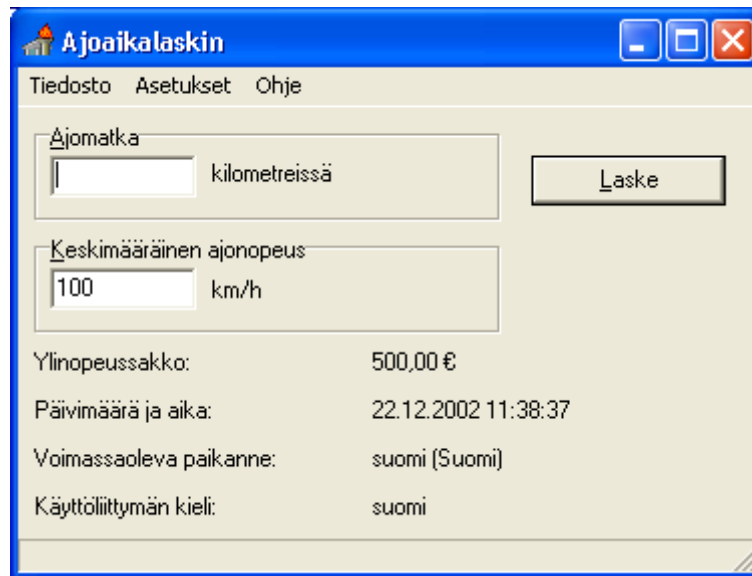


Figure 68 Dcalc in Finnish

Dcalc now has the ability to adapt to the current language and locale settings of the user. What about changing the language and/or locale at run-time? Instructions on how to do that below. Close the Dcalc application and return to Delphi/C++ Builder.

Double click the MainMenu1 component. The menu editor appears. Add a Language... menu item to the Language menu.

The result should look like this:

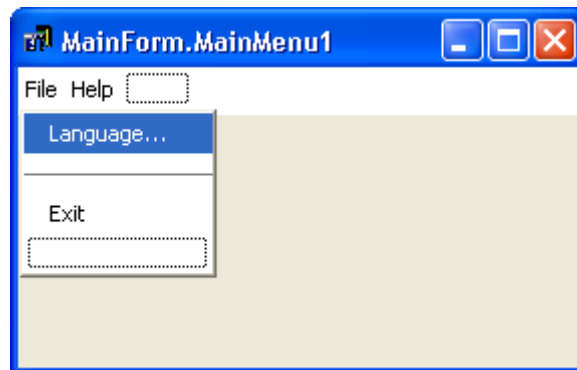


Figure 69 Add the Language menu item to the main menu

Write the following event handler to the Language... menu item:

Delphi

```
procedure TMainForm.LanguageMenuClick(Sender: TObject);
var
    language: Integer;
begin
    if SelectLanguage(language) then
        IvBinaryDictionary1.Language := language;
end;
```

C++Builder

```
void __fastcall TMainForm::LanguageMenuClick(TObject *Sender)
{
    int language;

    if (SelectLanguage(language))
        IvBinaryDictionary1->Language = language;
}
```

The SelectLanguage shows the Language Select dialog box of Multilizer. It contains a list of available languages that the user can select. After calling the SelectLanguage function

the event sets a new active language by setting the Language property of the dictionary component.

Add the `IvLanguD` unit to the uses clause of the implementation part. This is because the `SelectLanguage` function locates on that unit.

Run the application and choose `Language | Language...` (Or `Kieli | Kieli...` if you have Finnish active). The following dialog box appears:

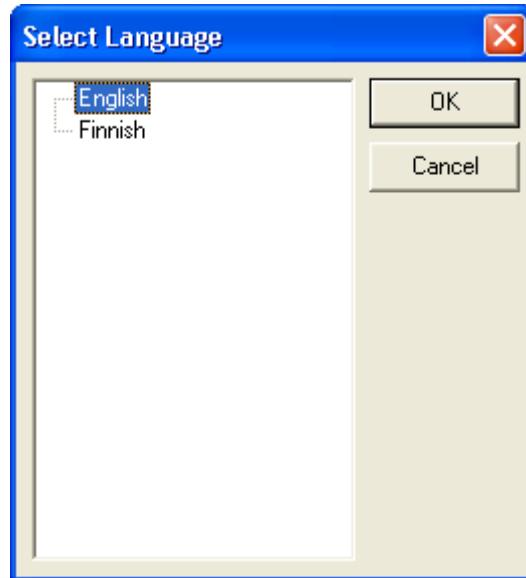


Figure 70 *Select Language dialog box lets the user select the active language at run-time*

The `SelectLanguage` dialog box shows all available languages in a tree view. There may be more languages in the project file than are shown in the available languages list. It may be that your operating system cannot cope with all character scripts of the languages in the dictionary, and by default such languages are not shown in the list.

For example you need to have a bi-directional OS to properly display Arabic or Hebrew. Also most Western OS versions lack support for Cyrillic or Far Eastern languages. You can make the dictionary component and `SelectLanguage` function display every language by setting the `CheckLevel` property of the dictionary component to `ivclNone`.

Now select a new language from `Dcalc` and click the `OK` button. The active language (user interface) of `Dcalc` changes to that language. By default the active locale also changes to the default locale of the language. You can set the active language and locale independently by removing the `ivdoBindLocale` from the `Options` property of the dictionary component.

The `SelectLanguage` dialog box itself contains strings that need to be translated as well. Also the `IvMessageBox` uses several strings (e.g. "OK", "Cancel"). The master string table of the Multilizer contains all these constant strings. All you need to do is to add them to your project file.

If the Multilizer is not running, start it now. Open the project file, and choose **Project | Include | System Strings**. The System Strings dialog box appears. Check `Language Dialog`, `Message Box`, and `System Menu` check boxes.

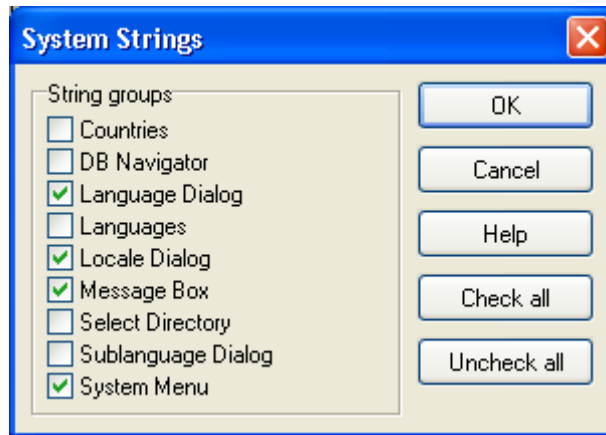


Figure 71 The System Strings dialog box lets the developer add strings used by the system or by the standard components

Press the OK button. Multilizer adds the strings used by the Select Language dialog, Message box and the system menu. Now translate these strings. For more information about translating the project, please refer to chapter 7 in this manual.

Now the Dcalc application is fully multilingual. The user interface, locale settings and input measures match the local settings. The user can even change the language at run-time.

10

Java

In this tutorial we are going to create a multilingual application with Multilizer. First we use java resource bundles and then we show how to do the same thing with Multilizer java beans. The application will be a simple driving-time calculator, Dcalc that a user can use to calculate the average driving time for a given distance.

This tutorial is written for JBuilder 7. With each instruction there is also an explanation how to accomplish it using the plain JDK. If you use some other Java IDE (e.g. Sun ONE Studio, Forté, Visual Café, Visual Age, PowerJ, etc.) you can easily modify the procedure to match your Java IDE.



This symbol indicates that the information given applies to JBuilder only. In the front of a header it applies to the whole chapter, otherwise it applies to the current paragraph.

This symbol indicates that the information given applies to plain JDK only. In the front of a header it applies to the whole chapter, otherwise it applies to the current paragraph.

To see more about how to use Multilizer read the online help and study the other sample applications found in the `samples` subfolder.

Opening a Monolingual Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. This is what we are going to do. The `samples\tutorial` contains the English Dcalc. Open it, compile it, and finally run it.

The application should look like this:

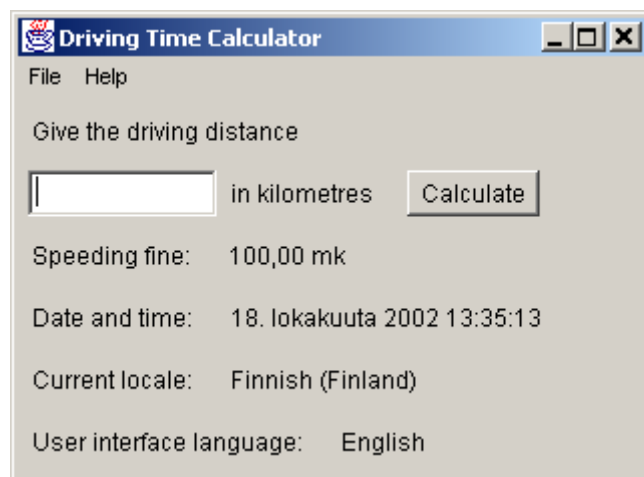


Figure 72 The monolingual application using an English user interface

The user interface language is English and the application uses the default locale, which in this case is Finnish (Finland). The speeding ticket is formatted using the Finnish currency format (mark) and the date and time is also formatted using the Finnish format. Java enables this type of localization automatically.

In the following chapters we will make Dcalc truly multilingual step-by-step. First by using Java's resource bundles and then by using Multilizer beans. Use the method of your

preference. See online help topic "Multilizer Components vs. Resource Bundles" for comparison between resource bundles and Multilizer components.

Resource Bundle Localization

Java standard edition provides support for internationalization in `java.util` and `java.text` packages. Localization is mainly done through resource bundles. This chapter covers only the basic internationalization (I18N). Prefer I18N books and web sites to get more information about. An excellent start is the Java tutorial at <http://www.javasoft.com/>.

The following picture describes the Java standard edition localization process with resource bundles and Multilizer.

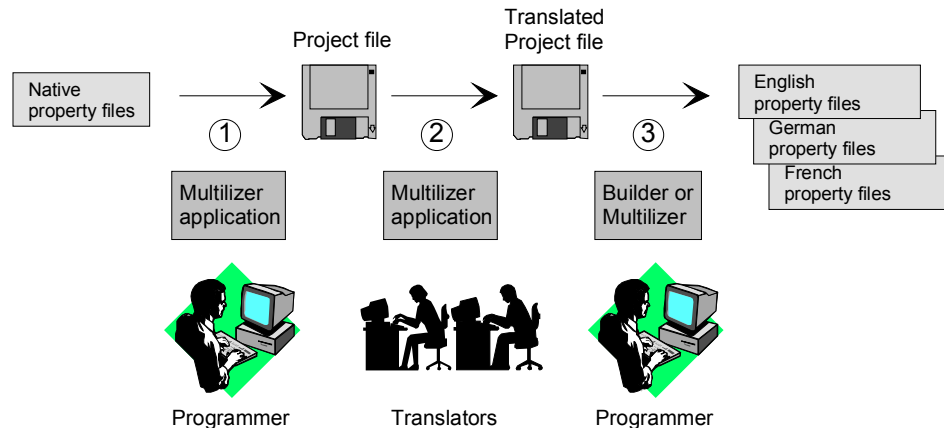


Figure 73 Java localization process with resource bundles

The programmer uses Multilizer to extract strings from the original resource bundle (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource bundles (3). As the result there will be one resource bundle for each localized language.

The following figure shows the files that Multilizer uses in the Java standard edition localization process.

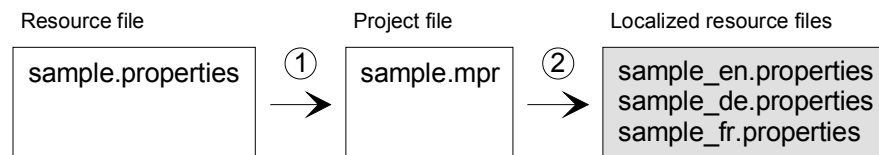


Figure 74 The files of the Java localization process with resource bundles

Resource Bundle Internationalization

To globalize your application you have to resource the application. Resourcing means removing hard coded strings. Most applications contain strings that have been inserted inside the source code. These strings are hard coded. It is impossible to localize such a code without changing and recompiling the code. When you resource the code you will take the strings from the source code and place to a resource bundle that can be easily translated.

The main source code file of your application is `tutorial/MainFrame.java`. It contains several hard coded strings. For example:

```
fineLabel.setText("dummy");
label4.setText("Speeding fine:");
label5.setText("Date and time:");
dateLabel.setText("dummy");
label7.setText("Current locale:");
localeLabel.setText("dummy");
```



```
label9.setText("User interface language:");
languageLabel.setText("English");
```

Now we have to extract all the hard coded strings to a resource bundle. In this example we use `dcalc.properties` property resource bundle.



In JBuilder you can create the resource bundle easily by using a wizard: **Wizards | Resource Strings**. The wizard scans your source code, creates the bundle and makes the necessary modifications in your source code.

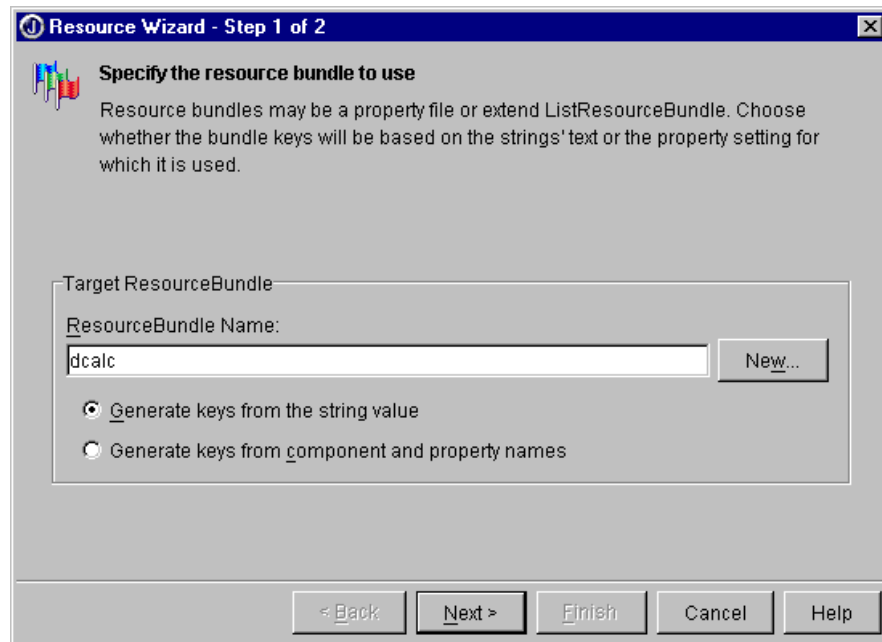


Figure 75 JBuilder Resource wizard

Press New and change the Name to `dcalc` and Type to `PropertyResourceBundle`. Press OK. Press Next twice. The wizard extracts strings from the source code. Press Finish to complete the wizard.



In plain JDK create the `PropertyResourceBundle` by hand and take it in use.

```
public class MainFrame extends Frame
{
    static ResourceBundle res = ResourceBundle.getBundle("dcalc");
```



Wrap all hard coded strings inside the `res.getString()` method. Add all keys and their native translations to the bundle. Remember that key values in a bundle aren't allowed to contain e.g. space characters and you have to use Unicode escapes for non-ASCII characters.

```
fineLabel.setText(res.getString("dummy"));
label4.setText(res.getString("Speeding_fine_"));
label5.setText(res.getString("Date_and_time_"));
dateLabel.setText(res.getString("dummy"));
label7.setText(res.getString("Current_locale_"));
localeLabel.setText(res.getString("dummy"));
label9.setText(res.getString("User_interface"));
languageLabel.setText(res.getString("English"));
```

The last String ("English") should not be localized by translating it in a resource bundle. Remove the line and add the following code to the constructor of `MainFrame`.

```
// Update the user interface language
languageLabel.setText(res.getLocale().getDisplayLanguage());
```

Use also elsewhere in the MainFrame `Locale.getDefault()` instead of `res.getLocale()`. For example:

```
// Update the locale label
localeLabel.setText(res.getLocale().getDisplayName());
```



`ResourceBundle.getLocale()` is supported only by JDK 1.2 or later. With JDK 1.1.8 you have to use `Locale.getDefault()` instead of `res.getLocale()`.

Creating a Resource Bundle Project

Double click the Multilizer icon from the Multilizer program group to launch Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

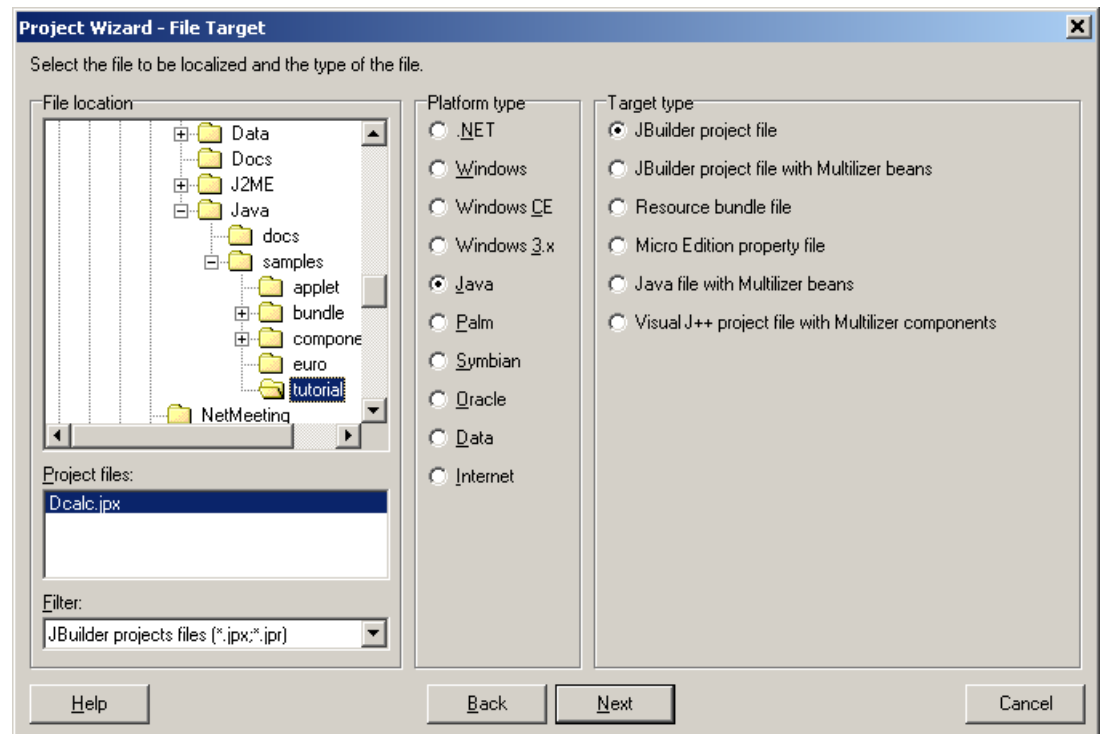


Figure 76 The Select Target sheet is used to specify the resource file to be localized.

This sheet specifies the directory where your application is located. Choose the `<mldir>\java\samples\tutorial` subfolder of your Multilizer setup. Project Wizard detects the application type. The Platform type should be *Java* and the Target type should be *JBuilder project file*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the `>>` button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

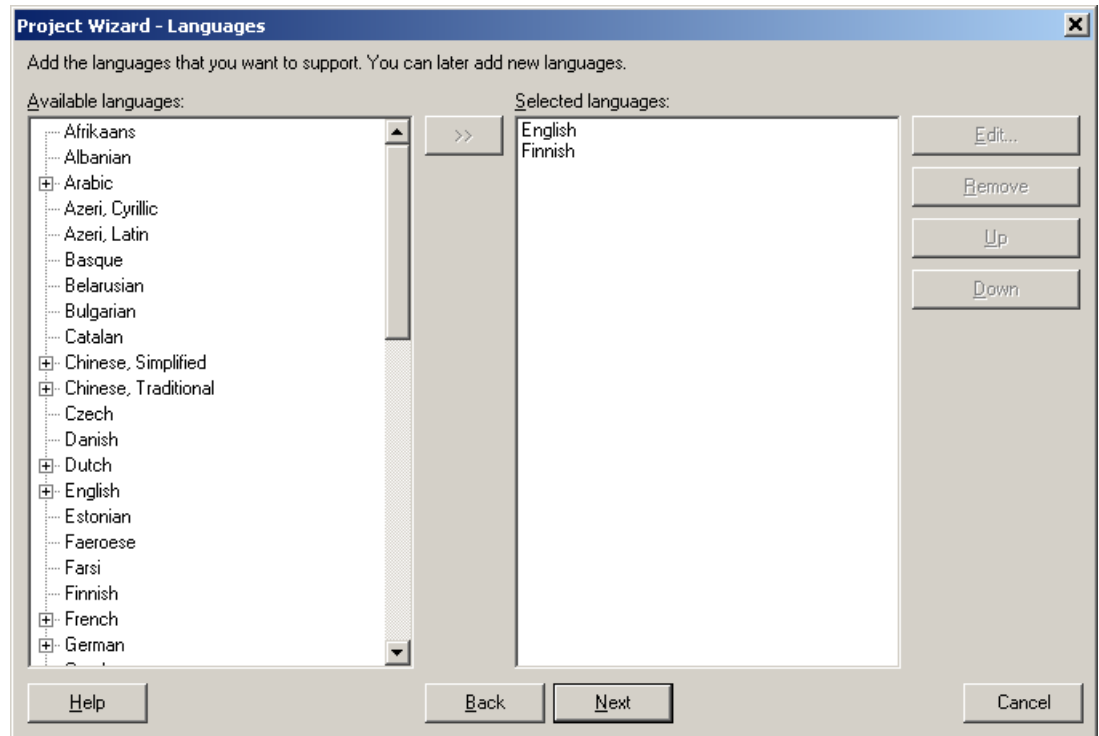


Figure 77 English and Finnish added to the project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

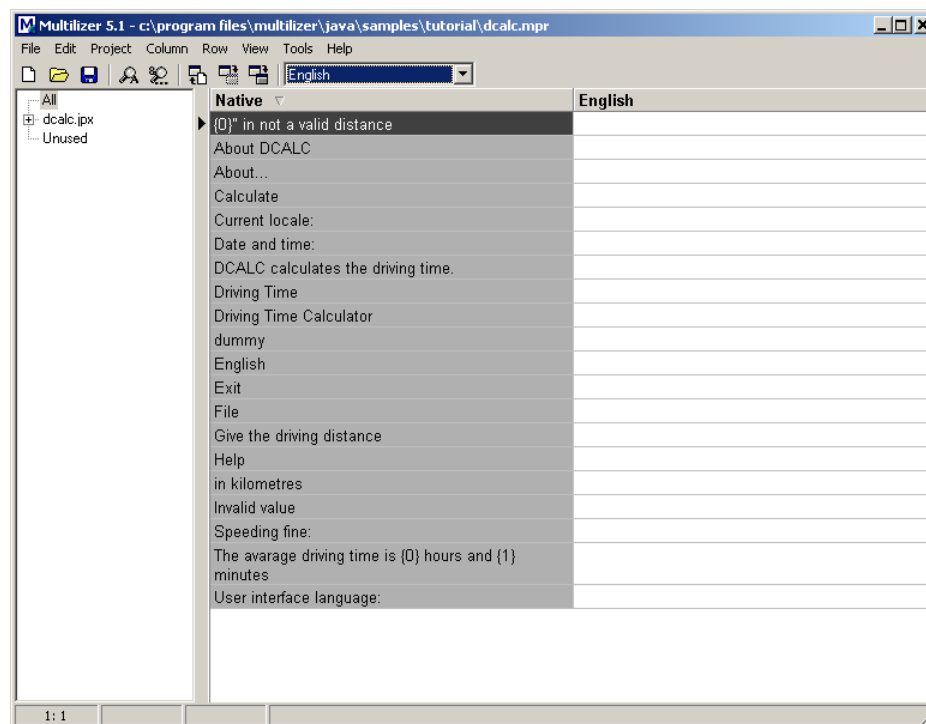


Figure 78 The translation grid

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part.

Create localized resource bundles by choosing **Project | Create Localized Items**.

Now all the strings in Dcalc are localized and next time you start it, you get the system default language version (if you translated the corresponding language).

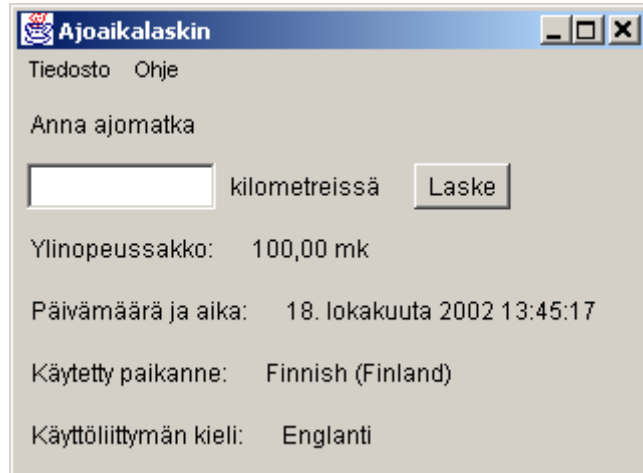


Figure 79 Localized Dcalc application

Localization with Multilizer Components

Now the same Dcalc application will be localized with Multilizer beans. The Dcalc application is very simple but still it uses most of the features of Multilizer beans. The creation of the application is divided into several lessons each covering one or more Multilizer feature.

Before you can start building multilingual Dcalc, you have to install Multilizer beans. To get the information on how to install them, see the help files. Double-click the *Java Help* icon of your Multilizer program group for more information.

Making the Application Multilingual



The first step is to make Dcalc multilingual, by just dropping two components to the form. Select the Translator component from the Component Palette. Drop the `multilizer.Translator` to the form. Drop the `multilizer.TestDictionary` component to the form as well.



The result should look like this:

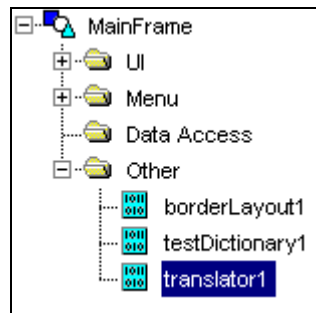


Figure 80 Translator and dictionary components have been added to the form

Add the following code below the import lines:

```
import multilizer.*;
```





Add the following code just before the constructor of MainFrame:

```
TestDictionary testDictionary1 = new TestDictionary();
Translator translator1 = new Translator();
```

What are these two components for? TestDictionary is one of the dictionary components of Multilizer. A dictionary component provides a string or a phrase translation for the application. Normally each application contains one dictionary component that contains all the translation data of the application. Multilizer contains several different dictionary components: one for getting the translation data from a text file, another for getting data from a database, etc.

The TestDictionary is a special case. It does not require any dictionary data but it makes the translation on-the-fly by changing the original string to a test string. In a normal case you can not use the test dictionary in your final application because the translations are not any real language. However, the test dictionary is really handy in the development phase.

The Translator on the other hand is the component that does all the work. It scans the form before it becomes visible and translates the user interface string from the original value to the current language.



Select *translator1* component and move it to the Properties sheet. Drop down the value list of the *host* property. Select *this*. This specifies the control that the translator should translate. In the most cases it is the frame containing the translator component.



Add the following code to the `jbInit` function:

```
translator1.setHost(this);
```

Add the `translator1.translate()` line to the constructor of the MainFrame:

```
public MainFrame()
{
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try
    {
        jbInit();

        fineLabel.setText(NumberFormat.getCurrencyInstance(
            Locale.getDefault()).format(new Integer(100)));

        dateLabel.setText(DateFormat.getDateTimeInstance(
            DateFormat.LONG,
            DateFormat.MEDIUM,
            Locale.getDefault()).format(new Date()));

        localeLabel.setText(Locale.getDefault().getDisplayName(Locale.UK));

        translator1.translate();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

The `translate` method makes the translator to translate its host control. A proper place to call this is the last line of the constructor.

Compile and run Dcalc. It should look like this:

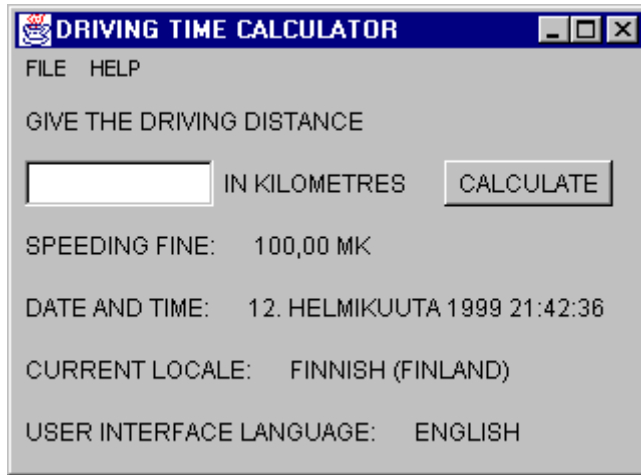


Figure 81 "Translated" application. The test dictionary translated every string to upper cased string

As you can see, every user interface string is now in upper case. The translator changed every string type property after the form had been loaded from the resource. By default the test project translates every string by putting it in upper case.

For additional information on using the test dictionary, see the online help topic "TestDictionary".

This was a quick demonstration of the power of Multilizer. In the next chapter we will create a real project that contains real languages.



Creating a Component Project

Creating a project for an application using components is fairly similar to the binary project. The only differences are that you have to select a different target type and specify the dictionary.

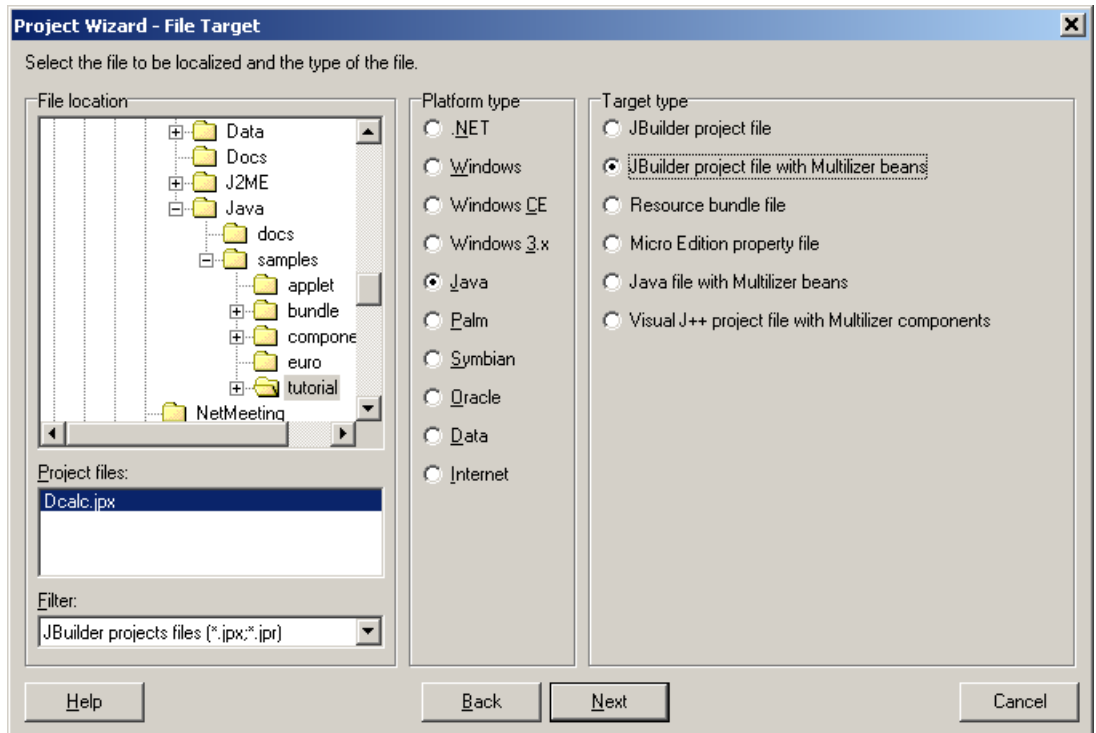


Figure 82 The Select Target sheet is used to specify the project file to be localized.

A few sheets later the wizard shows the Dictionary sheet that specifies the type of run-time dictionary to be used by the dictionary component:

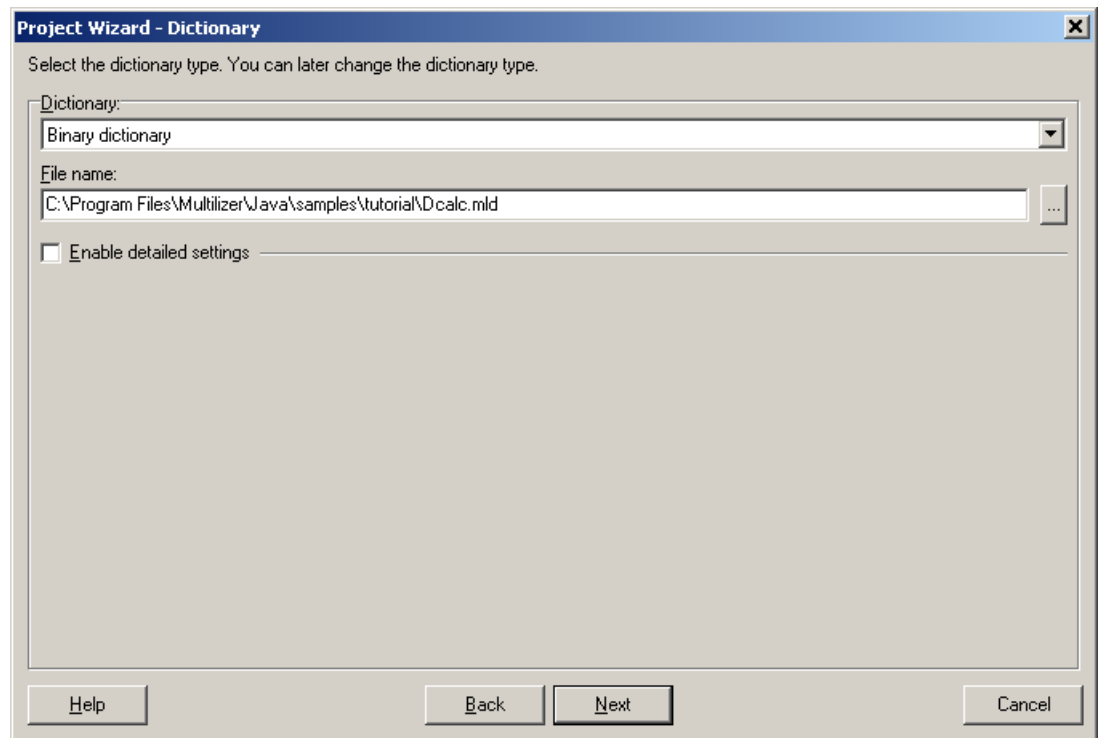


Figure 83 Dictionary sheet is used to specify the dictionary type

This sheet specifies the type of dictionary. The online help describes each dictionary type. Accept the default dictionary (Binary file) by pressing the **Next** button. The rest of the wizard is similar to binary localization.

We recommend that you use the Binary dictionary, unless you specifically need to use another type.

Now you can translate the project. When you are done translating, save the project and create the run-time dictionary by selecting **Project | Build Localized Items** from the menu.

Component Internationalization



We have the project file now. Let's use them. Delete the TestDictionary component from the form. Add the `multilizer.BinaryDictionary` component. Choose the component and move to the Properties sheet. Set the `fileName` property to `dcalc.mld` (i.e. the dictionary that you created in previous lesson). Set the `name` property to `dictionary1`.



The JBuilder window should look like this:

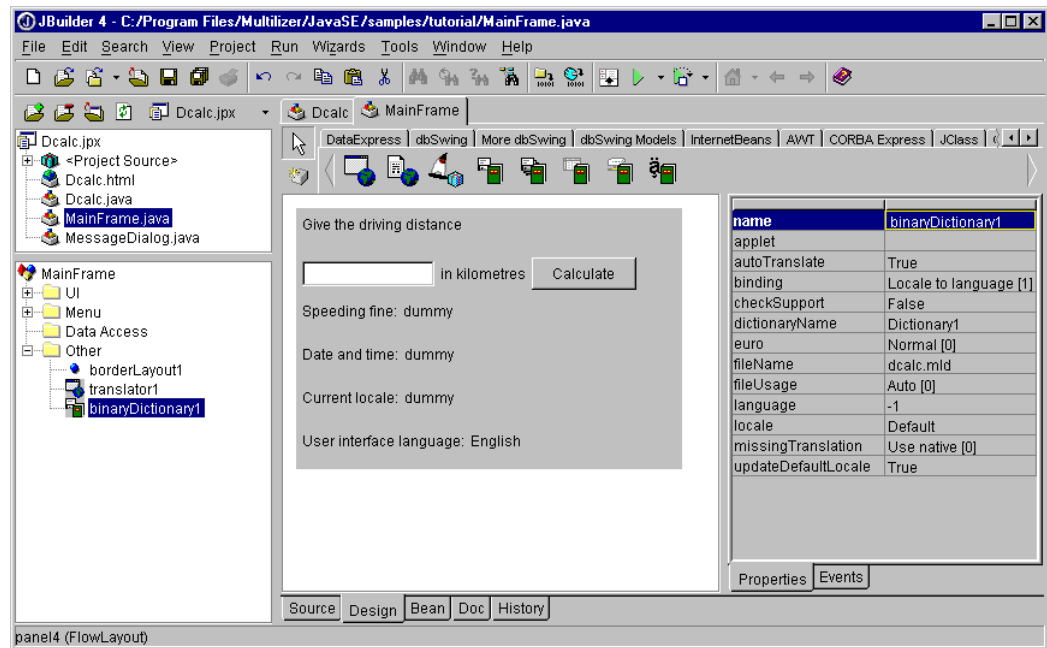


Figure 84 The Properties window showing the properties of a binary dictionary component

Let's study some of the properties. The *language* property specifies the active language. By default it is `-1`. This makes Multilizer check the current locale of the user and find the language that matches the locale. If none is found the first (non-native) language is used.

The *locale* property specifies the active locale. The active language determines the language of the user interface. The active locale, however, determines the locale used by the application. The locale is a country and language specific object that controls how the date, time, currency, number, etc. are formatted.

In our case the project file contains English and Finnish. If the locale setting of the user is Finnish (Finland) the user interface of Dcalc will be in Finnish and the locale will be Finnish (Finland).



Add the following code just before the constructor of MainFrame:

```
BinaryDictionary dictionary1 = new BinaryDictionary();
```



Add the following code to the `jblnit` function.

```
dictionary1.setFileName("dcalc.mld");
translator1.setDictionary(dictionary1);
```

Run the application. It should look like this (if your system default locale is Finnish):

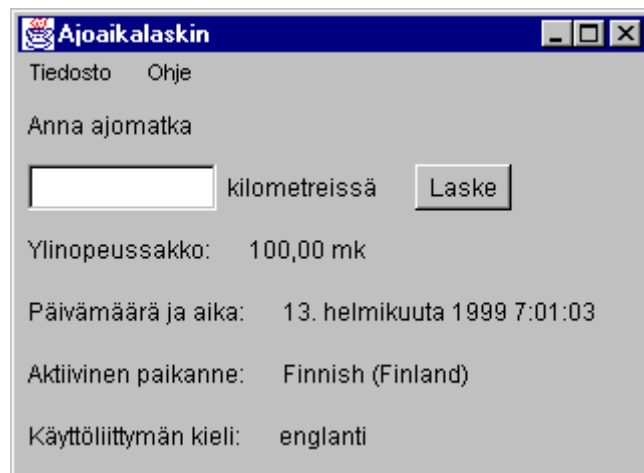


Figure 85 Dcalc in Finnish

Making a multilingual application is this simple. In a simple case, this is all you have to do to make a multilingual application. In most other cases you have to do a little bit more.

If the program contains items which are just country (locale)-specific and hard coded in the source, they must be removed. This phase is called internationalization: it makes your software international and language/country independent. The next phase would then be to localize the program, i.e., add for each target country the locale-specific issues. This is done easily by using Multilizer beans. The remaining document discusses how to do this.

Dcalc calculates the average driving time. Most countries use the metric system, where the distance is expressed in kilometers. However in the US miles are used. Let's study how to make Dcalc compatible with both kilometers and miles.

When pressing the Calculate button Dcalc calls the following event:

```
void calculateButton_actionPerformed(ActionEvent e)
{
    int distance;

    try
    {
        distance = Integer.valueOf(textField.getText().trim()).intValue();
        if (distance < 0)
            throw new NumberFormatException();

        String[] params =
        {
            new Integer(distance/100).toString(),
            new Integer((int) (60*(distance%100)/100)).toString()
        };

        MessageDialog.showMessageDialog(
            this,
            "Driving Time",
            MessageFormat.format("The average driving time is {0} hours and {1}
minutes", params),
            MessageDialog.OK);
    }
    catch (NumberFormatException ex)
    {
        String[] params = { textField.getText() };

        MessageDialog.showMessageDialog(
            this,
            "Invalid value",
            MessageFormat.format("\ \"{0}\ " in not a valid distance", params),
            MessageDialog.OK);
        textField.requestFocus();
    }
}
```

When the English (United States) locale is on the user gives the distance in miles. To convert miles to kilometers add the following just before

```
String[] params =;
if (Utils.getMeasurementSystem(dictionary1.getActiveLocale()) ==
Utils.US_MEASUREMENT)
    distance = (int)Utils.MILE_IN_METERS*distance/1000;
```

This is enough for the system to convert miles to kilometers but not enough for the user. The user will most definitely be a bit confused if the user interface still prompts in kilometers. To make user interface react on the locale change the *languageChanged* event to the translator component and writes the following code:

```
void translator1_languageChanged(DictionaryEventObject e)
{
    if (Utils.getMeasurementSystem(dictionary1.getActiveLocale()) ==
Utils.US_MEASUREMENT)
        unitLabel.setText(translator1.translate("in miles")); //mlz
    else
```

```

        unitLabel.setText(translator1.translate("in kilometers")); //mlz

        fineLabel.setText(NumberFormat.getCurrencyInstance(
            dictionary1.getActiveLocale()).format(new Integer(100)));

        dateLabel.setText(DateFormat.getDateTimeInstance(
            DateFormat.LONG,
            DateFormat.MEDIUM,
            dictionary1.getActiveLocale()).format(new Date()));

        localeLabel.setText(Utils.getLocaleName(
            dictionary1.getActiveLocale(), dictionary1));
        languageLabel.setText(dictionary1.translate(
            dictionary1.getLanguageData().englishName));
    }

```



Add the following code to the `jblnit` function. It adds the `languageChanged` event to the translator.

```

translator1.addLanguageChangeListener(new multilizer.DictionaryListener()
{
    public void languageChanged(DictionaryEventObject e)
    {
        translator1_languageChanged(e);
    }
});

```

First the code checks the measurement system. This is done by comparing the `measurementSystem` variable of the active locale. The code updates the text and help string. Let's study the following code in more detail:

```
unitLabel.setText(translator1.translate("in kilometres"));
```

In a monolingual application you would have used the following code:

```
unitLabel.setText("in kilometres");
```

This isn't the proper way in a multilingual application because the same EXE file must work on every language and locale. That's why the native string is translated before being assigned to the `Caption` property.

The lower part of the event updates the speeding fine, current time, active language name, and active locale name.

The constructor monolingual `MainFrame` contains the following code:

```

fineLabel.setText(NumberFormat.getCurrencyInstance(
    Locale.getDefault()).format(new Integer(100)));

dateLabel.setText(DateFormat.getDateTimeInstance(
    DateFormat.LONG,
    DateFormat.MEDIUM,
    Locale.getDefault()).format(new Date()));

localeLabel.setText(Locale.getDefault().getDisplayName(Locale.UK));

```

This code is not required any more because the `dictionary1_languageChanged` event updates the labels. You can remove it.

We need to make a few modifications to the `calculateButton_actionPerformed` event to make the message boxes multilingual. Consider the following code:

```

MessageDialog.messageBox(
    this,
    "Driving Time",
    MessageFormat.format("The average driving time is {0} hours and {1}
minutes", params),
    MessageDialog.OK);

```

Multilizer can not translate the standard message dialogs. You must use Multilizer's own *multilizer.MessageDialog* or add a translator component to the message dialog.

```
multilizer.MessageDialog.messageBox(
    this,
    "Driving Time", //mlz
    Utils.formatMessage(
        "The average driving time is {0} hours and {1} minutes", //mlz
        params,
        dictionary1),
    MessageDialog.OK,
    new Translator(dictionary1));
```

Remember that you have to translate the exception message as well.

```
multilizer.MessageDialog.messageBox(
    this,
    "Invalid value", //mlz
    Utils.formatMessage(
        "\"{0}\" in not a valid distance", //mlz
        params,
        dictionary1),
    MessageDialog.OK,
    new Translator(dictionary1));
```

Translate the message box in the *aboutMenuItem_actionPerformed* event.

```
multilizer.MessageDialog.messageBox(
    this,
    "About DCALC", //mlz
    "DCALC calculates the driving time.", //mlz
    MessageDialog.OK,
    new Translator(dictionary1));
```

In this case you do not have to translate the text parameter but you can let the *MessageDialog* translate it. This is because the message is not parametrized.

Most string constants in the above code examples are trailed with the mlz comment. The comment is a tag for the Multilizer. You can use tags to control Multilizer when extracting the strings and adding them to the project file. You can configure tags by editing your Multilizer project's Scan Targets.

Changing Language at Run-Time

Dcalc now has the ability to adapt to the current language and locale settings of the user. What about changing the language and/or locale at run-time? Is this possible? Yes.

Double click the File menu and move to the white area (Type Here) on the bottom of the menu. Type *&Language...* Move the memo on the top of the *Exit* menu.

The result should look like this:

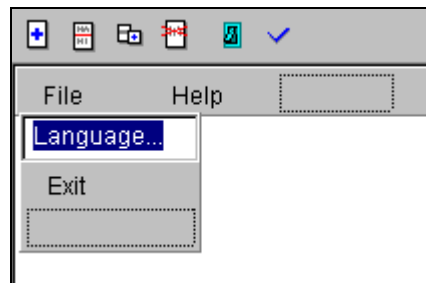


Figure 86 Add the Language menu item to the main menu

Add the following code just before the constructor of *MainFrame*:

```
MenuItem menuItem1 = new MenuItem();
```

Add the following code into the *jblnit* function:



```
menuItem1.setLabel("Language...");
menuItem1.addActionListener(new java.awt.event.ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        menuItem1_actionPerformed(e);
    }
});
fileMenu.add(menuItem1);
```

Write the following event handler to the Language... menu item:

```
void menuItem1_actionPerformed(ActionEvent e)
{
    SelectLanguageDialog dialog = new SelectLanguageDialog(
        this,
        new Translator(dictionary1),
        false);

    if (dialog.showModal())
        dictionary1.setLanguage(dialog.getLanguage());
}
```

The dialog box contains a list of available languages that the user can select. After showing the dialog the event sets the new active language by setting the *language* property of the project file component.

Run the application and choose File | Language... (Or Tiedosto | Kieli... if you have Finnish active). The following dialog box appears:

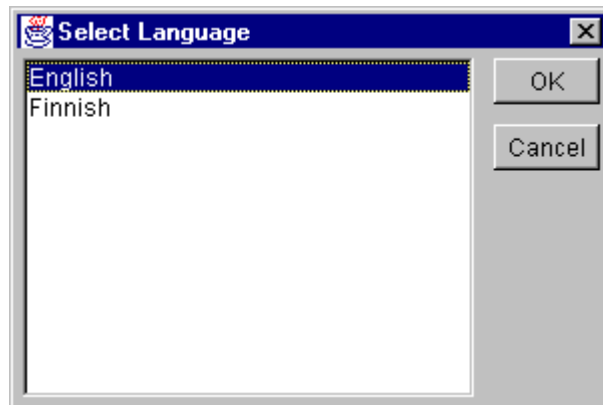


Figure 87 The Select Language dialog box lets the user select the active language at run-time

The SelectLanguage dialog box shows all available languages in a list box.

Select a new language and press the OK button. The active language (user interface) of Dcalc changes to that language. By default the active locale also changes to the default locale of the language. You can set the active language and locale independently by setting the *binding* property of the dictionary component to false.

The *SelectLanguageDialog* dialog box contains strings that need to be translated as well. Also the *MessageDialog* uses several strings (e.g. "OK", "Cancel"). The string tables of the Multilizer contain all these constant strings. All you need to do is to add them to your project file. Launch Multilizer. Open the project file. Choose **Project | Include Strings | System Strings**. The System Strings dialog box appears. Check Language Dialog, and Message Dialog check boxes.

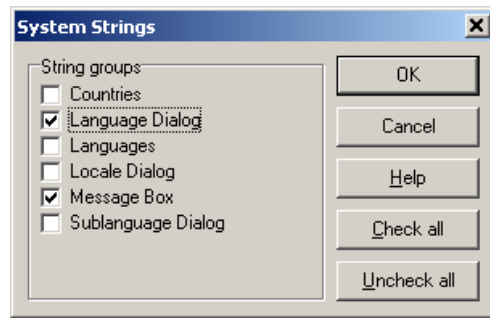


Figure 88 The System Strings dialog box lets the developer add strings used by the system or by the standard components

Press the OK button. Multilizer adds the strings used by the selected items. Let Multilizer get the translations from the translation memory: Select any cell in the Finnish column. Choose **Language | Translate | Using Translation Memory**.

Now our Dcalc application is fully multilingual. The user interface, locale settings, and input measures match the local settings. The user can even change the language at run-time. The remaining chapters describe some of the advanced features of Multilizer.

Writing Multilingual Applets

Writing multilingual applets is as easy as writing multilingual applications. However you have to notice the following items:

- To make the applet compatible with most browsers use the AppletTranslator component instead of the Translator component.
- Set the applet property of the dictionary component to refer to the applet.
- Take the dictionary away from the destroy method. For example:

```
public void destroy() {
    dictionary1.dispose();
}
```



MORE INFO

For additional information on writing applets, see the Dcalc and Euro applets from the samples subfolder.

Writing Multilingual Swing Applications

Writing multilingual Swing application is as easy as writing multilingual AWT applications. However you have to notice the following:

- Add the SwingModule bean to the main frame. This adds the Translator bean the ability to translate the Swing components.

Learn more from the online documentation and samples.

Swing applets are not fully supported by Multilizer beans. See swing applet sample from the samples subfolder for additional information.



MORE INFO



NOTE!

Font Issue with Non-Western Languages

Java uses Unicode strings. That's why in theory every Java application can display any characters. Unfortunately Java must always work on the top of the host platform (e.g. Windows, Linux, Solaris). The host platform does not necessary contain the font support needed by the language.

You might need to update the font.properties files of your Java run-time environment to add Far Eastern and Middle Eastern fonts. See JDK documentation for more information.

11

Java Micro Edition

In this tutorial we are going to create a localized J2ME (Java Micro Edition) application. The application will be a simple driving-time calculator, Dcalc that a user can use to calculate the average driving time for a given distance. We use MID profile (MIDP).

The Dcalc application is very simple but still it uses most features of Multilizer. The creation of the application is divided into several lessons, each covering one or more Multilizer functions.

J2ME Localization

Java Standard Edition (J2SE) contains very rich support for localization. J2SE contains locale class, resource bundles and formatting classes. Unfortunately Java Micro Edition (J2ME) does not contain these classes. It only contains very low-level resource class that the application can use to access resource file. The current CLDC-configuration has the following I18N-support:

- `java.lang.Class.getResourceAsStream(String name)` that returns the input stream for the resource file.
- `java.lang.System.getProperty(String name)`. When passed "microedition.locale" as a parameter this returns the system locale of the configuration.

In theory this could be just enough support for I18N for simple applications. However using the `getResourceAsStream` to get the localized user interface strings is rather complicated. This is because `getResourceAsStream` is a very low level function. It just gives you the access to raw resource file data. The J2ME programmer needs to write a large amount of code to get the localized user interface strings from the resource file.

J2SE contains two kinds of resource files: property files and list files. The property file contains the translations of the strings in one language, one translation in a row. List files use similar approach but they represent data inside a Java class. Without having the resource bundle class a J2ME programmer has two choices to localize his or her application.

- Write own code on the top of `java.lang.Class.getResourceAsStream`. This is a complicated task and the memory consumption might be too high.
- Change the strings in the java source code. This might seem as an easy solution but it leads the programmer into troubles if the application source code changes. Either the programmer needs to do the same changes to every single localized java code or retranslate the localized java source codes again. Both approaches are difficult to implement, slow and error prone.

Because J2ME doesn't support property files (through `PropertyResourceBundle`) nor list files (through `ListPropertyFiles`), Multilizer contains a small footprint resource bundle class and format that is suitable for J2ME.

With Multilizer you are able to globalize your J2ME applications. The `multilizer.microedition.Properties` class has the main role. Using it is pretty similar to using J2SE `PropertyResourceBundles`. It's even possible to use your old J2SE property files directly through it.

`multilizer.microedition` works on a top of the CLDC configuration. `multilizer.microedition` is profile independent so it works with any CLDC profile such as MIDP and PDA profile. It is also small (3 Kbytes) and memory efficient.

Property files may be UTF-8 or ISO8859-1 encoded. This makes the Properties class backward compatible to ISO8859-1 encoded property files. Using UTF-8 encoding makes the files more understandable, because you don't have to use Unicode escapes for non-ASCII characters. UTF-8 also helps conserving the memory because none English UTF-8 files are considerably smaller than Unicode escaped ASCII files.

The following picture describes the J2ME localization process.

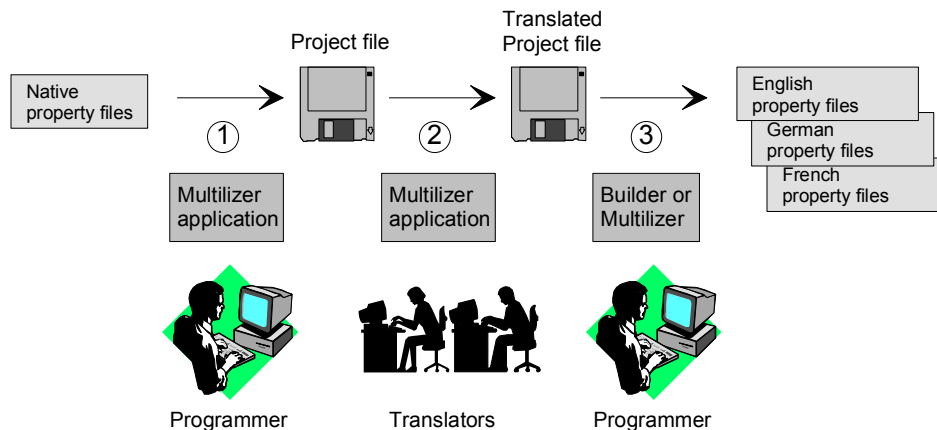


Figure 89 J2ME localization process

The programmer uses Multilizer to extract strings from the original property file (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized property files (3). As the result there will be one property file for each localized language.

The following figure shows the files that Multilizer uses on the J2ME localization process.

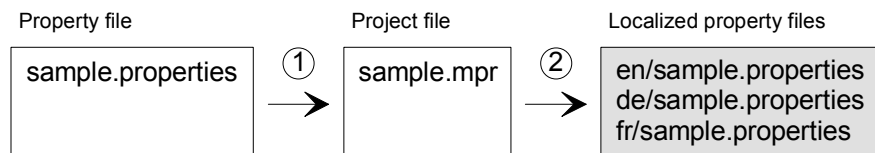


Figure 90 The files of the J2ME localization process

Add the localized property file (e.g. `de/sample.properties`) instead of the original property file (`sample.properties`) to the setup package when building a localized application.

Application With An English User Interface

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. This is what we are going to do. The `JavaME\samples\tutorial` contains the English Dcalc. Open it, compile it, and finally run it.

The application should look like this:

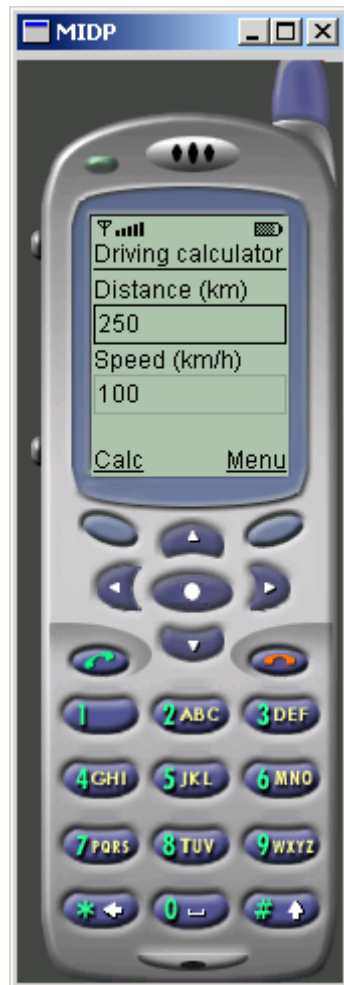


Figure 91 J2ME application with an English UI

The user interface language is English. In the following chapters we will localize Dcalc step-by-step.

Internationalization

To globalize your application you have to resource the application. Resourcing means getting rid of hard coded strings. Most applications contain strings that have been inserted inside the source code. These strings are hard coded. It is impossible to localize such code without changing and recompiling the code. When you resource the code you take the strings from the source code and place them to a resource file that can easily be translated.

The main source code file of your application is `dcalc/Dcalc.java`. It contains several hard coded strings. For example:

```
public class Dcalc extends MIDlet implements CommandListener
{
    ...
    private StringItem distanceLabel = new StringItem(
        null,
        "Distance (km)");
    ...
}
```

This contains a hard coded string: "Distance (km)". The class contains several others as well.

Our first task is to create the property file and take it in use in the midlet. Let's give the property file the same name as the midlet: `dcalc.properties`. Using the property file

in the midlet is easy. Just create a `multilizer.microedition.Properties` instance and pass the property file name as the parameter.

```
public class Dcalc extends MIDlet implements CommandListener
{
    private Properties prop = new Properties("/dcalc/Dcalc.properties");
    ...
}
```

Use `Properties(String)` if your property files are in UTF-8 format. If you want to use the same format (ISO8859-1) as J2SE property files use `Properties(String, int)` instead.

Resourcing the code is done with the `getString` method. In general you should wrap every single string that need to be translated inside the `getString` method. So change the above code to:

```
public class Dcalc extends MIDlet implements CommandListener
{
    private Properties prop = new Properties("/dcalc/Dcalc.properties");
    ...
    private StringItem distanceLabel = new StringItem(
        null,
        prop.getString("Distance (km)"));
    ...
}
```

Add the string to the property file:

```
Distance (km)<tab>Distance (km)
```

The property file format is `key<separator>value`. Where the key is the string inside the `getString` method and the value is the translation. The separator can either be a tab or the '=' character.

Process every hard coded string in the same way. As a result you will have a completed property file and an internationalized midlet source code file.

Distance (km)	Distance (km)
Speed (km/h)	Speed (km/h)
Driving calculator	Driving calculator
Calc	Calc
About	About
Exit	Exit

The `Message.java` file also has to be resourced. We need also complete one extra task with this file. Let's consider the following code in the `showItem` method:

```
alert.setString(
    new Integer(hours) + " hours " +
    new Integer(minutes) + " minutes");
```

We could resource "hours" and "minutes" by wrapping them with the `getString` method. However that would not be very good internationalization because the above code always assumes that the message has the following form `<hour value> <hour label> <minute value> <minute label>`. There are languages that use the label first and the value next. Also some countries prefer to show the minutes first and hours next.

To solve this problem we will use `multilizer.microedition.MessageFormat` class where the whole message is put in the message pattern and the pattern is combined with data at run-time to produce the actual message.

```
Object[] args = {new Integer(hours), new Integer(minutes)};

alert.setString(MessageFormat.format(
    prop.getString("{0} hours {1} minutes"),
```

```
args) ;
```

Now the property file contains the message pattern, {0} hours {1} minutes. The translator can easily relocate the items in the pattern.

This chapter has covered only basic internationalization (I18N). Refer to I18N books and web sites to get more information. An excellent start is the Java tutorial at www.javasoft.com.

Creating a New Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

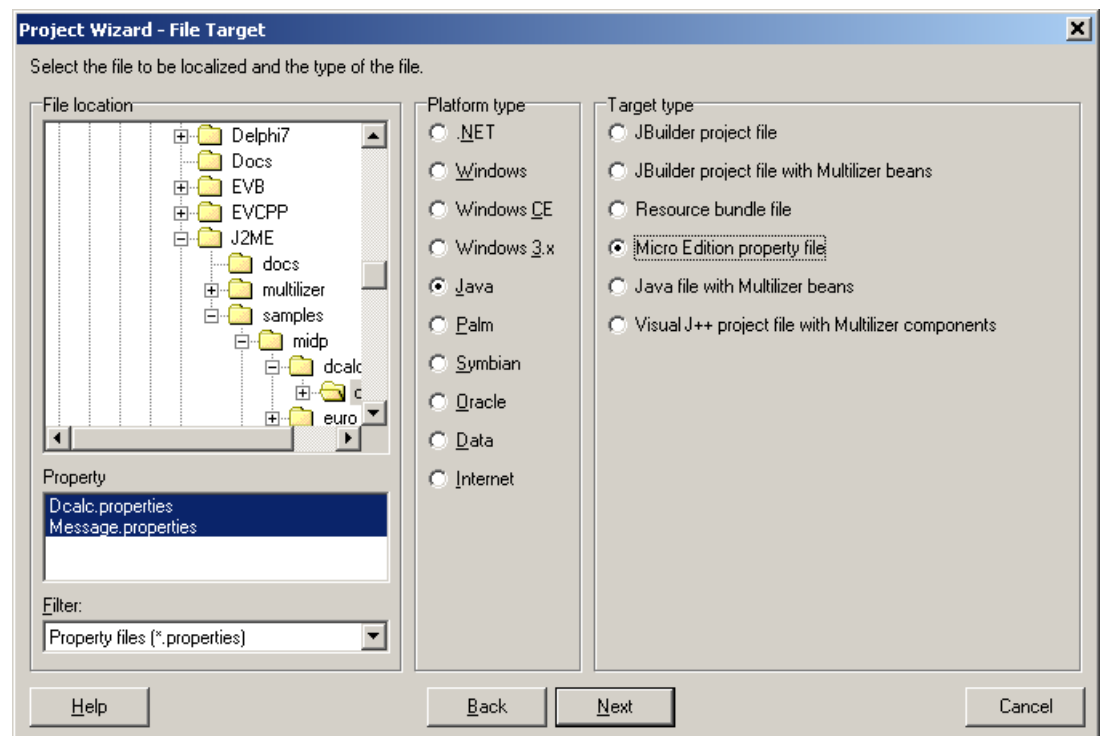


Figure 92 The File Target sheet is used to enter the source directory

This dialog specifies the directory where your application is located. Choose the `<multilizer>\JavaME\samples\tutorial\dcalc` folder. Your application contains two property files (.properties). Select the `Dcalc.properties` file. Project Wizard detects the platform and project types. The Platform type should be *Java* and the Target type should be *Micro Edition property file*. If they are wrong, check the right types.



If you have not completed the previous chapter the above directory does not contain the property file. In that case copy the property files from the `<multilizer>\JavaME\samples\midlet\dcalc\dcalc` directory.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

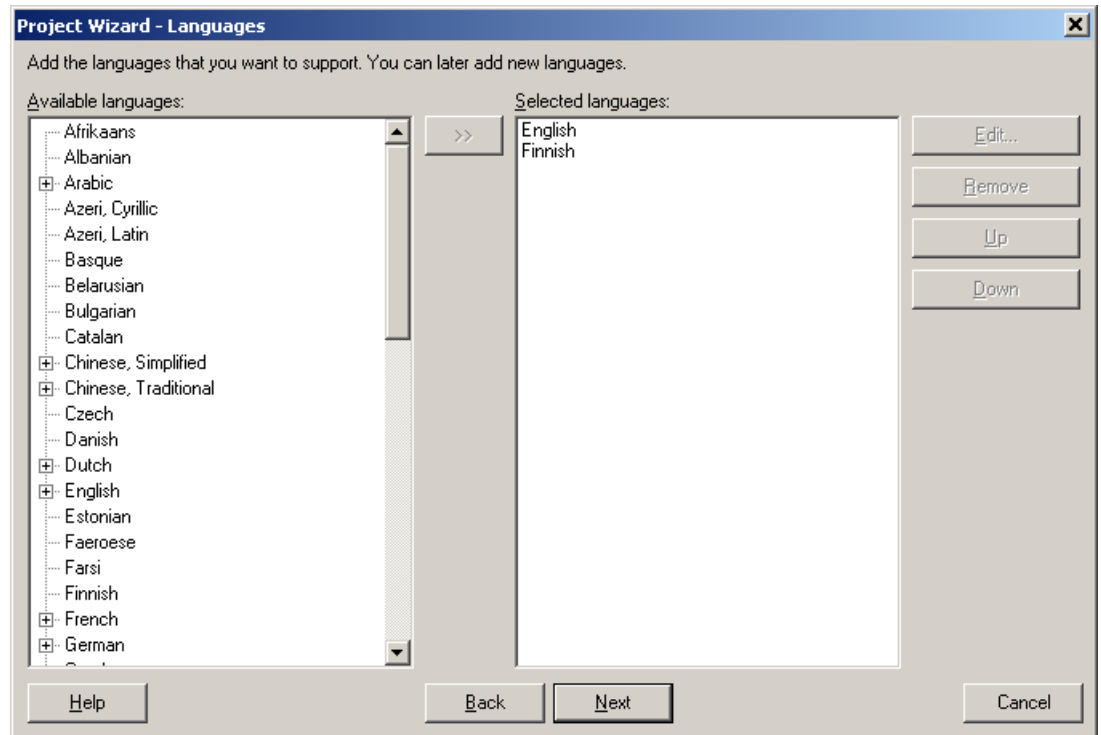


Figure 93 English and Finnish added to the project file

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

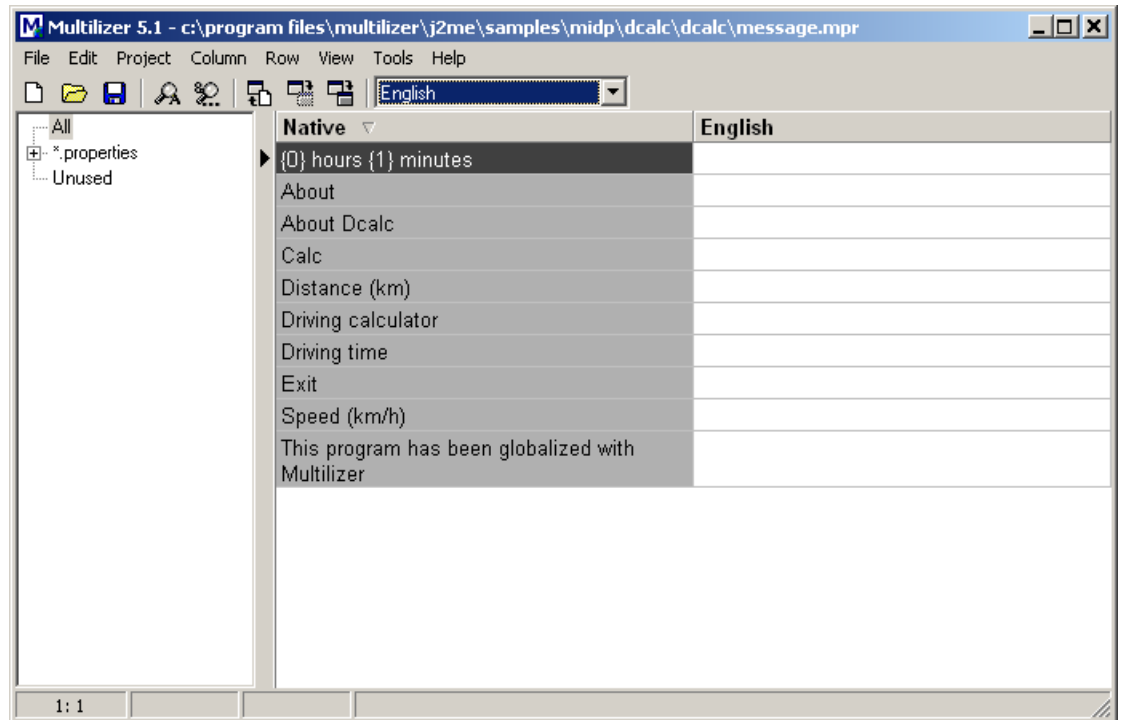


Figure 94 The project file grid

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Create the localized resource files by choosing **Project | Create Localized Items**. This creates the localized property files (.properties) in the language specific subdirectories ('en' and 'fi').

Now you can make the localized versions of the application. Edit for example your build script to include the right properties files for each languages setup packages.

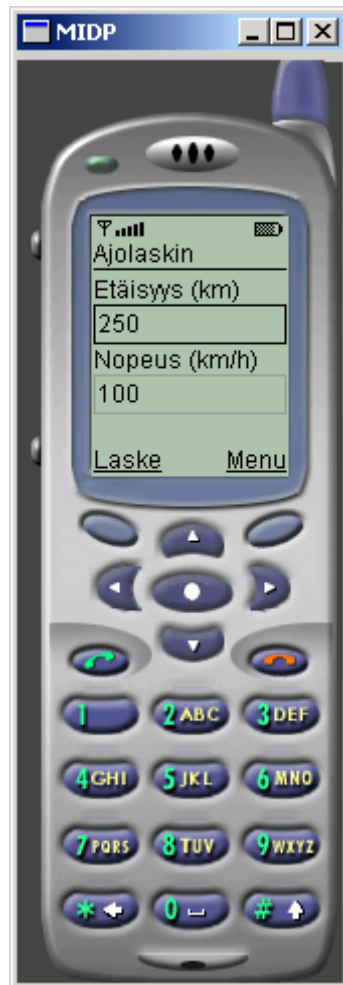


Figure 95 Localized Dcalc application

12

Visual J++

In this tutorial we are going to create a multilingual application. The application will be a simple driving-time calculator, Dcalc that a user can use to calculate the average driving time for the given distance.

Dcalc application is very simple but still it uses most features of Multilizer. Application creation is divided into ten lessons each covering one or more Multilizer issues.

This tutorial is for Visual J++ 6.

Dcalc sample application is located in the `WFC\Samples\Dcalc` subfolder of your Multilizer directory.

To study more about how to use Multilizer read the online help and study the other sample applications found from the `WFC\Samples` subfolder.

Before you can start building Dcalc you have to install Multilizer components to the Component Palette. To get the information how to install them, see the readme files. Double click the *WFC Readme* icon of the compiler to open the readme.

Open a Monolingual Application

We could start from scratch but in most cases it is a finished application or at least an application under development that you want to globalize. We are going to work with a finished application. The `WFC\Samples\Tutorial` contains the English Dcalc. Open it, compile it, and finally run it.

The application should look like as this:

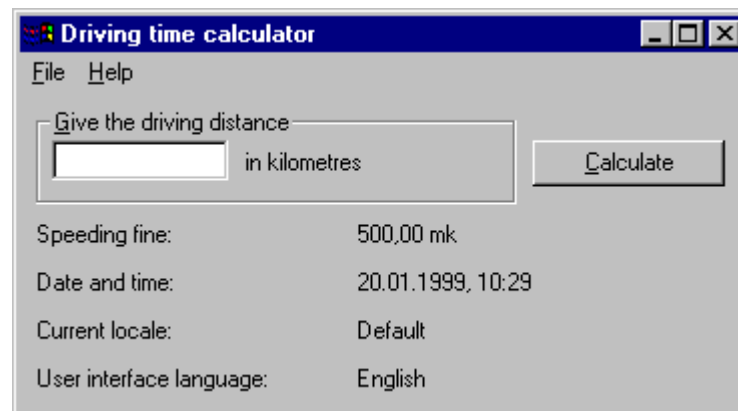


Figure 96 The monolingual application with an English user interface

The user interface language is English and the application uses the default locale that is in this case Finnish (Finland). The speeding ticket is formatted using the Finnish currency format (mark) and the date and time are also formatted using the Finnish format. Standard WFC provides this kind of globalization through NLSAPI.

In the following chapters we will make Dcalc truly multilingual step-by-steps.

Make Application Multilingual

The first step is to make Dcalc multilingual just dropping two components to the form. Select the Translator component from the Toolbox. Drop the Translator to the form. Drop the TestDictionary component to the form as well.

The result should look like this:

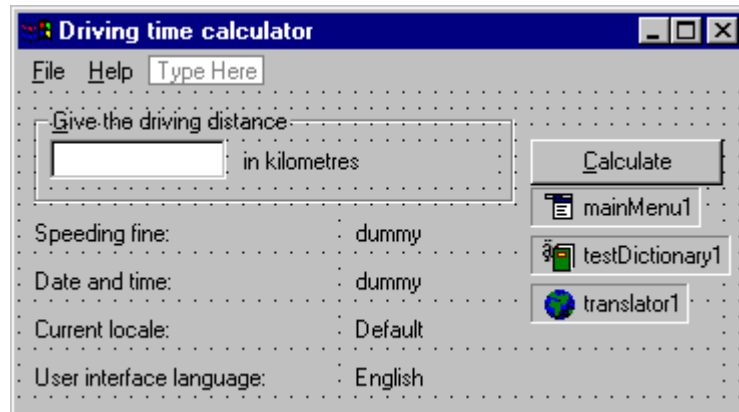


Figure 97 Translator and dictionary components have been added to the form

TestDictionary is one of the dictionary components of Multilizer. A dictionary component provides string or phrase translation for the application. Normally each application contains one dictionary component that contains all the translation data of the application. Multilizer contains several different dictionary components: one getting the translation data from a text file, the other from a database, etc.

The TestDictionary component is a special case. It does not require any dictionary data but it makes the translation on-the-fly by changing the original string to a test string. In a normal case you can not use the test dictionary in your final application because the translation is not a real language. However, the test dictionary is really handy in the development phase.

The Translator on the other hand is the component that makes all the work. It scans the form before it comes visible and translates the user interface string from the original value to the current language.

Move to the Properties windows. Drop down the value list of the Host property. Select Form1. This specifies the control that the translator should translate. In most cases it is the form containing the translator component.

Add the following line to the constructor of the Form1:

```
public Form1()
{
    initForm();
    translator1.translate();
}
```

The translate method makes the translator to translate its host control. A proper place to call this is just before the form becomes visible. Another possible place would have been the activate event:

```
private void Form1_activate(Object source, Event e)
{
    translator1.translate();
}
```

Compile and run Dcalc. It should look like this:

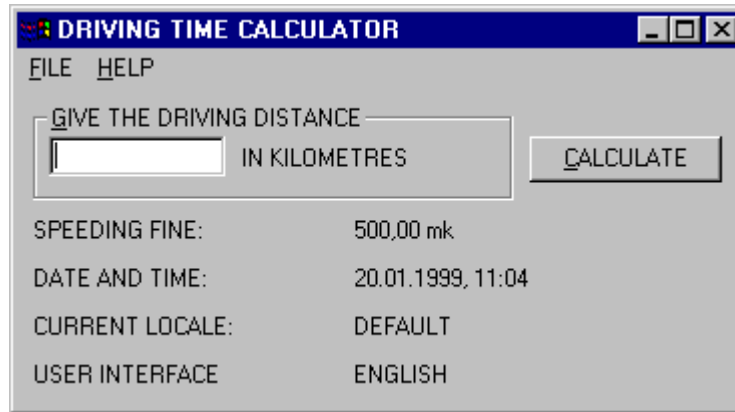


Figure 98 “Translated” application. The test dictionary translated every string to upper cased string

As you can see, every user interface string is now in upper-case. The translator changed every string type property after the form had been loaded from the resource. By default the test dictionary translates every string by upper-casing it.

For additional information on using the test dictionary, see the online help topic "TestDictionary".

This was a quick demonstration of the power of Multilizer. In the next chapter we will create a real dictionary that contains real languages.



Create a Project for the Application

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

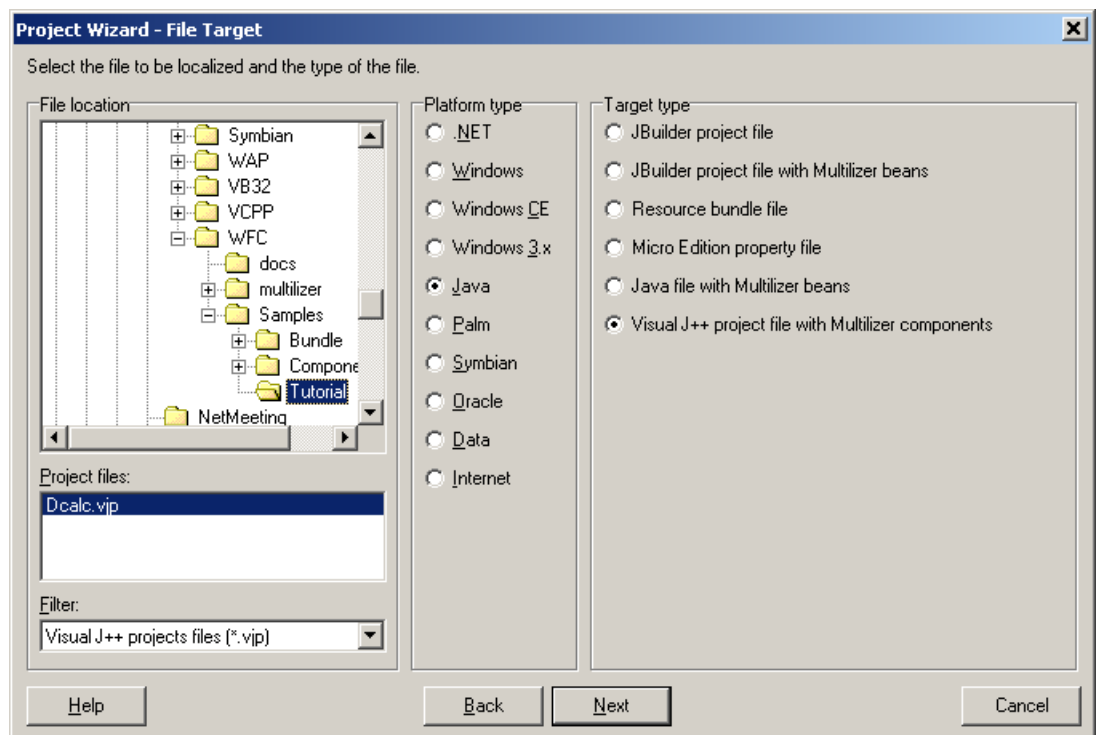


Figure 99 The File Target sheet is used to specify the project file

This sheet specifies the location of your application. Choose the `<mldir>\WFC\Samples\Tutorial` subfolder. Project Wizard detects the platform and

project types. The Platform type should be *Java* and the Target type should be *Visual J++ project file with Multilizer components*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the >> button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

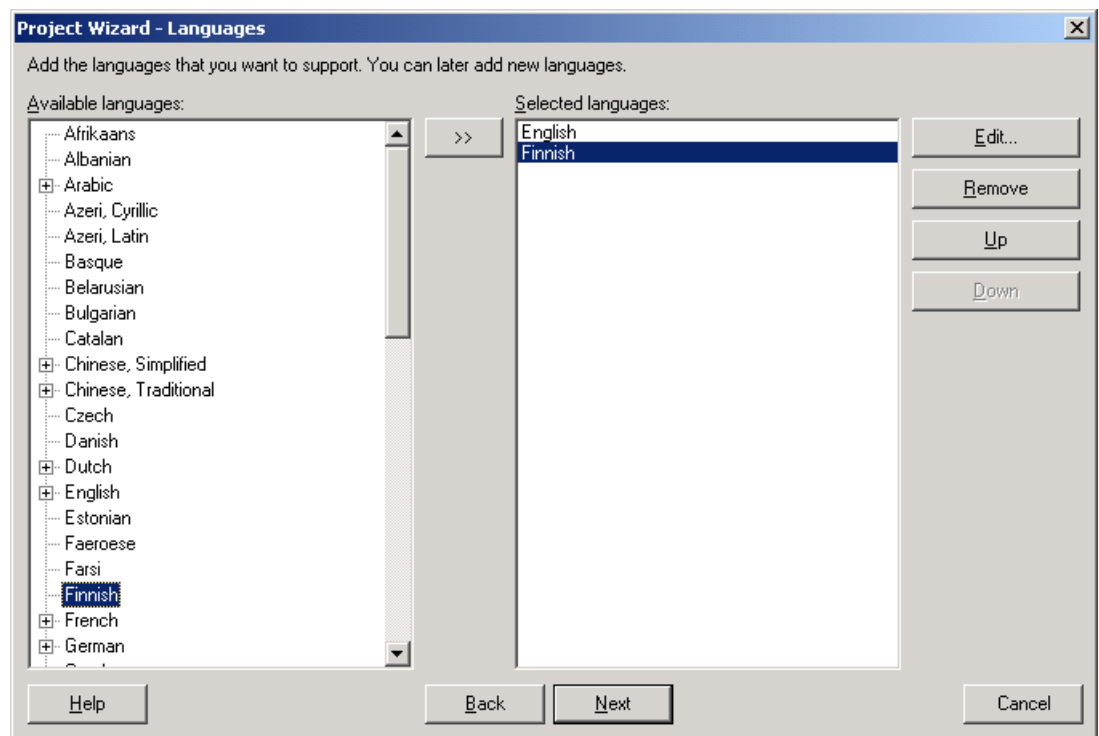


Figure 100 English and Finnish added to a project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

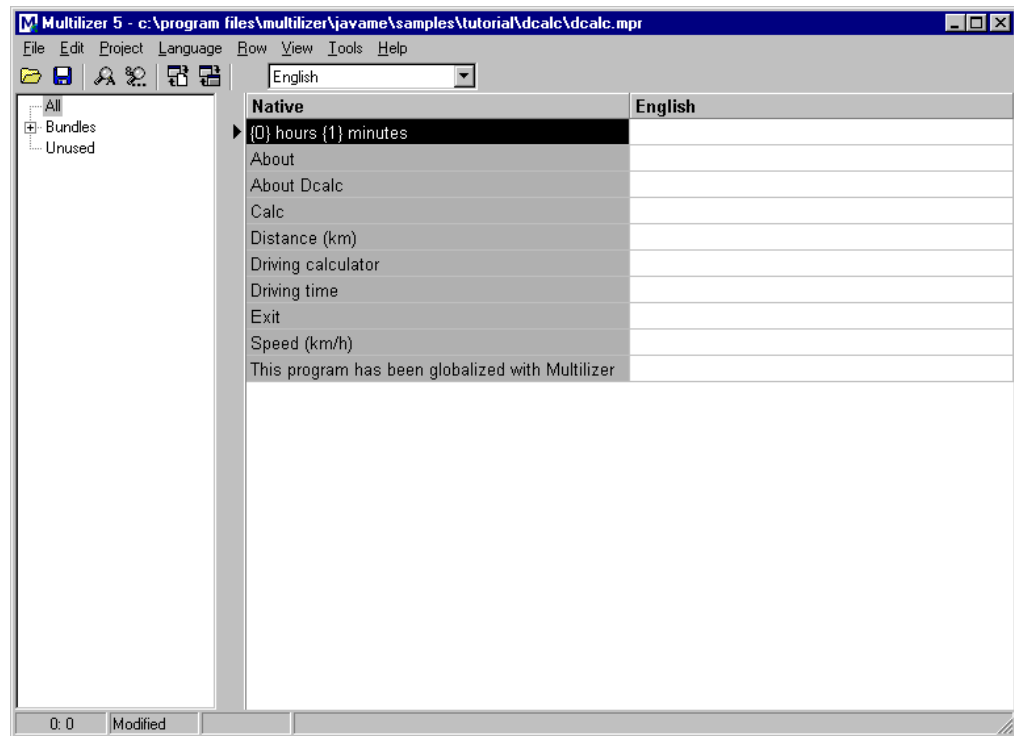


Figure 101 The project grid

Save the project by choosing File | Save As.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part.

Internationalize Your Code

We now have a project file. The next step is to delete the TestDictionary component from the form. Next, add the BinaryDictionary component. Choose the component and move to the Properties window. Set the *fileName* property to `dcalc.mld` (i.e. the dictionary that you created in previous lesson). Set the *name* property to `dictionary1`.



NOTE!

If you pressed the ... button and browsed the file name, the Properties window would add the full file name (e.g. `C:\Program Files\Multilizer\WFC\Samples\Tutorial\dcalc.mld`). This may cause troubles when deploying your application so remove the path part from the file name.

The Properties window should look like this:

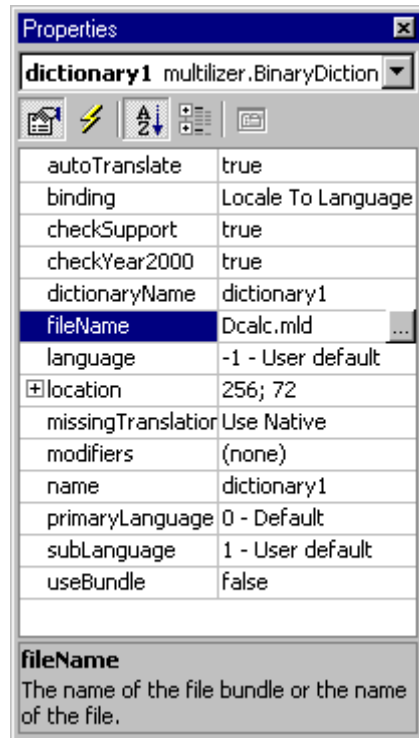


Figure 102 The Properties window showing the properties of a binary project file component

Let's study some of the properties. The *language* property specifies the active language. It is -1. This makes Multilizer check the current locale of the user and to find the language that matches the locale. If none is found the first (none-native) language is used.

The *primaryLanguage* and the *subLanguage* properties specify the active locale. The active language determines the language of the user interface. The active locale however determines the locale used by the application. The locale is a country and language specific object that control for how the date, time, currency, number, etc are formatted.

The *primaryLanguage* property is 0 and the *subLanguage* property is 1. These makes Multilizer use the default locale of the user. See more from the online documentation at the Dictionary topic.

In our case the project file contains English and Finnish. If the locale setting of the user is Finnish (Finland) the user interface of Dcalc will be in finish and the locale will be Finnish (Finland).

Run the application. It should look like this:

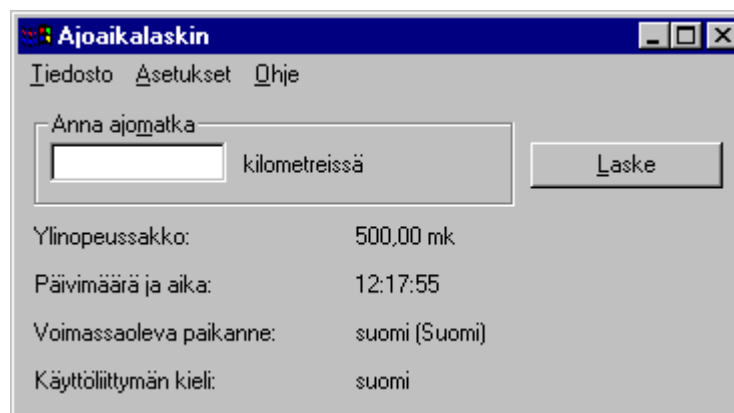


Figure 103 Dcalc in Finnish

Making a multilingual application is this simple. In a simple case, this is all you have to do to make a multilingual application. In most cases you have do a little bit more.

If the program contains code, which is just country (locale) specific and hard coded in the source, it must be removed. This phase is called internationalization: it makes your software international and language/country independent. Next phase would then be to localize the program, i.e., add for each target country the locale specific issues. This is done easily by using Multilizer. Remaining document discuss how to localize your application.

Dcalc calculates the average driving time. Most countries use metric system where the distance is expressed in kilometers. However in US miles are used. Let's study how to make Dcalc compatible both with kilometers and with miles.

When pressing the Calculate button Dcalc calls the following event:

```
private void calculateButton_click(Object source, Event e)
{
    try
    {
        int distance = Integer.parseInt(distanceEdit.getText());
        if (distance < 0)
            throw new Exception();

        int hours = distance/100;
        int minutes = (int)(0.6*(distance%100));

        MessageBox.show(
            Utils.format(
                "The avarage driving time is %0 hours and %1 minutes.",
                new String[] {
                    new Integer(hours).toString(),
                    new Integer(minutes).toString()}),
            getText(),
            MessageBox.ICONINFORMATION | MessageBox.OK);
    }
    catch (Exception ex)
    {
        MessageBox.show(
            Utils.format(
                "\"%0\" is not a valid distance!",
                new String[] {distanceEdit.getText()}),
            getText(),
            MessageBox.ICONERROR | MessageBox.OK);
        distanceEdit.focus();
    }
}
```

When the English (United States) locale is selected, the user gives the distance in miles. To convert miles to kilometers, add the following to just before line

```
int hours = distance/100;
if (dictionary1.getLocaleData().measurementSystem == Measurement.US)
    distance = (int)(Measurement.MILE_IN_METERS*distance/1000);
```

This is enough for the system to convert miles to kilometers but more has to be done to make a truly localized application. The user will most definitely be a bit confused if the user interface still prompts in kilometers. To make user interface react on the locale add the *languageChange* event to the dictionary component and write the following code:

```
private void dictionary1_languageChange(Object source, LanguageEvent e)
{
    if (dictionary1.getLocaleData().measurementSystem ==
        Measurement.METRIC)
    {
        unitLabel.setText(dictionary1.translate("in kilometres"));
        helpProvider1.setHelpString(
            unitLabel,
            dictionary1.translate("Give the driving distance in kilometres"));
    }
    else
    {
        unitLabel.setText(dictionary1.translate("in miles"));
    }
}
```

```

        helpProvider1.setHelpString(
            unitLabel,
            dictionary1.translate("Give the driving distance in miles"));
    }

    speedingFine.setText(java.text.NumberFormat.getCurrencyInstance(
        dictionary1.getLocaleData().getJavaLocale()).format(500));

    currentTime.setText(DateFormat.getTimeInstance(
        DateFormat.MEDIUM,
        dictionary1.getLocaleData().getJavaLocale()).format(new Date()));

    currentLocale.setText(dictionary1.getLocaleData().getDisplayName(
        multilizer.Locale.TRANSLATED,
        dictionary1));

    currentLanguage.setText(
        dictionary1.translate(dictionary1.getLanguageData().englishName));
}

```

At first the code checks the measurement system. This is done by comparing the *measurementSystem* variable of the active locale. The code updates the text and help strings. Let's study the following code in more detail:

```
unitLabel.setText(dictionary1.translate("in kilometres"));
```

In a monolingual application you would have used the following code:

```
unitLabel.setText("in kilometres");
```

This isn't a proper way in a multilingual application because the same EXE file must work on every language and locale. That's why the native string is translated before assigned to the Caption property.

The lower part of the event updates the speeding fine, current time, active language name and active locale name.

The monolingual Dcalc contains the following event:

```

private void Form1_activate(Object source, Event e)
{
    speedingFine.setText(Value.formatCurrency(500, 2));
    currentTime.setText(new Time().formatShortDate() + ", " +
        new Time().formatShortTime());
}

```

The event is not required any more because the *dictionary1_languageChange* event updates the labels. You can remove it.

We need to make a few modifications to the *calculateButton_click* event to make the message boxes multilingual. Read the following code:

```

MessageBox.show(
    Utils.format(
        "The avarage driving time is %0 hours and %1 minutes.",
        new String[] {
            new Integer(hours).toString(),
            new Integer(minutes).toString()}),
    getText(),
    MessageBox.ICONINFORMATION | MessageBox.OK);

```

Multilizer can not translate the standard message dialogs. You must use Multilizer's own *MessageDlg*.

```

MessageDlg.show(
    Utils.format(
        dictionary1.translate("The avarage driving time is %0 hours and %1
minutes."),
        new String[] {
            new Integer(hours).toString(),
            new Integer(minutes).toString()}),
    getText(),

```

```

    MessageDlg.INFORMATION_ICON,
    MessageDlg.OK_BUTTON);

```

Remember, that you have to translate the exception message as well.

```

MessageDlg.show(
    Utils.format(
        dictionary1.translate("\"%0\" is not a valid distance!"),
        new String[] {distanceEdit.getText()}),
    getText(),
    MessageDlg.ERROR_ICON,
    MessageDlg.OK_BUTTON);

```

Translate the message box in the AboutMenuClick event.

```

MessageDlg.show(
    "Dcalc is a multilingual application that calculates the average
driving time", //mlz
    getText(),
    MessageDlg.NO_ICON,
    MessageDlg.OK_BUTTON);

```

In this case you do not have to translate the text parameter but you can let the MessageDlg translate it. This is because the message is not parametrized.

Change Language at Run-Time

Dcalc has now the ability to adapt to the current language and locale settings of the user. What about changing the language and/or locale at the run-time? This is possible as well. Double click the File menu and move to the white area (Type Here) on the bottom of the menu. Type *&Language....* Move the memo on the top of the *Exit* menu.

The result should look like this:

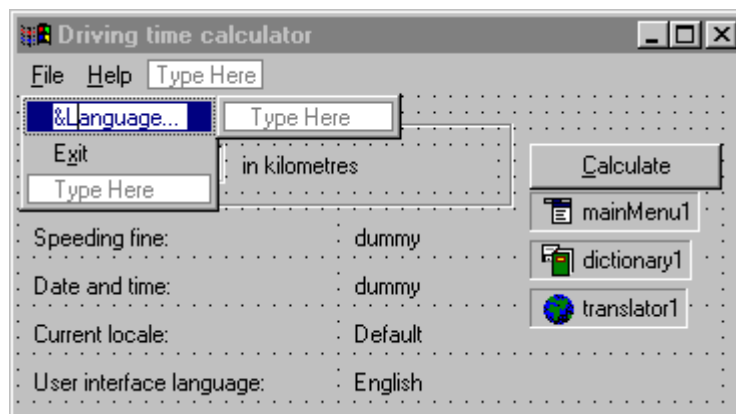


Figure 104 Add the Language menu item to the main menu

Write the following event handler to the Language... menu item:

```

private void languageMenu_click(Object source, Event e)
{
    int language = SelectLanguage.selectLanguage(dictionary1);

    if (language >= 0)
        dictionary1.setLanguage(language);
}

```

The *selectLanguage* shows the Language Select dialog box of Multilizer. It contains a list of available language that the user can select. After calling the *selectLanguage* function the event sets the new active language by settings the *language* property in dictionary component.

Run the application and choose File | Language... (Or Tiedosto | Kieli... if you have Finnish active). The following dialog box appears:



Figure 105 The Select Language dialog box that lets the user select the active language at run-time

The SelectLanguage dialog box shows all available languages in a tree view. The available language does not necessarily mean that all the languages of the project file are being displayed. It may be that your operating system cannot cope with all character scripts.

For example you need to have a bi-directional OS to properly display Arabic or Hebrew. Also most Western OS versions lack the support for Cyrillic or Far Eastern languages. You can force dictionary component and SelectLanguage function to display every language by setting the *checkSupport* property of the dictionary component to false.

Select a new language and press the OK button. The active language (user interface) of Dcalc changes to that language. By default the active locale also changes to the default locale of the language. You can set the active language and locale independently by setting the *binding* property of the dictionary component to false.

The SelectLanguage dialog box contains strings that need to be translated as well. Also the MessageDlg uses several strings (e.g. "OK", "Cancel"). The string tables of Multilizer contain all these constant strings. All you need to do is to add them to your project file. Launch Multilizer. Open the project file. Choose Project | Include | System String. The System String dialog box appears. Check Language Dialog, Message Box, and System Menu check boxes.



Figure 106. The System Strings dialog box lets the developer add strings used by system or by the standard components

Press the OK button. Multilizer adds the strings used by the Select Language dialog, and the system menu. The glossaries contain translations of these strings. You do not have to translate them but let Multilizer get translations from the glossaries: select any cell in the Finnish column. Choose Language | Translate | Using Translation Memory.

Now our Dcalc application is fully multilingual. The user interface, locale settings, and input measures match the local settings. The user can even change the language at run-time. The remaining chapters describe some of the advance features of Multilizer.

13

Symbian

In this tutorial we are going to localize a Symbian application. The application is a simple driving-time calculator, Dcalc, which a user can use to calculate the average driving time for a given distance. The Dcalc application is very simple but nevertheless uses most Multilizer's features. The localization of the application is divided into several lessons, each covering one or more Multilizer functions.

Symbian Localization

Symbian applications use resource files (.rsc). Each application file (e.g. sample.app) must have a corresponding resource file (e.g. sample.rsc). Multilizer creates the localized resource files from the original resource source file (.rss). The following picture describes the Symbian localization process.

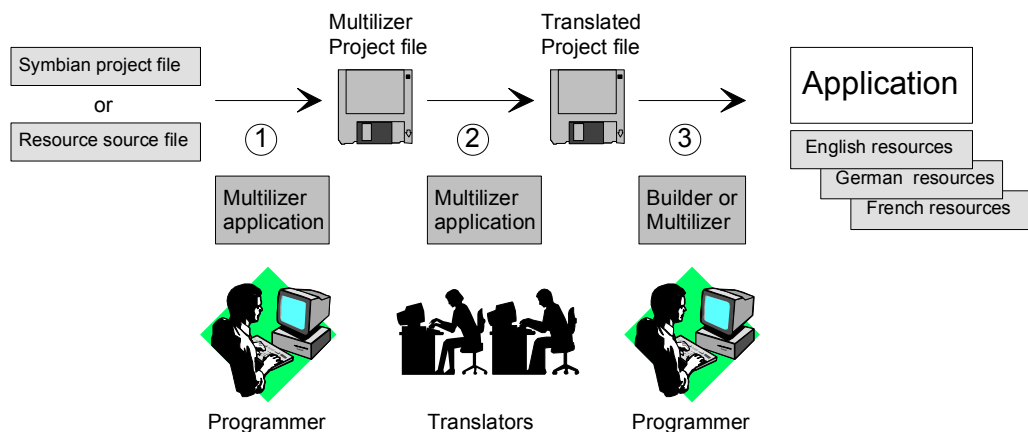


Figure 107 Symbian localization process

The process of localizing Symbian applications can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the resource source file(s) (.rss) belonging to the selected Symbian project file (.mmp). Multilizer saves these strings into its project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file. Translators send the translated project files back to the programmer who imports them to the project files.
3. The programmer uses Multilizer or Builder to create the localized preprocessed resource source files (.rpp) and compiles them to the resource files (.rsc).

Multilizer creates three files for each language. The files will be written to a language specific directory (e.g. de for German). The first file is the localized preprocessed resource file (.rpp). It is a file that has been first preprocessed and then translated. It is written in Unicode (UTF-8) format even if the native resource source file uses Windows code page (1252). The second and third files are compiled resource files. They differ only by their file extensions. The first one has the default file extension .rsc. The second one has the localized file extension. Each language or locale has its own extension. For example .r01 is English, .r02 is French, .r10 is English (United States).



NOTE!

In addition Multilizer creates two skeleton files. The first is the installation package file. The name equals to the Symbian project file but the file extension is `.pkg.ml`. Read more information from the *Deploying* paragraph later on this chapter. The second is the AIF resource file. The name equals to the Symbian project file but the file extension is `aif.rss.ml`.

The following figure shows the files that Multilizer uses in the Symbian project file localization process.

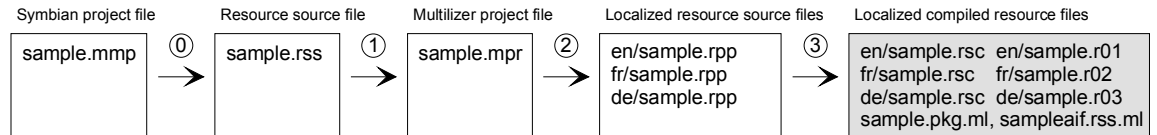


Figure 108 The files of the Symbian project file localization process

Instead the project file (`.mmp`) the programmer can specify a Symbian resource file (`.rss`). The process is similar to the project file localization.

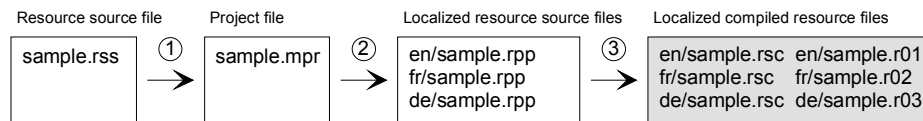


Figure 109 The files of the Symbian resource file localization process

To use the localized resource file (e.g. `de/sample.rsc` or `de/sample.rsc`), deploy it to the Symbian device instead of the original resource file (e.g. `sample.rsc`).



In order to preprocess and compile the resource file, and to run the localized Symbian application, Multilizer needs to know the locations of the Symbian preprocessor, resource compiler and emulator. When Multilizer runs first time it tries to find them from the hard disk. To edit these settings start Multilizer and choose **Tools | Options | Symbian** menu. The following dialog box appears:

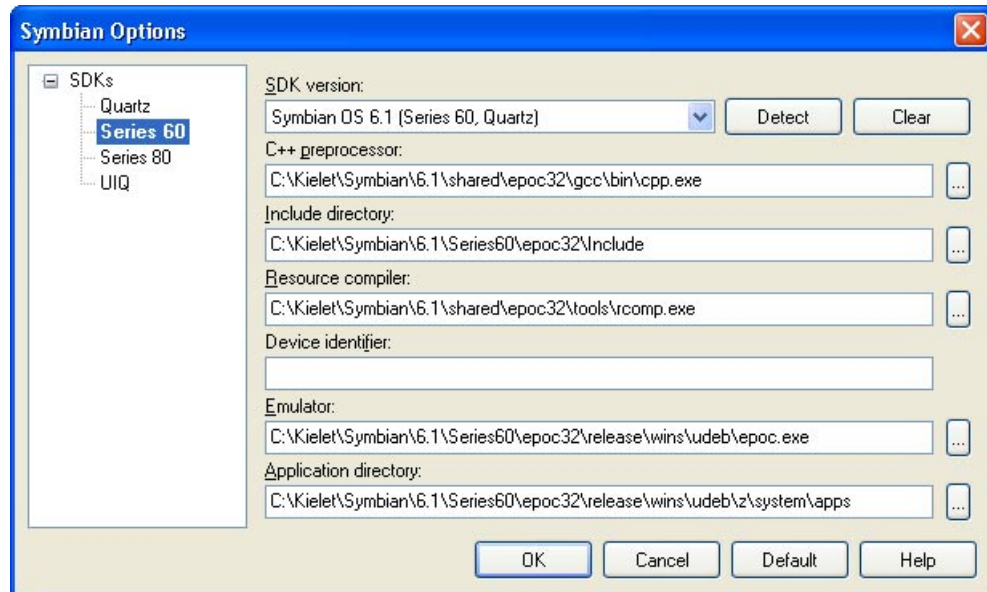


Figure 110 The Symbian Options dialog box

By default the tree contains all the Symbian SDKs that have been installed to the computer. If the SDK tree is empty, right mouse click the tree, choose Add, and enter SDK settings manually. You must select at least the SDK version and enter the preprocessor path and the include directory. In order to build the localized files you have to enter the resource compiler path and device identifier (OS 7 or later).

Application Using an English User Interface

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to localize. This is what we are going to do. The `Symbian\Samples` directory contains several subdirectories. One for each Symbian SDK. Each subdirectory contains the Tutorial subdirectory. For example `Symbian\Samples\Series60\Tutorial` contains the English Dcalc using Series 60 SDK. Select the tutorial for the SDK that you have installed. Open it, compile it, and finally start the emulator. To start the application, press the Dcalc icon. The application launches:



Figure 111 Series 60 emulator and Dcalc application with an English user interface

The user interface language is English. In the following chapters we will localize Dcalc step-by-step.

Internationalization

Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development. I18N is defined as the set of processes, tools, coding techniques and procedures used to write a software program that supports all of the language requirements and country conventions of all of the countries where the software will be used. For instance, writing an I18N ready application that supports the writing systems for Japan and English, including the special sorting for the different alphabets. The user interface of an I18N ready application is still in English, but the base code supports the language requirements for both languages.

Apart from general considerations about I18N processes, there are no specific requirements for Symbian localization. Naturally all the localizable resources will have to be placed in the .rss files and not within the application's code.

The I18N process for the Symbian applications is nevertheless rather transparent to developers and translators alike. Developer should not be concerned with how different the application might look when localized because the sizes of most of the visual components are determined at run time by the operating system and also the translator is quite free when choosing the appropriate translations without the constraints coming from the differences between languages.

By default Multilizer localizes all strings in the .rss file. However you can disable localization of a string by adding an exclude tag. It is a `nomlz` comment at the end of the line where the string is located.

```
RESOURCE TBUF text1 { buf="Do not scan me"; } //nomlz
```

Multilizer does not localize the above string.

Sometimes it is useful to add a comment for a string. The comment gives translator an extra information about the string. Comments are added using the include tag. It is similar to the exclude tag but the tag is `mlz`. Any string after the tag is considered as a comment.

```
RESOURCE TBUF text1 { buf="Hello world"; } //mlz This is a comment
```

Multilizer add "This is a comment" comment for the "Hello world" string.

Symbian devices have limited screen size. This is why the length of the string must be as short as possible. Resource files support rather long strings. Up to 518 characters. In most cases this is way too long. This is why you can set the maximum string that the translation can have. Use the include tag to set the maximum length. The length can be given in characters or in pixels. To use characters add `MaxChars=N` at the end of the comment where N is the maximum length of the string in characters. To use characters add `MaxPixels=N` at the end of the comment where N is the maximum length of the string in pixels.

```
RESOURCE TBUF text1 { buf="Hello world"; } //mlz MaxChars=15
RESOURCE TBUF text2 { buf="Hello world"; } //mlz This is a comment
MaxPixels=30
```

The first string has a maximum length of 15 character. The second string has maximum length of 30 pixels and also a comment.

The tagging works only on resource files for Symbian OS 6.1 or later. You can change the tag strings. See online help to get more information.



NOTE!

Creating a New Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch Multilizer. Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

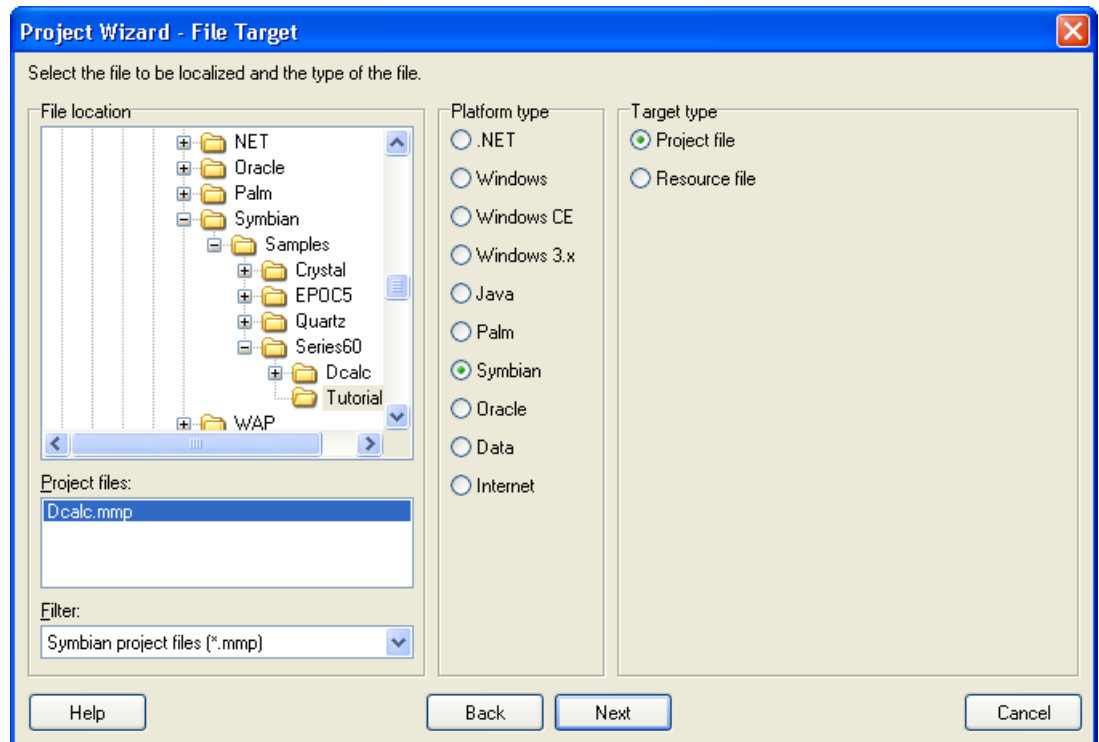


Figure 112 The File Target sheet is used to specify the project file to be localized

This sheet specifies the file to be localized. Choose the `<multilizer>\Symbian\Samples<sdk>\Tutorial` subfolder. The Platform type should be *Symbian* and the Target type should be *Symbian project file*. If they are wrong, check the right types.

Press the **Next** button. The Symbian sheet appears:

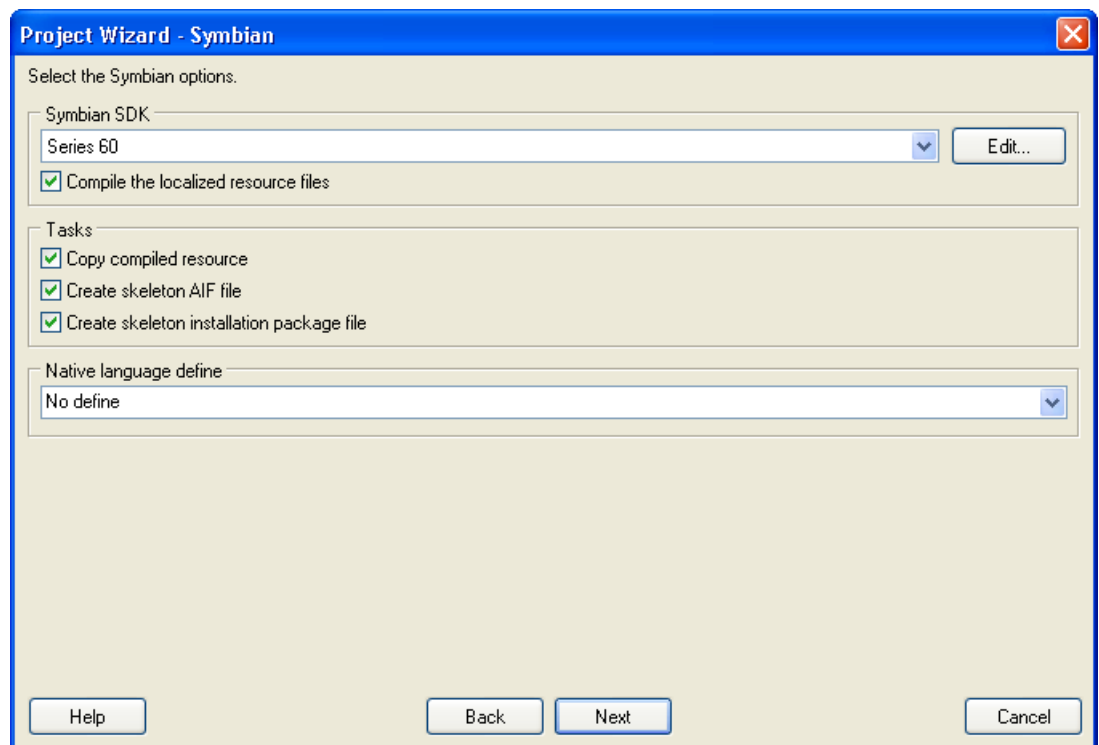


Figure 113 The Symbian sheet is used to set the Symbian specific options.

This sheet specifies the Symbian SDK and other Symbian specific options. Selecting the right Symbian SDK is important. If you have multiple Symbian SDKs installed on the computer the list will have several items. Select the one that the project file (.mmp) you

selected uses. If the list does not have that SDK press **Cancel**, choose **Tools | Options | Symbian** to add the SDK, and run Project Wizard again. To select the right native language define refer the *Conditional Compiling* paragraph later on this chapter.

Accept the values by pressing the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

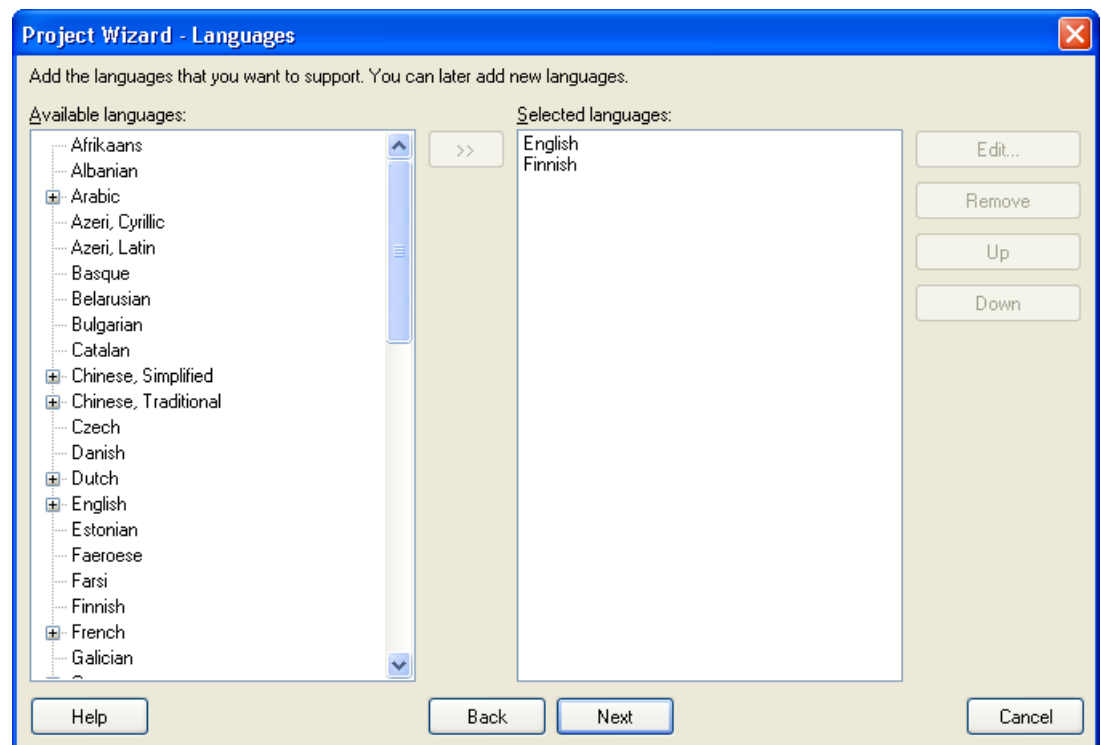


Figure 114 English and Finnish added to project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

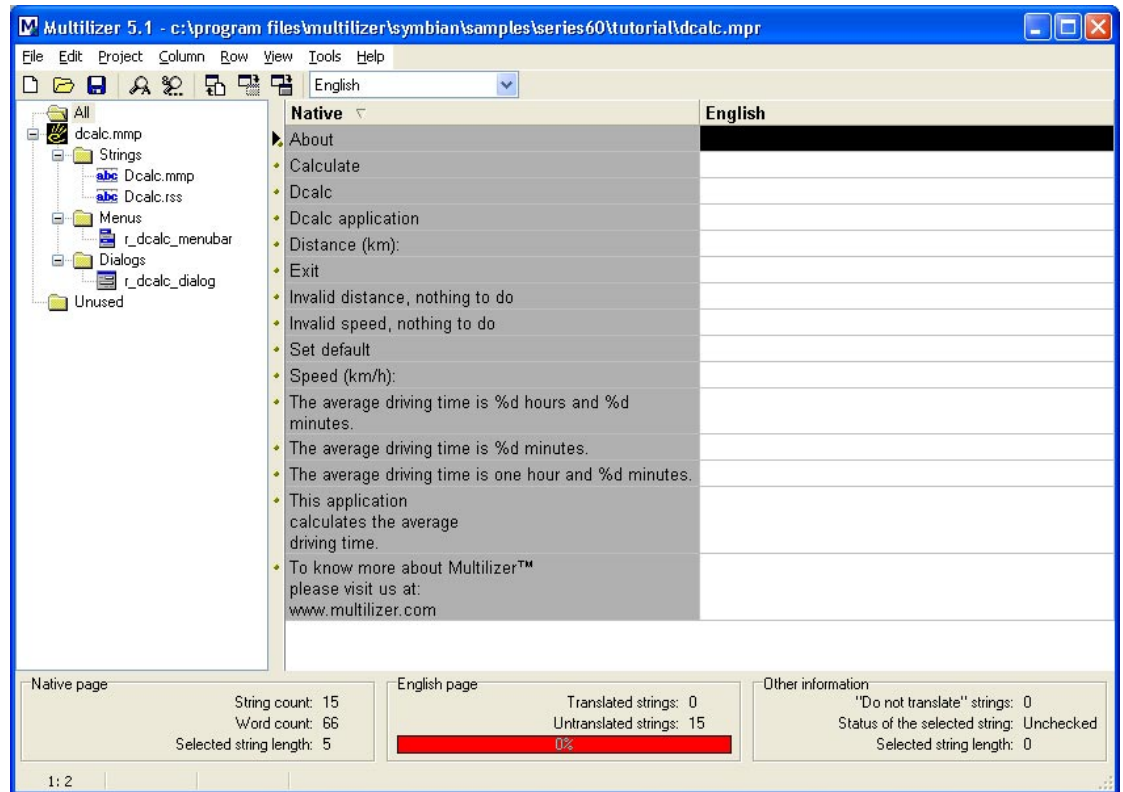


Figure 115 The project grid

On the left side there is a tree view that contains the targets and the items they contains. Our project contains one target, dcalc.mmp, and it contains two string tables, one dialog and one menu. The first string table is a special one: it contains the application name to be used in the AIF resource file and the installation name to be used in the package file. To show only the string belonging to a specific item select the item in the tree. On the right side there is the editing grid. It contains the native column and the English column. To show another language select the language from the combo box above the grid.

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project, save it by choosing **File | Save**.

Create the localized resource files by choosing **Project | Create Localized Files**. This creates the localized resource files (.rsc) in the language specific sub-directories ('en' and 'fi').

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run. This will upload the localized resource file to the Symbian emulator and launches the emulator.

After the emulator appears, press the Dcalc icon to start the application. The localized Dcalc appears.



Figure 116 Dcalc application with a Finnish user interface



For more information about translating a project with Multilizer, see the Translator's Manual.

Deploying

Symbian applications are deployed using installation files (.sis). It is a binary file containing the application files and the resource files. The installation package can contain resource files in several languages. To create an installation file you have to first write a package file.

Multilizer creates a skeleton package file for you. The name of the skeleton package file equals to the Symbian project file (.mmp) but it has .pkg.ml extension. To take the package file in use remove the .ml extension from the file, add the missing parts and finally run `makesis` tool to create the .sis file.

When scanning a Symbian project file Multilizer adds an application name string to the Multilizer project. The string is "<filename> application" where filename is the file name of the Symbian project file without extension (e.g. `Dcalc.mmp` -> "Dcalc application"). This string is the installation name. Multilizer writes its translations to the package file.

The following package file is for Dcalc.

```
&AM, FI
#{ "Dcalc application", "Dcalc-ohjelma"}, (0x10008ace), 1, 0, 0, TYPE=SUSAPP
"Dcalc.app" - "!:\system\apps\Dcalc\Dcalc.app"
{
  "en\Dcalc.rsc"
  "fi\Dcalc.rsc"
} - "!:\system\apps\Dcalc\Dcalc.rsc"
```


The file contains two languages: English and Finnish. The English installation name is "Dcalc application" and the Finnish one is "Dcalc.ohjelma". The English resource file is `en\Dcalc.rsc`, the Finnish one is `fi\Dcalc.rsc`.

**NOTE!**

The default English country in Multilizer is United States. If you add a country neutral language to the Multilizer project (e.g. English) Multilizer uses the Symbian installation language code of English (United States) that is "AM". In order to use "EN" add English (United Kingdom) to the Multilizer project.

**NOTE!**

Multilizer creates a skeleton package file only if you have specified a Symbian project file (`.mmp`) to be localized. If you specify a Symbian resource file (`.rss`) you have to create the package file yourself.

Conditional Compiling

Symbian SDK documentation encourages to use conditional compiling when localizing resource files. The active language is specified by defining a define that enables language related sections. The following sample code contains three languages: English (US), English (UK), and Finnish.

```
#ifndef defined LANGUAGE_01
    LBUF { txt="Localise"; },
#elif defined LANGUAGE_09
    LBUF { txt="Lokalisoi"; },
#elif defined LANGUAGE_10
    LBUF { txt="Localize"; },
#endif
```

By defining `LANGUAGE_01` the resource file will be in English (United Kingdom). By defining `LANGUAGE_09` the resource file will be in Finnish. By defining `LANGUAGE_10` the resource file will be in English (United States).

Similarily the `.mmp` file contains the `lang` statement.

```
lang 01 09 10
```

When compiling the project the resource file will be compiled once for each language. The resource will be three resource files each having different file extension (`.r01`, `.r09`, `.r10`).

Some resource files use different approach. They do not write any strings to the `.rss` file but use constant defines instead that are locating in the language files (`.l??`).

```
#ifndef LANGUAGE_01
    #include "Language.l01"
#endif
#ifdef LANGUAGE_09
    #include "Language.l09"
#endif
#ifdef LANGUAGE_10
    #include "Language.l10"
#endif
```

...

```
LBUF { txt=LANG_TXT; },
```

Each `Language.l??` file contains string defines that hold the actual strings. For example `Language.l01` would be:

```
#define LANG_TXT "Localise"
```

This is very complicated. You have to resource strings twice. First you have to resource strings from the source code to the resource files (`.rss`) and then from the resource file to the language files (`.l??`). Moreover, you loose the string context when you move it from its location in the `.rss` file to the define in the `.l??` file. The developer does not see the actual string by viewing `LBUF { txt=LANG_TXT; }` code. Instead he or she have to open `Language.l01` file, search for `LANG_TXT`, and only after that he or she will see the actual string. Updating resource files containing conditional compiling is hard, slow and error prone.

Multilizer makes this all much easier. When writing Symbian resource files you do not have to use any conditional language defines, language file, or lang statements but write resource files using only one language (e.g. English). When building Multilizer creates the localized files for you. This makes the original resource files much easier to maintain. There is no translation data in the resource file – only the original resource data. Also there is not risk that your translator will accidentally corrupt the resource file. The previous resource example code will be much more simpler.

```
LBUF { txt="Localize"; },
```

If you still use conditional compiling Multilizer tries to detect the defines used by the resource file. Select the define that you want to use. If not present type the define. Use **Project | Targets** dialog to edit the define of the target. This makes it possible for your to select the native language.

If you have previously used the conditional compiling to localize you application switching to Multilizer is very easy. All you have to do is to create a Multilizer project. When creating the project the following message box will appear.

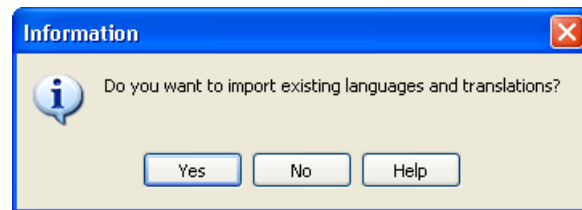


Figure 117 A message box that shows that there are existing translations.

Press **Yes** to import the initial languages and translations from a resource file using thr conditional compiling. From now on you do not have to use conditional compiling any more. You can delete the loc files and start hard coding the string into the resource files (.rss).

14

Palm

In this tutorial we are going to create a localized Palm application. The application will be a simple driving-time calculator, Dcalc, which a user can use to calculate the average driving time for a given distance.

The Dcalc application is very simple but nevertheless uses most Multilizer features. The creation of the application is divided into several lessons, each covering one or more Multilizer functions.

Palm Localization

Palm applications contain the resource data in the application files (.prc). Multilizer creates the localized application files from the original application file. The following picture describes the Palm localization process.

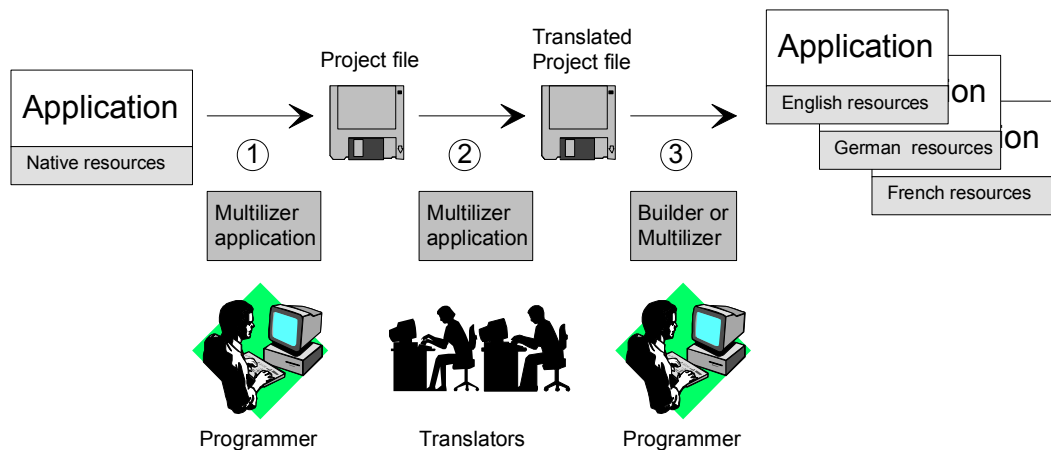


Figure 118 Palm localization process

The process of localizing a Palm application can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the original application file. Multilizer saves these strings into the project file.
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to create the localized application files.

The above will result in one application file for each language and one localized overlay file for each localized language.

The following figure shows the files that Multilizer uses in the Palm localization process.

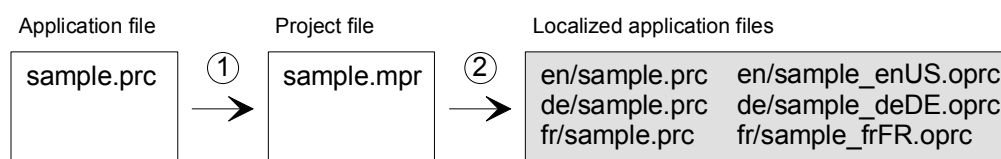


Figure 119 The files of the Palm localization process



To use a localized application, upload the localized application file (e.g. `de/sample.prc`) instead of the original application file (`sample.prc`) to the Palm device, or upload the original application file (`sample.prc`) and the overlay file (`de/sample_deDE.oprc`) to the Palm device.

In order to run the application Multilizer needs to know the location of the Palm emulator or simulator. The emulator can be used to test Palm application up to OS version 4. The simulator can be used to test any Palm application. By default Multilizer tries to find the emulator in its default installation directory. The location of the simulator can not be detected. You must manually set it. To check or change Palm settings, launch Multilizer and choose the **Tools | Options | Palm** menu. The following dialog box appears:

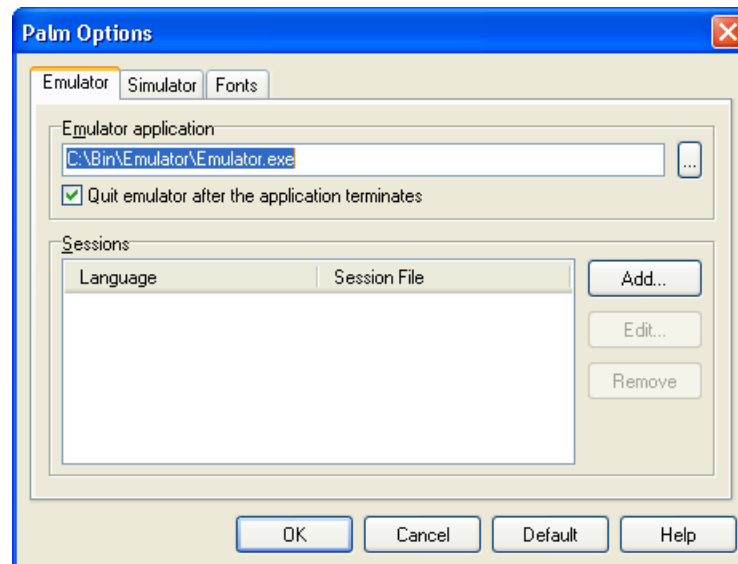


Figure 120 The Palm options dialog box

If the emulator or simulator edit boxes are empty press the **Default** button. If they are still empty press the ... button to set the paths manually. Press Help to get more information about Palm options.

Application With An English User Interface

We could start from scratch but in most cases it is a completed application or at least an application under development that you want to globalize. In this example an example application shipped with Multilizer is used. The `Palm\Samples\Tutorial` contains the English Dcalc. Open it, compile it, and finally run it.

The application should look like this:

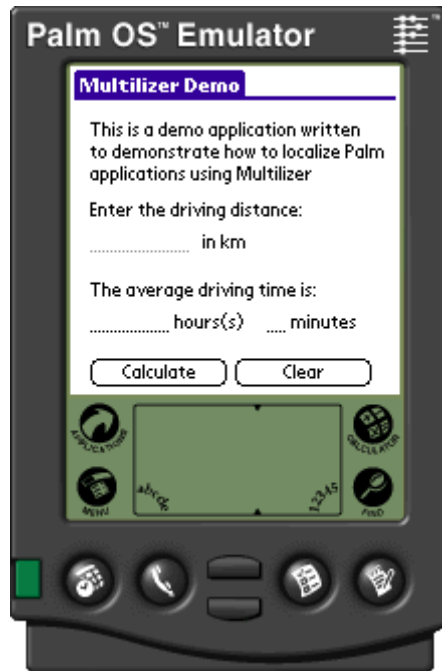


Figure 121 An English Palm application

The user interface language is English. In the following chapters we will localize Dcalc step-by-step.

Internationalization

Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development. I18N is defined as the set of processes, tools, coding techniques and procedures used to write a software program that supports all of the language requirements and country conventions of all of the countries where the SW program will be used. For instance, writing an I18N ready application that supports the writing systems for Japan and English, including the special sorting for the different alphabets. The user interface of an I18N ready application is still in English, but the base code supports the language requirements for both languages.

Apart from general considerations about I18N processes there are some specific requirements for Palm applications to be localized correctly. For a correct and complete I18N of Palm applications all the localizable resources will have to be placed in an external resource file (.rsrc) that will be eventually compiled and linked to the final .prc file (Palm executable).

All strings that are “hard coded” into the code section of the application will not be extracted and therefore not localized. In order for the strings to be localized they must be outside your source code.



NOTE!

Multilizer will scan and localize the following resources:

- tFRM, form resources
- Talt, alert dialog boxes resources
- tSTR, single string resources
- tSTL, string list resources
- tAIN, application icon name resources
- tAIS, application info string list resources
- MBAR, menu resources

- tver, version string resources

Multilizer will not scan user-defined resources, so if you want to ensure maximum compatibility with the I18N process you should use tSTR and tSTL to store the strings used by your application.



If the application has been developed using J2ME (Java to Micro Edition) then you should localize it using Multilizer's Java support and not Multilizer's Palm support. The reason is because even though the compiled file is a prc file the resources are not all of the types defined above and therefore not all the strings contained in the application will be reported correctly.

Developers who are aiming at a fully localizable Palm application should be aware of the limitations of the translation process, being the source of a compiled application. If you want to make sure that your application's characters can be displayed fully also in other languages you should think in advance about some extra space in forms' components. For instance button should, as a rule of thumb, have space to accommodate a text twice as large the one you have in the native version. Using this approach, at design time, will ensure a certain freedom to the translator and reduce the possibilities of having to redesign the visual components to fit new strings. Also the positioning of the components in forms should be considered as important. Text labels, for instance, will resize correctly when translated but if two labels are placed one after the other in a form then there is always the chance of the two of them overlapping which will result in hidden text.

Another possible cause of hidden text might come from string lists and combo boxes in the case when the length of the combo box is enough to show only the largest of the native strings in the list. During translation it is most likely that some strings will be shorter but also that some will be longer and if such is the case then the combo box will not be showing those strings in full. Also in this case it is advisable, when possible, to have some extra space in the combo box to fit possibly larger texts.

Menus should not be of concern because Multilizer will recalculate the menu items sizes during the localization process resulting in correct resizing and repositioning.

Creating a Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

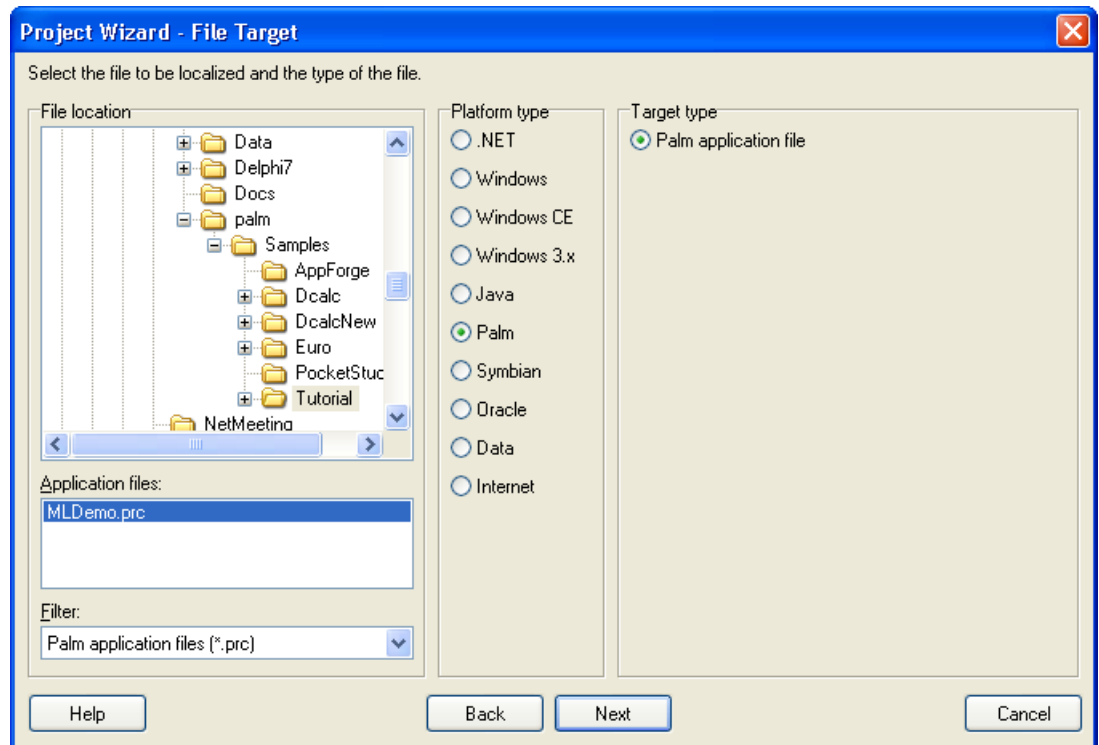


Figure 122 The File Target sheet is used to specify the file to be localized

This sheet specifies the location of your application. Choose the <mldir>\Palm\Samples\Tutorial subfolder of your Multilizer setup. Project Wizard detects the platform and project types. The Platform type should be *Palm* and the Target type should be *Palm application file*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the >> button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

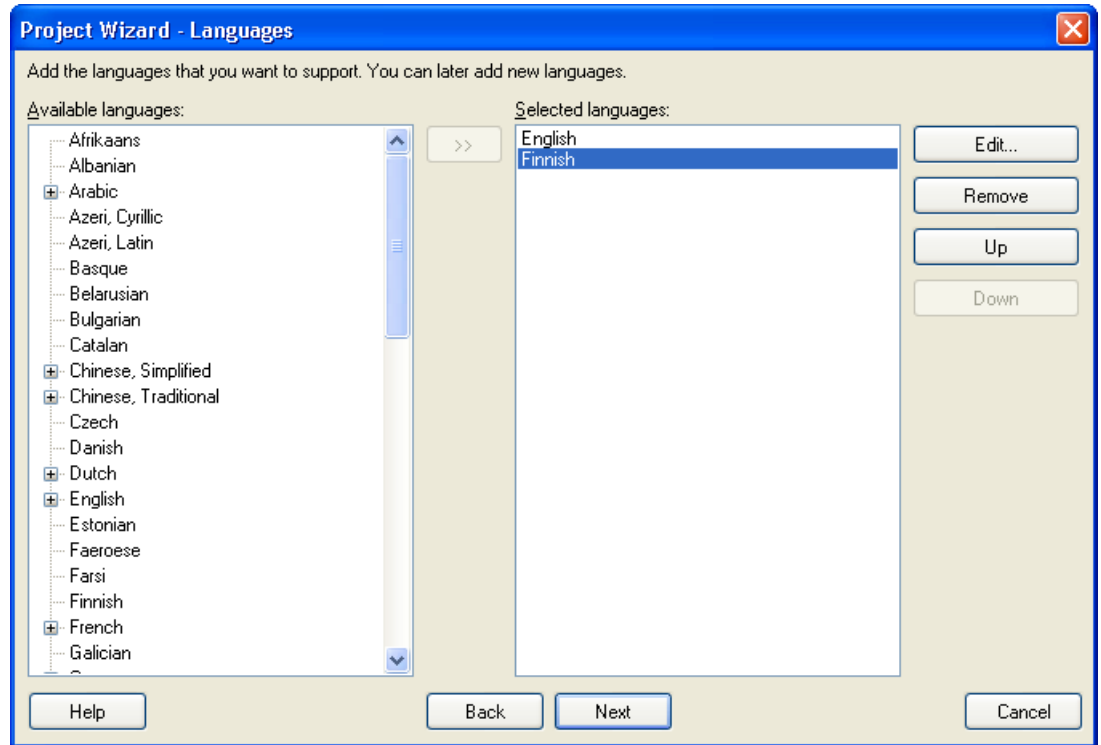


Figure 123 English and Finnish added to project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

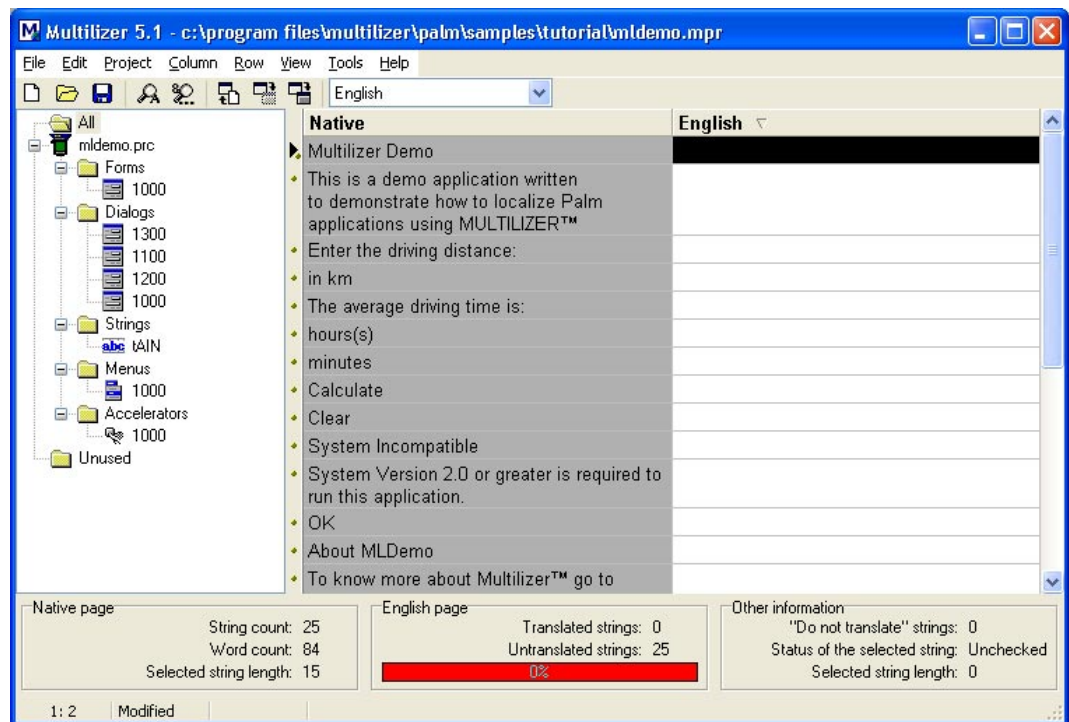


Figure 124 The project grid

On the left side there is a tree view that contains the targets and the items they contains. Our project contains one target, `mldemo.prc`, and it contains one form, several dialogs, one string table, one menu and one accelerator table. To show only the string belonging to a specific item select the item in the tree. On the right side there is the editing grid. It contains the native column and the English column. To show another language select the language from the combo box above the grid.

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project, save it by choosing **File | Save**.

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized application files (`.prc`) in the language specific sub-directories ('en' and 'fi').

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and choosing **Run**. This will upload the localized resource file to the Palm emulator, launches the emulator and finally starts the application.



Figure 125 Localized Dcalc application

PalmOS 3.0 or later support overlays. They are PRC files that contain only resources. You deploy the overlay with the application PRC file. By default Multilizer creates the localized application files and the overlay files. To change the settings select `mldemo.prc` from the left side tree view and right-click to open the popup menu. Choose **Edit target** to open the Palm Binary File Target dialog.

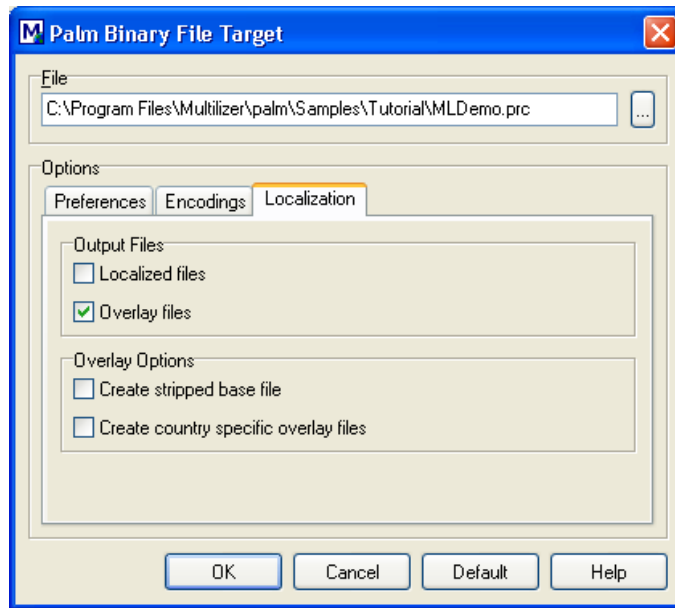


Figure 126 The Palm Binary File Target dialog

Select the Localization sheet and uncheck the *Localized files* check box. Next time you build the localized files Multilizer will not create the localized files, only the overlay files.

If you have configured both the Palm emulator and simulator, Multilizer will use the simulator. You can force it to use the emulator by selecting the Preferences sheet and checking the *Use emulator when both emulator and simulator exist* check box.

For more information about translating a project with Multilizer, see the Translator's Manual.



15

XML

In this tutorial we are going to localize XML files. The file will be a simple product catalog, which contains consumer products.

XML Localization

XML file contains data. Multilizer creates the localized XML files from the original XML source file. The following picture describes the XML localization process.

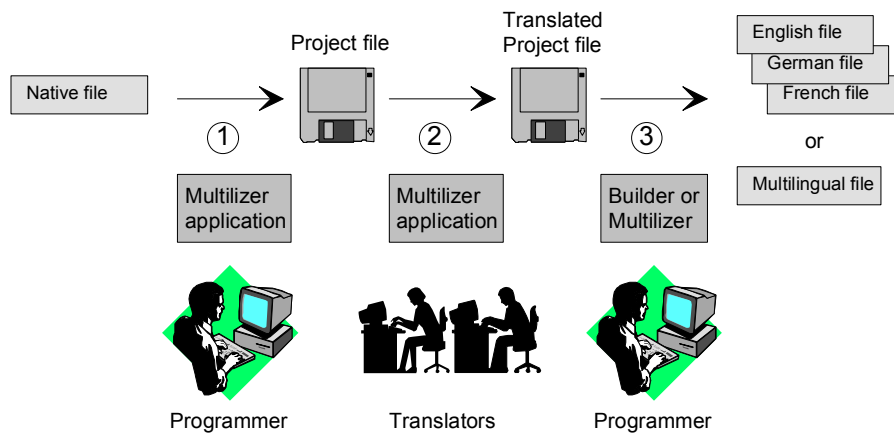


Figure 127 XML localization process

The process of localizing XML files can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the XML file. Multilizer saves these strings into the project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to create the localized XML files.



The above will result in one XML file for every language. The structure of the localized file is identical to the original one. The only difference is that the selected string data has been translated to the target language. Multilizer can create one XML file that contains selected tags in several languages.

The following figure shows the files that Multilizer uses in the XML localization process.

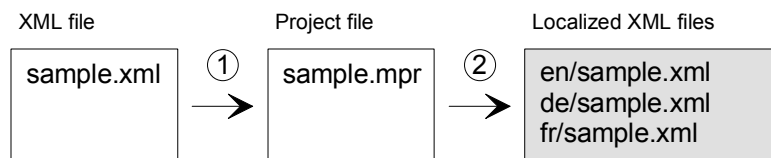


Figure 128 XML localization process files

To use the new localized resource (e.g. de/sample.xml), deploy or use it instead of the original XML file (sample.xml).

English File

The `<mldir>\Data\Samples\XML\Tutorial` contains the English XML file. It is a simple product catalog file that contains three products. Each product has a name, a price and a description.

```
<?xml version="1.0" encoding="UTF-8"?>
<PRODUCTS>
  <PRODUCT Id="0">
    <NAME>Nokia 9210</NAME>
    <PRICE>800</PRICE>
    <DESCRIPTION>The third generation communicator having a color
display, Symbian operating system and Java</DESCRIPTION>
  </PRODUCT>

  <PRODUCT Id="1">
    <NAME>Fiskars Handy</NAME>
    <PRICE>39</PRICE>
    <DESCRIPTION>This ax has fiber class arm that is virtual
unbreakable</DESCRIPTION>
  </PRODUCT>

  <PRODUCT Id="2">
    <NAME>Polar M52</NAME>
    <PRICE>129</PRICE>
    <DESCRIPTION>A heart rate meter that has a build in fitness
test</DESCRIPTION>
  </PRODUCT>
</PRODUCTS>
```

The DESCRIPTION tag contains English text that needs to be localized.

Creating a New Project

Double-click the Multilizer icon from the Multilizer program group to launch Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button.

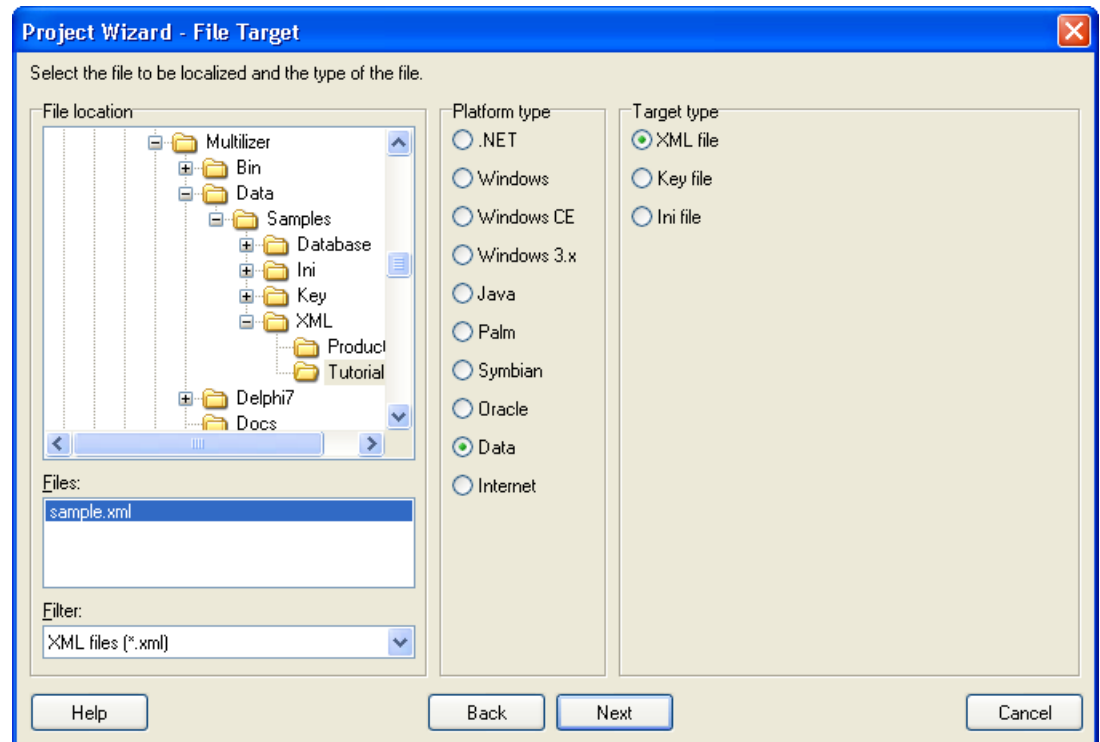


Figure 129 The File Target sheet is used to specify the XML file to be localized.

This dialog specifies the directory where your application is located. Choose the `<multilizer>\Data\Samples\XML\Tutorial` subfolder of your Multilizer setup. Project Wizard detects the platform and project types. The Platform type should be `Data` and the Target type should be `XML file`. If they are wrong, check the right types.

Press the **Next** button. The XML sheet appears:

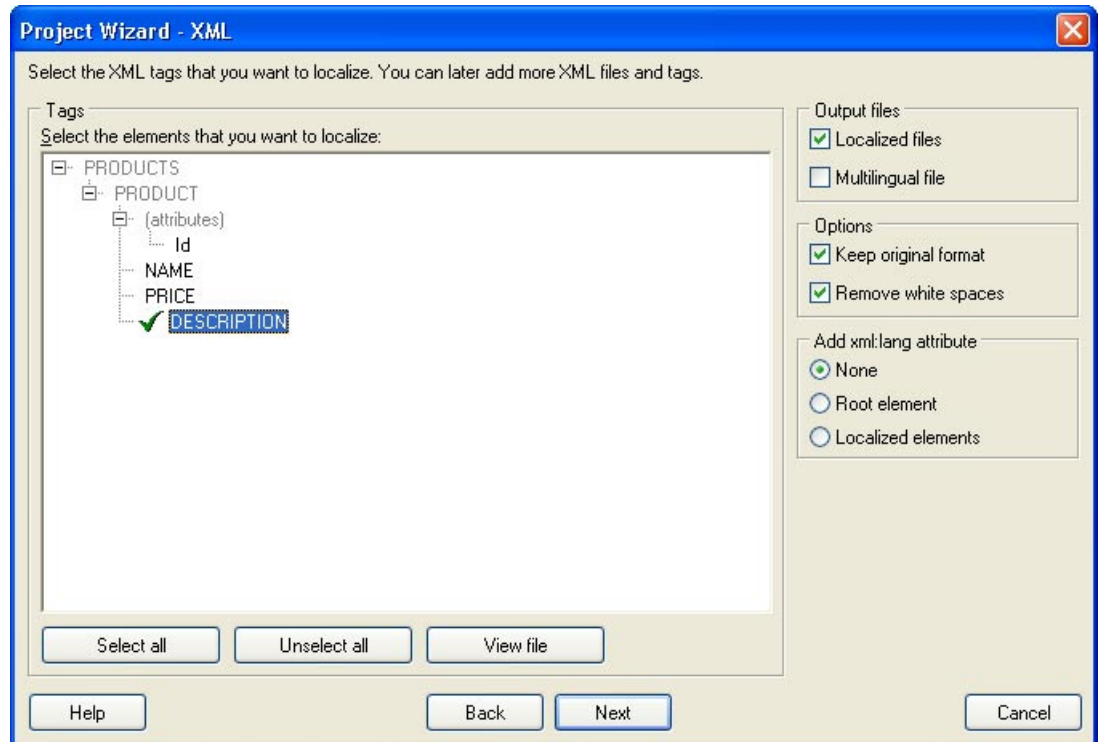


Figure 130 The XML sheet is used to select the tags to be localized.

This sheet specifies the tags that will be localized. Open the tree and double click the DESCRIPTION leaf. A green check marks appears. This makes Multilizer to localize all DESCRIPTION blocks that belong to the PRODUCT block that belongs to the PRODUCTS block. Double click the DESCRIPTION node to check it.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Press the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

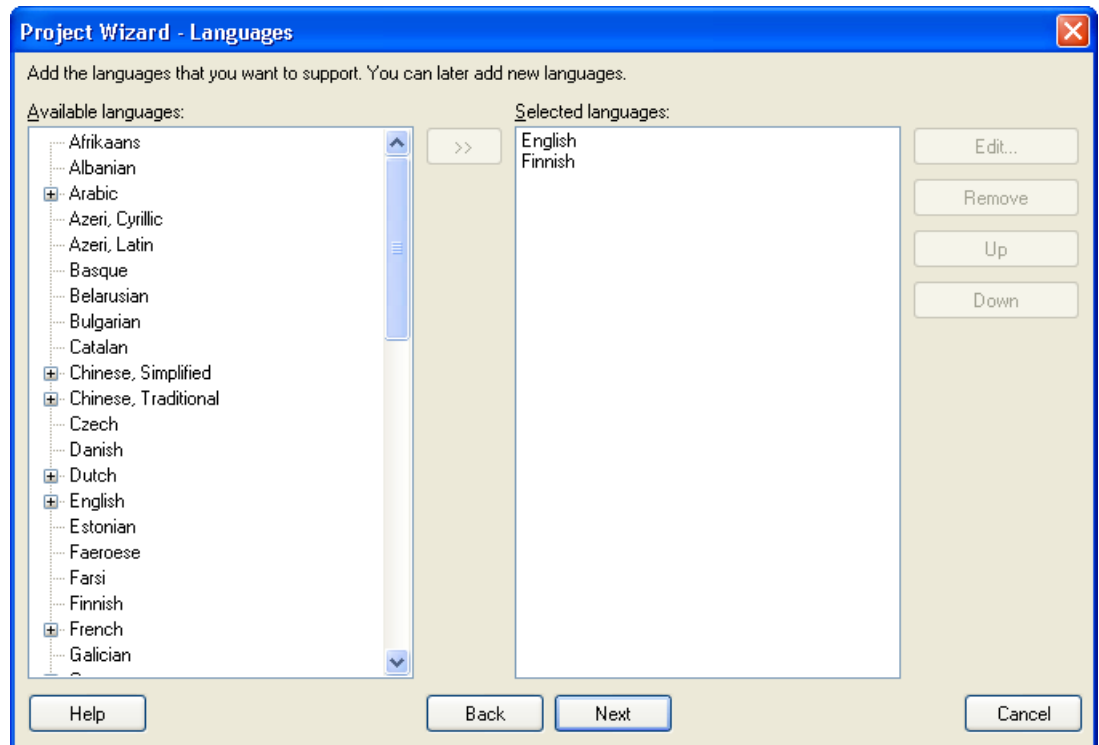


Figure 131 English and Finnish added to a project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

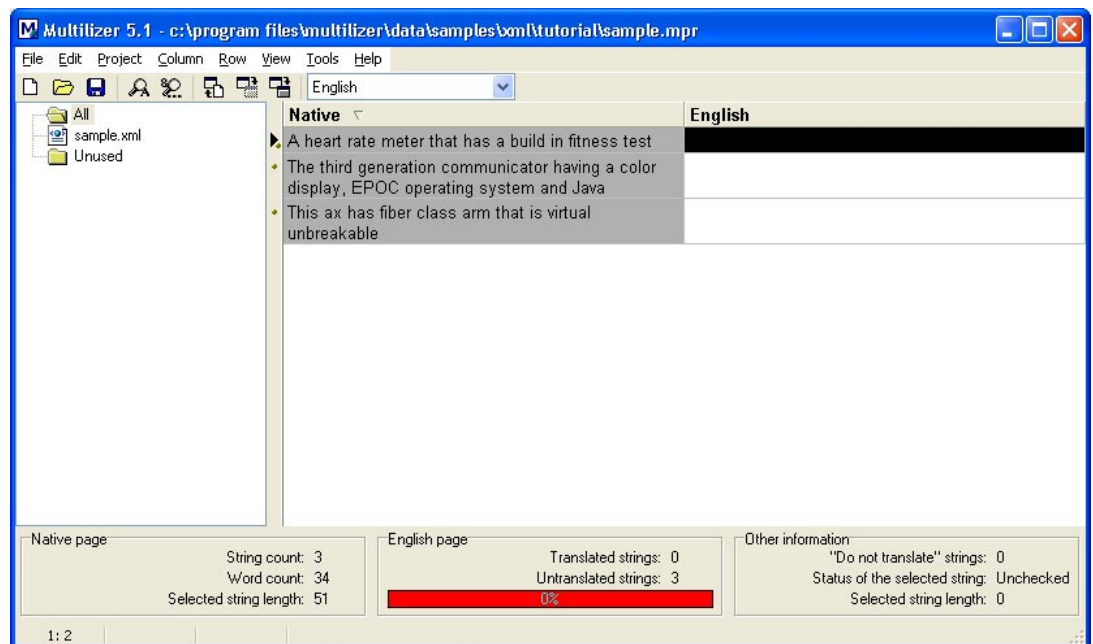


Figure 132 The project grid

Save the project by choosing **File | Save**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Create the localized XML files by choosing **Project | Create Localized Files**. This creates the localized XML files in the language specific sub directories (e.g. 'en\sample.xml' and 'fi\sample.xml').

For example the Finnish XML file (fi\sample.xml) would look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<PRODUCTS>
  <PRODUCT Id="0">
    <NAME>Nokia 9210</NAME>
    <PRICE>800</PRICE>
    <DESCRIPTION>Kolmannen sukupolven kommunikaattori, jossa on
vÄrÄnÄyttö, Symbian-kÄyttöjärjestelmä ja Java</DESCRIPTION>
  </PRODUCT>
  <PRODUCT Id="1">
    <NAME>Fiskars Handy</NAME>
    <PRICE>39</PRICE>
    <DESCRIPTION>Lasikuituvartinen kirves, joka on käytännössä
rikkoutumaton</DESCRIPTION>
  </PRODUCT>
  <PRODUCT Id="2">
    <NAME>Polar M52</NAME>
    <PRICE>129</PRICE>
    <DESCRIPTION>Sykemittari, jossa on kuntotesti</DESCRIPTION>
  </PRODUCT>
</PRODUCTS>
```

The structure is identical to the native (English) one but the DESCRIPTION block has been translated to Finnish.

Multilizer makes it possible to create multilingual XML files as well. Choose **Project | Targets**. Select the XML file and press **Edit**. The XML File Target dialog appears. Check the Multilingual file in the Preferences sheet.

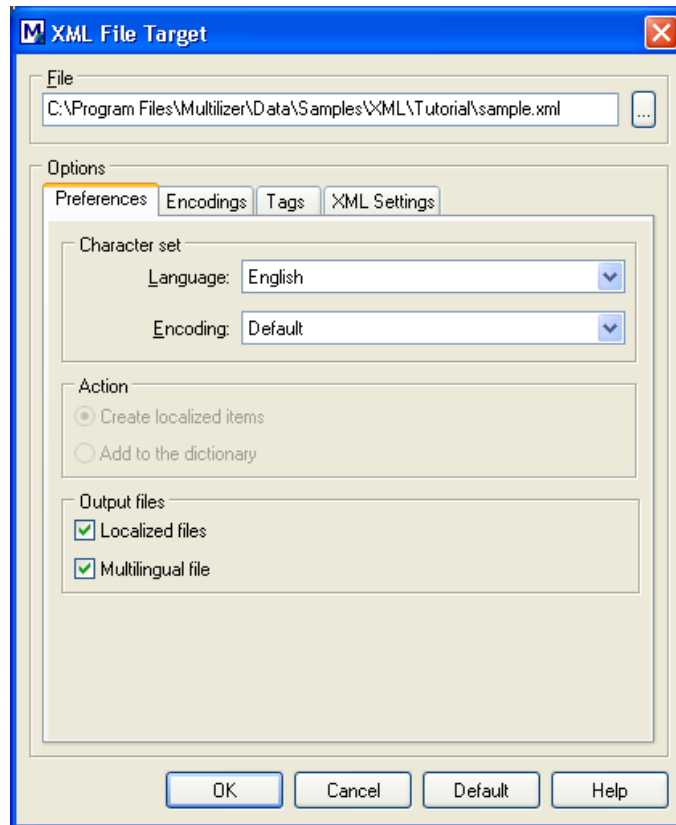


Figure 133 XML File Target dialog

Next time you create the localized files Multilizer will create a multilingual XML file in the all subdirectory (e.g. all\sample.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<PRODUCTS>
  <PRODUCT Id="0">
    <NAME>Nokia 9210</NAME>
    <PRICE>800</PRICE>
    <DESCRIPTION>The third generation communicator having a color
display, EPOC operating system and Java</DESCRIPTION>
    <DESCRIPTION xml:lang="en">The third generation communicator having a
color display, EPOC operating system and Java</DESCRIPTION>
    <DESCRIPTION xml:lang="fi">Kolmannen sukupolven kommunikaattori, jossa
on värinäyttö, EPOC-käyttöjärjestelmä ja Java</DESCRIPTION></PRODUCT>

  <PRODUCT Id="1">
    <NAME>Fiskars Handy</NAME>
    <PRICE>39</PRICE>
    <DESCRIPTION>This ax has fiber class arm that is virtual
unbreakable</DESCRIPTION>
    <DESCRIPTION xml:lang="en">This ax has fiber class arm that is
virtual unbreakable</DESCRIPTION>
    <DESCRIPTION xml:lang="fi">Lasikuituvartinen kirves, joka on
käytännössä rikkoutumaton</DESCRIPTION></PRODUCT>

  <PRODUCT Id="2">
    <NAME>Polar M52</NAME>
    <PRICE>129</PRICE>
    <DESCRIPTION>A heart rate meter that has a build in fitness
test</DESCRIPTION>
    <DESCRIPTION xml:lang="en">A heart rate meter that has a build in
fitness test</DESCRIPTION>
    <DESCRIPTION xml:lang="fi">Sykemittari, jossa on
kuntotesti</DESCRIPTION></PRODUCT>
</PRODUCTS>
```


The structure is identical to the native (English) one but there is one DESCRIPTION block for each language. Each added and localized tag also contains the XML language attribute (`xml:lang`).

16

WAP

In this tutorial we are going to create a localized WAP application that consists of one wml file and one wmls file. WML is a markup language based on Extensible Markup Language (XML). It is designed to be used to specify application content for devices like mobile phones. This content can be represented with text, images, selection lists, etc. Simple formatting is also supported. This content, however, is all static and there is no way of modifying this without modifying wml itself. WMLScript (WMLS) was designed to overcome these limitations and to provide programmable functionality that can be used over narrowband communication links in clients with limited capabilities. The application will be a simple driving-time calculator, Dcalc that a user can use to calculate the average driving time for a given distance. The dcalc WAP-application is relatively simple, but still it uses most features of Multilizer. The creation of the application is divided into several lessons, each covering one or more Multilizer functions.

WAP Localization

The wml files contain language data in specific places, i.e. inside tags after type definitions that determine whether the string will be a title, label or something different. Multilizer can find all the localizable strings from the wml files after which it extracts them and places the strings to the Multilizer project file.

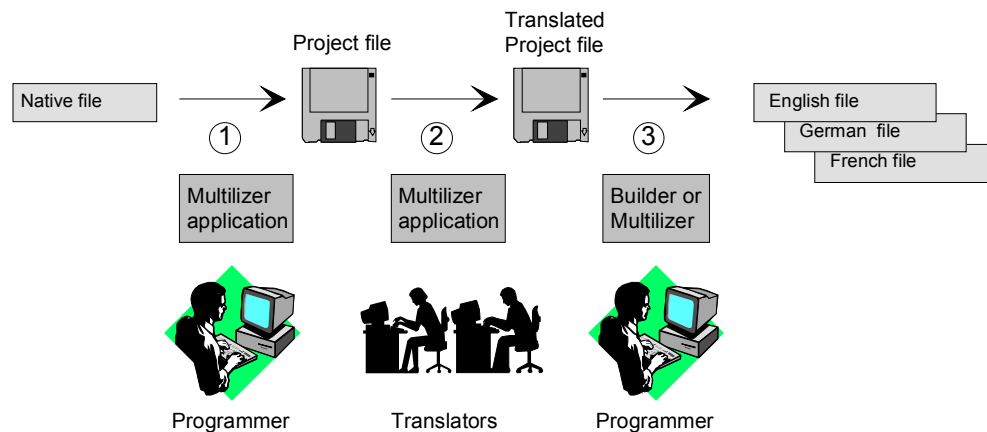


Figure 134 The WML localization process

The process of localizing WML files can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the WML file. Multilizer saves these strings into the project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to create the localized WML files.

The above will result in one WML file for every language. The structure of the localized file is identical to the original one. The only difference is that the selected string data have been translated to the target language.



The following figure shows the files that Multilizer uses in the WML localization process.

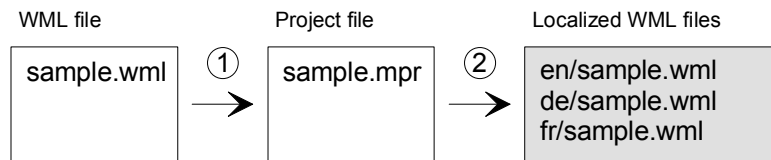


Figure 135 The files of the WML localization process

Most WAP applications use a template or scripting technology to create WML files dynamically at run-time. You can use the above localization process for XSL, JSP and ASP files.

Application with an English User Interface

We could start from scratch but in most cases it is a completed application or at least some specific application under construction that you want to globalize. This is what we are going to do. The `<mldir>\WML\Samples\Tutorial\dcalc.wml` contains the Dcalc sample application for WAP. Compile and run the application.



Figure 136 Dcalc driving time calculator with English user interface

The user interface language is English. In the following chapters we will localize the wml source code of the Dcalc application step-by-step.

Internationalization

Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development. I18N is defined as the set of processes, tools, coding techniques and procedures used to write a software program that supports all language requirements and country conventions of all countries where the application will be used. For instance, writing an I18N ready application that supports the writing systems for Japan and English, including the special sorting for the different alphabets. The user interface of an I18N ready application is still in English, but the base code supports the language requirements of both languages.

When writing code that can be easily localized later on, i.e. internationalization, the following things should be taken into consideration with wml pages:

- Having the right character set in the wml document, i.e. document's character set has to correspond to the actual language used in the document. For example with Western languages it can be either ISO-8859-1 or UTF-8, but for example with Japanese it is SHIFT-JIS or UTF-8. The setting is in the beginning of the document: e.g. "`<?xml version="1.0" encoding="ISO-8859-1"?>`". You can check all existing character sets from <http://www.czyborra.com/charsets/>. UTF-8 works as a character set with all languages.
- Language has to be separately categorized in the wml document; e.g. `<wml xml:lang="fi">` for Finnish or `<wml xml:lang="en">` for English.
- User interface limitations are especially important to take into consideration in the sense that the displays that are used to read wml pages are usually very small. String lengths have to be checked with emulators or with actual equipment to see how everything fits in to the displays of different equipment.
- In WML the user interface and in WML Scripting the data input for example have to be taken into consideration. The normal country specific things, i.e. the user interface must support multiple different countries' cultures and settings. Internationalizing WAP code is getting more and more popular as the popularity of WAP grows.

Creating a New Project

We have now internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

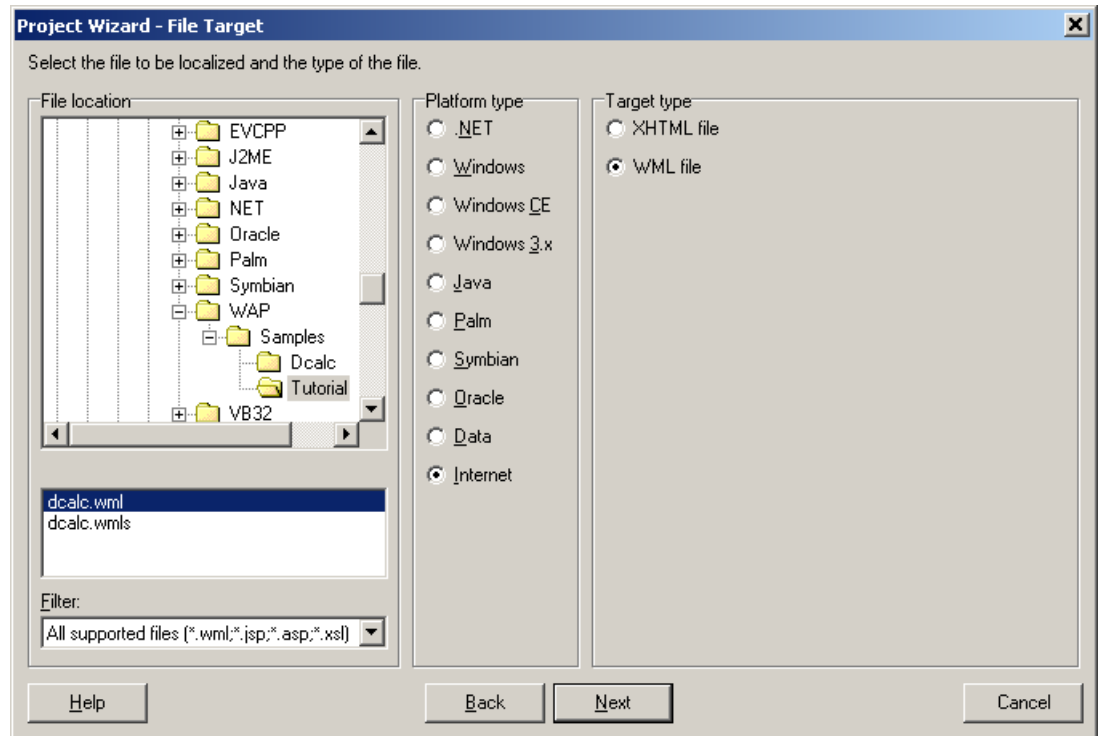


Figure 137 The File Target sheet is used to specify the WML files.

This dialog specifies the directory where your application is located. Choose the `<multilizer>\WML\Samples\Tutorial` subfolder. Project Wizard detects the platform and target types. The Platform type should be *Internet* and the Target type should be *WML files*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

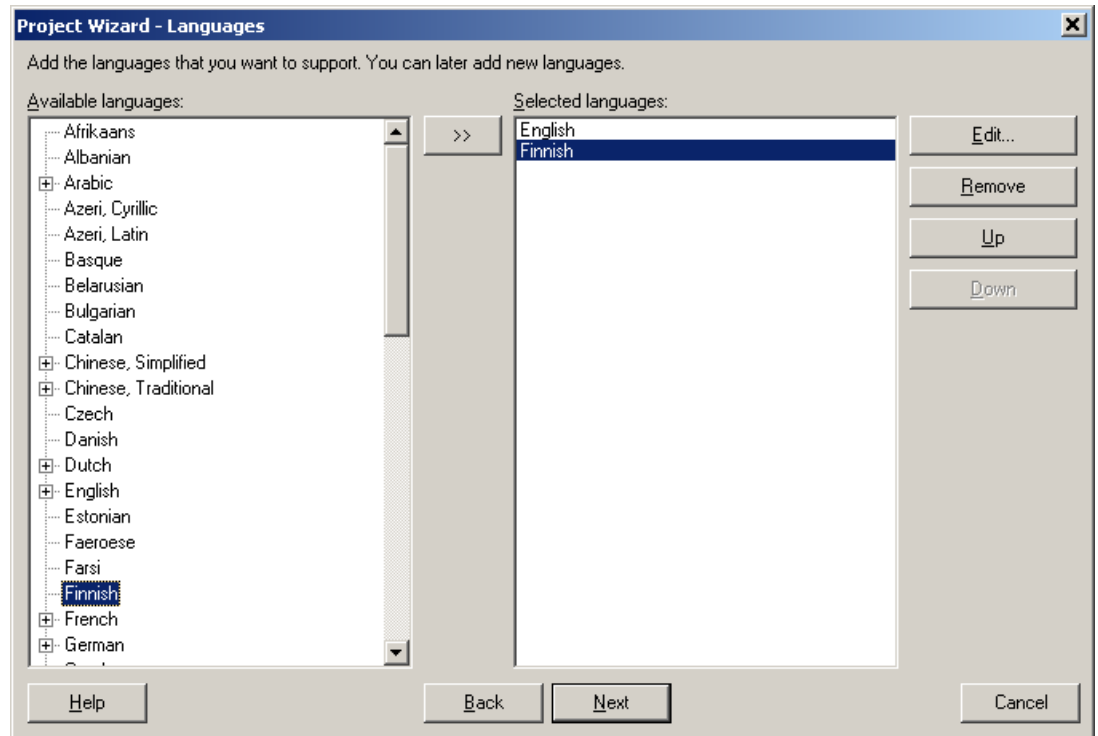


Figure 191 English and Finnish added to the project

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

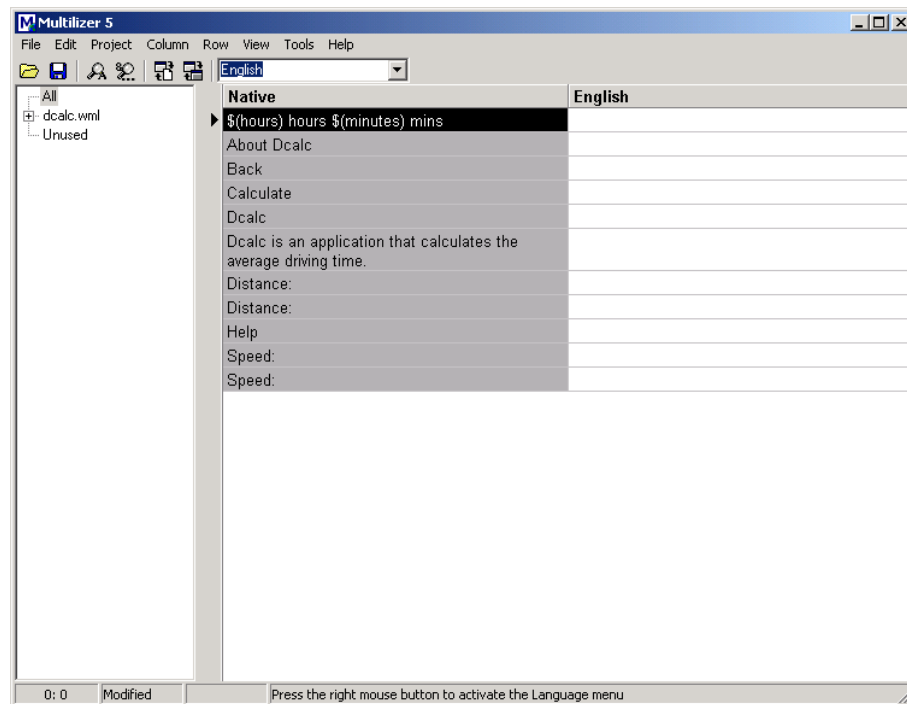


Figure 138 The project grid

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project, save it by choosing **File | Save**.

Create the localized application files by choosing **Project | Create Localized Files**. This creates the localized application files (.wml) in the language specific sub directories ('en' and 'fi').

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.



Figure 139 Localized Dcalc application

The Finnish WML file (fi\dcalc.wml) would look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card id="main" title="Dcalc" newcontext="true">
    <p>
      Etäisyys:
      <input format="N*M" name="distance" value="150" title="Etäisyys:"/>
      Nopeus:
      <input format="N*M" name="speed" value="100" title="Nopeus:"/>
      <br/>
      =
      <u>$ (hours) tuntia $(minutes) min.</u>
    </p>
    <do type="accept" label="Laske">
```

```
        <go href="dcalc.wmls#calculate('hours', 'minutes', '$(distance)',
'$(speed)')"/>
    </do>
    <do type="help" label="Ohje">
        <go href="#about"/>
    </do>
</card>
<card id="about" title="Tietoa Dcalc:sta">
    <p>Dcalc is ohjelma, joka laskee keskimääräisen ajoajan.</p>
    <do type="prev" label="Takaisin">
        <prev/>
    </do>
</card>
</wml>
```

The structure is identical to the native (English) one but strings have been translated to Finnish and file uses UTF-8 character set.

17

Database

In this tutorial we are going to localize database contents. Databases in this tutorial are simple catalogs that contain consumer products.

Multilizer supports three different kinds of database localization. They are: fields localization, tables localization and single table localization. The following chapters will describe them in more detail.

General Considerations on Database Contents Localization and Used Terminology

Before moving to the actual tutorials let's read about issues related to databases in general, issues such as security, architectures, existing localized data and a brief description of the terminology used through this document including the description of some icons used in Multilizer.

Security

Every Relational Database Management System (RDBMS) has built-in security features, obviously to protect the data stored in the databases. If you are planning to localize the contents of databases stored in such systems, it is necessary that you have the required access rights to do so. When you use Multilizer as a database user, you should make sure that you have SELECT, INSERT and UPDATE rights on the tables you want to localize; otherwise the localization will not be possible.

Under some circumstances the database administrator might choose to create different roles to the database according to the tasks of the user.

In your organization, for instance, there could be a user (or role) that has only SELECT rights and another user (or role) that has SELECT, INSERT and UPDATE rights. In this case the first user can use Multilizer to scan the database contents and create a new Multilizer project while the second user can create the localized contents after the project has been translated.

How you, or your database administrator, will share the tasks between different users is up to your organization.



NOTE!

Data visibility is dependent also on RDBMS security that can vary from system to system. On certain systems (such as Oracle), when using Multilizer, the first user might not "see" the database tables whose owner is second user. In this case the first user, when logging into the database from Multilizer, should use second user's username (and a dot) as a prefix to his/her username.

For instance, let's suppose that first user's name is "translator" and second user's name is "owner". The first user wants to use Multilizer to scan the data contained in a database created (or otherwise owned) by the second user. When the first user selects the database to log into, he/she will use the username "owner.translator" and his/her own password.

On some systems it might be enough to prefix the user name with the Schema associated to the database tables.

Choosing the right connection type

Even though Multilizer supports ADO/ODBC and BDE compatible connections to remote servers it is a better idea to choose the correct database server type before attempting to

connect. Multilizer will detect which client drivers are present in your system and offer you choices accordingly. **If the server type you want to connect to is listed in the connection dialog drop-down list, please select it instead of ADO/ODBC or BDE compatible.**

Some servers are supported by other than ADO/ODBC or BDE and we strongly advice not to use ADO/ODBC or BDE for the following:

- Oracle
- MySQL
- Interbase
- IBM DB/2

If these types are not listed in the Multilizer database connection dialog then you will need to install the required vendor client drivers, usually supplied with your database software distribution.

Architectures

As you will see in the following chapters, we have defined three specific database architectures on which Multilizer can successfully localize data: fields, tables and single table localization architectures. In fact, even though the requirements are quite different for each of them (please refer to the next chapter), they can be easily mixed according to your specific needs as a result of Multilizer's great versatility.

Each one of them offers advantages and disadvantages compared to the remaining two. The choice should be made based on how the database needs to be structured and accessed, the data it will contain and of course if it has been already planned with the localization process in mind or if you are trying to localize an existing database that was not originally planned for localization.

Chapters 4, 5 and 6 offer more detailed views on these different architectures.

Existing Data

We at Multilizer have tried very hard to build a system that can preserve your existing translations but still **we suggest you to back up your database before proceeding with the localization process.**

The reason is that if you are not an experienced Multilizer user you might accidentally overwrite your existing localized data by, for instance, selecting the wrong field when creating a new project.

In general when Multilizer scans the tables for the first time (or after each rescan) it will detect the data in the native fields, of course, but also in the localized ones so that any existing data will be added to Multilizer's localized columns (whether any data found will replace the translations already existing in your Multilizer is an option in the target editor).

In order for the process to work correctly you need to instruct Multilizer on which table and/or field contains the data for a certain language (please see chapters 7 to 9 on how to do this).

Used Terminology & Icons

Through this document we will use the following terms:

Localizable Field or Table, a field or a table that will contain localized data but that is at the moment empty

Localized Field or Table, a field or a table that contains localized data either as a result of Multilizer use or in the case of existing data prior to Multilizer use.



This icon represents the database. In the tree view it represents the root.



This icon represents a table. It may contain localized fields.



This icon represents a localizable table. It is nested within a table.



This icon represents a field within a table. It may contain localizable fields.



This icon represents a localizable field. It is nested within a field.



This icon shows that a field has been tagged as localizable (for Single Table localization).



This icon shows that a field has been selected as language id field (for Single Table localization).

Field and Table Naming Conventions and Restrictions

In order to facilitate the auto detection of localized fields and tables Multilizer is suggesting a naming convention that, if followed, will make the localization process much faster and easier.

Localized fields within the same table should use the following

Native field:

FieldName

Localized Fields:

FieldNameidID or *FieldName_id_ID* or *FieldNamelD_ID* or *FieldNameID*

Where *id* and *ID* represent the language (ISO-369) and country (ISO-3166) codes. For instance in a table containing a native field *Description* with two localized fields, one for Finnish, the other for English, you may use the following:

Description, *Description_Fi*, *Description_En*

or

Description, *DescriptionFi*, *DescriptionEn*

or any other combination of the above as long as the *FieldName* section of the name is maintained the same.

The naming convention for the localized tables is the same as for the fields except that the convention now applies to tables' names instead of the fields

Native table:

TableName

Localized Tables:

TableNameidID or *TableName_id_ID* or *TableNamelD_ID* or *TableNameID*

Where *id* and *ID* represent the language (ISO-369) and country (ISO-3166) codes. For instance in a database with a native table called *Products* and two localized tables, one for Finnish the other for English, you may use the following:

Products, *Products_Fi*, *Products_En*

or

Products, *ProductsFi*, *ProductsEn*



In the case of tables localization, the field convention naming is **mandatory** and very simple:

Even though the localized table may contain fewer fields than the native, every field contained in the localized table must have the same name as the correspondent field in the native table

Furthermore:

The localized tables and the native table must have the same primary key.

These rules are not as restrictive as they might seem and in fact they help in SQL code as well because the selection of one or the other language is achieved easily by changing only the table name in the queries and not the fields names.

In the above database example the fields in the native table (Products) could be as follows:

Id, Name, Description, Code

and in the localized tables (*ProductsFl* and *ProductsEn*):

Id, Description



In case of a **single table** localization it is essential that the table contains one field dedicated to the language id. Multilizer will use the ISO codes for the language (ISO-369) and country (ISO-3166) combination for every RDBMS except Oracle where Oracle's own language/country codes are used instead. The name of the field can be chosen freely as long as it is a string type and at least 5 characters long (except in Oracle where the maximum length will probably be 3 characters).

So where Oracle will contain the code 'US' for English (United States) rows, all other databases will contain 'en_US'.

Fields Localization

Sometimes database fields contain information that needs to be localized. For example in a product catalog, a description field may contain a short description about the product that the record stores. The description strings need to be localized. The easiest way is to add a new description field for each language. The following picture describes the field localization process.

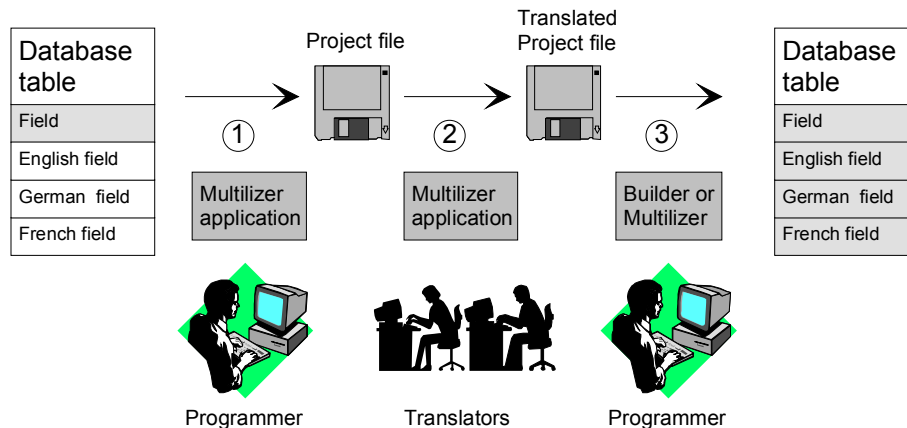


Figure 140 The field localization process.

The database table in the above example contains one field that needs to be localized, Description. Before you can localize the field you have to add the localizable fields into the table. The table in the example above contains localizable fields for three languages: DescriptionEn (English), DescriptionDe (German) and DescriptionFr (French),

The process of localizing table fields can be divided into 3 basic steps:

4. The programmer uses Multilizer to extract strings from the native field(s). Multilizer saves these strings into the project file (.mpr).

5. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
6. The programmer uses Multilizer or Builder to fill the localizable fields with the localized data.

The following figure shows the items that Multilizer uses in the field localization process.

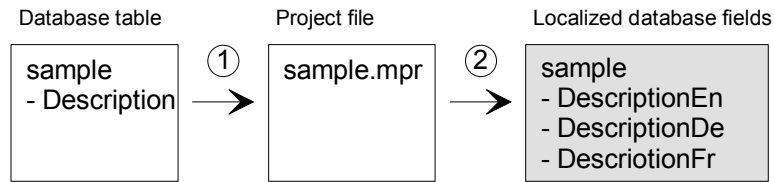


Figure 141 The items of the field localization process.

To use the new localized data replace the native field name (e.g. `Description`) with the localized field name (e.g. `DescriptionDe`) in the SQL statement.

Examples

The following SQL statement selects the description field of the first row in the table

```
SELECT Description FROM sample WHERE Id=1
```

If the active language is German you need to change the statement so that `DescriptionDe` is used instead of `Description`.

```
SELECT DescriptionDe FROM sample WHERE Id=1
```

A more generic way is to store the name of the description field into a resource string in your application and use the localized resource (e.g. localized field name).

Then the SQL statement might be constructed like:

```
'SELECT ' + LoadStr(Description) + ' FROM sample WHERE Id=1'
```

This gets the localized field name from the application resource. If the application language is German the description resource contains "`DescriptionDe`", if the language is French the resource contains "`DescriptionFr`".

The field localization method has quite minimal impact to your source code. The only thing that you have to do is to select the right field name in the SQL statement. Unfortunately you cannot always add new fields easily to database tables. Either the insertion of the fields afterwards is impossible or the insertion would break the existing code. In this case you can use the tables localization.

See the Field localization samples in the
`<mldir>\Data\Samples\Databases\Field` directory.

Tables Localization

In the tables localization architecture the situation is quite different from the fields localization (see previous chapter). In this case we need not to worry about the final number of languages that will be in our database because insertion of a new language is (as we'll see later on) much easier.

The basic principle of this architecture is to isolate all the localizable fields and create a new localizable table for each language.

The localizable tables can be:

- **Clones of the native table:** each localized table will contain ALL the data contained in the native with localized data instead of native
- **Subsets of the native table:** each localized table will contain ONLY the localized fields data plus as many extra fields as those contained in the primary index of the native table.

In both cases the localizable table MUST contain the localizable fields. The choice is dependent on the data your database contains. If there is a large amount of data that need not be localized and maybe requiring large amount of storage (like blobs) then the obvious choice is to subset the translatable tables; in this way there is no unnecessary data cloning. If these are no issues to be concerned with then cloning the tables will work just the same.

The following picture describes the tables' localization process.

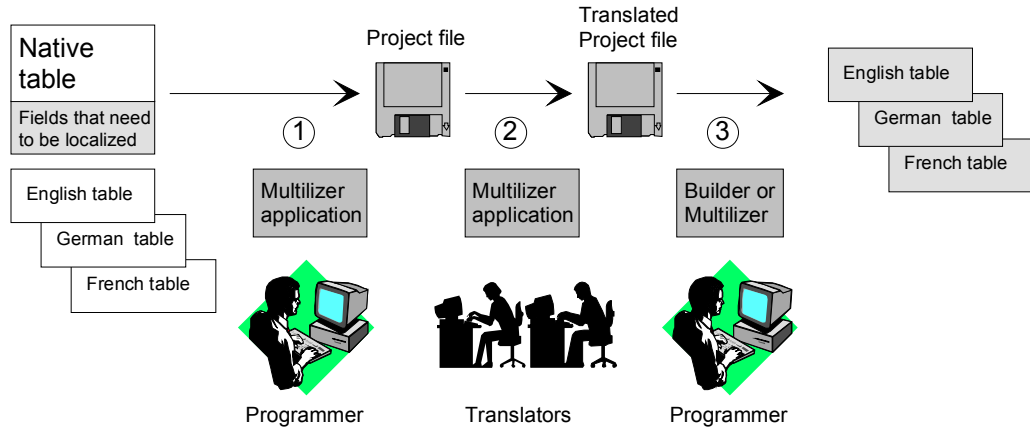


Figure 142 The tables localization process.

The table in the above example contains one field that needs to be localized. Before you can localize the fields you have to create the localizable tables (English, German and French) containing the fields that need to be localized plus an id field used as primary key referencing the native table.

The tables can be empty only the structure needs to be defined, Multilizer will add the rows to the localized tables during the localization process.

The process of localizing table fields can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the native field(s). Multilizer saves these strings into the project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to fill the localized tables.

The following figure shows the items that Multilizer uses in the field localization process.

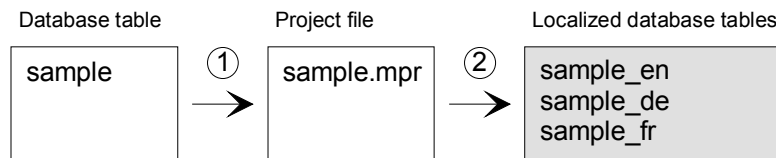


Figure 143 The items of the tables localization process.

To use the new localized data replace the native table name (e.g. sample) with the localized table name (e.g. sample_de) in the SQL statement.

Examples

The following SQL statement selects the description field from the native table.

```
SELECT Description FROM sample WHERE Id=1
```

If the active language is German you need to change the statement so that the table sample_de is used instead of sample.

```
SELECT Description FROM sample_de WHERE Id=1
```

Obviously such a simple statement is not that useful, in real situations you might have something like:

```
SELECT sample.Price, sample_de.Description FROM sample, sample_de WHERE
sample.Id = 1 AND sample_de.Id = sample.Id
```

And if your RDBMS supports the JOIN clause most likely you will use something like:

```
SELECT sample.Price, sample_de.Description FROM sample JOIN sample_de ON
sample_de.Id = sample.Id
```

The above examples might clarify as well why one of the requirements is that the primary keys in the native and localizable tables have to be the same.

See the Tables localization samples in the
<mdir>\Data\Samples\Databases\Tables directory.

Single Table Localization

Again, this architecture differs from the previous two enough to deserve a chapter on its own.

Even though tables localization is rather effective it might be problematic to adjust existing code to use the newly created localized tables. An answer to this problem comes from what we call "Single Table" localization. The name is a bit of a misnomer and might lead to the wrong conclusion that only one table can be localized. In fact it is not at all so. What we refer to is a single table containing ALL the data: native and localized.

Some planning is required to be able to insert the localized data in such a table. The first and most obvious addition is that we need to have a primary key containing at least two fields, one for the language id and one for the string id (or whatever field would be unique within a set of row of the same language).

In other words each row in the table is identified uniquely by the combination of the language id and the string id (please look at the examples at the end of the chapter for more information).

The following picture describes the single table localization process.

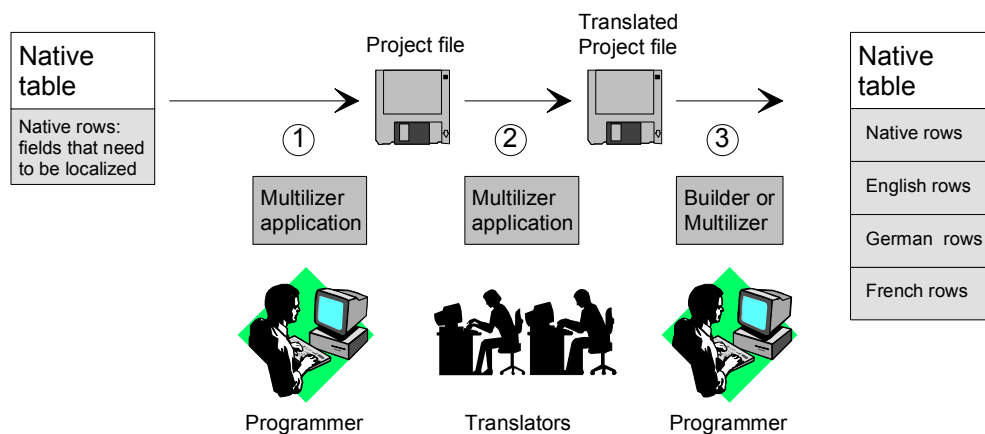


Figure 144 The single table localization process.

The table in the above example contains one field that needs to be localized. One field is required for the language data and one is for the string id.

The process of localizing fields within a single table can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the native field(s). Multilizer saves these strings into the project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to fill the table with the localized data.

The following figure shows the items that Multilizer uses in the single table localization process.

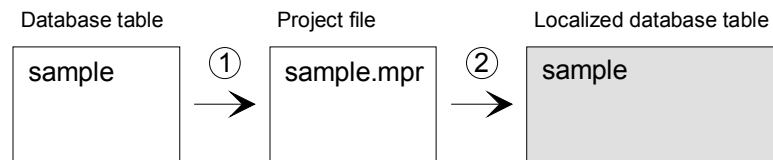


Figure 145. The items of the single table localization process.

To use the new localized data pass the required value for the language id field in the SQL statements.

Examples

The following SQL statement selects the native description field from the table. In our example English is the native language

```
SELECT Description FROM sample WHERE Id=1 AND languageId='en'
```

If the active language is German you need to change the statement in such way that the languageId will have to contain the new locale value

```
SELECT Description FROM sample WHERE Id=1 AND languageId='de'
```



Oracle users will probably be more familiar with Oracle's language/country codes. In Oracle's systems the above statements would probably be

```
SELECT Description FROM sample WHERE Id=1 AND languageId='GB'
```

and

```
SELECT Description FROM sample WHERE Id=1 AND languageId='DE'
```

See the Single Table localization samples in the
<mdir>\Data\Samples\Databases\Single directory.

Creating a New Project with Localized Fields

The <mdir>\Data\Samples\Database\Field\Product contains a database with localized fields

Id	Name	Price	Description	Description_en	Description_fi
0	Nokia 9210	800	The third generation communicator having a col		
1	Fiskars Handy	39	This ax has fiber class arm that is virtual unbrea		
2	Polar M52	129	A heart rate meter that has a build in fitness tes		

Figure 146 The table containing the native and localized fields

As you can see two fields, Description_en and Description_fi, have been added to our original Product table. These fields are of the same type and size of the native field Description and currently contain no data.

Double click the Multilizer icon from the Multilizer program group to start Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a Database** button. The Database sheet appears. This sheet is used to select the database type, the connection string and the fields and/or tables to be scanned and localized.

Set database type to *Any ADO/ODBC compatible*. Press the ... button. The Data Link Properties dialog box appears. Select Microsoft Jet 4.0 OLE DB Provider. Press the **Next** button. Press the ... button. Choose the

<mdir>\Data\Samples\Database\Field\sample.mdb database file. Press **OK**. Press the **Connect** button. Multilizer will detect those fields that can be localized and will try to detect the correct languages according to the names suffices. The Database sheet should look like this.

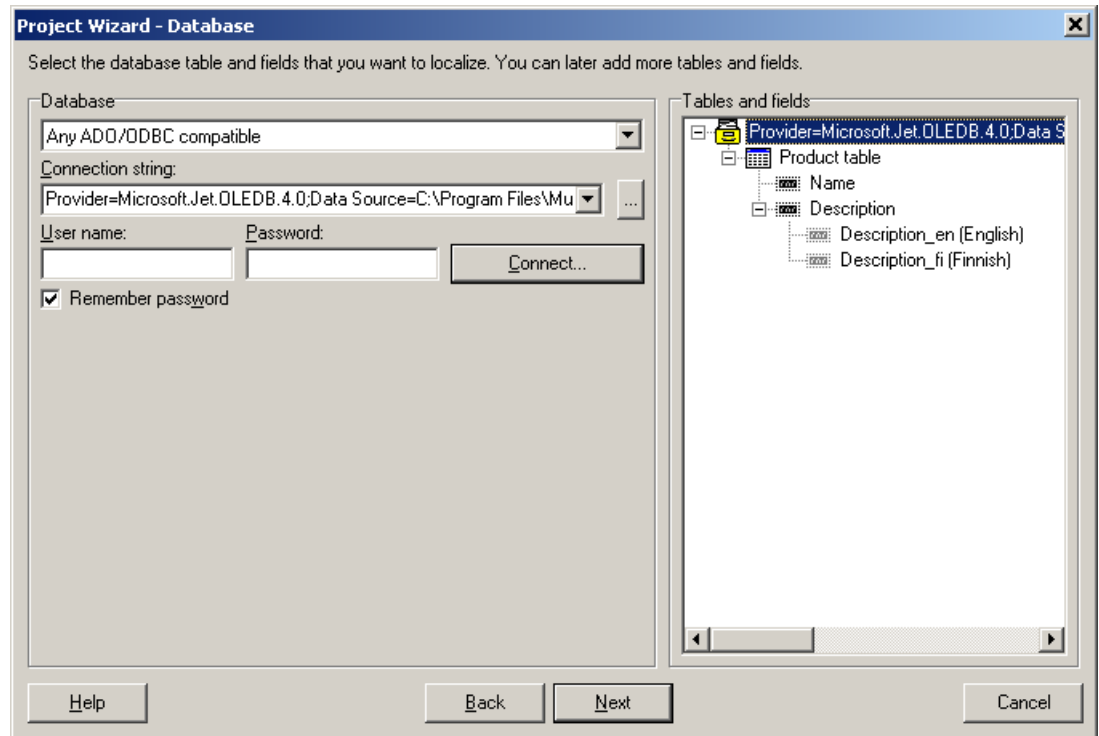


Figure 147. The Database sheet is used to specify the database, tables and files.

Our sample database contains a table with localizable fields. Multilizer will nest the localizable fields within the native fields in the tables that contain them.



The auto-detection will not work if the fields and/or table names follow a different naming convention than the one suggested by Multilizer (please refer to chapter 3). In this case you can manually drag fields and/or tables within other fields or tables though the language might still remain undetected. To set the correct language for the localized field (or table), select the item you are interested in and right click on it. A sub menu appears with the current language selection from which you can choose.

Press the **Next** button, the Languages sheet appears. It lets you add string languages to the project. From Available languages select English and drag the item to the Selected languages list box or press the >> button. This adds English to the project.

Add some other European language. If you add Finnish the result should look like this:

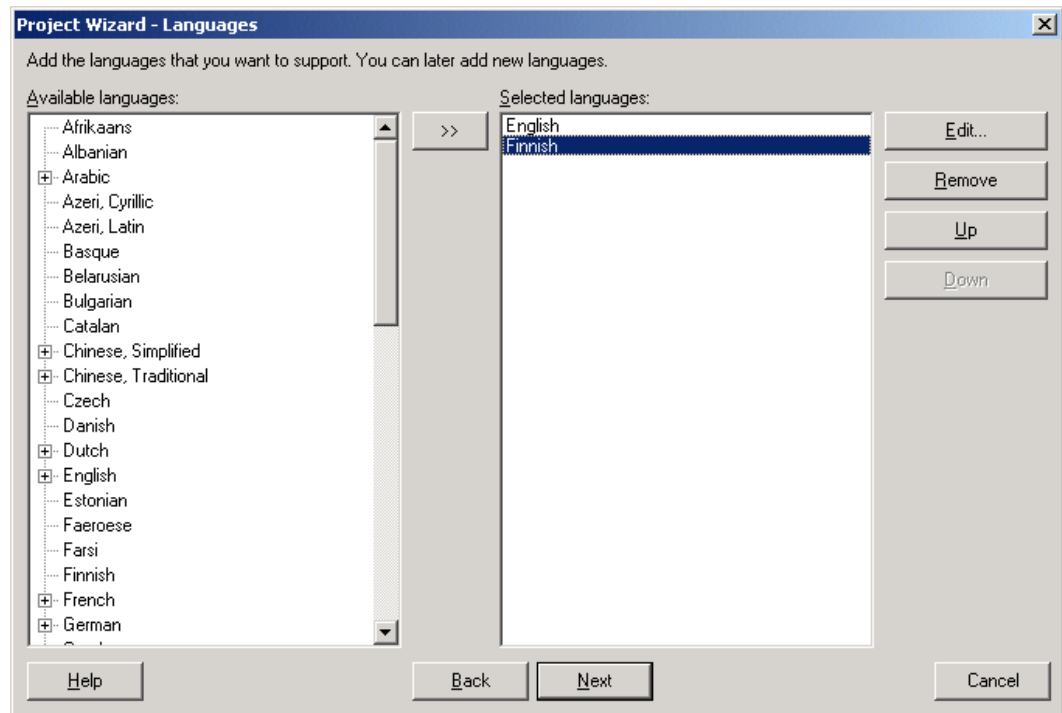


Figure 148. English and Finnish added to project.

When you are satisfied with the fields and language selection click **Next** and the targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Project Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Finish sheet appears. Press the **Finish** button to finish the Project Wizard. The following project grid appears.

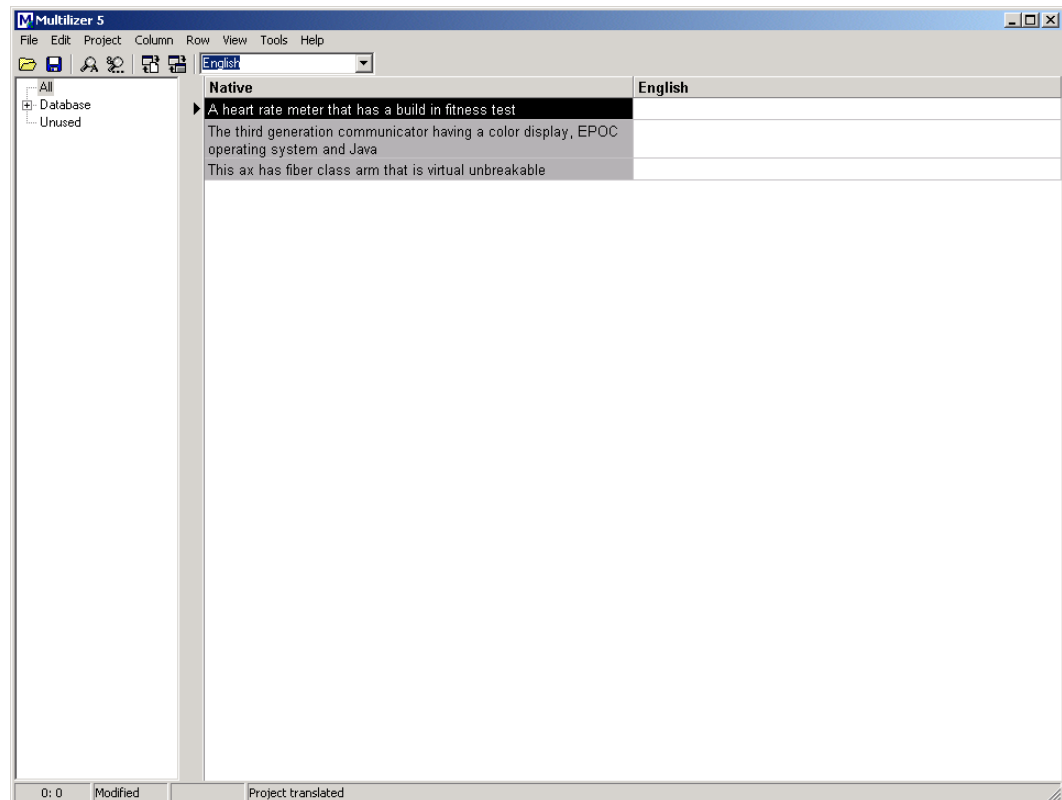


Figure 149 The project grid.

Save the project by choosing **File | Save**.

Creating a New Project with Localized Tables

The `<mdir>\Data\Samples\Database\Tables\Product` contains the database with localized tables.

The figure shows three screenshots of database tables. The first, 'Product : Table', contains three records with columns Id, Name, Price, and Description. The second, 'ProductEn : Table', is empty with columns Id and Description. The third, 'ProductFi : Table', is also empty with columns Id and Description.

Product : Table	Id	Name	Price	Description
	0	Nokia 9210	800	The third generation communicator having a color display, EPOC operating system and Java
	1	Fiskars Handy	39	This ax has fiber class arm that is virtual unbreakable
	2	Polar M52	129	A heart rate meter that has a build in fitness test
*	0		0	

ProductEn : Table	Id	Description
	0	

ProductFi : Table	Id	Description
	0	

Figure 150 The native and localized tables

In this case two tables, ProductEn and ProductFi, have been added to our original database. These tables contain a primary key (Id) and a Description field of the same type and size as the Description field in the Product table (please refer to Chapter 3 for more information on naming conventions). Obviously they are empty, ready to be localized.

Creating a project with localized tables is very similar to creating a project with localized fields. The only difference is the Database sheet after connection.

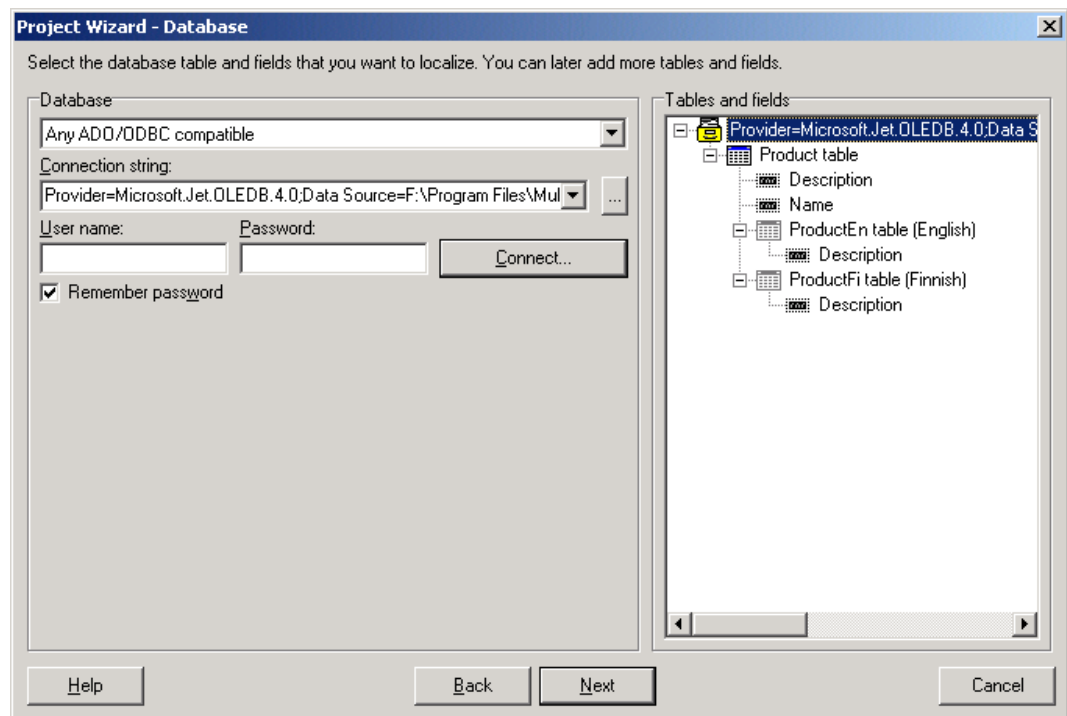


Figure 151 Tables and fields in the database.

Our sample database contains a table with localizable fields. Multilizer will nest the localizable fields within the native fields in the tables that contain them.

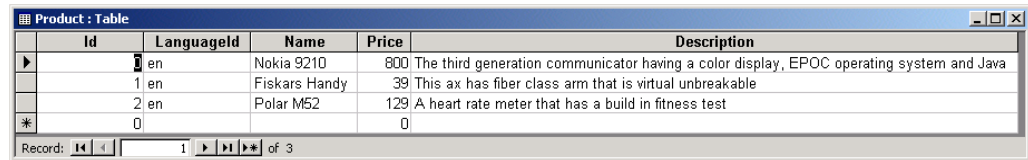


The auto-detection will not work if the fields and/or table names follow a different naming convention than the one suggested by Multilizer (please refer to chapter 3). In this case you can manually drag fields and/or tables within other fields or tables though the language might still remain undetected. To set the correct language for the localized field

(or table), select the item you are interested in and right click on it. A sub menu appears with the current language selection from which you can choose.

Creating a New Project with a Single Table

The <mdir>\Data\Samples\Database\SingleTable\Product contains the database with a native table.



Id	LanguageId	Name	Price	Description
1	en	Nokia 9210	800	The third generation communicator having a color display, EPOC operating system and Java
2	en	Fiskars Handy	39	This ax has fiber class arm that is virtual unbreakable
0	en	Polar M52	129	A heart rate meter that has a build in fitness test

Figure 152 A single table containing native rows

In this case one extra field, LanguageId, has been added to the table. The table primary key contains now two fields: Id and LanguageId.

Creating a project with localized tables is very similar to creating a project with localized fields. The only difference is the Database sheet after connection.

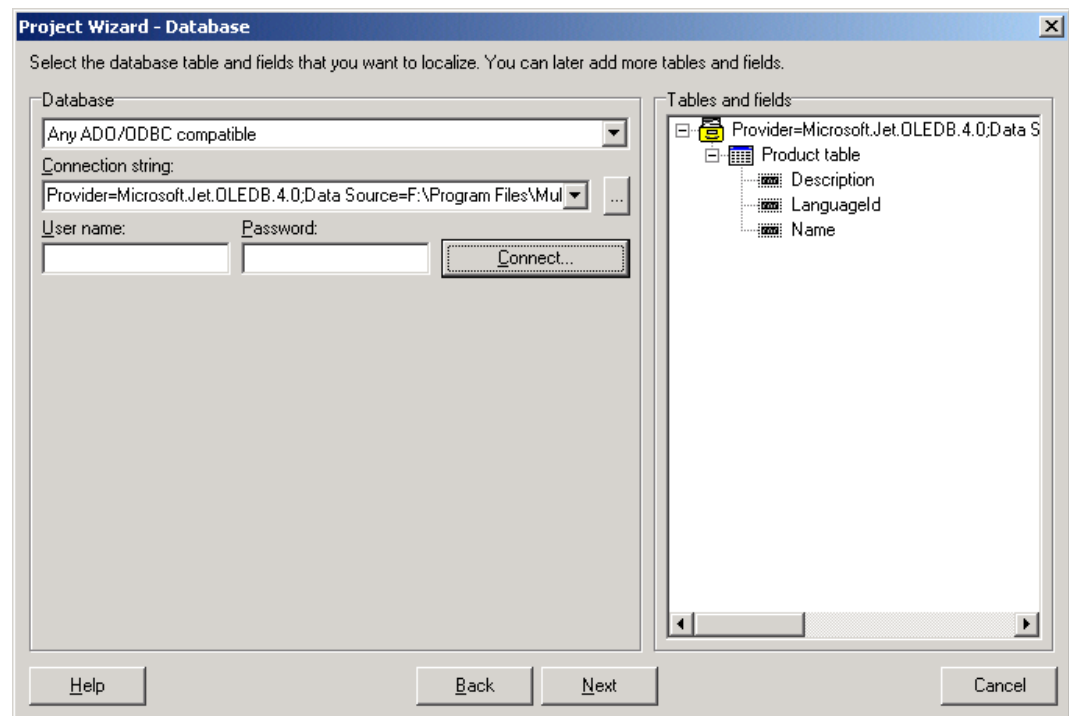


Figure 153 Tables and fields in the database

We need to tag the field Description as localizable and LanguageId as our language id field.

To tag the field Description as localizable, select it in the tree and then right-click with your mouse, a menu will popup with two items, *Localizable Field* and *Language Id Field*.

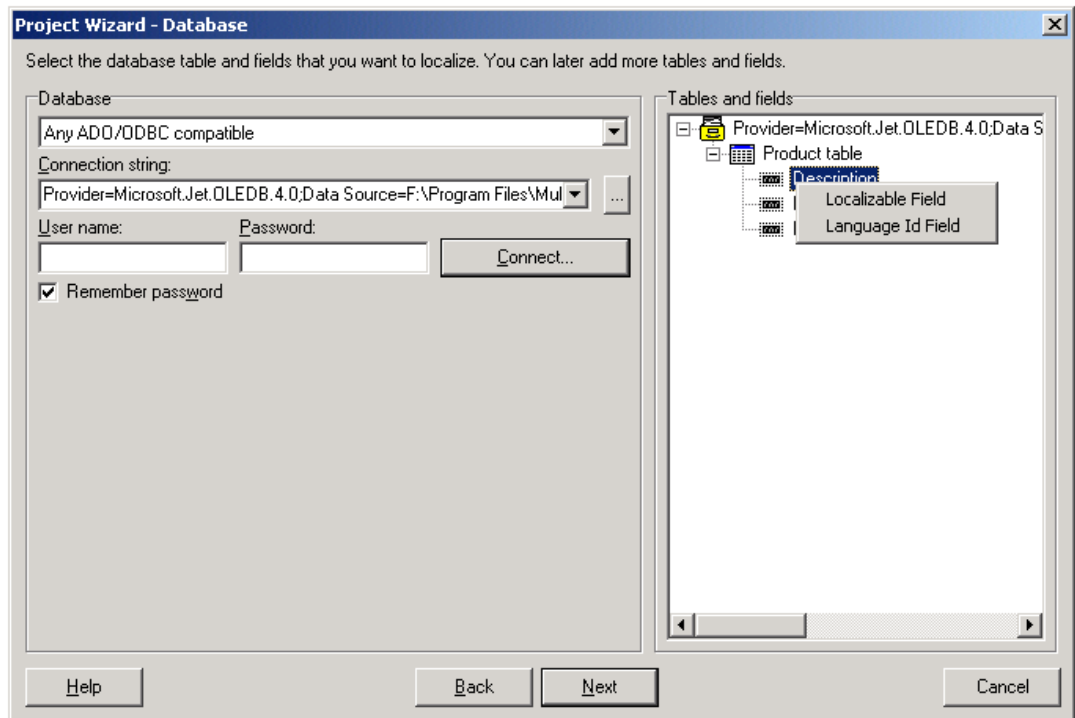


Figure 154 The popup menu for selecting fields

Now click *Localizable Field*. Now the icon of the Description field has changed into a green check mark.

To select the field LanguageId as language id, select it in the tree and then right-click with your mouse, the same menu will popup, this time click the *Language Id Field*. Now the icon of the LanguageId field has changed into a hand pointer.

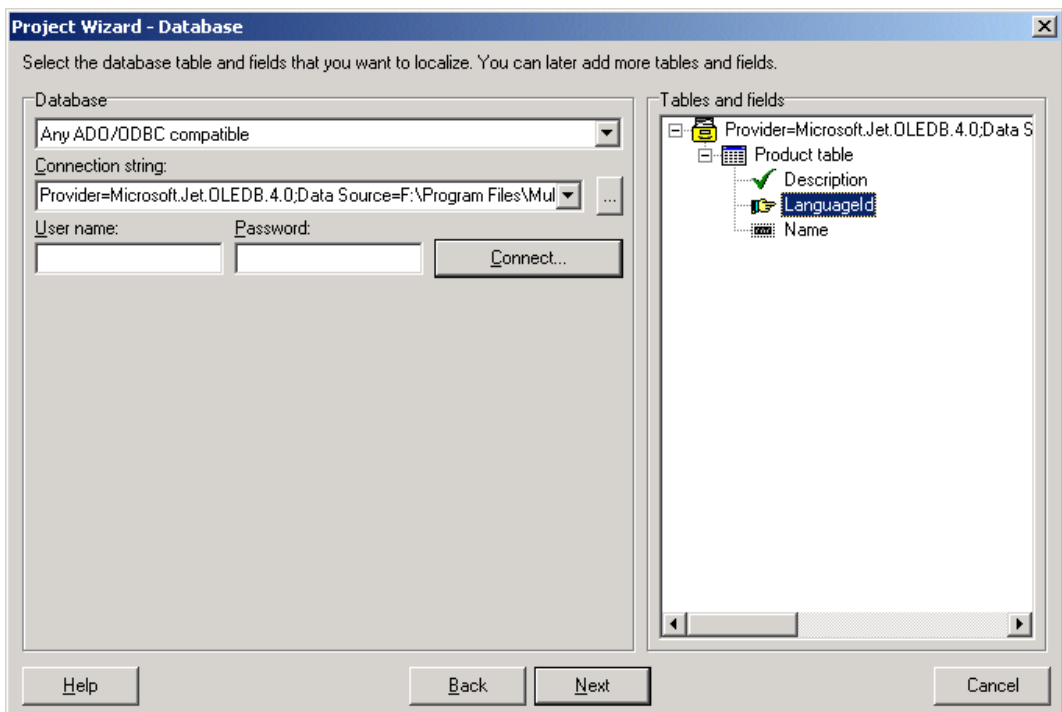


Figure 155 Selected fields and language id for localization

Now Multilizer will know which field to scan (Description) and which field in the table contains the language id field (LanguageId).

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Create the localized database files by choosing **Project | Create Localized Files**. This will write the localized rows to the tables according to the type of project (field and/or table localization).

18

Oracle® Forms

In this tutorial we are going to localize a simple Oracle Forms application. The application is not connected to any database because it is for demonstrative purposes only but this will not affect at all the localization process.

Main differences between Oracle Translation Manager/Builder and Multilizer

Before moving to the actual tutorials let's read about issues related to the differences between the tools supplied with Oracle and Multilizer. This is important in order to fully understand what Multilizer does to your forms files or how it can reuse the translations you have previously created with OTM. Furthermore, we will address also issues like the translation of object libraries and PL/SQL source code.

Once the differences are clear then the migration to Multilizer will be more transparent and easy.

Choosing the right native language

If your native form has been previously localized using OTM you should make sure that your NLS_LANG environmental variable is set to the correct value before scanning it with Multilizer. Multilizer relies on Oracle Forms API for certain functionalities and the NLS_LANG settings will determine the language that Multilizer will scan as native. In other words if you have an English application that has been localized to Italian and your NLS_LANG is set to Italian then Multilizer will present you with the Italian strings as native even though you would consider English as the native language of the application. So, in order to make sure that the correct native is displayed please check your NLS_LANG settings.

How Multilizer localizes fmb files

Unlike OTM, Multilizer creates a copy of each fmb file in your project into the localized directory. This means that after having localized a form file, the settings of the NLS_LANG variable will not affect anymore the language displayed in the localized forms. This might seem like a waste of space but it has advantages especially if your application uses object libraries. When you localize a form that contains object libraries Multilizer will localize every item in your form and store it in a copy into the localized subdirectory and finally compile the file for immediate use. If you change, at a later time, the object library of your native application, all you need to do is a rescan of the native forms and, after saving the project, all your changes will be stored in the new localized forms without any loss of previous translations.

This removes also the necessity to localize the object libraries separately.



After localizing a form with Multilizer you should not reopen it using the Form Builder because you might end up losing your translations especially if you are using olb files.

Multilizer project versus Oracle Translation Manager/Builder database

Multilizer stores the translations in projects and in order to guarantee reusability the same translations are also stored in its translation memory. When you import a previously

created OTM database into Multilizer's translation memory, all the strings that are imported are available to be reused in every new project you create with Multilizer. Duplicated native-translation pairs are not stored into the translation memory because there is no necessity to do so. On the other hand every new translation you create using Multilizer will be reusable in other projects too. Later on in this chapter we will see how to import your existing data from an OTM database into Multilizer's translation memory.

PL/SQL code localization

Multilizer can localize your PL/SQL code. Currently functions that accept one string parameter can be localized. The advantages are evident if you need to localize an application that contains, for instance, several calls to a function to display some message, for instance the function Message.

By instructing Multilizer to scan the Message function you will be able to localize the parameter passed to it exactly in the same way in which you localize any of the visible components in the form.

We will see later in the tutorial how to set Multilizer to scan PL/SQL code.

Oracle Forms Localization

Oracles Forms applications contain the localizable data in the application file (e.g. .fmb). Multilizer creates the localized application files from the original file. The following picture describes the binary localization process.

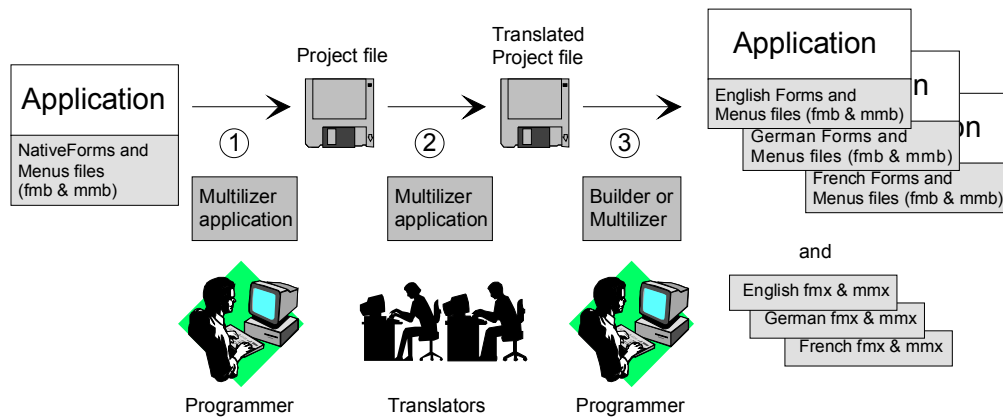


Figure 156 The Oracle Forms localization process.

The programmer uses Multilizer to extract strings from the original application file (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (1). The programmer uses Multilizer or Builder to create the localized application files (2). As a result there will be one application binary file and one compiled file for each localized language.

Multilizer creates subdirectories under original file folder containing the localized file(s). I.e. there might be subfolders called `..en\<localized file>` and `..fi\<localized file>`.

The following example figure shows the files that Multilizer uses in the Oracle binary localization process.

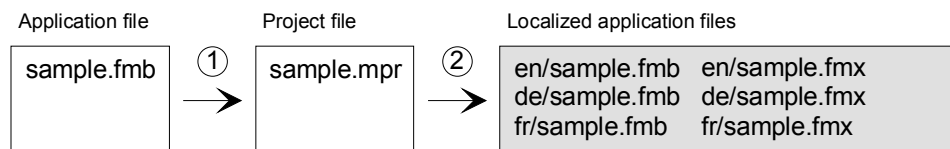


Figure 157 The items of the Oracle Forms localization process.

When deploying the application it is enough to deploy the localized compiled files (e.g. `de\sample.fmx`).

Creating a New Project with Oracle Forms

The `<mdir>\Oracle\Samples\Tutorial` contains a form file called `sample.fmb`.

The form looks like:

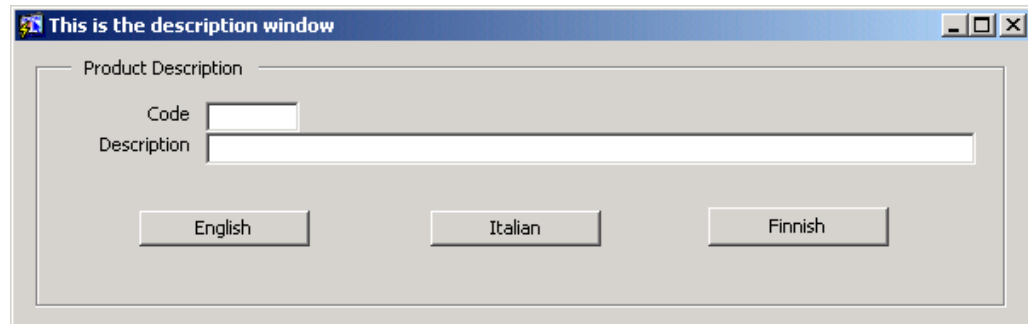


Figure 158 Sample form application with the English user interface

Double click the Multilizer icon from the Multilizer program group to start Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button. The File Target sheet appears.

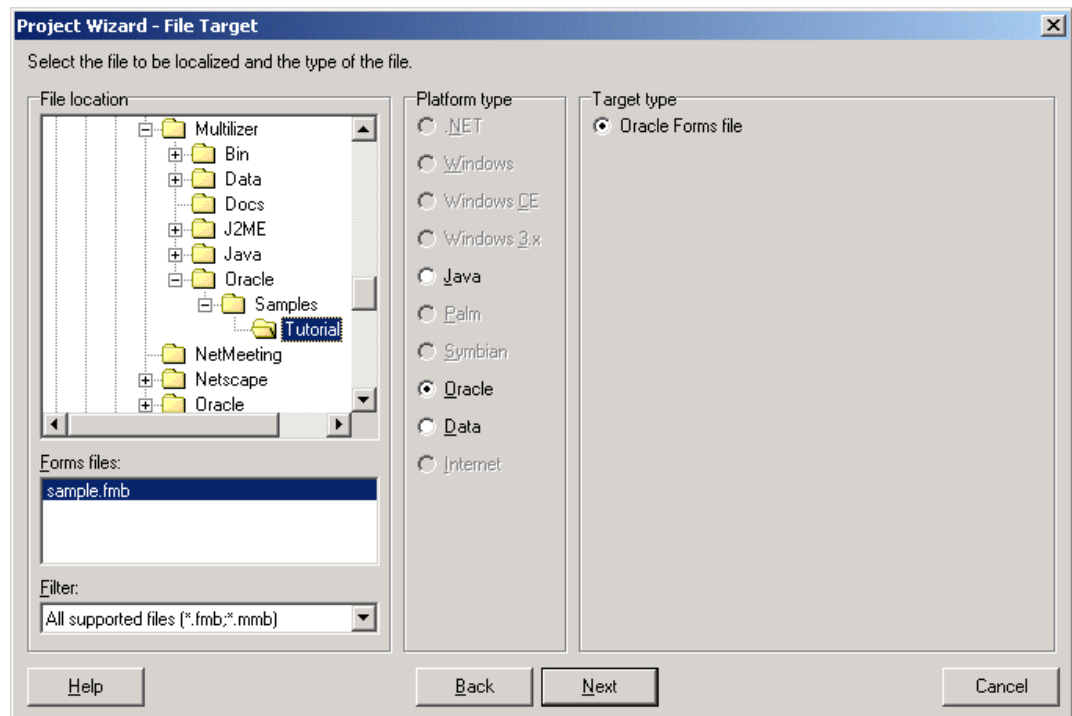


Figure 159 The Source sheet is used to enter the source directory.

In this sheet you will specify the directory where your application is located. Choose the `<mdir>\Oracle\Samples\Tutorial` of your Multilizer setup. Project Wizard detects the platform and target types. The Platform type should be *Oracle* and the Target type should be *Oracle Form and Menu file*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Accept the default values by pressing the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

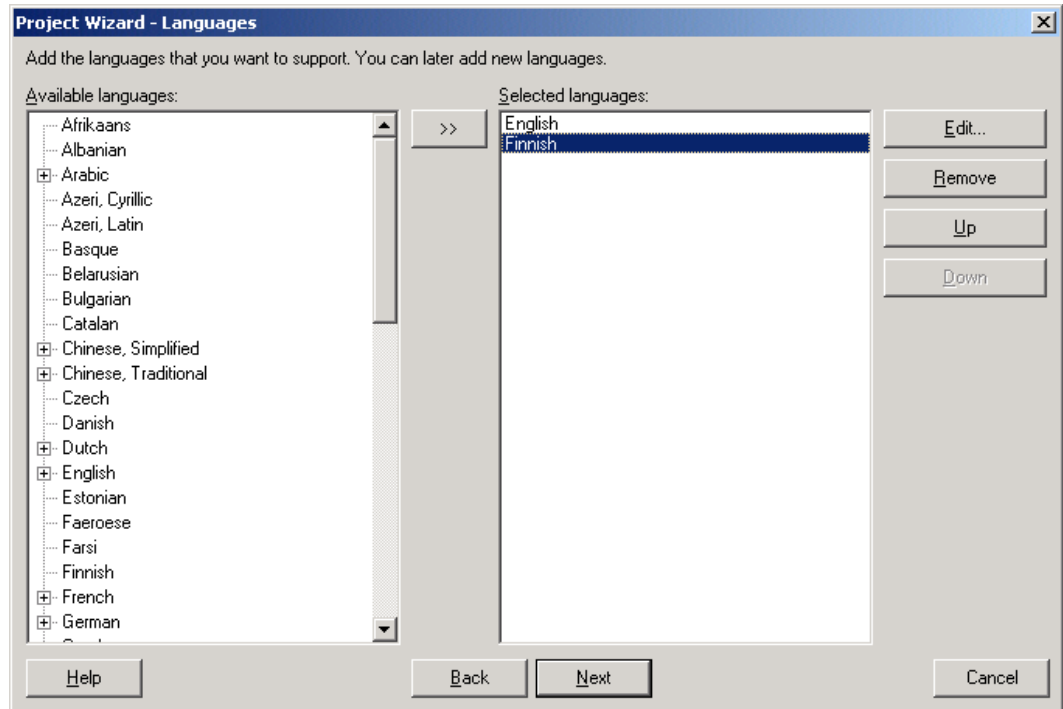


Figure 160 English and Finnish added to project.

Press the **Next** button. The Targets sheet appears. This sheet is used to add new targets to your project or to edit those already contained in it. Usually for most projects types the default settings are good enough, in our case we would like to scan the PL/SQL code contained in our application so we will need to edit the targets in order to instruct Multilizer on which functions to scan.

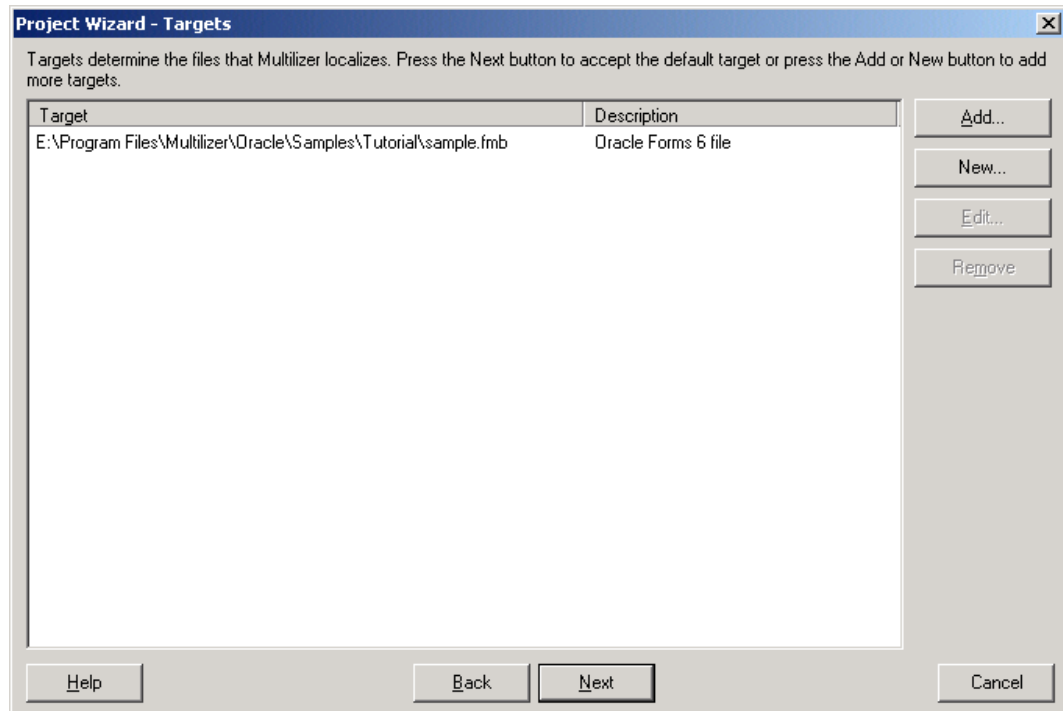


Figure 161 The Targets sheet

Select the first, and only, target currently in our project and click the **Edit** button. The following dialog appears.

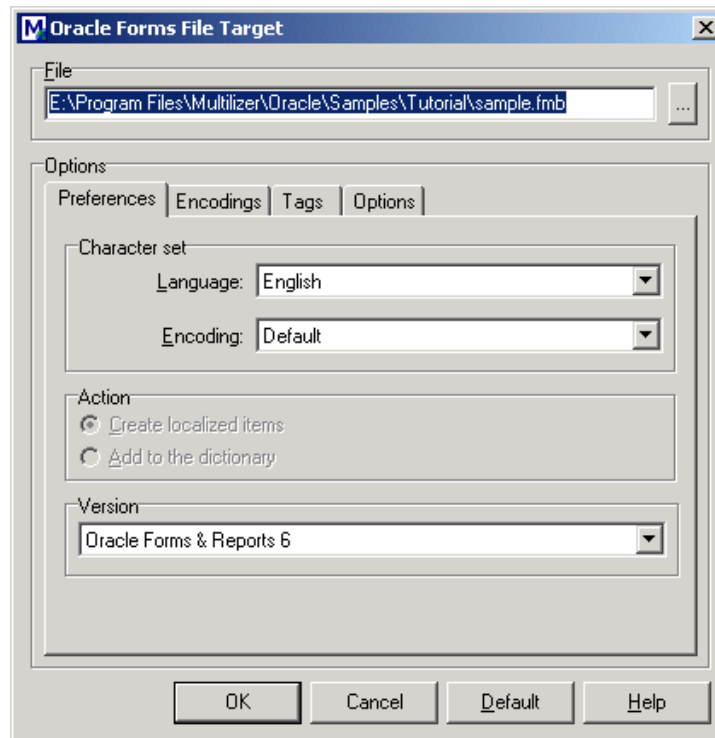


Figure 162 The Targets dialog

Now select the Tags tab.

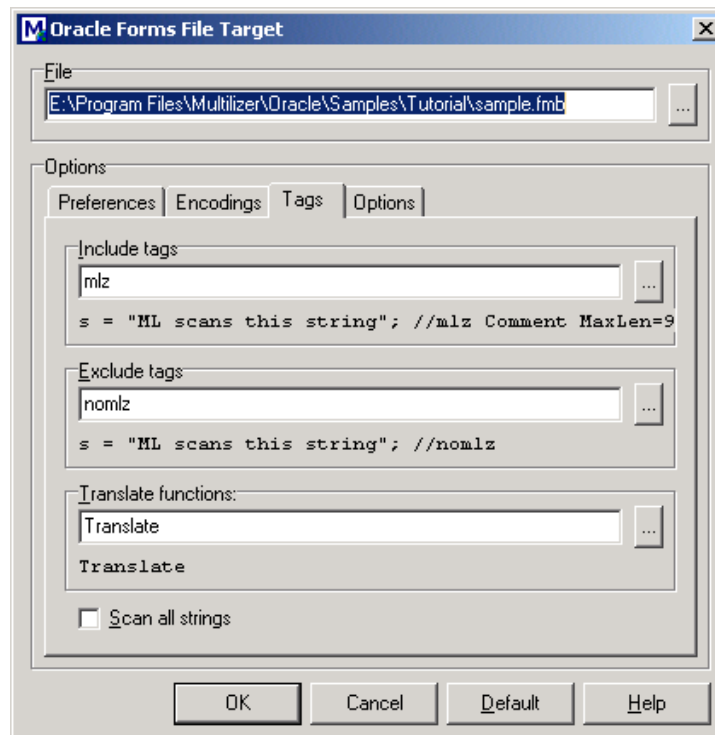


Figure 163 The Tags tab in the Targets dialog with default settings

The default settings are visible and we need to change some of them. In our PL/SQL code we have inserted some calls to the function message in response to the WHEN-BUTTON-PRESSED trigger of all three buttons in the form:

```
message('Language changed');
```

Obviously the function call we would like to scan is the function “message” so write the function name in the Translate functions field.

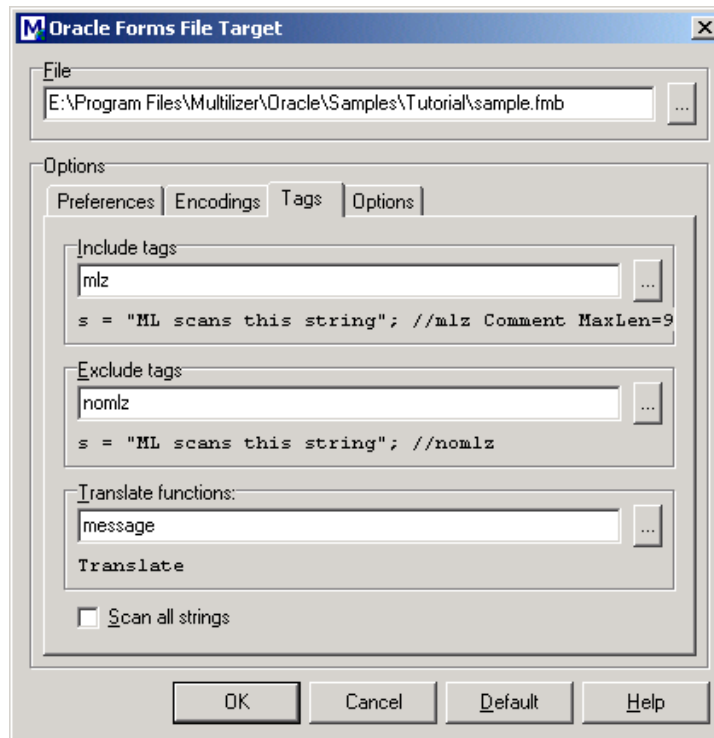


Figure 164 The Tags tab in the Targets dialog with the modified settings



If you have more than one function call to scan, you can write all the names separated by a semicolon in this way:

```
message;newMessage;anotherCall
```

The function names are NOT case sensitive.

Now click OK to return to the Targets sheet.

Because we do not want to add any more files, and we are ready with the target editing, you can press the **Next** button. The Finish sheet appears. Press the **Finish** button to finish the Project Wizard. The following project grid appears.

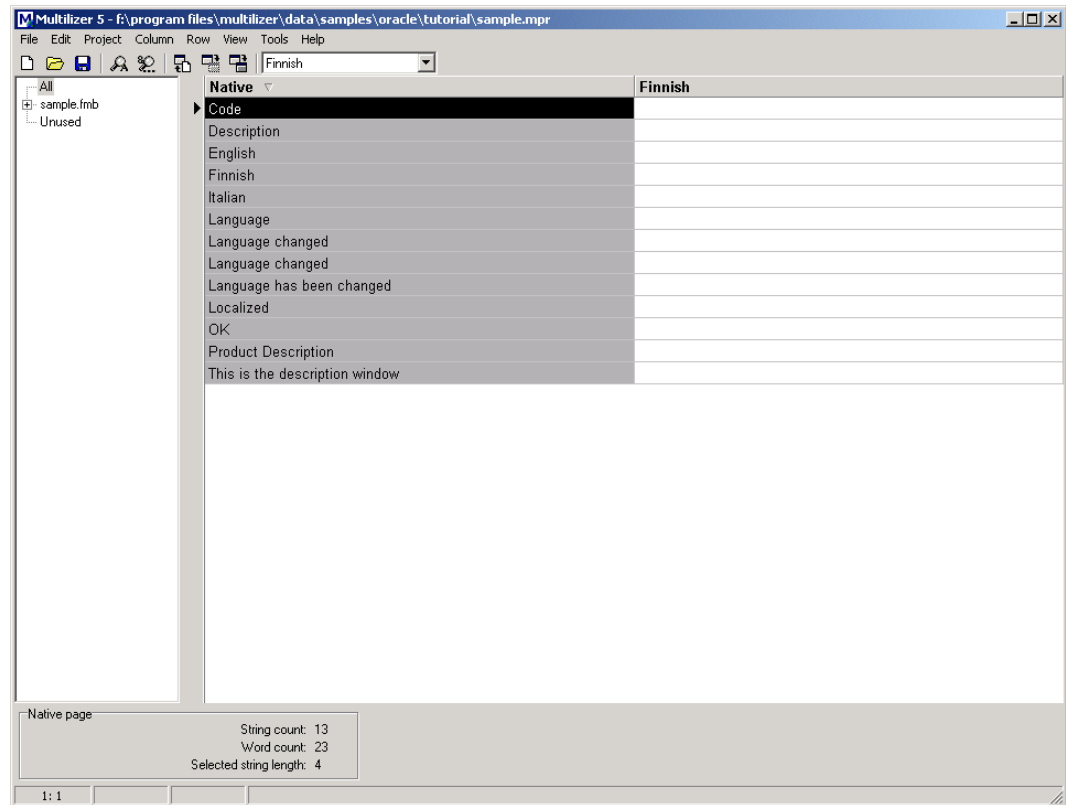


Figure 165 The project grid.

Save the project by choosing **File | Save**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project, save it by choosing **File | Save**.

Create the localized database files by choosing **Project | Create Localized Files**. This will write the localized rows to the tables according to the type of project (field and/or table localization).

Importing Translations From OTM/OTB into Multilizer's Translation Memory

One of the biggest advantages of using Multilizer is the reusability of translations across multiple projects. This reusability is achieved through Multilizer's translation memory and of course it becomes essential for the user to be able to import existing translations into the translation memory.

Importing of OTM/OTB databases into Multilizer is quite straightforward; in this next example we'll show how the import works.

Open the Translation Memory dialog from **Tools | Translation Memory** menu and choose the import sheet. In the import sheet click the **Add** button to open the Import Wizard.

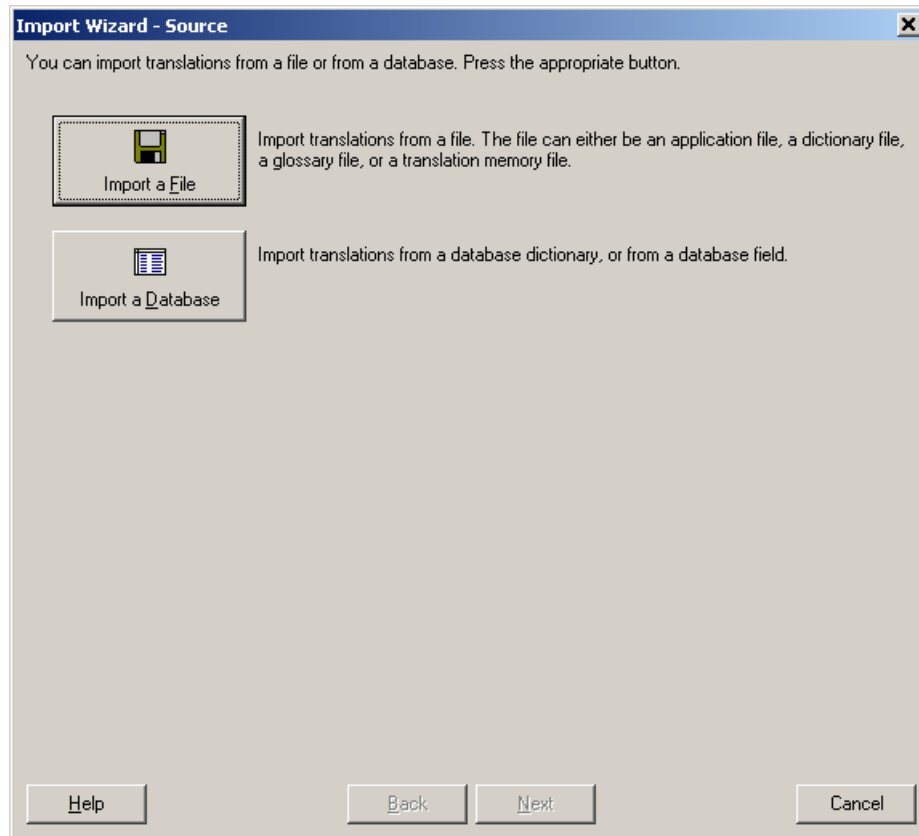


Figure 166 *The translation memory import wizard.*

Now you can choose the import source that in our case is Database, so click the **Import a Database** button.

In the Database drop down list select Oracle, and enter the login information for your OTM/OTB database in the login dialog.

Import Wizard - Database

Select the database that you want to import.

Database: Oracle

Address: oracleserver

Database: translations

Parameters:

User name: user Password: xxxxxxx

Remember password

Connect...

Help Back Next Cancel

Figure 167 The login dialog

Now click the **Connect** button to login into your Oracle's Translation Manager database.

Select the **Oracle Translation Builder** button and select the native language, in our example English (United States).

Import Wizard - Database

Select the database that your want to import.

Database: Oracle

Address: oracleserver

Database: translations

Parameters:

User name: user Password: [masked]

Remember password

Dictionary or Glossary Single Field **Oracle Translation Builder**

Native language: English (United States)

Help Back Next Cancel

Figure 168 The login dialog after a successful log

After having selected the native, click the **Next** button. Now you can select which languages you'd like to import, in our case we will import Italian, Korean and Finnish.

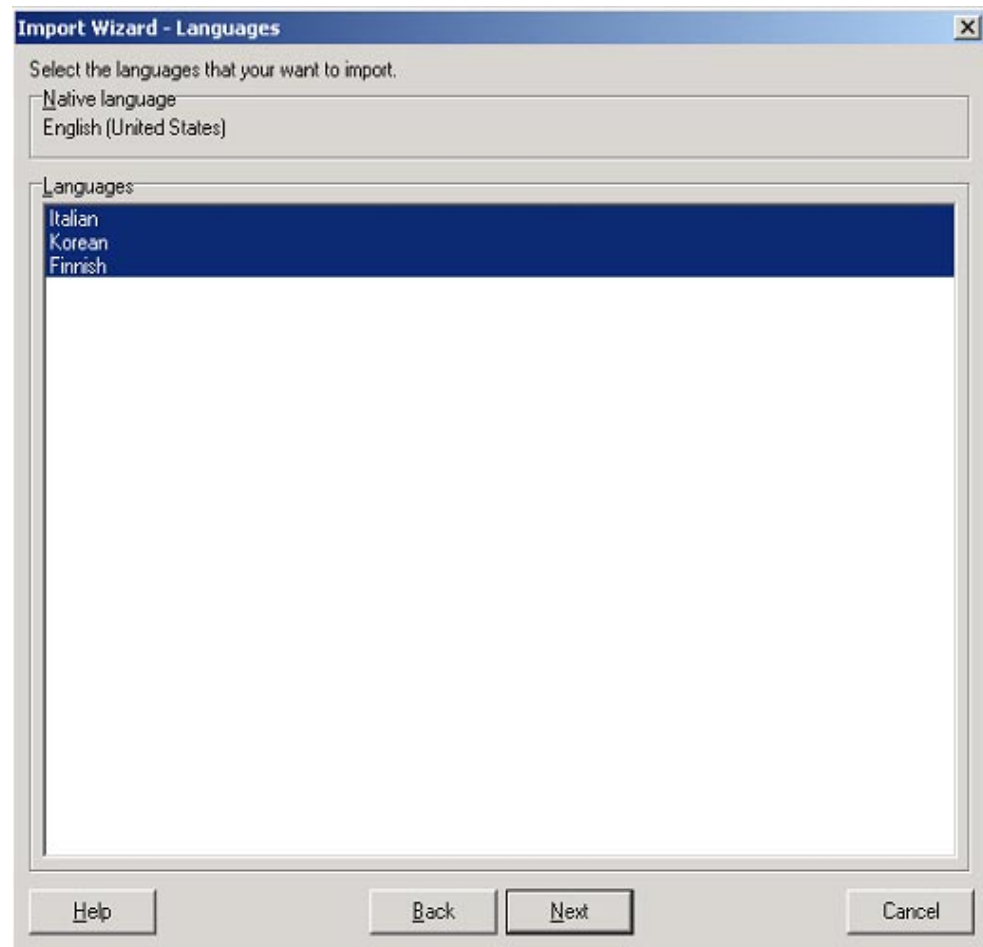


Figure 169 The languages selection in the Import Wizard

Now click **Next** and the import will start.



NOTE!

Only the strings that have been flagged as “Done” in OTM will be imported. Multilizer will ignore the strings flagged as “Start” and “Revision Needed”.

Now the translations you had in OTM have been imported into Multilizer’s translation memory and are ready to be reused in your Multilizer projects.

19

StreamServe

In this tutorial we are going to localize StreamServe's SLS files. They contain the localized items of reports. Our sample SLS file is used in a product catalog application.

StreamServe Localization

SLS file contain string data. Multilizer creates the localized SLS files from the original SLS file. The following picture describes the SLS localization process.

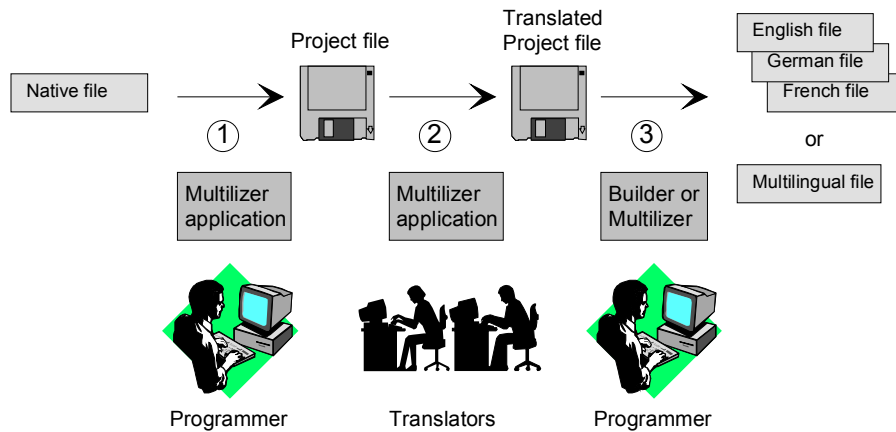


Figure 170 SLS localization process

The process of localizing SLS files can be divided into 3 basic steps:

1. The programmer uses Multilizer to extract strings from the SLS file. Multilizer saves these strings into the project file (.mpr).
2. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file.
3. The programmer uses Multilizer or Builder to create the localized SLS files.



The above will result in one SLS file for every language. The structure of the localized file is identical to the original one. The only difference is that the selected string data has been translated to the target language. Multilizer can create one SLS file that contains strings in several languages.

The following figure shows the files that Multilizer uses in the SLS localization process.

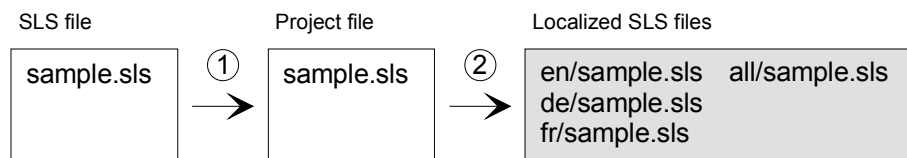


Figure 171 XML localization process files

To use the new localized resource (e.g. de/sample.sls), deploy or use it instead of the original SLS file (sample.sls).

English File

The `<mdir>\Data\Samples\StreamServe\Tutorial` contains the English SLS file. It is a simple file that contains strings for a product catalog report. The strings that need to be localized are marked with bold typeface.

```
product_name      en      Name
product_description en      Description
product_company  en      Company
sport_skiing     en      Skiing
sport_hockey     en      Ice Hockey
```

The SLS file format is simple. Each line contains one string. The line starts with the string id, following with the language code and finally the string value. These three parts are separated by white spaces.

Creating a New Project

Double-click the Multilizer icon from the Multilizer program group to launch Multilizer.

Choose **File | New** from the main menu to start the Project Wizard. The Target Type sheet appears. Press the **Localize a File** button.

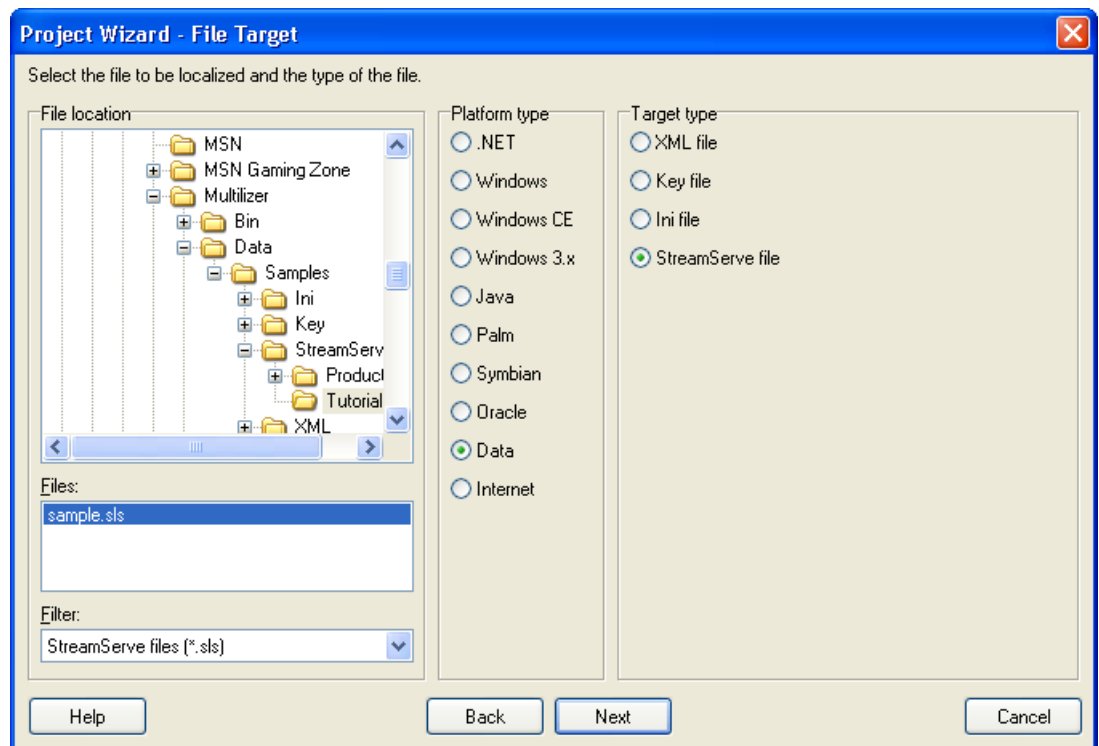


Figure 172 The File Target sheet is used to specify the XML file to be localized.

This dialog specifies the directory where your application is located. Choose the `<multilizer>\Data\Samples\StreamServe\Tutorial` subfolder of your Multilizer setup. Project Wizard detects the platform and project types. The Platform type should be *Data* and the Target type should be *StreamServe file*. If they are wrong, check the right types.

Press the **Next** button. The Information sheet appears. This sheet specifies the project name and other project related information. Press the **Next** button. The Languages sheet appears. This sheet lets you select the initial languages you would like to localize in the project. You only need to select one or a few initial languages, as you can always add more languages later.

From the Available languages list select English and drag the item to the Selected languages list box, or press the **>>** button. This adds English to the project.

Add some other language to the project as well. If you are new to Multilizer, it might be easiest to add Finnish, so that you can follow the examples shown in this tutorial directly. If you add Finnish the dialog box should look like this:

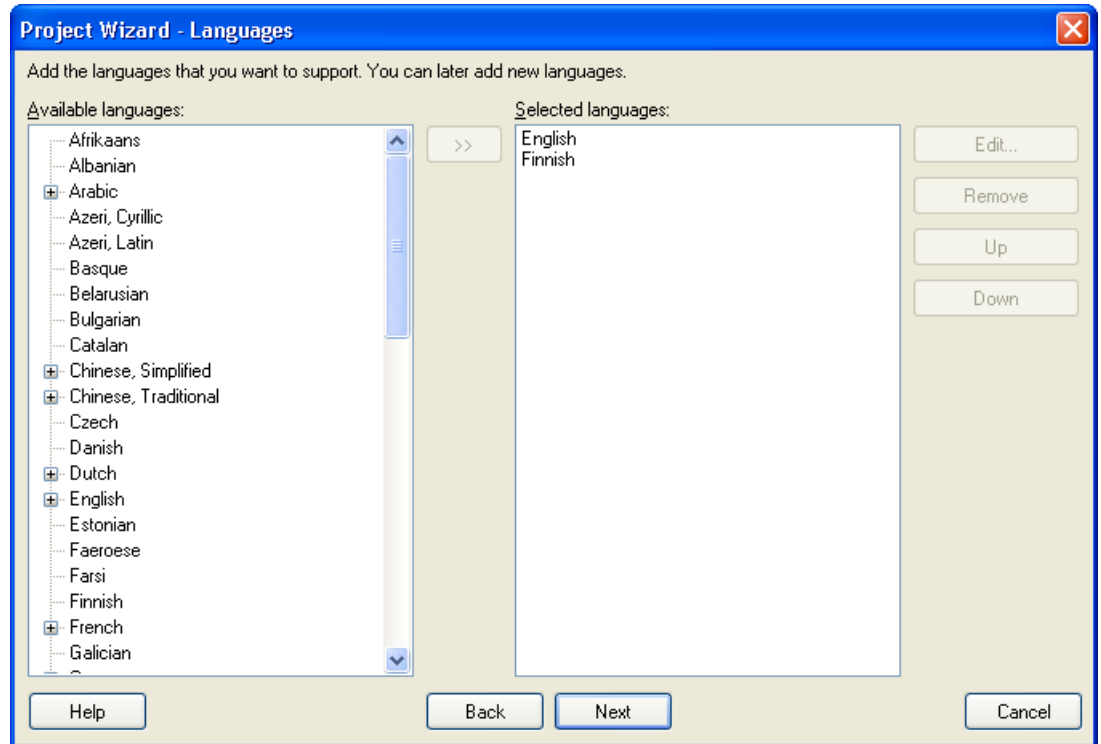


Figure 173 English and Finnish added to a project

Press the **Next** button. The StreamServe sheet appears:

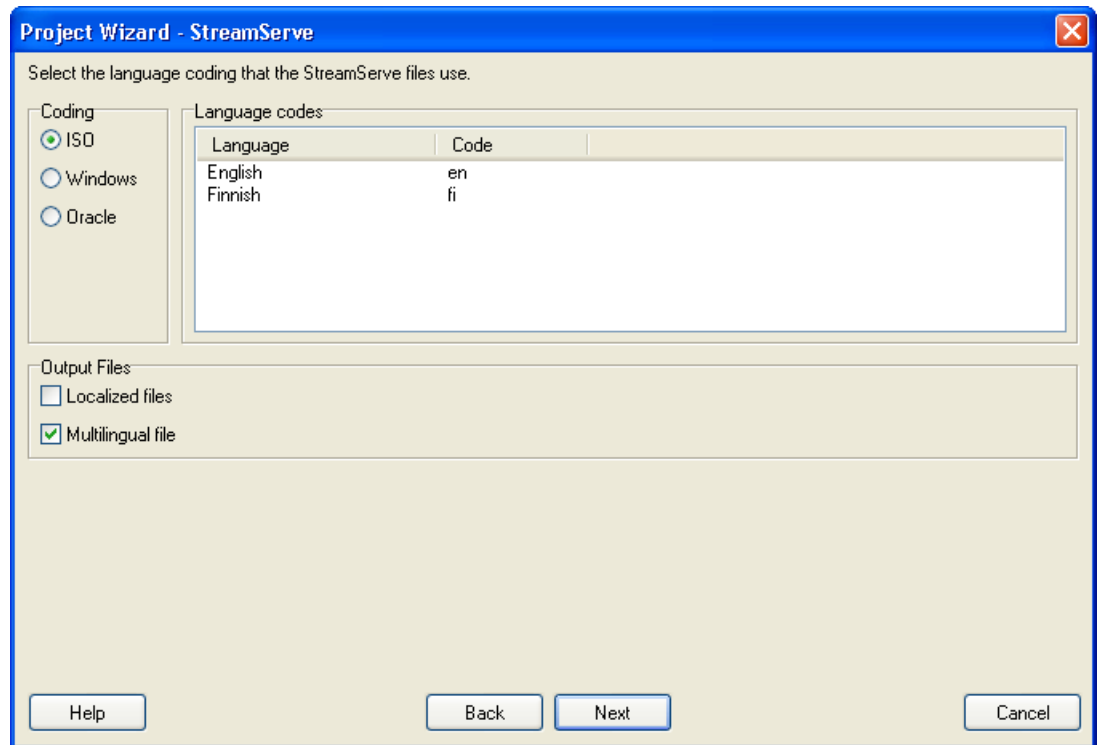


Figure 174 The StreamServe sheet is used to set the language codes.

This sheet specifies the language codes used in SLS files. SLS files can use any language code. It is up to the application. You can select between three language coding. They are ISO, Windows and Oracle. If they do not match you needs, double click the language row in the *Language codes* list and enter the language code that you want to

use for that language. We recommend to use ISO language codes because they are standard way to represent language id. Most operating systems and application use them.

By default Multilizer creates only the multilingual SLS file. If you want to create a separate SLS files for each language, check Localized files in the *Output Files* group box.

Press the **Next** button. The Targets sheet appears. This sheet lets you add more files to be localized. We do not want to add any more files. Press the **Next** button. The Ready to create project sheet appears. Now you have almost finished creating the project.

Press the **Finish** button to end Project Wizard. Multilizer then scans the application, and extracts all resource strings from it, and builds a project file of them. It only takes a few seconds for a project as simple as the Dcalc, but if you had a larger project you can monitor the progress from the status bar.

When the scanning is done, the following project grid appears:

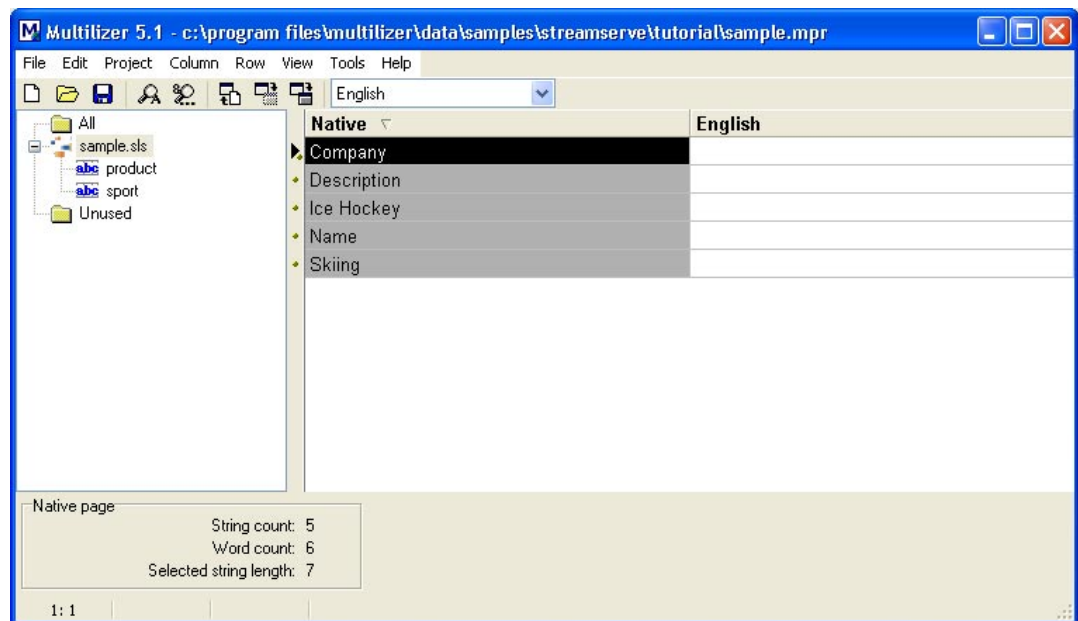


Figure 175 The project grid

On the left side there is a tree view that contains the targets and the items they contains. Our project contains one target, sample.sls, and it contains two groups. To show only the string belonging to a specific group select the group in the tree. On the right side there is the editing grid. It contains the native column and the English column. To show another language select the language from the combo box above the grid.

Save the project before moving on by choosing **File | Save As**.

Translating a Project

To translate the project read the *Translating a Project* chapter in the end of this part. When you have translated the project save it by choosing **File | Save**.

Create the multilingual SLS file by choosing **Project | Create Localized Files**. This creates the multilingual SLS file in the language specific sub directories (e.g. 'all\sample.sls'). The file looks like this.

```
product_name      en      Name
product_name      fi      Nimi
product_description en      Description
product_description fi      Selitys
product_company   en      Company
product_company   fi      Yrityks
sport_skiing      en      Skiing
sport_skiing      fi      Hiihto
sport_hockey      en      Ice Hockey
```

```
sport_hockey      fi      Jääkiekko
```

The structure is identical to the native (English) one but the file contains also the Finnish strings.

Tagging

In many times the space available for a string is limited. For example the layout of the report is designed in such way that there is only certain space for the string. You could use Multilizer's Row | Max Characters or Row | Max Pixels menu to set the maximum length of a string. However this can be automated by adding the maximum length data into the string id. This makes it possible for the developer to set the maximum lengths for the string.

Multilizer uses tags in the comment to specify a comment and to set the maximum length. The format is:

```
id language      string //mlz [comment] [MaxChars=C] [MaxPixels=P]
```

where

mlz It the multilizer tag.

comment is an optional comment to be added to the Multilizer project.

C is a positive integer number specifying the maximum length of the translation in characters.

P is a positive integer number specifying the maximum length of the translation in pixels.

In the following example the English string ("This is a string") does not belong to any group, has SampleLabel id, and has no size limitation.

```
SampleLabel      en      This is a string
```

In the following example the English string ("Name of the product") belongs to the Main group, has NameLabel id, and the maximum string size in characters is 25.

```
Main_NameLabel   en      Name of the product //mlz MaxChars=25
```

In the following example the English string ("User name") belongs to the Login group, has UserLabel id, has a "A comment" comment, and the maximum string size in pixels is 120.

```
Login_UserLabel  en      User name //mlz A comment MaxPixels=120
```

20

Translating a Project

The grid displays one language at a time. There is also additional information available for each string such as context information or a comment. You can toggle these views on and off from the View menu. To change between languages, click on the drop-down combo box at the top of the grid and select the language you want to work on.

We could translate Finnish (or your own language) manually by entering the translations to the grid. However there is an easier way: automatic translation with the translation memory.

By default the translation memory is empty. You can add items to it in two ways. Every time you enter a string to the grid, Multilizer will automatically add every translated string to the translation memory. This means that you don't need to translate the same string twice. The second way to add items to the translation memory is to import glossary files, e.g. TMX or Microsoft® glossary files. You can also import previous translation memory files, e.g. mld files. You can even fetch translation data from databases or database servers.

For additional information about the translation memory, see the online help topic "Translation Memory".

Multilizer Translation Memory can also be configured to work on a database server. This enables multiple users to work with it concurrently. (→ Translation Memory on database server, p. 213)

The next task is to translate the Finnish (or your own language) column. If you have used Multilizer before to translate strings, or you have imported glossaries into the translation memory, we can now use translation memory to translate some of the strings. Right-click the header of the Finnish column, where it says "Finnish". A popup menu appears. Choose **Translate | Using Translation Memory**. Multilizer translates all the strings it can find from the translation memory. It is up to you or your translator to translate the rest of the project.

The editor is easy to use. Use the arrow keys to move the selection to the a cell and start typing. If you leave a translatable cell empty, the native string will be used instead. Because the native language is English you do not have to translate the English column at all. You can translate strings in the English column, if you wish to correct a typing-error or alter spelling of the native string, but if the native string is acceptable, you don't need to change it.

On the left side of the grid is a tree view from which you can select the types of strings you want to view in the grid. You can, for example, work on the menus first and then concentrate on the forms.

When you have translated the project, save it by choosing **File | Save**. For now you do not have to translate all strings, just those strings that you want to. Where there is no translation available, the native string is used instead, thus making it possible to test the localization as the work progresses.



MORE INFO

21

Adding Languages

It is quite likely that you need to add new languages to your applications after the original languages. This is the most powerful features of Multilizer. After making your application multilingual, adding new languages is a simple operation. You do not have to change the source code at all. Neither do you have to change resources (forms) in any way. In fact, you do not even have to recompile the application.

Let's add Swedish to the project file. Launch Multilizer. Choose **File | Open** and browse to the folder containing the dcalc.mpr project file. Choose **Project | Languages**. The Languages dialog box appears:

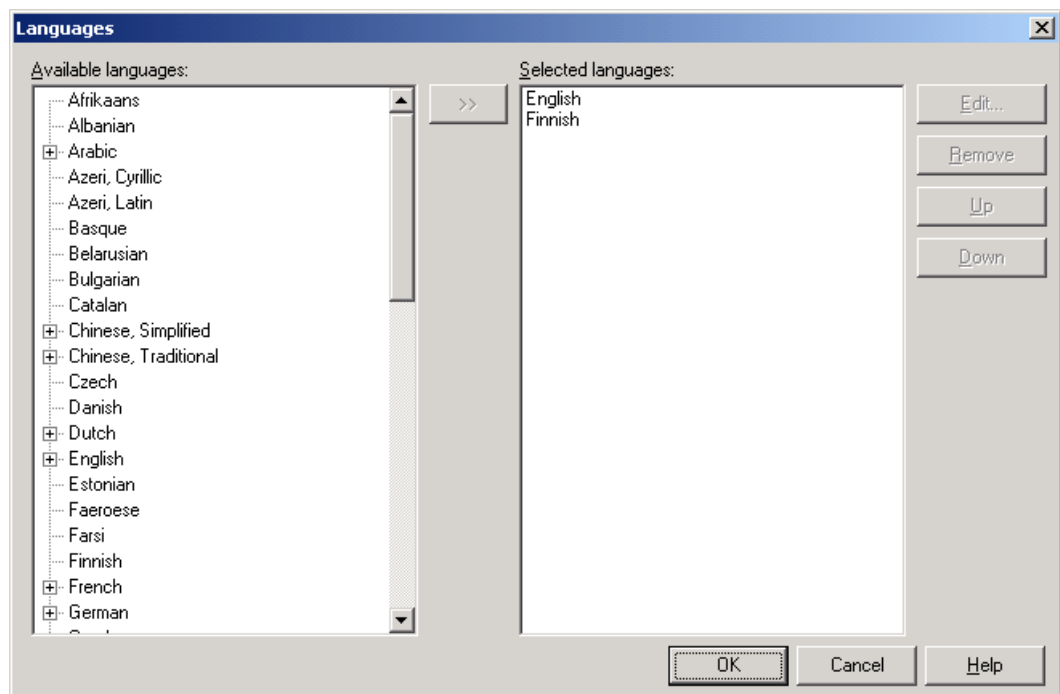


Figure 176 The Languages dialog box lets the user add or remove languages

Select Swedish from the available languages and press the >> button. Press the **OK** button to close the dialog box. Swedish column appears to project file's grid.

Changing the System Language

Some Windows applications are so called Ansi applications. This means that they do not use Unicode but code page related character sets. When running such an application the system code page of the operating system must match the code page used by the language of the applications. For example if you have an English Windows and you are trying to run an Japanese software you might get most of the strings containing garbage strings. The following image shows such an application.

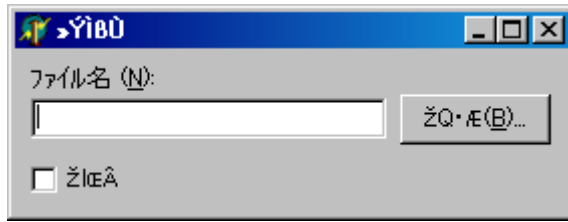


Figure 177 A Japanese application on English system code page

In order to show Japanese text correctly you must change the system code page to Japanese. The procedure depends on the Windows version but is always done using the Control Panel.

Windows XP

Start Control Panel and launch Regional and Language Options. From the Regional Options sheet set the Standard and format to Japanese and Location to Japan.

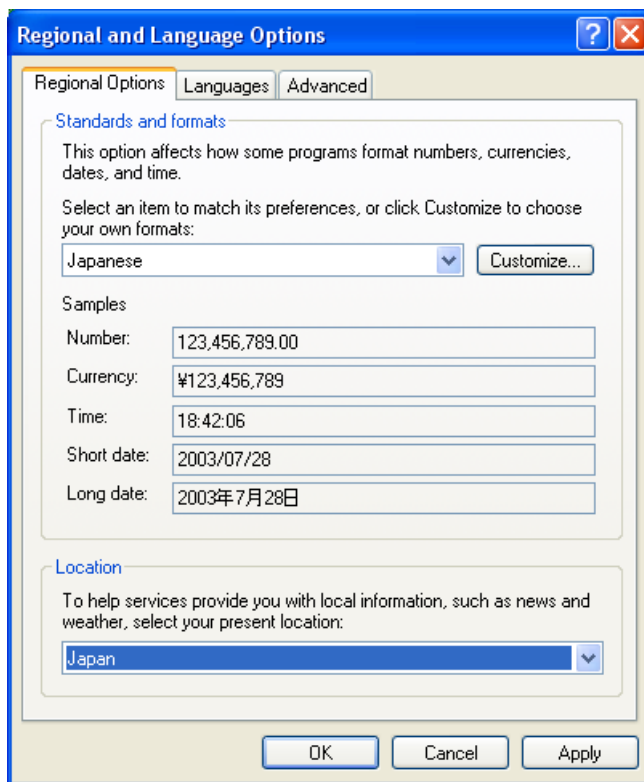


Figure 178 A Japanese regional options

From the Advanced sheet set the Language for non-Unicode programs to Japanese.

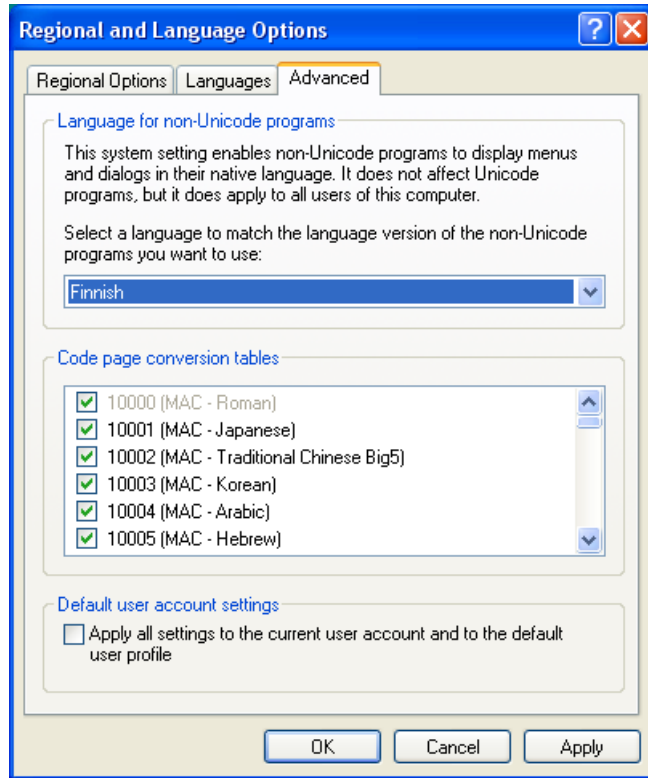


Figure 179 A Japanese non-Unicode options

Press OK. The system prompts to reboot. Do so.

Windows 2000 and Windows NT

Start Control Panel and launch Regional Options. From the General sheet set the Settings for the current user to Japanese.

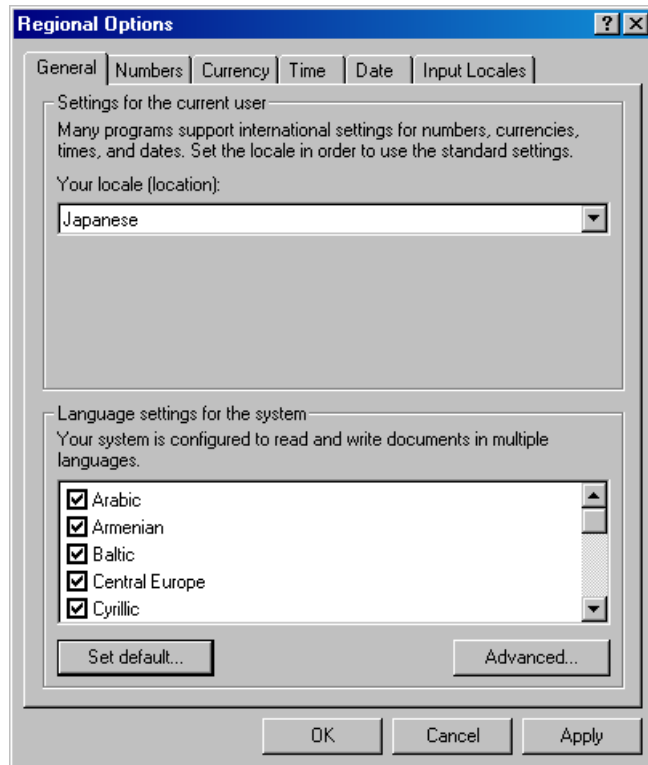


Figure 180 A Japanese locale options

Press the Set default button and select Japanese.

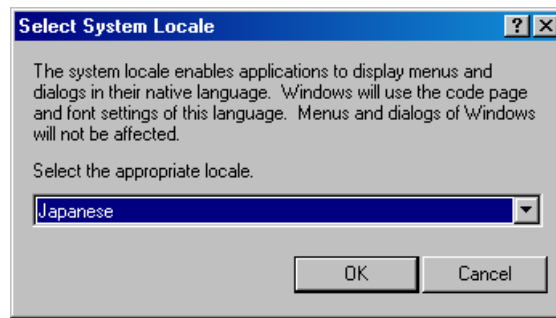
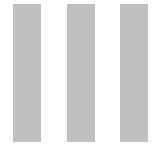


Figure 181 A Japanese non-Unicode options

Press OK twice. The system prompts to reboot. Do so.

Windows 95, 98 and ME

It is not possible to change the system code page. Either upgrade to 32-bit Windows or install a localized Windows 9x (e.g. Japanese Windows ME).



Part III: Using Multilizer

The purpose of this part is to familiarize you with the software globalization using Multilizer tools. To obtain the most precise definitions on use of components and technical details, please refer to the on-line help.

Using Multilizer part includes the following:

- Introduction to Multilizer user interface.
- Multilizer Translation Memory maintenance.

22

Globalization Process

The technical part of the globalization process can be divided roughly in three phases:

- Internationalization (p. 188-).
- Localization, including translation (p. 194-).
- Quality assurance (p. 210-).

It is essential to understand the needs and requirements of the international marketplace as early as possible in your development cycle, and then to build the capability to support these in your design and development processes.

We are designing customer-focused products and services already. It is a question of shifting that focus to embrace international customers as well. Every member of your team has to acknowledge this.

By doing this, you will be internationalizing your product - i.e. designing and developing a product in a way which allows a reduced time to market, reduced cost and higher customer satisfaction when the need arises to localize the product for a particular market.

The following chapters will go through the different phases and show how Multilizer helps to accomplish the aforementioned goals.

Globalization team

Background

Traditionally, globalization has been outsourced to globalization service providers. This has meant that the entire software project has been frozen to its current stage and sent to the service provider.

Multilizer technology is designed in a way that it provides software houses with a possibility to do a great part of the globalization in-house, with the following benefits:

- Continue development during on-going globalization.
- Produce globalized versions faster.

Team members

The globalization team that works with Multilizer consists typically of the following members:

- Project Manager.
- Software Engineer.
- Translator.

Project managers and software developers are found in software companies, and translators are normally freelancers or services bought from a translation agency. Following paragraphs go through the tasks that belong to different teams.

Tasks

In a typical software company, the developer will take care of all technical tasks. He will use Multilizer to create a new project, do the internationalization and localization.

Furthermore he might take care of quality assurance tasks. In addition to using Multilizer to accomplish these tasks, he typically needs to make changes to the software using the development tools.

Team member	Tasks
Project Manager	Project management.
Software Engineer	Internationalization, localization engineering, technical quality assurance.
Translator	Translation, linguistic quality assurance.

Following paragraphs will explain the role of Multilizer in the tasks above.

Work-flows

As mentioned above, the globalization team consists of a project manager, a software engineer and a translator. The picture below shows the globalization project workflow in its simplest: there is a software engineer developing software and doing the internationalization/localization related software development and a translator that does the translation.

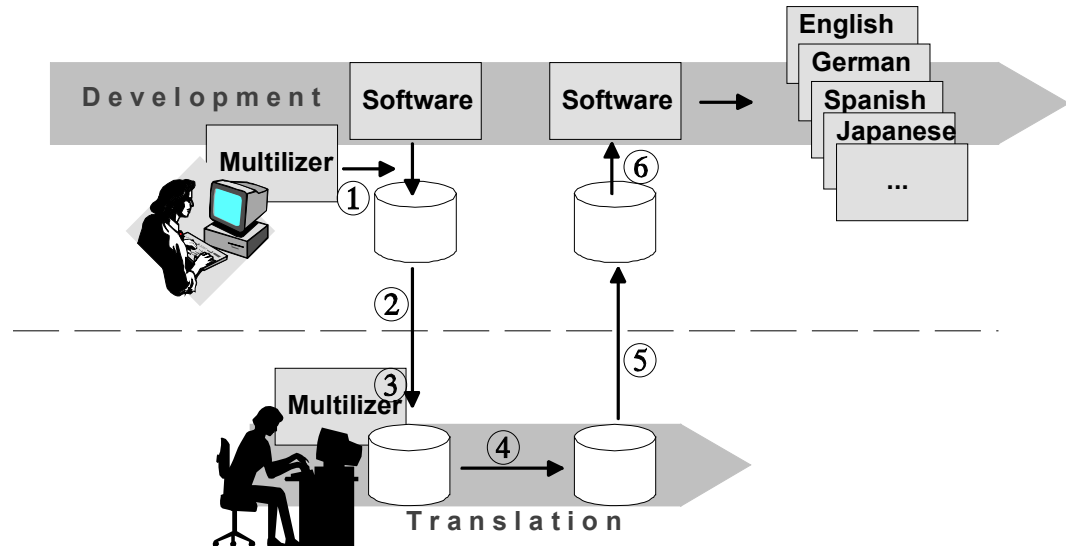


Figure 182 Multilizer globalization workflow

1. The programmer uses Multilizer to extract strings to be localized from the application and saves them to a project file.
2. The programmer sends the Localization Kit (the project file, Multilizer® Translator Edition™ and related files, such as documentation, schedule or terminology glossaries/translation memory) to the translator(s).
3. The Localization Kit is a self-installing Windows executable that installs in translator's PC.
4. Translators will complete the translations using Multilizer, and benefit from features like translation memory and glossaries that automate translation work.
5. When the project is translated, the translator sends the translations back to the developer.
6. Finally, the developer uses Multilizer *Import Wizard* to integrate the translations to the project file and build localized resources or executables.

Enabling concurrent work

It is possible to proceed simultaneously with the development and translation. Therefore, the savings in time are bigger the earlier the translation starts. The Import Wizard ensures



NOTE!

that translations are easily integrated in the software even after major changes in the software.

Localization type

Localization type impacts the work-flow a lot. Technically the differences affect just the way localized versions are built (After phase 6 in the preceding figure). Following table describes what building localized versions means in the context of different localization types:

Localization Type	Build localized versions
Binary localization	The strings of the original executable (binary) are replaced with translations. The result is one localized executable for each language.
Source localization	The strings in the source code files (string resources, form files, etc.) are replaced with the target language strings. After creating localized source files, the project(s) are compiled into localized executable(s).
Component localization	The strings in the source code files (string resources, form files, etc.) are included in a database along with the equivalents in target languages. Multilizer components are added to the project source code to link the database in the software to enable multilingual behavior. Finally the software is compiled into a multilingual executable.

It is important to understand that none of the localization types require the source code be sent to the translator as long as Multilizer is used in the company that develops the software.



NOTE!

If the entire localization process is outsourced to a localization vendor, then binary localization is the localization type required, if software source won't be sent to the localization vendor.

More info on localization types is available in the Getting Started section of this manual.

23

Internationalization

Internationalization (I18N) is the process of generalizing the software so that it can handle multiple languages and cultural conventions without the need for re-design.

An important aspect of internationalization is the separation of text from the software source code. There are two reasons for this:

- Replacing the original texts with translations will not affect the source code.
- Translators won't be able to change – or break – the program code.

Basically internationalization is moving translatable text (i.e. any text visible to the end-user) to separate resource files. This makes software's executable code language-independent, preventing localization from resulting in multiple code-bases.

Different software development environments, development tools and operating systems provide different technical foundations for internationalization. To add the best possible globalization support, Multilizer provides different ways to do the internationalization. These are introduced in the subsequent chapters.

To do's

In the Multilizer context internationalization covers the following:

- Internationalizing software (manual work).
- Extracting strings from software (automated).
- Checking internationalization quality (automated check routines).

The degree of manual work in internationalization depends on the localization type. With binary localization all strings must be isolated into resources. In source-code localization, Multilizer is capable to extract also strings that are not in string resources, but within program code, for example. In component localization, Multilizer components add extra code to software, doing essential parts of the internationalization automatically. These are explained in the end of this chapter.

Extracting strings from software is done automatically. This is done for the first time when creating a Multilizer project file. This is explained in the following chapter.

Internationalization QA checks are explained in the QA section of this manual. C.f., Quality Assurance – QA, p. 210.

Creating a new project

By launching Multilizer and choosing **File | New**, Project Wizard will start guiding you through the necessary steps to create a new project file.

Specifying the target type

The purpose of the Project Wizard is to add the initial target to the project. A target specifies an item to be localized. The item can either be a file or a database.

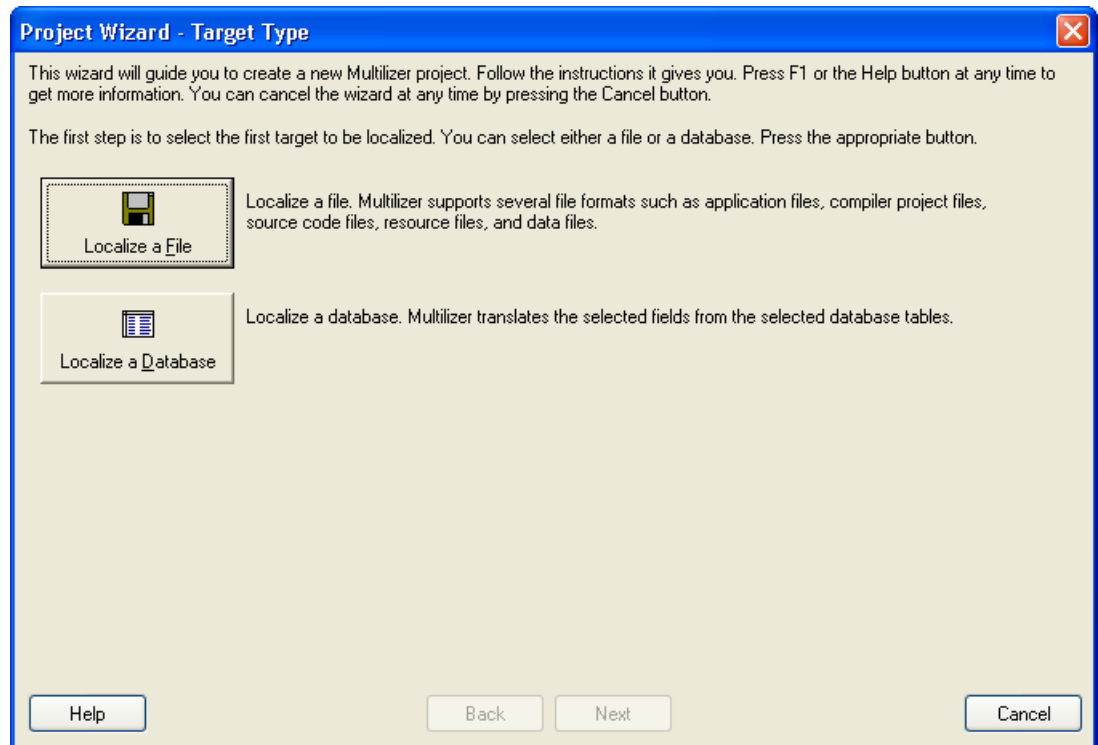


Figure 183 Starting the Project Wizard

If you plan to add a file based target press the **Localize a File** button. If you plan to add a database based target press the **Localize a Database** button.

Specifying a file target

If you pressed the Localize a File button your first task is to select the directory where the file is located. If the directory contains file, Multilizer tries to detect the type of the file. if succesfull it sets the platform and the target types to match the file type.

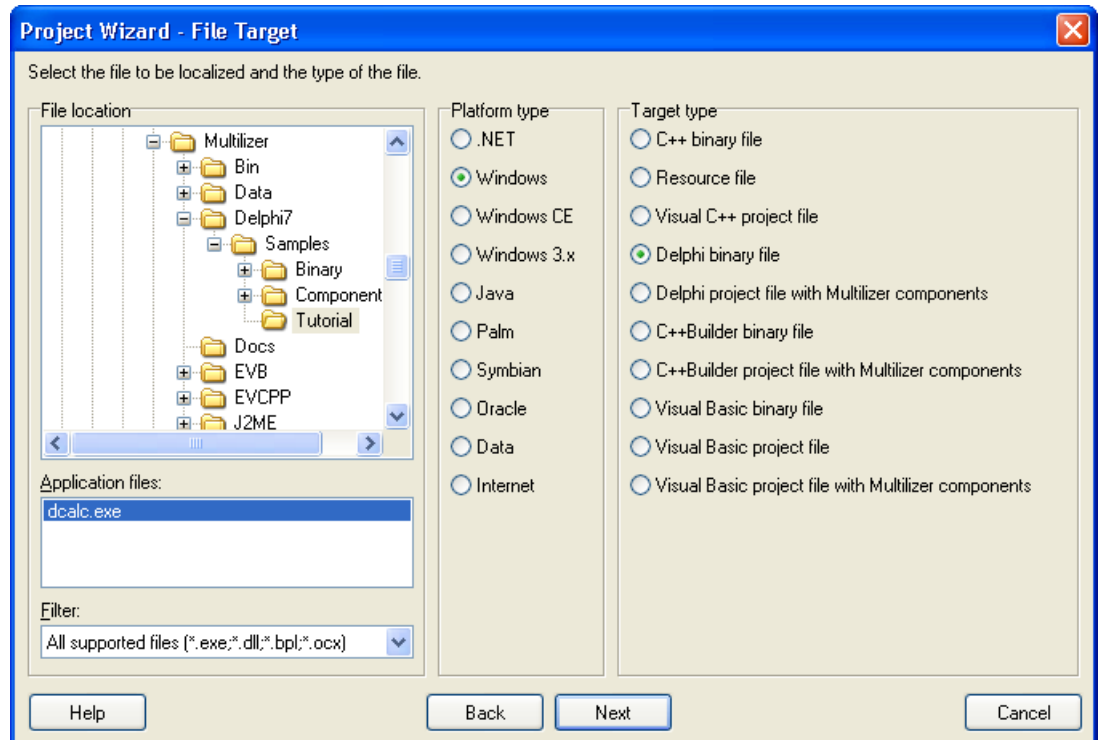


Figure 184 Selecting the directory of the program source and application type

**MORE INFO****NOTE!**

The different platform and target types are discussed in the Getting Started section of this manual.

In the same view as where you choose the directory, you define the application type. Multilizer automatically detects what kind of source is located in the specified folder. If detection fails, you can force the type to be one of those mentioned above.

Entering project information

The next screen is for entering information to the project. This information is useful for later identification of the project and the project file. You always have to enter the File Name.

Project Wizard - Information

Accept the default project file name or change it. You can enter some other project related information as well.

File name
C:\Program Files\Multilizer\Delphi7\Samples\Tutorial\dcalc.mpr

Project information

Author:
Organization:
Description:

Help Back Next Cancel

Figure 185 Entering project information

Selecting languages used in the project

After entering project information, you proceed to the language selection view. In this view you specify which languages to include in your project.

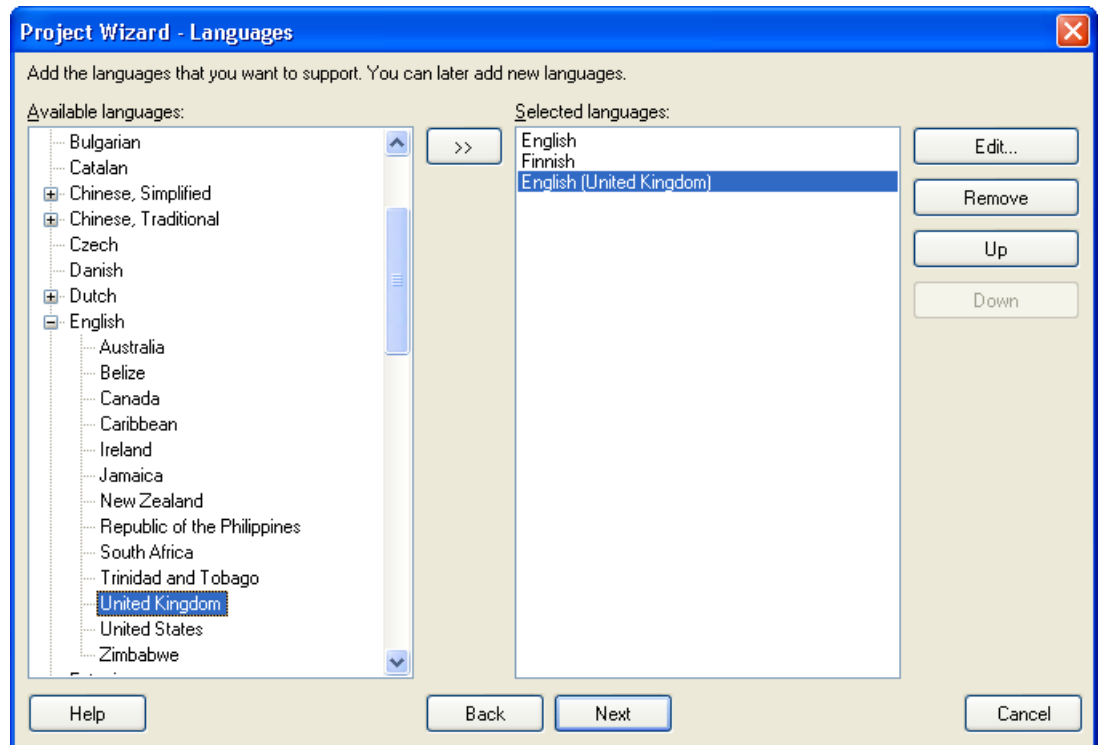


Figure 186 Selecting languages and sub languages

Although the original software was created in English, you can add English also to the selected languages. It gives you the possibility to change the original language's texts.

Both languages and sub-languages can be selected to be used at the same time in one project. The difference is that a sub-language is country specific. E.g., British English differs from US English. Therefore you can specify which language to use. Of course, you can select both English versions for your project.

You don't have to know at this phase which languages you might add in the future. New languages can be added later, even during the localization project.



NOTE!

Adding Targets

The next screen lets the user add different targets to the same project. To add a new target press the Add button. There can be any number and kinds of targets in the project. For example there can be a Delphi executable, a Palm executable and an XML file all in the same project.

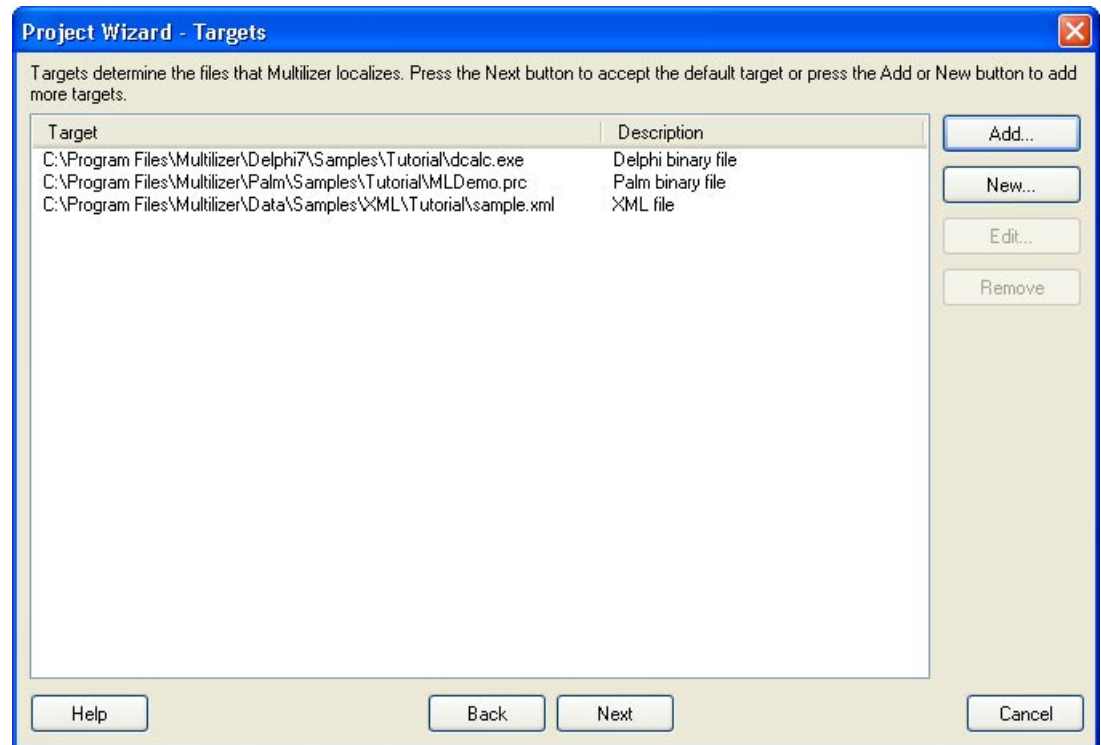


Figure 187 Adding targets to a project

Finishing the Wizard

The next step is to scan the targets and create a new project file – press Finish. Now your applications are being scanned in order to find the strings to be translated. The project file will include strings from the targets

You can now proceed to editing the translations. The following chapter familiarizes you with the Multilizer interface.

I18N essentials

Although Multilizer is optimized to scan strings from each supported software project type, there are situations that not all strings get extracted. This is due to the fact that the project source code isn't prepared for localization, i.e. it is not internationalized.



NOTE!

The developer has to take care of re-working software so that Multilizer can be used efficiently. The following list contains some very essential issues that can affect the entire globalization project if not taken into account:

- Eliminate UI text length restrictions. Translated strings are typically longer than the English strings. This is a typical GUI design issue, which involves re-designing dialogs and frames.
- Ensure support for accented characters, including double byte (if your software is intended for Far Eastern markets).
- Check for hard coded strings in the source-code. Move them to string-resources or use the proposed way in the development platform that you work with.
- Enable support for foreign keyboard layouts.
- Avoid fixed date, time, currency or number formats, use the operating system's support to handle these.
- Avoid text in bitmaps as they are hard to edit. Multilizer scans for strings, thus it doesn't recognize any text in bitmaps.

Keeping the previous issues in mind, following chapters will show what Multilizer does in the internationalization phase of source-code, component and binary globalization projects.

Source-code globalization

In source-code globalization projects, Multilizer will do the following in the internationalization phase:

- Scan through the source files.
- Add strings to the localization database.

Multilizer will locate all strings, assuming that they contain locale-dependent data. The source-code will not be altered. Instead, Multilizer will create localized source-codes of the native version (C.f., Localization, p. 194).



NOTE!

By default, Multilizer extracts all strings from the source-code. Because some strings are used for other purposes other than to communicate with users, Multilizer let's you alter source code scanning options. This way you can ensure that the localization database will include locale-specific data only. In Multilizer workspace the strings can also be locked to prevent their localization.

Component globalization

Multilizer component globalization projects are done much in the same way as source-code globalization projects. The biggest difference is the use of Multilizer components that enable the creation of multilingual software that lets the user change the language at run-time.

Multilizer will do the following in the internationalization phase:

- Scan through the source files.
- Add strings to the localization database.

In addition to using Multilizer, Multilizer components are added to the software project:

- Add the Dictionary component to attach the localization database to the software.

Binary globalization

In binary globalization projects, Multilizer will do the following in the internationalization:

- Scan through the binary.
- Add strings to the localization database.

Multilizer will locate all strings, assuming that they contain locale-dependent data. The original binary will not be altered. Instead, Multilizer will create localized binaries of it (C.f., Localization, p. 194).



NOTE!

By default, Multilizer extracts all strings. Because some strings are used for other purposes other than to communicate with users, Multilizer let's you modify the way Multilizer performs the scan. This way you can ensure that the localization database will include locale-specific data only. In Multilizer workspace strings can also be locked to prevent localization.

24

Localization

Localization involves making the software linguistically and culturally appropriate to the target locale (country/region and language) where it will be used and sold.

This chapter focuses on the engineering aspects and the following chapter will cover the translation tasks.

To do's

In Multilizer context, localization covers the following:

- Preparing a project to be outsourced (manual work; optional).
- Translation of the software (manual work; automated).
- Integrating translated texts in Multilizer project file (automated; → Importing translation to a project, p. 204).
- Creating localized versions of the original software. (→ Build localized software versions, p. 208)
- Checking the localization quality (automated check routines; → Quality Assurance – QA, p. 210).

The translation is initiated by creating localization kits that are sent off to the translators. Building a kit with the *Exchange Wizard* is explained in this chapter, the translation work is explained in the following chapter.

When translations are completed, translators send them back to the developer who integrates them into project file with Import Wizard.

Once the translations are integrated in the project file, the developer can build localized versions of the software. Depending on the localization type, the result of the build process varies (C.f., Localization type, p. 187).

The localization quality checks are explained in the QA section of this manual.

Preparing a project to be outsourced

Multilizer automates the creation of the project file and picks the translatable data from the original software. Although this might be sufficient, it is recommended to check out the project data and adjust Multilizer project settings in the way that the translation can start as effortlessly as possible.

Before sending off the translations to linguists, the developer can include supportive information in the project file. The purpose is to prevent errors in the translation process, thus stream-lining the overall process flow.

Key tasks:

- Adding visual context.
- Adding comments.
- Locking strings – preventing translations of strings.
- Adding project strings.
- Adjusting scanning options.

- Updating the project file.
- Pre-translation.

The supportive information is easily included in the Localization Kit sent to the translator. Multilizer provides, automatically if wanted, all (Multilizer) documents, tutorials, translatable material and if wanted the translation tool, documentation and source in an easily deployable kit – the localization kit.

Adding visual context

Adding visual context means that images (screenshots) are attached in the project.

For example, in the following picture, the user has right-clicked item #14 in Dialogs. If the user has copied the dialog from the original software to the clipboard, he can choose **Paste Image**, and the screenshot is automatically added in the project.

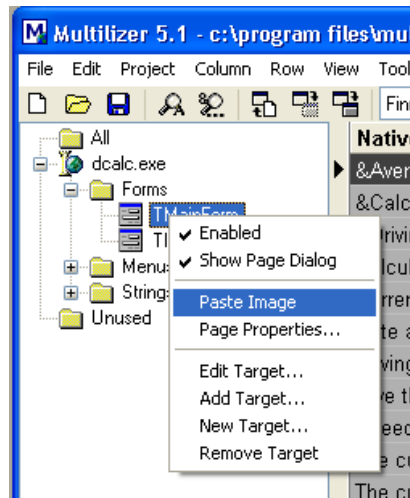


Figure 188 Multilizer User Interface - the left-hand side pane

By choosing **Page Properties**, the user can also add textual information that lets the developer give the dialog resource a more descriptive name and comments. Targets can also be added, edited or removed from this menu. Finally, with the extra information attached, the working place looks as follows.

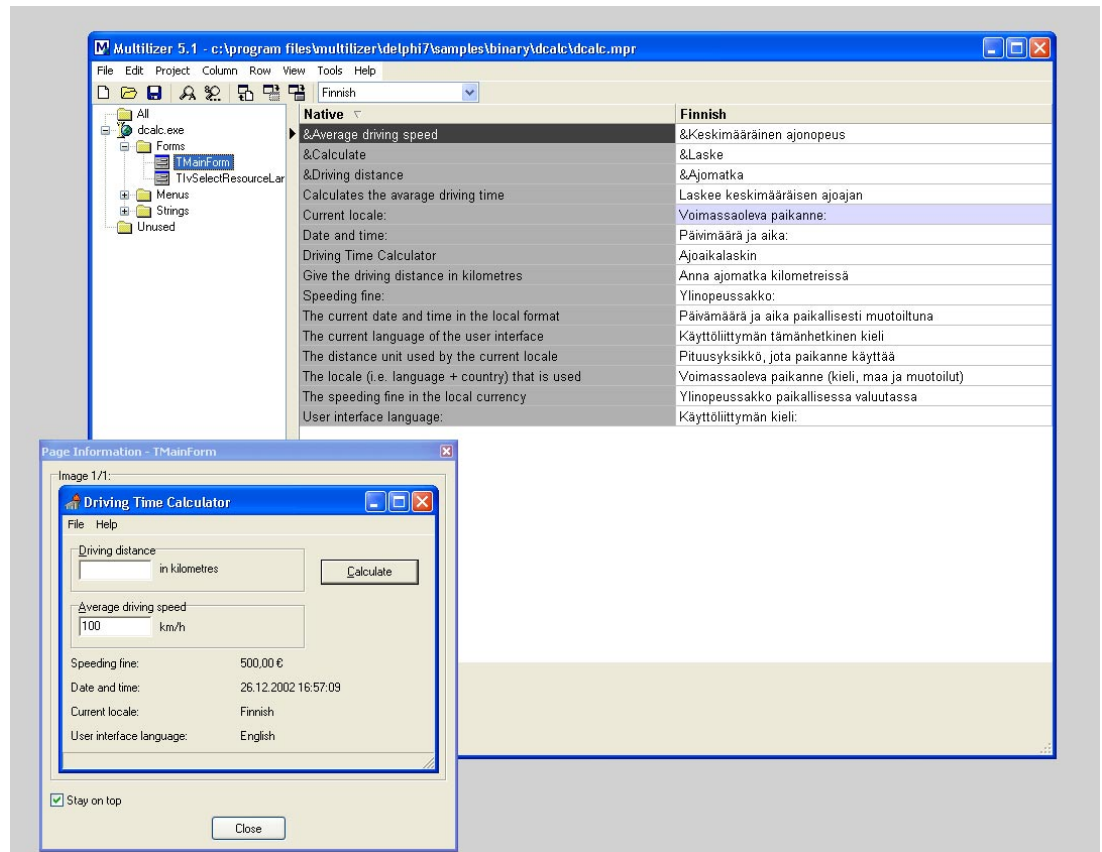


Figure 189 Multilizer User Interface - the right-hand side pane

With the visual context, it is easy for the translator to do the translations. He immediately sees where the translations are placed. Furthermore the description gives a hint how to find the dialog in the original software. The 'goto dialog' explains more than just a resource id.

Adding comments

With the same principles as adding comments to a program part, such as a dialog, you can add comments that are related to a single text item in the project.

The simplest way to add comments is to right-click the left margin of the translation grid and choose from the context menu **Comment...** A comment dialog opens and lets you type in the comment.

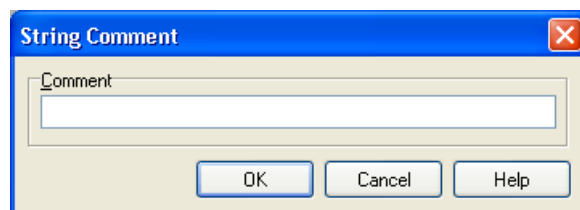


Figure 190 Translation Comment dialog box

You can also turn on showing comments in the Translation grid by choosing **View | Comment** from the main menu.

Locking strings – preventing translation of strings

There are strings that shouldn't be translated in any occasion. For example, by default Multilizer finds SQL Query strings from the software and adds them to the project. However, these are seldom translated. To prevent the translation you can:

- Tell Multilizer not to scan SQL Query strings (C.f., Adjusting scanning options, p. 198).

- Exclude strings from the project file. **Choose Project | Exclude Strings | By Native...** and type in the native strings that mustn't be included in the project.
- Lock the string.

Locking strings is easy. Right-click the left margin of the translation grid and choose from the context menu **Do not translate**. The line gets gray and editing translations is prevented.

Delphi & C++Builder; component localization only.

Adding project strings

It may happen that not all the strings are added to the project after the first scanning. You may also need to maintain separately a group of strings that you need to include in your software. These conditions can become true in component localization projects with Delphi or C++ Builder.

For the purposes mentioned above there is a possibility in Multilizer to add strings pertaining to a certain logical group.

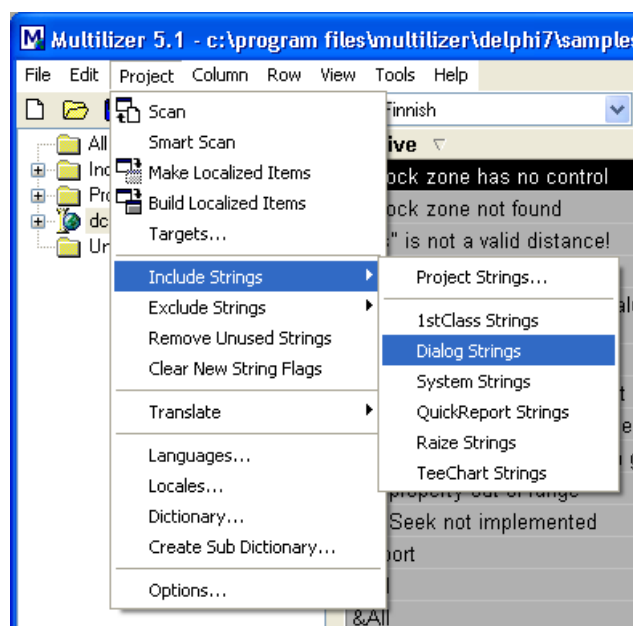


Figure 191 Multilizer User Interface - the right-hand side pane, including special purpose strings to the project

If you choose e.g. Dialog strings, you get the following Dialog, which lets you add all the strings encountered in the dialog that you choose.



Figure 192 Standard Dialog strings

The actual dialog names may vary depending on your application type.

These additional included strings are defined in the Multilizer/bin program directory in *.mls files.



TIP!

You can easily define your own string groups for any third party component. Multilizer lets you then add the strings to the project. To do this, you can either create a new *.mls file or edit any of the existing ones. The syntax of these files is described in the Multilizer help. It is also easy to modify an existing one and get the syntax from it. See more in the online documentation under the topic mls.

Adjusting scanning options

After creating a new project, you might encounter unwanted strings in the project. To prevent Multilizer from adding such strings, you can adjust scanning options and after that update the project.

Scanning options are defined by choosing **Project | Targets...** from the menu. It opens the following dialog.

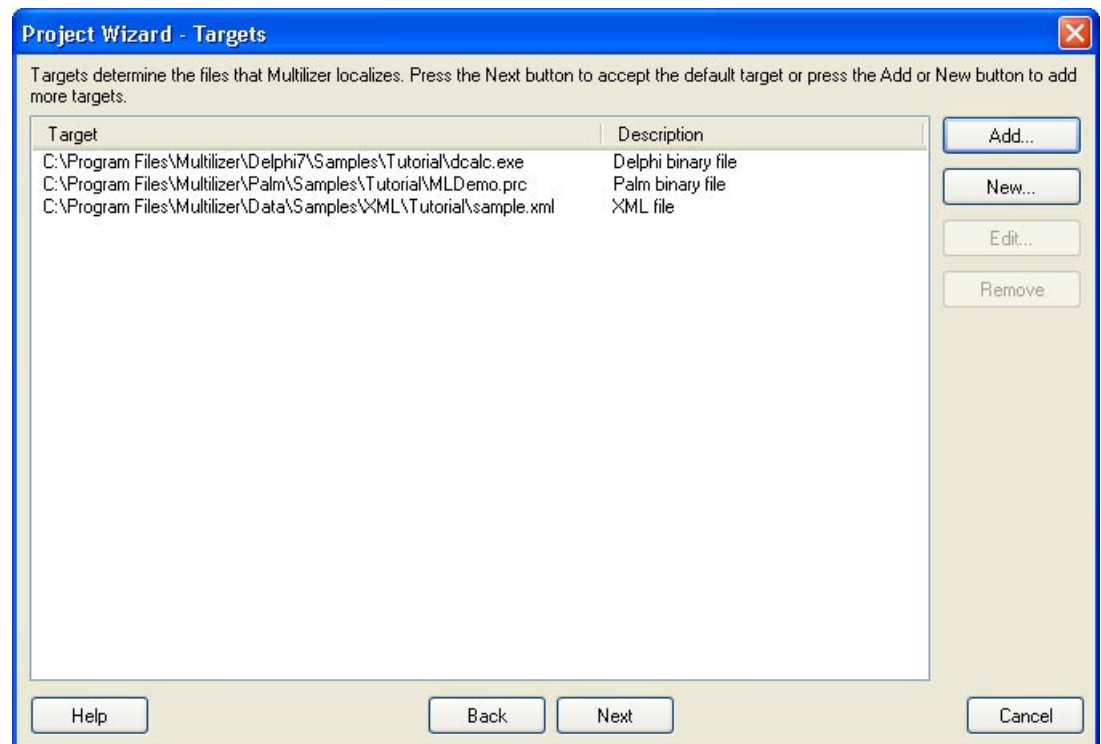


Figure 193 Defining targets

Using the Targets dialog, you are able to define the targets to be scanned in your project. For each target you can also define file-type specific scanning properties.

To add new files to be scanned, press the **Add** button. Multilizer shows the Target Wizard. It is similar to the Project Wizard used to create the project.

Another way to add a target is to press the **New** button. Multilizer shows *Add Target* dialog box that displays a list of the different target types recognized and supported.

In typical projects, there is only one or a couple of targets in one project. For example, a small Palm project might include only the Palm Application in the targets list.

To get software development-environment specific guidance in adjusting scanning options, consult the on-line help. The development tool / platform-specific tutorials also give more information of the file types mentioned above.



MORE INFO

Updating the project file

You have to ensure that the project file is up-to-date. This means that when you have made changes in the software during the localization process, Multilizer must be told to

synchronize the software's strings with the strings in the updated software. This is done easily by:

- Pressing the scan button on the tool bar or
- Choosing **Project | Scan...** from the menu.
- Choosing **Project | Smart Scan...** from the menu.

Scanning

Once scanning has been started, the following window appears on the screen showing how the scanning is proceeding.

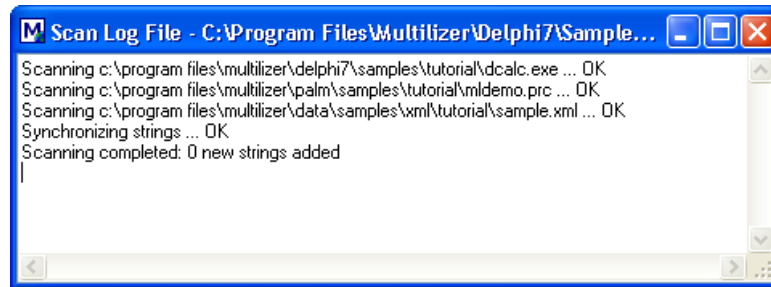


Figure 194 Window showing the project update process

By default Multilizer scans all the strings to be translated, using the default scanning options. The next chapter describes how you can customize the scanning of your program source.

Pre-translation

Automated pre-translation of the software before sending it off to the translators has many advantages. The two most important are:

- Company-specific terminology will be translated following the company policy
- Automated translation saves time and costs.

Choosing **Project | Translate | Using Translation Memory...** does the pre-translation. Multilizer compares the original 'Native' string and searches from the translation memory the translation for it.

To be able to do the pre-translation, the translation memory must be configured properly and there must be translations in it. More info on this is available in Translator's Manual. Also see: Translation Memory on database server, p. 213

Although Multilizer is capable of re-using translations by storing them in a Translation Memory, it is recommended that a linguist is given the possibility to validate/correct them.



Building a Localization Kit

The Exchange Wizard provides an easy way to send and distribute project data. It lets you create a package including both Multilizer (without database support) and the project.

Exchange Wizard works in two ways:

- The developer uses Exchange Wizard to send the (sub)project to the translator.
- The translator uses Exchange Wizard to return the (sub)project back to the developer.

To start the Exchange Wizard choose **File | Exchange**. The first step is the specify the delivery method. Multilizer lets you either create an exchange file or upload the exchange file to a service. Services are enabled if you have installed one or more service DLL.



Figure 195 Selecting the delivery method

The next step is to specify the languages to be sent. In the same dialog the strings that are included can also be selected based on their status. Also the application(s) that belong to the project can be included to the package.

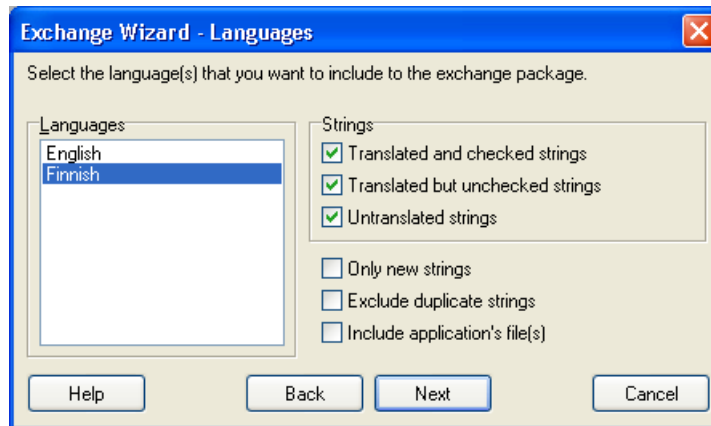


Figure 196 Selecting languages

Next step is to specify the properties for the project to be deployed.

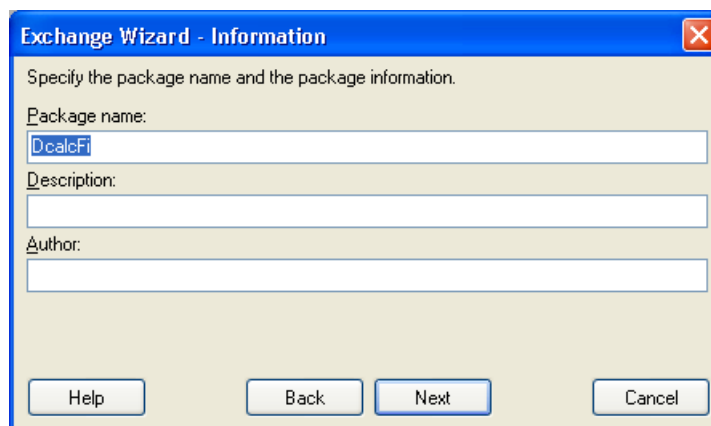


Figure 197 Specifying project properties

Next it can be specified whether the Multilizer application itself will be included in the package. If selected, Multilizer will automatically be installed when translator receives the package. At this point, also Translator's online manual can be included in the package.

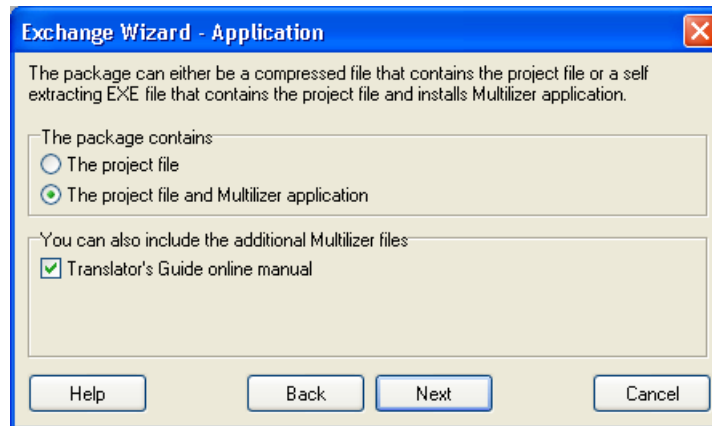


Figure 198 Selecting the type of the package

Press the next dialog button.

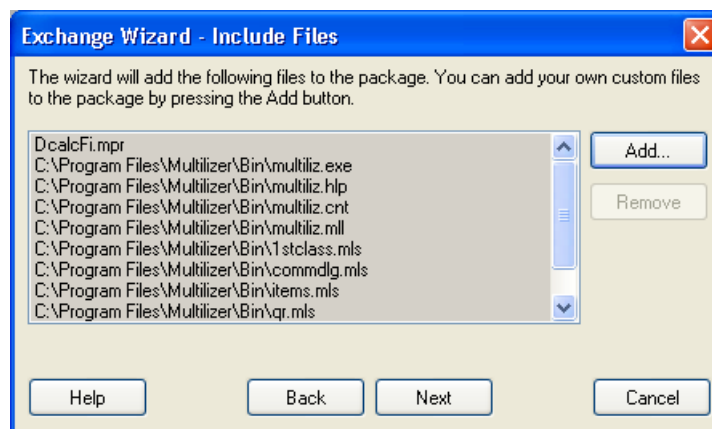


Figure 199 Files included in the deploy package

You can add any other files to the package by pressing the Add button.

In the last step, you have to specify the name for the package. Then press **Finish** to create the package.

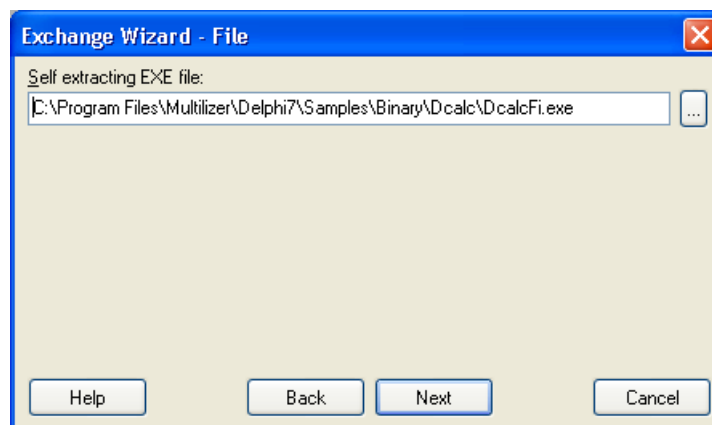


Figure 200 Specifying the package executable name

Finally the Wizard informs you what file(s) you have to send to the translator.

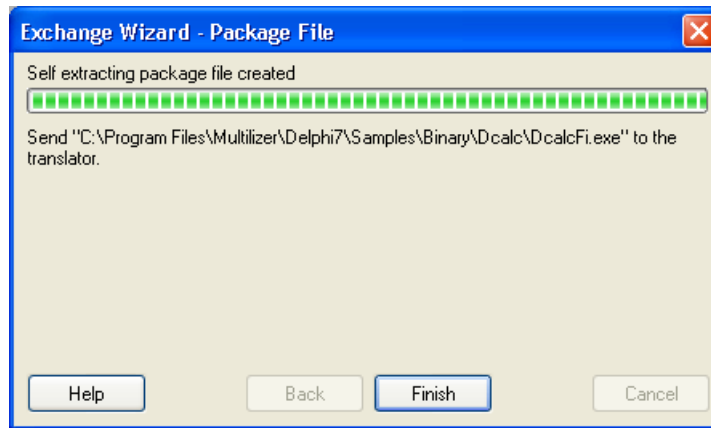


Figure 201 Package was created successfully

25

Translation

Translation is an essential part of software globalization. Persons with no technical background generally do this part of the globalization project. Therefore, in over 90% of globalization projects the translation is outsourced.

Translator's Guide goes through the translation related tasks. It describes the Multilizer user interface from translator's point of view. Furthermore, it tells how to maximize the re-use of translations using Multilizer's Translation Memory.

Both MULTILIZER® 5.1 and MULTILIZER® 5.1 Translator Edition are based on the same technology and formats. Therefore, using Multilizer throughout the localization process – including translation – guarantees the best quality and shortest project loop-through time.

26

Build localized software

Building localized software is the final phase of localizing software with Multilizer. Binary and source localization projects will generate localized software, while component localization projects generate multilingual software.

To do's

Following steps have to be completed to build the localized software versions:

- Integrate the translations into a project if translation was outsourced. (→ Importing)
- 1) Build localized executables (binary localization project) or
2) Build localized source files (source localization) or
3) Create a Project file that is integrated in the software with components (component localization).
- Compile the software (not needed in binary localization nor in certain component localization projects).

Importing translation to a project

By using the Import Wizard you are able to integrate outsourced translations back into the project. Importing translations is a straightforward process. However, imprecise use of the import command may cause loss of data. Therefore the user must use this command with care.

Starting the Import Wizard

The Import Wizard is started by choosing **File | Import...** from the menu. The following dialog appears.

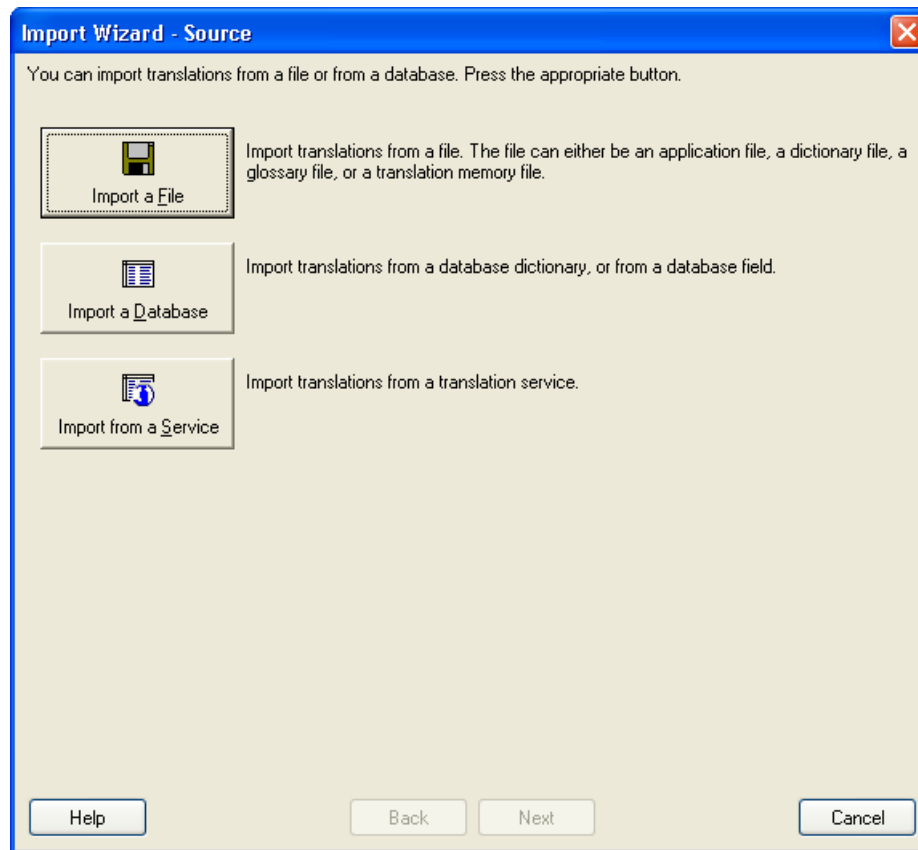


Figure 202 Select the source for the import

In this dialog you select whether you want to import a file (e.g. project file, dictionary file, tmx file etc.), a database (any ADO/ODBC compatible, DBISAM, DB2, Interbase, MySQL or Oracle), or import data from a service.



Depending on the Multilizer version and the configuration of your computer, there may be 'Import from a Service' button disabled.

In the dialog below, 'Import a File' was selected and a dictionary file was entered.

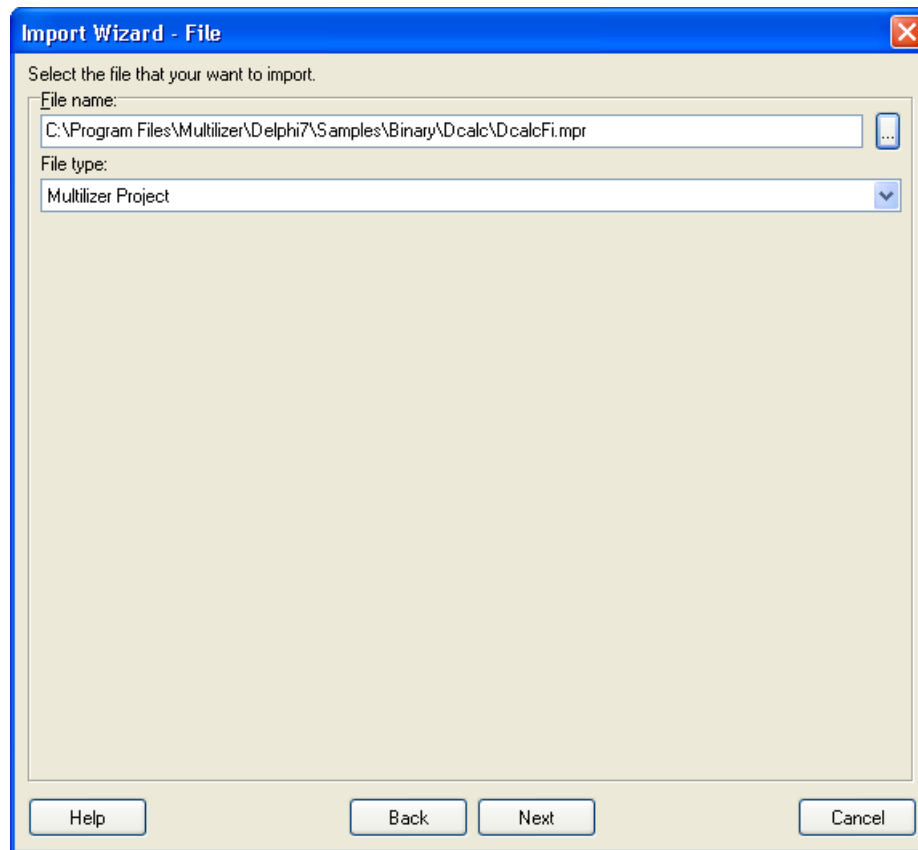


Figure 203 Select the source for the import. Here 'File' was selected

When you are ready, press the **Next** button to proceed.

Specifying import properties

After specifying the import data type, you have to select the languages that will be imported.

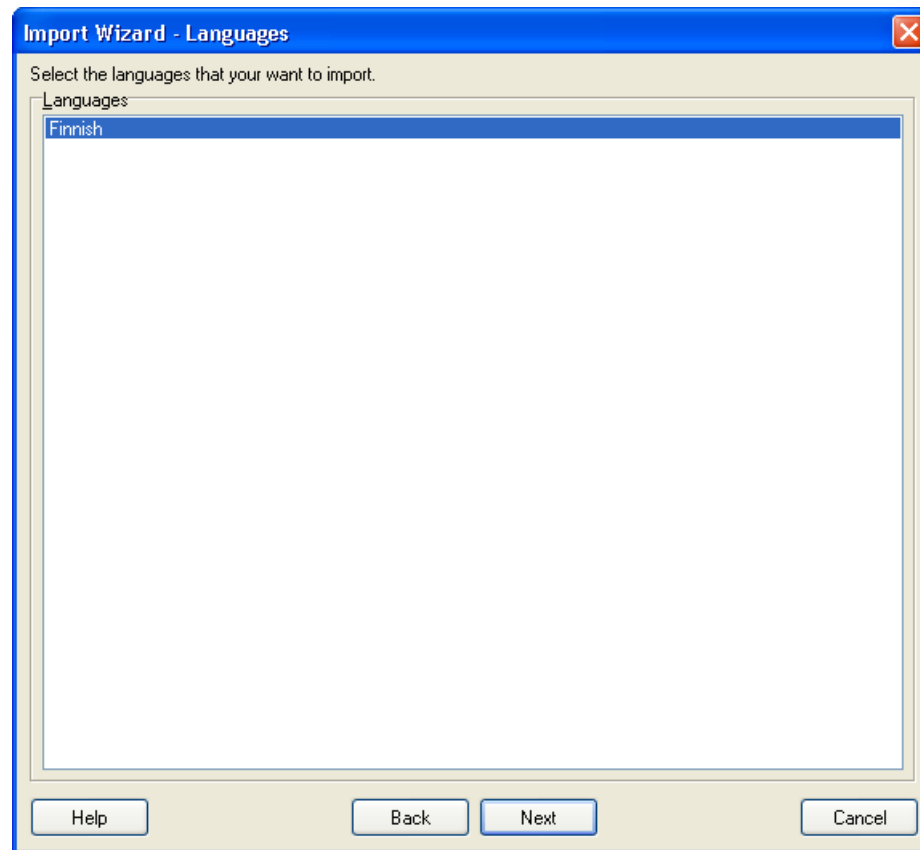


Figure 204 *Select the language(s)*

The Languages list box contains the language that will be imported. By default all the languages are selected. Deselect the language(s) that you don't want to import. (In the dialog above, there is only one language to be imported.)

Press the **Next** button.

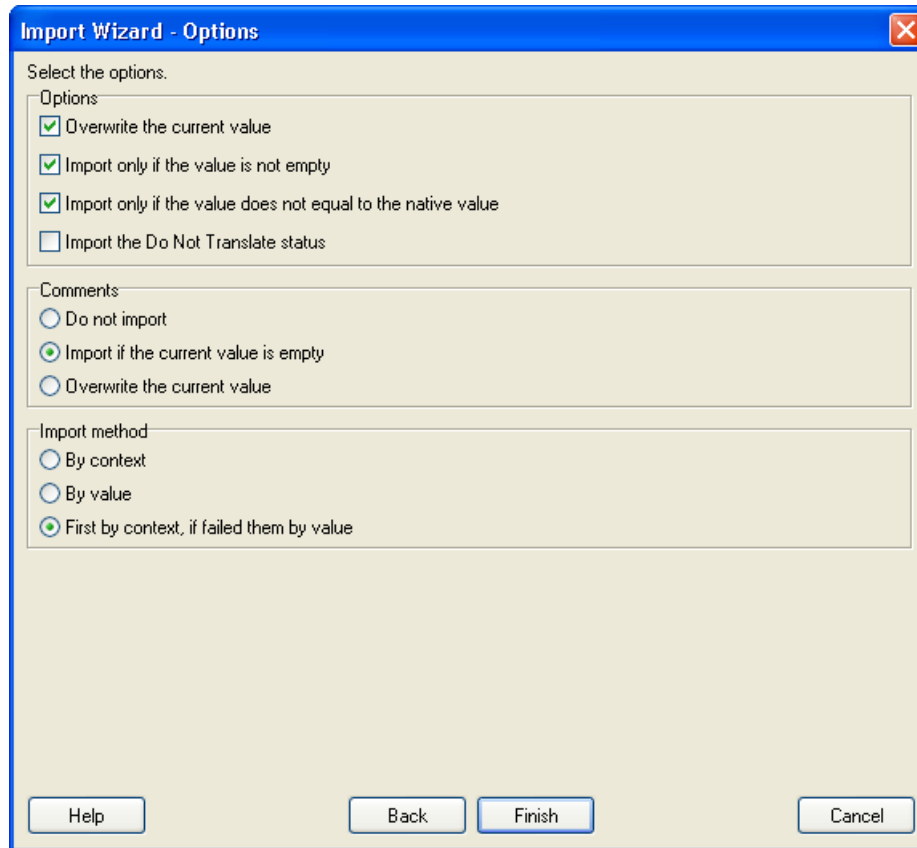


Figure 205 Import Options

Select the import method. Press F1 to get detailed information about the options. Press the **Finish** button to import the project.

Build localized software versions

When the translations are completed in the project, it's time to create the localized versions of the original software. Choose **Project | Build Localized Items** or **Project | Make Localized Items** to start building the localized software versions.

The Build or Make commands can also be executed the toolbar:

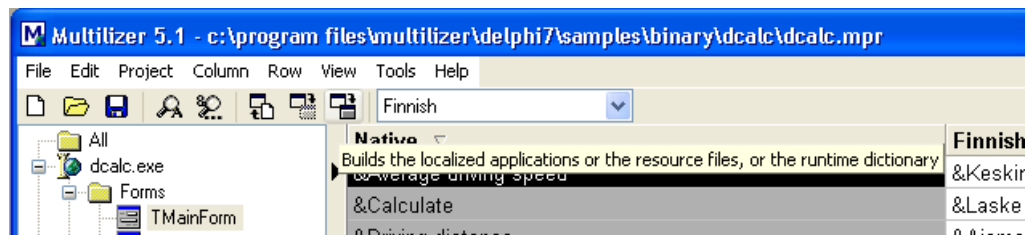


Figure 206 Multilizer user interface

You can also use the command-line tool MIBuild.exe to create localized software versions. This enables the use of efficient build batches.



TIP!



MORE INFO

More information of the localization types is in the Getting Started manual and the tutorials.

Binary localization

In binary localization the build process creates localized executables, one for each language. These are ready-to-run language versions of the original software and should just pass the quality assurance before distribution.

Because binary localization affects just the resource-part of the original software, there is no difference in the localized software's performance compared with the original one.

Source localization

In source localization the build process creates localized resource/source/project source files. After this, the software has to be compiled into localized executables.

Consult you software development environment documentation, if you want to add the compilation into your batch process.

These compiled language versions should pass the quality assurance before distribution.

The source localization and the support for different source code formats differ heavily between the software development environments supported. More info is available in the tutorials.



Component localization

In component localization the build process creates a Run-time Dictionary that contains the translations and the locale data.

The Run-time Dictionary is attached to the software with Multilizer components, and finally the software is compiled into one multilingual executable.

The Run-time Dictionary can be stored in different formats, such as Ansi text file, Unicode text file, Multilizer Binary file as well as database tables. Text files and Multilizer binary file can be embedded in the executable, to create a single multilingual executable.

Multilizer components add also code into the software that enables advanced L10N features, such as changing language at run-time.

If translations are stored in an external Run-time Dictionary, additional language support can be added without recompiling the software. Updating the Run-time Dictionary will upgrade the software's language support.

27

Quality Assurance – QA

Although following the guidelines of this manual prevent many quality assurance concerns, there are several issues that cannot be predicted. Multilizer includes features that make it easier to detect such issues.

The different levels of testing software in an ongoing software localization process are the following:

- Internationalization testing
- Localization testing
- Functionality testing

The two first testing levels are applied during the ongoing Multilizer localization process. This chapter introduces some basic features in this area.

Before releasing the final version of the localized software, the functionality checking should be done. Functionality testing verifies, for instance, how compatible the localized application is with localized operating systems and applications, or local hardware standards.

Test languages

You can define and use Test languages in Multilizer to detect I18N issues before even sending out any data to the translators.

After first scan, you can define that some of the languages included in the project is a test language. Right-click the header of any language column to open the language properties. Choose **Properties...** and the **Test** tab.

You can choose between following Test languages:

- Cover – converts all characters into a sequence of identical characters.
- Minimum – truncates the string into one character.
- Pseudo language – gives a rich set of features to simulate real languages.

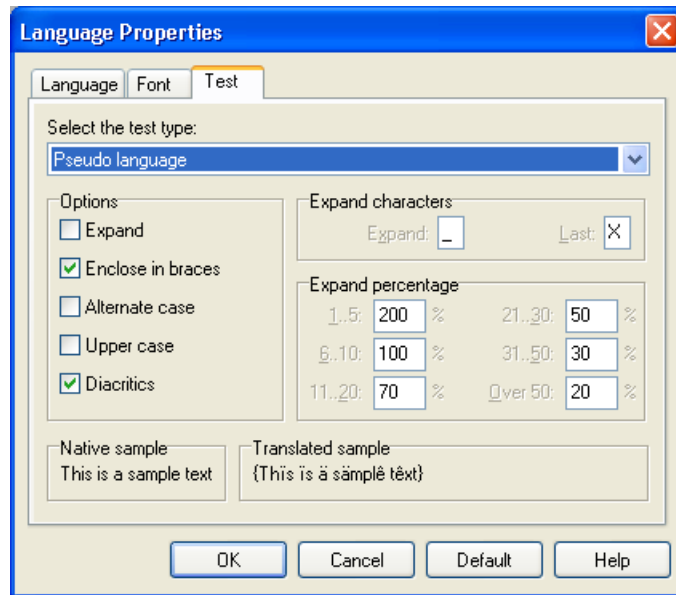


Figure 207 Language Properties dialog box

Once you have defined the Test language, you can fill the test in the language column. After that, you can build a localized version of the software based on the test language and run it.

Running the software with a test language lets you detect several problems such as:

- Overlapping strings.
- Non-translated strings caused by issues in internationalization.
- Memory allocation errors caused by strings expansion in size.
- Etc.

Cell highlighting

Cell highlighting is used to visualize changes in the string length. This feature helps to detect issues caused by strings that expand (C.f. Test languages). To toggle Cell highlighting on, choose **Tools | Options | General...** and the **Grid** tab.

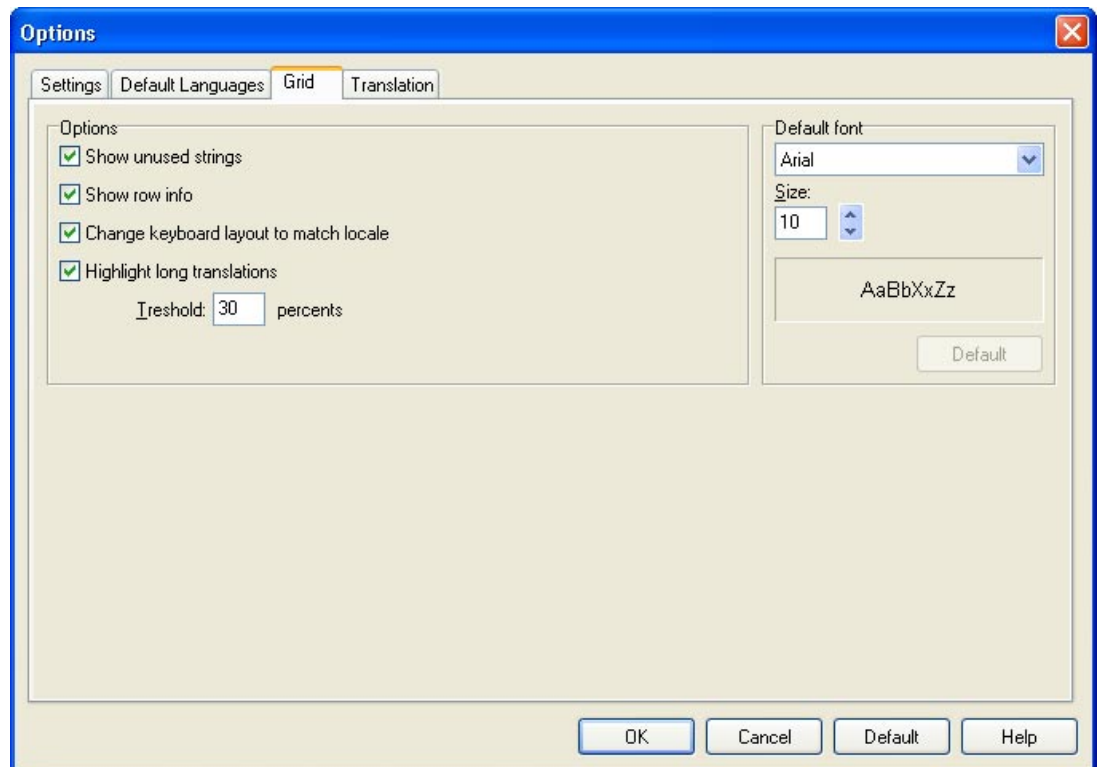


Figure 208 Project options dialog box

If you specify the Threshold value to 20 percent, cells will be displayed in blue if the translation becomes 20% longer or more than the native string. The bigger the differences, the darker shades of blue are used.

Component localization only.

Translator components

In component localization, you can specify the behavior of the software, if a translation fails due to missing translation.

You can either tag the native string at run-time, or throw an exception in the software on translation failure. Using this feature systematically you can reduce the time in locating internationalization issues in your software source.

28

Translation Memory on database server

General considerations

By default Multilizer Translation Memory (MTM) uses built-in single-user database. To enable more efficient use of it, MTM can be installed on database server.

Before switching MTM to server database, ensure the following:

- This feature requires *Multilizer for Enterprise* or *Multilizer for Oracle*. Ensure that you have the required licenses.
- Ensure that your RDBS is installed and configured properly. Refer to the RDBS documentation/vendor in this.
- Multilizer should not store translations in MTM when saving project (See Tools→Options→General, Translation tab).
- Set access levels according to users' tasks (See: Managing MTM rights, p. 218)

All examples of this document are shown for MS SQL Server.

You need to have Multilizer for Enterprise or Multilizer for Oracle in order to use this tool.



Database related tasks

Create new database

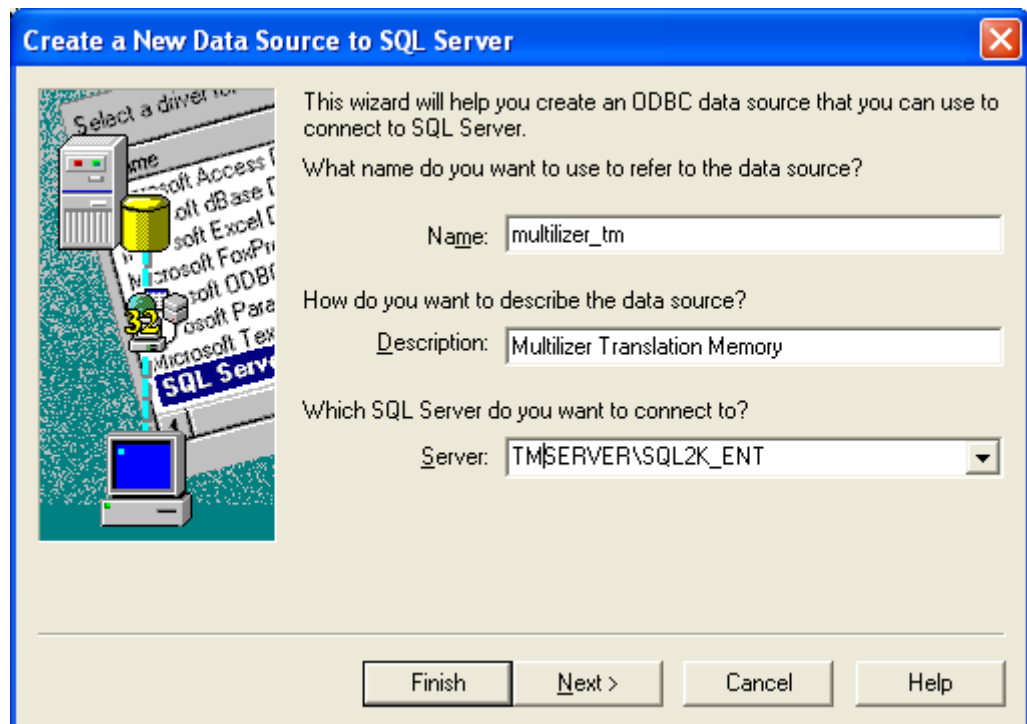
This is done with SQL Server Enterprise Management Tools.

Create a new database and add the permitted users in it. See 'Connecting from Multilizer to MTM' (p. 214) for required privileges.

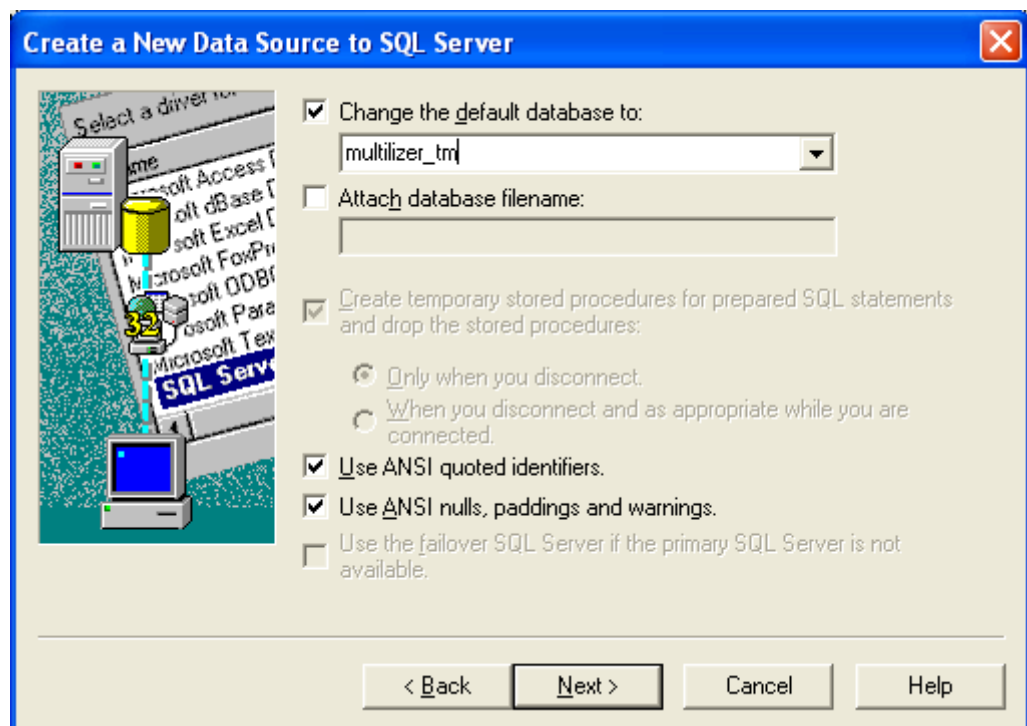
Define Connection parameters

Multilizer recommends that you create a DSN on the client computer.

Enter the connection parameters according to the SQL Server installation and database in use.



Set authentication method according to your company's security policy.



Connecting from Multilizer to MTM

To connect to a new MTM, choose in Multilizer Tools→New Translation Memory...

If the database is empty, Multilizer will create the required tables for MTM. Ensure that you have privileges to modify database metadata (create/alter tables, indexes, etc.). This user becomes 'owner' of the MTM, and can't be changed afterwards.

For later MTM use, user needs read and write access to the tables.

Translation Memory Wizard - Select ✖

Select the type of the translation memory

Translation Memory Type

Use the default translation memory database

Specify the translation memory database

Help Back Next Cancel

Translation Memory Wizard - Database ✖

Specify the translation memory database.

Any ADO/ODBC compatible

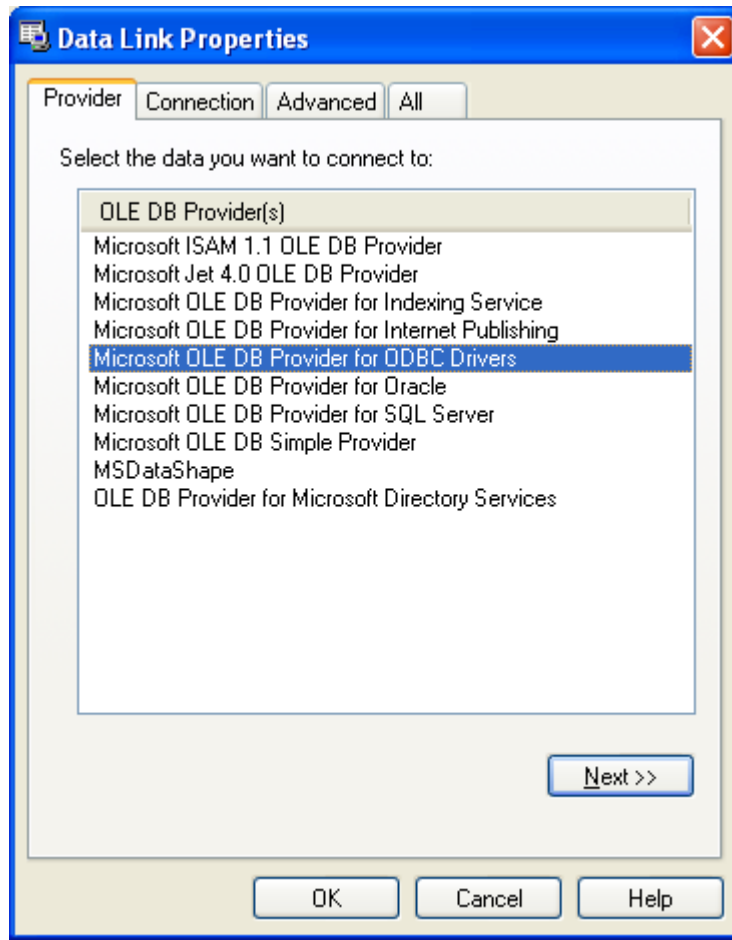
Connection string:

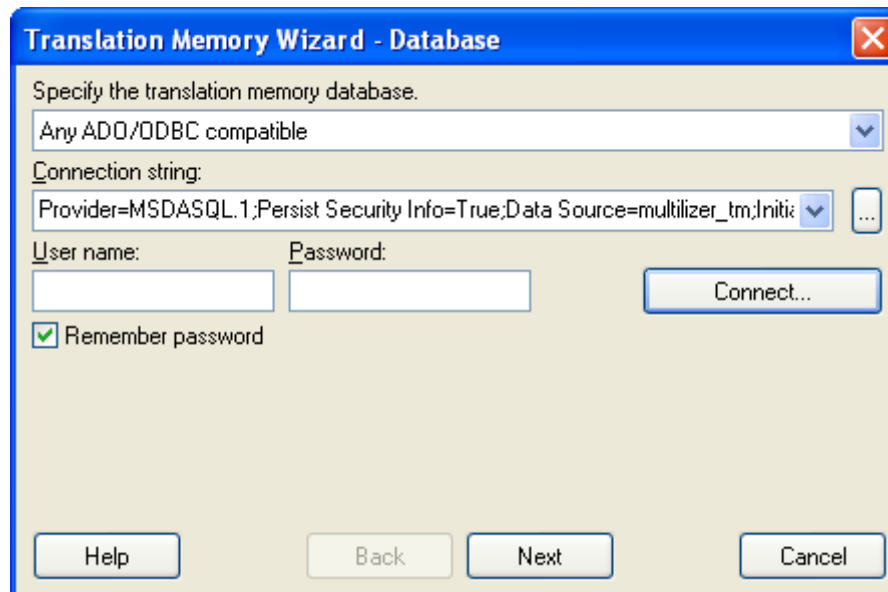
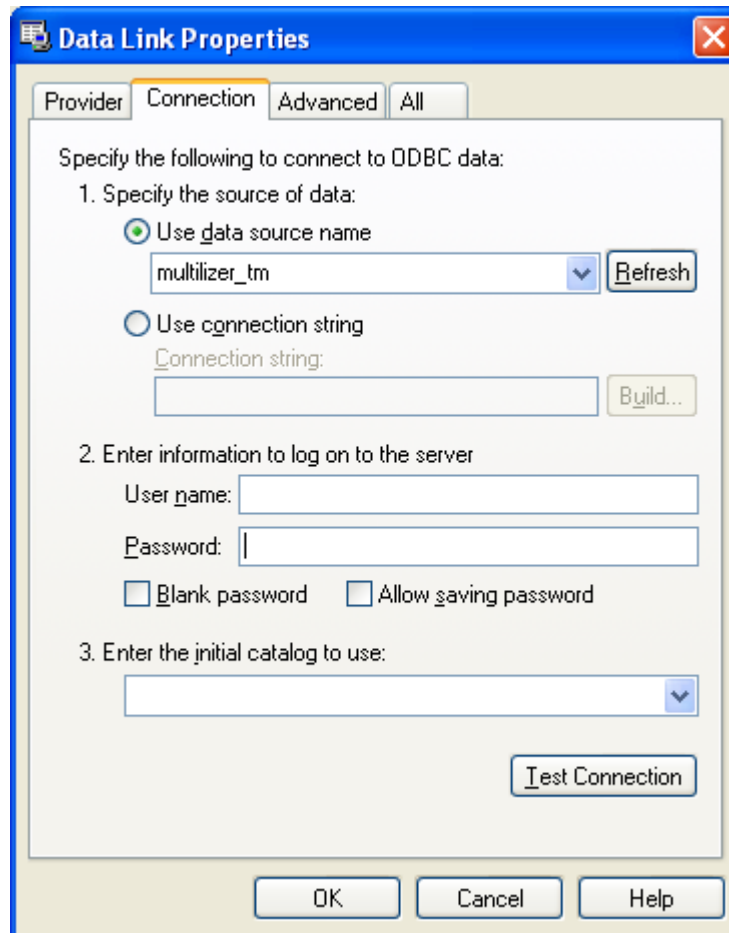
User name: Password:

Remember password

Connect...

Help Back Next Cancel





After specifying Connection string, press Connect... to test connection. If test succeeds, Next button becomes enabled. Press Next to proceed.

Enter your details in dialog.

Translation Memory Wizard - User

You can enter the default user name and the organization.

User name:
ML User

Organization:
My organization

Maximum string width that the translation memory can store:
200

Help Back Next Cancel

Managing MTM rights

Installing MTM on a server enables multiple users to concurrently work with it.

In order to control how MTM is used, it is highly recommend attaching different permissions to users according to their role.

To set access rights, connect to MTM with owner privileges. Information tab will show users and their access right. To change access right level, right click any user and select desired rights from popup-menu.

Translation Memory

Information Import Maintenance

Translation memory
ADO Compatible translation memory
Location: TMSERVER\SQL2K_ENT\multilizer_tm
Version: 1.2

User Name	Access Rights
guest	No access
MYDOMAIN\No Translator	No access
dbo	Owner
MYDOMAIN\Tim Translator	Translate access

Close Help

Translate Access

Typically Translators and QA Personnel are granted *Translate access*. This setting enables them to use MTM for translating projects.

Ensure that project translations are not saved in MTM automatically. (See Tools→Options→General, Translation tab).



Translate and add access – Full access

Localization Managers are granted *Translate and add access* or *Full access*. In addition to being able to use MTM for translation, they can add/update glossaries in it.

Full access enables removing of glossaries from MTM.

29

Builder Command Line Tool

It is common to have a build file that compiles the release application without debug information, creates the setup applications, etc. You cannot use the interactive Multilizer application in such a process. Fortunately Multilizer contains a command line tool called Builder. It provides a command line interface to most Multilizer functions such as scanning, building, exchanging, importing and exporting.

The syntax of the Builder tool is:

```
mlbuild commands project [-q] [-h]
```

commands One or more commands and their parameters. The available commands are:

- add Adds a new target to the project
- scan Scan the project
- remove Remove the unused strings from the project
- import Import a file to the project
- export Export data to a file
- exchange Create an exchange project
- translate Translate the project
- build Create the localized items and/or the dictionary
- dictionary Create the sub dictionary

project Multilizer project file (.mpr)

-q Quiet mode. Write no output except errors.

-h Show the detailed command specific help.

`mlbuild scan -h` shows the detailed help of the scan command.



NOTE!

You need to have Multilizer Enterprise edition or Multilizer for Oracle in order to use this tool.

Adding

You can add new targets to the project. The command equals to the **Project | Targets -> New** menus of the Multilizer application.

The syntax of the add command is:

```
mlbuild a[dd] fileName [-type:X] [-q] project
```

fileName	The target file to be added. The file name can contain wildcards (* and ?).
-type:X	The target type: <ul style="list-style-type: none">delphibin Delphi binary file (.exe, .dll)delphi16 16-bit Delphi project file (.dpr)delphi32 32-bit Delphi project file (.dpr)cbbin C++Builder binary file (.exe, .dll)cb32 C++Builder project file (.bpr)cpp C++ binary file (.exe, .dll)rc16 16-bit resource file (.rc)rc32 32-bit resource file (.rc)rcce Windows CE resource file (.rc)vbbin Visual Basic binary file (.exe, .dll)vb16 16-bit Visual Basic project file (.vbp)vb16ml 16-bit Visual Basic project file (.vbp) with Multilizer componentsvb32 32-bit Visual Basic project file (.vbp)vb32ml 32-bit Visual Basic project file (.vbp) with Multilizer componentsvbce Embedded Visual Basic project file (.ebp) If no type is specified MLBuild detects the target type.
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (.mpr)

Examples

The following example adds d:\MyProj\Project.exe target to the MyProject project file.

```
mlbuild add D:\MyProj\Project1.exe -type:delphibin MyProject.mpr
```

Scanning

You can scan the project to pick up the new strings by using the scan command. After scanning, MLBuild saves the new items to the project file. If no new items are found the project file is not touched. The command equals to the **Project | Scan + File | Save** or **Project | Smart Scan + File | Save** menus of the Multilizer application.

The syntax of the scan command is:

```
mlbuild s[can] [-smart] [-q] project
```

-smart	Scan only those targets that have been changed since the last scan.
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (.mpr)

Examples

The following example scans the targets belonging to the MyProject project file.

```
mlbuild scan MyProject.mpr
```

The following example equals to the above example but it does not write any output except errors.

```
mlbuild s -q MyProject.mpr
```

Removing

You can remove the unused string from the project by using the remove command. Selecting this command make the MLBuild to scan the project. If no unused items are found the project file is not touched. The command equals to the **Project | Remove Unused Strings + File | Save** menus of the Multilizer application.

The syntax of the remove command is:

```
mlbuild r[emove] [-q] project
```

-q Quiet mode. Write no output except errors.
projectfile Multilizer project file (.mpr)

Examples

The following example remove the unused strings from MyProject project file.

```
mlbuild r MyProject.mpr
```

The following example equals to the above example but it does not write any output expect errors.

```
mlbuild r -q MyProject.mpr
```

Importing

You can import a Multilizer project file, a text file or a TMX file to the project by using the import command. After importing, MLBuild saves new or changed items to the project file. If no new items are found the project file is not touched. The command equals to the **File | Import + File | Save** menus of the Multilizer application.

The syntax of the import command is:

```
mlbuild i[mport] file [-type:X] [-lang:X] [-separat:X] [-overwrite]
[-different] [-status] [-method:X] [-comment:X] [-columns:X] [-q] project
```

file	The file (.mpr, .txt, or .tmx) to be imported. The file name can contain wildcards (* and ?)
-type:X	Specifies the type of the import file. See the Add command -type to get the possible values. If no type is specified MLBuild detects the target type.
-lang:X	List of language codes to be imported. Separate multiple codes with semi colon (;). The code uses the following format: ll [_CC] ll is the two character ISO-639 language code (e.g. en) CC is the optional two character ISO-3166 country code (e.g. US)
-separat:X	ASCII hex value of the column separator character of the text file. Default value is 9 (tab).
-overwrite	Overwrite the current values
-different	Import the translation only if it differs from the native value
-status	Import the translation status
-method:X	Specifies how strings are imported: 0 By context 1 By native value 2 First by context then by native value (Default)
-comment:X	Specifies how comments are imported: 0 No comments are imported 1 Comment is imported if the current comment is empty (Default) 2 The imported comment overwrites the current one
-columns:X	List of text columns. If the imported file is a text file you must give the columns. Separate multiple columns with semi colon (;). Language columns use the same ISO code as the lang option. The following special column can be used: na Native column co Context column cm Comment column ig Ignore this column
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (.mpr)

Examples

The following example imports Translated.mpr to MyProject.mpr.

```
mlbuild import Translated.mpr MyProject.mpr
```

The following example imports only the Finnish column. Overwrite the current values.

```
mlbuild i Translated.mpr -lang:fi;se -overwrite MyProject.mpr
```

The following example imports all the text files in the current directory to MyProject.mpr. The text files must contain the native, Finnish and Swedish columns:

```
mlbuild i *.txt -columns:na;fi;se MyProject.mpr
```

The following example imports Translated.exe that is a Delphi binary to the German column of MyProject.mpr.

```
mlbuild import Translated.exe -type:delphibin -lang:de MyProject.mpr
```

Exporting

You can export data from the project file to a text file or TMX file by using the export command. The command equals to the **File | Export** menu of the Multilizer application.

The syntax of the export command is:

```
mlbuild ex[port] file [-format:X] [-lang:X] [-nocheck] [-nouncheck]
[-noempty] [-separat:X] [-context:X] [-quote:X] [-nosig] [-addcomment]
[-tmxver:X] [-tmxdtd:X] [-tmxcase:X] [-empty] [-nocontext] [-nocomment]
[-admlang:X] [-srclang:X] [-q] project
```

file	The file (.txt, or .tmx) to be exported.
-format:X	File format of the export file. 0 Ansi 1 UTF-8 (Default) 2 UTF-16, little endian 3 UTF-16, big endian
-lang:X	List of language codes to be exported. Separate multiple codes with semi colon (;). The code uses the following format: ll [_CC] ll is the two character ISO-639 language code (e.g. en) CC is the optional two character ISO-3166 country code (e.g. US)
-nocheck	Do not export translated and checked strings
-nouncheck	Do not export translated but unchecked strings
-noempty	Do not export untranslated strings
-separat:X	ASCII hex value of the column separator character of the text file. Default value is 9 (tab). Note! This option is used only when exporting to a text file.
-context:X	The position of the context column. 0 No context column (Default) 1 Context column is the first column 2 Context column is the second column after the native column 3 Context column is the last column Note! This option is used only when exporting to a text file.
-quote:X	The quotes that are used. 0 No quotes (Default) 1 Single quotes (') 2 Double quotes (") Note! This option is used only when exporting to a text file.
-nosig	No UTF-8 signature or UTF-16 byte order mark is written. Note! This option is used only when exporting to a text file.
-addcomment	Adds the comment column as the last column Note! This option is used only when exporting to a text file.
-tmxver:X	TMX version. The value can be from 1.0 to 1.4. Default value is 1.4. Note! This option is used only when exporting to a TMX file.

-tmx.dtd:X	DOCTYPE and DTD usage. 0 No DOCTYPE tag 1 DOCTYPE tag with locale DTD file name 2 DOCTYPE tag with DTD URL file name (Default) Note! This option is used only when exporting to a TMX file.
-tmx.case:X	The case of the lang-attribute. 0 Default case (e.g. en-US) (Default) 1 Lower case (e.g. en-us) 2 Upper case (e.g. EN-US) Note! This option is used only when exporting to a TMX file.
-empty	Enable writing of empty translations. Note! This option is used only when exporting to a TMX file.
-nocontext	Disable writing of context. Note! This option is used only when exporting to a TMX file.
-nocomment	Disable writing of comments. Note! This option is used only when exporting to a TMX file.
-admlang:X	The admin language. Default value is en. Note! This option is used only when exporting to a TMX file.
-srclang:X	The source language. Default value is *all*. Note! This option is used only when exporting to a TMX file.
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (.mpr)

Examples

The following example exports Translated.txt from MyProject.mpr.

```
mlbuild export Translated.txt MyProject.mpr
```

The following example exports the translated and checked strings of the Finnish column.

```
mlbuild ex Translated.tmx -tmxver:1.3 -lang:fi -nouncheck -noempty  
MyProject.mpr
```

Exchanging

You can exchange translations to the translator by using the exchange command. The command equals to the **File | Exchange** menu of the Multilizer application.

The syntax of the exchange command is:

```
mlbuild exc[hange] file [-lang:X] [-nocheck] [-nouncheck] [-noempty]
[-dup:X] [-new] [-appfiles] [-name:"s"] [-des:"s"] [-author:"s"] [-q]
project
```

file	The exchange file to be created. If the file extension is <code>.mlp</code> the exchange package will be a compressed ZIP file (having <code>.mlp</code> extension) that contains the project file and the optional application files. If the file extension is <code>.exe</code> the exchange package will be an EXE file that installs Multilizer and opens the project file.
-lang:X	List of language codes to be exported. Separate multiple codes with semi colon (;). The code uses the following format: <code>ll[_CC]</code> <code>ll</code> is the two character ISO-639 language code (e.g. <code>en</code>) <code>CC</code> is the optional two character ISO-3166 country code (e.g. <code>US</code>)
-nocheck	Do not exchange translated and checked strings
-nouncheck	Do not exchange translated but unchecked strings
-noempty	Do not exchange untranslated strings
-dup:X	Specifies how duplicate strings are exchanged. 0 Exchange every instance of the string having the same native value. 1 Exchange the first string plus every other string having the same native value and different comment or maximum length (in characters or pixels). 2 Exchange only the first string if several strings having the same native value exists.
-new	Exchange only the new string
-appfiles	Exchange the application executable files.
-name:"s"	s is a string that contains the package name.
-des:"s"	s is a string that contains the package description.
-author:"s"	s is a string that contains the author name.
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (<code>.mpr</code>)

Examples

The following example creates a setup package containing Multilizer application and all the languages and strings of the project.

```
mlbuild exchange Translate.exe MyProject.mpr
```

The following example creates a Finnish package containing only the un-translated items.

```
mlbuild exc Translate.mlp -lang:fi -nocheck -nouncheck MyProject.mpr
```

The following example creates a package containing the new strings. If the string is duplicate the first and those having different properties are exchanged.

```
mlbuild exchange Translate.mlp -new -dup:1 MyProject.mpr
```


Translating

You can translate the project by using the exchange command. The command equals to the **Project | Translate | Translation Memory** menu of the Multilizer application.

The syntax of the translate command is:

```
mlbuild t[ranslate] [-lang:X] [-first] [-q] project
```

- lang:X List of language columns to be translated. Separate multiple codes with semi colon (;).
 The code uses the following format: ll [_CC]
 ll is the two character ISO-639 language code (e.g. en)
 CC is the optional two character ISO-3166 country code (e.g. US)
- first Use the first translation when two or more translations exists. The default feature is to skip all such translations.
- q Quiet mode. Write no output except errors.
- projectfile Multilizer project file (.mpr)

Examples

The following example translates all the columns of the MyProject.mpr file.

```
mlbuild translate MyProject.mpr
```

The following example translates the English and German columns. Use the first translation.

```
mlbuild t -lang:en;de -first MyProject.mpr
```

Building

You can build the localized file and/or the dictionary by using the build command. The command equals to the **Project | Build Localized Items** and **Project | Make Localized Items** menus of the Multilizer application.

The syntax of the translate command is:

```
mlbuild [build] [-make] [-q] project
```

- make Make operation. Build only those file that are older than the project file.
- q Quiet mode. Write no output except errors.
- projectfile Multilizer project file (.mpr)

Examples

The following example builds the localized files

```
mlbuild build MyProject.mpr
```

The following examples make the localized files

```
mlbuild b -make MyProject.mpr
```

```
mlbuild -make MyProject.mpr
```

The following build file compiles the Delphi application and creates the localized files.

```
dcc32 -$D-L-Y- -U..\..\ -I..\..\ dcalc.dpr
```

```
mlbuild dcalc.mpr
```

Creating a Sub Dictionary

You can create a sub dictionary by using the exchange command. The command equals to the **Project | Create Sub Dictionary** menu of the Multilizer application.

The syntax of the translate command is:

```
mlbuild d[ictionary] file [-lang:X] [-q] project
```

file	The dictionary file (.mld) to be created.
-lang:X	List of language columns to be included. Separate multiple codes with semi colon (;). The code uses the following format: ll [_CC] ll is the two character ISO-639 language code (e.g. en) CC is the optional two character ISO-3166 country code (e.g. US)
-q	Quiet mode. Write no output except errors.
projectfile	Multilizer project file (.mpr)

Examples

The following example creates a Finnish dictionary.

```
mlbuild dictionary Finnish.mld -lang:fi MyProject.mpr
```

The following example creates a dictionary containing the Nordic languages.

```
mlbuild d Nordic.mld -lang:da;fi;fo;is;no;se;se_FI MyProject.mpr
```

Appendix A: Glossary



This glossary describes common terminology used in software localization and in Multilizer. Multilizer White Paper contains a more comprehensive glossary.

Code page

A code page is a code array that maps the integer code to the character of the character set. The first 128 items of every code page contains the ASCII characters. The remaining items depend on the character set.

Windows 3.1 supports only one code page. It is the code page of the system and it is bound on the language version of your operating system.

Windows 95 and *Windows 98* support multiple code pages but only one can be the system code page. It is bound on the language version of your operation system. It can not be changed.

Windows NT, *Windows 2000* and *Windows XP* support multiple code pages but only one is active at a time. The system must be rebooted after a code page change.

Java, *Windows CE*, *Symbian* use Unicode so there is no need to use code pages.

Dictionary

Multilizer uses dictionaries to store the translation data. Each Multilizer edition contains one or more dictionary components that access that data. In most cases all the translation data of your application is stored in one dictionary. A dictionary component is needed in component localization only.

Globalization

Globalization is the compound of tools and methods that are applied to software in order to make it work globally. Thus, the globalized software works with the appropriate features of each of the target countries.

Globalization can be seen as the sum of internationalization and localization: when globalizing the software, it first has to be internationalized and after that localized. Properly developed software targeted for many locales must go through both internationalization and localization.

Internationalization

Designing and building products so they can be easily adapted into various target languages and regions without requiring subsequent engineering changes. May often include multi-byte character enablement work so that the application can be localized into Asian languages. Also includes making an application aware of OS locale settings; adding flexible font settings for display of various languages; externalizing strings built into the application; and/or adding mechanisms to allow language-specific features to be implemented more easily.

Internationalization "builds-in" language interchangeability at the early product development stage. Internationalization is the successor to retrofitting-the old methodology of simple linguistic translation after the product is finished.

However, internationalization tends to be used in the meaning of globalizing software.

Language IDs

Language ID The language ID specifies the language (e.g. English, Finnish, Japanese, etc.).

Multilizer uses Windows's primary language IDs (e.g. LANG_ENGLISH), and Java's language codes as defined by ISO-639 (e.g. "en").

Country ID The country ID specifies the country of the language (e.g. English in United States, English in Great Britain, English in Canada, etc.).

Multilizer uses Windows's sub-language IDs (SUBLANG_ENGLISH_US), and Java's country code as defined by ISO-3166 (e.g. "US")

Linguist

The linguist is the human translator who translates the project into the target language(s).

Locale

A locale is a combination of language and country IDs. In other word it is a language spoken in a country (e.g. German spoken in Austria). There can be multiple locales for one language. For example, English (United Kingdom), English (United States), English (Canada), etc.

Localization

Localization is the act of applying country specific features to the program. The name is derivative of locale, which is an OS-specified set of items of the target country's denominative features.

Software localization aims to make the software reflect the target country's cultural features, in order to make the software customer-friendly.

Some countries need several localized versions of the software. For instance, in Canada there is a need to produce both English and French locale versions of the software.

Module

Module is a Multilizer component that translates the complex properties of one ore more 3rd party component. A Module component is needed in component localization only.

Translator

Translator is a Multilizer component that translates the form (window) from the native (original) language to the active language just before the form becomes visible. Translator uses the translation data provided by the Dictionary component. A Translator component is needed in component localization only.

Index

- .
- .NET, 1, 15, 23
- A**
- Arabic, 19, 81
- B**
- BiDi, 19
- Bi-directional, 19
- Binary, 56
- Build, 219
- Builder, 6, 209
- C**
- C#, 23
- C++Builder, 15, 56
- Canada, 21
- Character sets
 - bi-directional, 19
 - double byte, 19
 - MBCS, 19
 - multi byte, 19
 - Unicode, 20
- Chinese, 19
- CLDC, 98
- Code page, 221
- Comment, 120
- Component, 57
- Components, 7
- Conditional compiling, 125
- Cyrillic, 81
- D**
- Database, 15, 148
- DateTimeToStr, 61
- Dcalc
 - Palm, 127
- Delphi, 15, 56
- Deploying, 124
- Dictionary, 221
 - binary, 78
 - sub, 220
 - test, 71
- DLL, 65
- Documentation, 3
- E**
- Edit
 - Far Eastern languages, 21
 - Middle Eastern languages, 21
- Embedded Visual Basic, 16, 47
- Embedded Visual C++, 16, 34
- Emulator, 128
- English, 79, 83
- Enterprise, 1
- Exchange, 217
- Export, 215
- F**
- Far Eastern, 81
- Finland, 79
- Finnish, 79
- Format, 61, 63
- Forté, 83
- French, 21
- G**
- German, 19
- Globalization, 221
- Greek, 18
 - ancient, 18
 - modern, 19
- H**
- Hebrew, 19, 81
- I**
- I18N, 119, 129
- ID
 - country, 221
 - language, 221
- Idioms, 19
- IME, 21
- Import, 213
- Installation, 2
- Installation file, 124
- Internationalization, 5, 25, 37, 51, 58, 71, 100, 119, 129, 143, 221
- ISO, 175
- ITE, 70
- J**
- J2ME, 15, 98, 130
- J2SE, 98
- Japanese, 19
- Java, 1, 15, 83
- JBuilder, 83
- JDK, 83
- K**
- Kilometer, 61
- km/h, 61
- Korean, 19
- L**
- Lak, 19
- Languages
 - Far Eastern, 21
 - Middle Eastern, 21

- Western, 20
- Latin, 19
- Lezgi, 19
- Ligature, 19
- Linguist, 222
- Locale, 222
- Localization, 5, 222
 - binary, 8, 56
 - component, 12
 - component, 57
 - database, 12
 - document, 11
 - file, 8
 - resource, 10
 - source code, 10
- Localized application, 6

M

- Maximum length, 177
 - in characters, 120
 - in pixels, 120
- MBCS, 19
- Metric system, 63
- MIDP, 98
- Mile, 61
- Module, 222
- mph, 61
- Multi byte, 19
- Multilingual application, 6
- Multilizer, 6

O

- Oracle, 1, 175
- Oracle Forms, 15
- Overlay, 127, 133

P

- Package file, 124
- Palm, 16, 127
 - emulator, 128
 - internationalization, 129
 - overlay, 127
 - simulator, 128
- PowerJ, 83
- prc, 127
- Project
 - Creating a new, 184
- Project Wizard, 29, 43, 52, 66, 86, 102, 120, 130, 136, 143, 174

R

- Registration, 3
- Remove, 212
- Report, 173
- Resource bundle, 84
- Resource file, 35
- Resource files, 69
- Resource string, 58

S

- Satellite Assembly, 33
- Scan, 211
- SDK, 24
- Series 60, 119
- Simulator, 128
- SLS, 173
- StreamServe, 16, 173
- Sun ONE Studio, 83
- Support, 3
- Swing, 97
- Symbian, 16, 117
 - emulator, 118
 - internationalization, 119
 - preprocessor, 118
 - resource compiler, 118
 - SDK, 118
- System
 - registry, 66

T

- Tag
 - exclude, 120
 - include, 120
- Target, 210
- Translate, 218
- Translation memory, 14
- Translator, 71, 222
- Translator edition, 17

U

- Unicode, 20, 221
- USA, 61, 63
- UTF-8, 99

V

- VCL, 1, 56
- Visual Age, 83
- Visual Basic, 16, 47
- Visual Basic .NET, 23
- Visual C++, 16, 34
- Visual Café, 83
- Visual J++, 16, 106
- Visual Studio .NET, 15

W

- WAP, 16, 141
- Windows
 - 2000, 221
 - 95, 221
 - 98, 221
 - Arabic language edition, 21
 - Far Eastern language edition, 21
 - Far Eastern language editions, 21
 - NT, 221
 - XP, 221
- Windows CE, 34
- WML, 16
- WMLScript, 141

X

XML, 16, 135