

Contents

Resource Localization Manager



[Introduction](#)

[Usage](#)

[Localization Process](#)

[Data Files and Formats](#)

Introduction

In the past, localization of a software product required the localizer to edit strings and controls embedded in source code and then rebuild the product in order to test the localized version. Such a process requires at least a rudimentary knowledge of computer programming and is often prone to human error. The Resource Localization Manager (hereafter referred to as RLMAN) was designed to automate localization of products that make use of the Windows resource model by allowing the localizer to extract localizable resources directly from the applications that use them, modify the resources, and use the modified resources to create localized versions of the original applications. All this can be achieved without rebuilding the product and with minimal knowledge of computer operations.

RLMAN was designed with several goals in mind. Some of these goals were:

1. Allow the product to be localized without access to source code and without re-compilation.
2. Allow localization to proceed concurrent with development (provide update capability).
3. Allow localizers to share glossaries of common terms among applications.

The localization model followed by the RLMAN is very simple. Localizable resources are extracted from a source resource file and put into special text file called a token file. Each localizable resource may generate one or more tokens. Each token is contained on a single line of text and consists of a unique identifier followed by the localizable data associated with that particular resource. These tokens can then be localized by using a standard text editor.

The localized tokens are then used in conjunction with the source resource file to generate a new localized resource file. The term "resource file" in this document means "any Windows executable format file (.EXE, .DLL, .CPL, etc.) or .RES file". The target resource file will contain exactly the same resources as the source resource file, the only difference is that the data will be localized.

This model has been expanded a little to allow for update tracking. When localization is done in conjunction with development a target resource file may change after the localizer has tokenized the file and begun translation. Update tracking allows the localizer to update the localized token file without losing any work that might have been done. Any resources that may have changed since the most recent update are marked "dirty" and the change is tracked in the token file so the localizer may see exactly what changed and exactly how it changed.

To allow for update tracking, the source resource file is used to generate a "master token file" which tracks changes. The master token file is then used to update any number of "language token files" (one for each target language). These language token files are then localized and used to generate the target resource files.

Usage

Any file name can include a UNC (Unified Naming Convention) or a drive letter, and a directory path.

The syntax of the 'RLMan' command is as follows:

rlman [-c *RDFFile*] [-p *CodePage*] [-f *ResType*] [-{n|o} *PriLang* *SubLang*] [-w] -{e|t|m|l|r|a} *files*

Note: The -e, -t, -m, -l, -r and -a options (together with their *files* file names) are mutually exclusive and must be last on the command line.

-e *InputExeFile* *OutputResFile*

Extract localizable resources from the resource file (.exe, .dll, .cpl, .etc.) *InputExeFile* and create a Win32 resource file *OutputResFile*. This output file can then be used, for example, by the dialog box editor or the bit map editors that come with the SDK.

-t *InputResOrExeFile* *OutputTokenFile*

This option will extract the localizable resources from the executable or resource format file *InputResOrExeFile* and create a project token file *OutputTokenFile*. Using this option will circumvent the history-keeping mechanism of RLMan. It is made available for those times when the user wants to simply see what localizable resources are in the input file or when the history mechanism is not needed. Using the -o option and the -p option with the -t option will allow one to extract and tokenize resources of a specific language. The resulting token file will contain only those resources that have the specified locale and the text will be in the given code page.

-m *MasterProjectFile* [*InputResOrExeFile* *MasterTokenFile*]

When the history mechanism is wanted, the first step when creating a new master project or updating an existing master project is to use this option. If the *MasterProjectFile* does not exist, the optional *InputResOrExeFile* and *MasterTokenFile* arguments must be provided. These last two arguments will be ignored if the *MasterProjectFile* exists.

-l *LanguageProjectFile* [*MasterProjectFile* *LanguageTokenFile*]

This option is used, when the history mechanism is wanted, to create a new localization project or to update an existing project. If the *LanguageProjectFile* does not exist, the optional *MasterProjectFile* (the one created via the -m option) and *LanguageTokenFile* arguments must be provided. These last two arguments will be ignored if the *LanguageProjectFile* exists.

-r *InputResOrExeFile* *LanguageTokenFile* *OutputResOrExeFile*

This option is used to create a localized version of the *InputResOrExeFile*. The resources in that *InputResOrExeFile* will be replaced with the localized resources in *LanguageTokenFile*. *OutputResOrExeFile* is the localized version.

-a *InputResOrExeFile* *LanguageTokenFile* *OutputResOrExeFile*

This option is used to create a localized version of the *InputResOrExeFile*. The resources in the resulting *OutputResOrExeFile* will include the original resources from *InputResOrExeFile* plus the localized resources in *LanguageTokenFile*. Use the -n option to specify what the new language is.

-n *PriLang* *SubLang*

Specifies what the language the tokens in the token file are in and consequently what language the new resources are in. Used when setting up a new Language Project (-l) and with the -r and -a options. Can also be used with the -m option when the resources in the original resource file are not in U.S. English. *PriLang* *SubLang* are decimal values from the list of allowed values in the SDK.

-o *PriLangID* *SubID*

Specifies what the language the resources being replaced in, or extracted from, the source resource file are in.

Used with the **-r** and **-a** options when the resources being replaced, or added to, in the original resource file are not in U.S. English. *PriLang SubLang* are decimal values from the list of allowed values in the SDK. For example, if the U.S. English resources had been replaced with German resources and now you wanted to add French resources to that German file, use the **-o** option (with arguments **7 1**) to indicate that the original resources are in German not U.S. English together with the **-a** option. Or, if you had created a file with U.S. English plus German resources in it, and now you wanted to replace the German resources with French (thus making a U.S. English plus French file), you would use the **-r** option with the **-o** option (with arguments **7 1**) and the **-n** option (with arguments **12 1**) to indicate that the old German resources are to be replaced with the new ones in the given French token file.

Used with the **-t** option, this identifies the language of the resources that are to be tokenized. You should also use the **-p** option to specify the code page the text in the token file is to be written in.

-c *RDFFile*

Use custom resources defined in the resource description file *RDFFile*.

-p *CodePage*

The default code page used in converting between the Unicode resources and the text in the token files is the Windows ANSI code page. To change the code page, use this option and use the IBM code page number as the *CodePage* argument. For example; -p 932.

-f *ResourceType*

By default, all localizable resources are extracted. To extract a single resource type, use this option with *ResourceType* set to the resource type's numeric value (1-16 for Windows resources).

-w

Print warnings about unknown custom resource types (**-c** option not given or resource type is not in the *RDFFile*), and about resources that are not tokenized because their language is not the language requested (**-o** option, or US English by default). It will also warn of any zero-length resources found.

Localization Process

There are two basic types of localization. The first is when a product is correctly enabled for localization ("globalized"), the product development is finished, and all that is needed is to modify the localizable resources. We'll call this the "One-Shot Process". The second is when a product is being localized in parallel with the development process and the localization work is to be preserved across new builds of the original (typically English) file. We'll call this the "Parallel Process". It is done in parallel with product development so the localized versions are ready as soon as possible after the domestic version is.

One-Shot Process

To simplify the file names on the command line, change directories to the place where the localized files are to be kept. Leave the source executable (typically the English version) in some other directory anywhere on the network or on the local machine.

1. Create the project token file "prog.tok".

```
rlman -t prog.exe prog.tok
```

2. Translate the text in the "prog.tok" file with your favorite text editor. Assume German in this example.

3. If dialog boxes are to be resized:

Create the file "tmpprog.exe" which will contain the translated text.

```
rlman -n 7 1 -r prog.exe prog.tok tmpprog.exe
```

Create the .RES file, in German, needed by the dialog editor in the SDK.

```
rlman -e tmpprog.exe prog.res
```

Resize the dialog boxes as appropriate to account for the different lengths of the translated text.

```
dlgedit prog.res
```

Update the project token file with the revised dialog box coordinates and sizes.

```
rlman -t prog.res prog.tok
```

4. Create the final, localized, executable.

```
rlman -n 7 1 -r prog.exe prog.tok newprog.exe
```

This completes the process. The German file "newprog.exe" is now ready to be tested.

See also [Parallel Process](#)

[Languages SupportedBy Windows NT](#)

Parallel Process

This process maintains the localization work from one build of the source executable (typically the English version) to the next. With this version of RLMan there is one caveat if the developers change the ID number of a localizable item, the previous translation will be lost. This is being addressed and a solution will be available in a future release of RLMan. New items can be added or old ones deleted but an item with a changed ID will show up as a new item.

See also [Project Creation](#)
[Maintaining the Master Project](#)
[Maintaining Each Locale Project](#)

and [One-Shot Process](#)

Project Creation

1. Create a directory for the master files and a separate directory for the project files. These directories may be anywhere on the net. The master project directory may contain any number of master projects, one need not create a new directory for each project as long as the base name for each master project is unique. There should be a separate directory for each localized version of the executable file. Typically this means one directory for each language.

2. Move to the master directory.

3. Copy the source executable to the master directory.

4. Create the master project file and the master token file.

```
rlman -m prog.mpj prog.exe prog.mtk
```

5. Move to the project directory (for German in this example).

6. Create the project file and the project token file.

```
rlman -n 7 1 -l prog.prj prog.mpj prog.tok
```

7. Steps 5 and 6 need to be repeated for each project directory (language). The resulting **prog.tok** files can then be translated to the appropriate language.

See also [Maintaining the Master Project](#)
[Languages Supported By Windows NT](#)

Maintaining the Master Project

1. Copy the newly built source executable to the master directory.
2. Move to the master directory, then update the master project file and the master token file.

```
rlman -m prog.mpj
```

See also [Project Creation](#)

Maintaining Each Locale Project

1. Follow steps 2 and 3 of the "one-shot process" (previous page) as often as desired until the resources are localized satisfactorily.
2. Every time the master project is been updated (step 2 in the "Maintaining the Master Project" section), update the project file and project token file.

```
rlman -l prog.prj
```

Repeat step 1 as needed to catch new or changed resource items.

See also [Maintaining the Master Project](#)
[Project Creation](#)

Data Files and Formats

RLMan uses a variety of special file types. All of the file formats described below are a special form of text file. Each file is human-readable and can be edited with any standard text file editor (such as Notepad).

As a general rule, all text in these files follows the C escape convention when dealing with non-displayable characters. This convention uses escape characters to represent non-displayable characters. For example, \n is newline and \t is tab.

Master Project Files (MPJ)

Project Files (PRJ)

Master Token Files (MTK)

Language Token Files (TOK)

Resource Description Files (RDF)

Master Project Files (MPJ)

Master project files consist of seven lines of text:

1. The first line contains the path to the source resource file. This may be either a .RES file, or an .EXE format file.
2. The second line contains the path to the master token file (MTK).
3. The third line contains zero, one or more paths to resource description files (RDFs) separated by spaces.
4. The fourth line contains a date stamp indicating the date of the source resource file as of the last update.
5. The fifth line contains a date stamp indicating the date of the master token file as of the last update of the master project.
6. The sixth line contains the primary- and sub-language components of the Language ID of the resources in the master token file (.MTK).
7. The seventh line contains the code page used when reading/writing the master token file. A zero (0) means the system's Windows ANSI code page. A one (1) means the syetem's default OEM code page. Other values are actual code page numbers.

See also [Project Files \(PRJ\)](#)

Project Files (PRJ)

Project files consist of seven lines of text:

1. The first line contains the path to the master project file (MPJ).
2. The second line contains the path to the language token file (TOK).
3. The third line contains the path to the target resource file. This may be either a .RES file or (if the source resource in the MTK is an .EXE file) an .EXE format file.
4. The fourth line contains a date stamp indicating the date of the master token file (MTK) as of the last update of the project.
5. The fifth line may be left blank or it may contain the path to a glossary file. (Not used in this release.)
6. The sixth line contains the code page used when reading/writing the project token file. A zero (0) means the system's Windows ANSI code page. A one (1) means the syetem's default OEM code page. Other values are actual code page numbers.
7. The seventh line contains the primary- and sub-language components of the Language ID of the resources in the language token file (.TOK).

See also Master Project Files (MPJ)

Master Token Files (MTK)

Master token files are text files which contain tokenized resources taken from some source resource file. Each token consists of a unique identifier followed by the text form of the resource data. Tokens are delimited by end-of-line characters.

Master token files are used for update tracking. They contain no localized resource data and should not be changed except by RLMan.

An example of what one token might look like is shown below:

`[[5|255|1|32|5|"FOO"]]=Localizable string containing text in C format.`

The token ID is surrounded by double square brackets and divided into 6 fields delimited by the vertical pipe '|' symbol:

1. The first field indicates the type of the resource
2. The second field is the resource name in the case of an enumerated resource, or it is 65535 if the name is a label (string) in which case the label itself is stored in the sixth field.
3. The third field is the internal resource id number taken from the resource header.
4. The fourth field is made up of a combination of data taken from the resource header and generated by the tools. This value is used in conjunction with the other values in the token ID to uniquely identify the resource.
5. The fifth field is a status field used by the update tools to determine the status of the current token.
6. The sixth field contains the name of the resource if the resource is identified by a label. Otherwise it contains a null string.

A token ID is followed by an equal sign which is in turn followed by the resource data. The data extends from after the equal sign to the end of the line (exclusive). Non printing characters (such as new-line and control characters) are represented using C escape sequences. Two of the most common are `\n` for new-line and `\t` for tab. Some characters are shown in the form `'\nnn'` where `nnn` is the decimal value of that character.

A token's status field is made up of combinations (bitwise OR'ing) of three basic flags:

| | | |
|-----------------|----------|---|
| CHANGED | 4 | Indicates that the token has changed since the last update |
| READONLY | 2 | Indicates that the token should not be localized. |
| NEW | 1 | Used in conjunction with the CHANGED flag to indicate that this is the new version of the token. |

For example, if a token has changed during an update, the current text would be stored in a token with a status of **CHANGED+NEW** ($4 + 1$) = 5. The previous text is also stored in the token file using the same token ID but the status field would contain a 4 (**CHANGED**). This way both the current and the previous text are retained.

See also [Language Token Files \(TOK\)](#)

Language Token Files (TOK)

Language token files are similar to master token files; the only difference being the meaning of the status fields found in the token identifiers.

Language token files are used during localization. They contain localized resource data.

A token's status field is made up by combinations of four flags:

| | | |
|-------------------|----------|--|
| TRANSLATED | 4 | Indicates that the token contains text that should be put in the target resource. If a token is not marked as TRANSLATED then it contains unlocalized text from the master token file which is maintained for update tracking purposes. |
| READONLY | 2 | Indicates that the token should not be localized. |
| NEW | 1 | Used only for tokens that are not marked with the TRANSLATED flag to indicate that this is the new version of the unlocalized token. |
| DIRTY | 1 | Used only for tokens that are marked with the TRANSLATED flag to indicate that the token is in need of attention (either the original translation has changed or the token has never been localized). |

For example, a clean, localized token is marked only with the **TRANSLATED** flag and therefore has a status value of 4.

As in the Master token files, non printing characters (such as new-line and control characters) are represented (and entered by the localizer) using C escape sequences. Two of the most common are `\n` for new-line and `\t` for tab. Some characters are shown or entered in the form `'\nnn'` where `nnn` is the decimal value of that character. The localizer may enter any character it's `'\nnn'` form.

See also [Master Token Files \(MTK\)](#)

Resource Description Files (RDF)

Custom resources are described in resource description files (RDFs) using c-like structure definitions. Each definition is identified with a specific resource type and the definition is applied to every resource of that given type.

An identifier is declared by the following syntax:

<type>

Types are numbers or quoted names unless they are normal windows types in which case the standard Windows type name may be used in place of a number or name. (CURSOR, BITMAP, ICON, MENU, DIALOG, STRING, FONTDIR, FONT, ACCELERATORS, RCDATA, ERRTABLE, GROUP_CURSOR, GROUP_ICON, NAMETABLE, and VERSION).

A structure definition follows normal 'C' syntax with the following limitations and differences:

1. Each definition must be fully enclosed in braces { }.
2. The standard 'C' types: char (single-byte OEM characters), int, float, long, short, unsigned, and near and far pointers are accepted. Additionally, the types wchar (Unicode character) and tchar (Unicode in the NT version, OEM otherwise) are accepted. (Labels and macros are not legal.)
3. Nested structures, arrays and arrays of structures are legal. All arrays must have a fixed count except for strings which are described below. int[10] is legal int[] is not.
4. Null terminated strings (sz's) are the only variable length structures allowed. They are represented as an array of characters with no length: char[]
5. Fixed length strings are represented as arrays of characters with a fixed length: char[10]
6. Comments may be included in the file using standard c comment delimiters (/ * */ and //) or by placing them after the pound symbol #.
7. Localizable types (types that need to be placed in token files) are indicated by all caps. Hence INT would generate a token while int would not.

See also [Sample RDF File](#)

Sample RDF File

This is a sample Resource Description File

```
<"type">
{
    int,          // no token will be generated for this integer
    CHAR          // this single-byte character will be placed in a token
}
<RCDATA>
{
    WCHAR[]       // a null terminated Unicode string that requires a token
    wchar[]       // no token will be generated for this Unicode string
}
<1000>
{
    TCHAR[],      // a null terminated Unicode or OEM string that requires
                  // a token (Unicode if running NT version, else OEM).
    {
        int,
        FLOAT,    // localizable floating point value
        far *,
        CHAR[20]  // localizable 20 character single-byte string
    }[3],         // an array of three structures (NOT IMPLEMENTED YET)
    int
}
END              // Optional
```

See also [Resource Description Files \(RDF\)](#)

Localization

Throughout this document, the term **localization** refers to the process of preparing a product for an international market. This process involves (among other things) translating text and resizing controls such as dialogs and buttons. A person performing localization is referred to as a **localizer**.

Languages Supported by Windows NT

| Primary Language | ID's | Sub language | ID's |
|------------------|------|----------------------------|------|
| Neutral | 0x00 | Neutral | 0x00 |
| Albanian | 0x1c | Default | 0x01 |
| Arabic | 0x01 | System Default | 0x02 |
| Bahasa | 0x21 | Arabic (Saudia Arabia) | 0x04 |
| Bulgarian | 0x02 | Arabic (Iraq) | 0x08 |
| Byelorussian | 0x23 | Arabic (Egypt) | 0x0C |
| Catalan | 0x03 | Arabic (Libya) | 0x10 |
| Chinese | 0x04 | Arabic (Algeria) | 0x14 |
| Czech | 0x05 | Arabic (Morocco) | 0x18 |
| Danish | 0x06 | Arabic (Tuinisa) | 0x1C |
| Dutch | 0x13 | Arabic (Oman) | 0x20 |
| English | 0x09 | Arabic (Yemen) | 0x24 |
| Estonian | 0x25 | Arabic (Syria) | 0x28 |
| Farsi | 0x29 | Arabic (Jordan) | 0x2C |
| Finnish | 0x0b | Arabic (Lebanon) | 0x30 |
| French | 0x0c | Arabic (Kuwait) | 0x34 |
| German | 0x07 | Arabic (U.A.E.) | 0x38 |
| Greek | 0x08 | Arabic (Bahrain) | 0x3C |
| Hebrew | 0x0d | Arabic (Qatar) | 0x40 |
| Hungarian | 0x0e | Chinese (Traditional) | 0x01 |
| Icelandic | 0x0f | Chinese (Simplified) | 0x02 |
| Italian | 0x10 | Chinese (Taiwan) | 0x04 |
| Japanese | 0x11 | Chinese (PRC) | 0x08 |
| Kampuchean | 0x2c | Chinese (Hong Kong) | 0x0C |
| Korean | 0x12 | Chinese (Singapore) | 0x10 |
| Laotian | 0x2b | Dutch | 0x01 |
| Latvian | 0x26 | Dutch (Belgian) | 0x02 |
| Lithuanian | 0x27 | English (US) | 0x01 |
| Maori | 0x28 | English (UK) | 0x02 |
| Norwegian | 0x14 | English (Australian) | 0x03 |
| Polish | 0x15 | English (Canadian) | 0x04 |
| Portuguese | 0x16 | English (New Zealand) | 0x05 |
| Rhaeto Roman | 0x17 | English (Ireland) | 0x06 |
| Romanian | 0x18 | French | 0x01 |
| Russian | 0x19 | French (Belgian) | 0x02 |
| Serbo Croatian | 0x1a | French (Canadian) | 0x03 |
| Slovak | 0x1b | French (Swiss) | 0x04 |
| Spanish | 0x0a | German | 0x01 |
| Swedish | 0x1d | German (Swiss) | 0x02 |
| Thai | 0x1e | German (Austrian) | 0x03 |
| Turkish | 0x1f | Hebrew (Israel) | 0x04 |
| Ukrainian | 0x22 | Italian | 0x01 |
| Urdu | 0x20 | Italian (Swiss) | 0x02 |
| Vietnamese | 0x2a | Japanese (Japan) | 0x04 |
| | | Korean (Korea) | 0x04 |
| | | Norwegian (Bokmal) | 0x01 |
| | | Norwegian (Nynorsk) | 0x02 |
| | | Portuguese (Brazilian) | 0x01 |
| | | Portuguese | 0x02 |
| | | Serbo Croatian (Latin) | 0x01 |
| | | Serbo Croatian (Cyrillic) | 0x02 |
| | | Spanish (Traditional Sort) | 0x01 |
| | | Spanish (Mexican) | 0x02 |
| | | Spanish (Modern Sort) | 0x03 |
| | | Thai (Thailand) | 0x04 |

