

The “Make” Process in MikroPascal

2012-12-01

1 Contents

2	Purpose of this document	2
3	Definitions	2
4	Global Overview of the Development Process.....	3
4.1	Project activities	3
4.2	Global activities	4
5	Project Components.....	4
5.1	Code components	4
5.1.1	Source Files.....	4
5.1.2	Compiled unit files.....	5
5.1.3	Libraries	5
5.2	Other components	5
5.2.1	The Project file.....	5
5.2.2	The Config file.....	6
5.2.3	Project Level Defines	6
6	Public and Private declarations	6
6.1	In Units	6
6.2	In the Program (main) file	6
7	Search Paths	6
7.1	Standard Paths	7
7.2	Additional Paths	8
8	Uses Clauses	8
9	External References.....	9
10	Command Line Compiler/Linker (CLC)	10
11	Project Manager	11
11.1	Sources	11
11.2	Binaries	12
11.3	Project Level Defines	12
12	MLK File	12

13	Library Manager	14
14	Package Manager	15
15	Libraries	17

2 Purpose of this document

The purpose of this document is an attempt to let shine some light on the “make” process used by the mikroElektronika tools, especially the mikroPascal for PIC development environment. At first sight this bundle of tools and methods seems to be very complex, but after a while one gets to know the way it all works together (*or thinks to know...*).

So, most probably this document is not (yet) complete and possibly inaccurate and/or incorrect because some of the information below is “second hand” information: it does not come from mE, it is “observed”. If so, please accept my apologies.

If the reader should see some of the above mentioned (or other) problems, please contact the author, so that the problems can be corrected.

Anyway, I hope the document is not too messy and provides some insight and help.

Thanks in advance.

Dany

3 Definitions

Some expressions/words and their meaning in this document:

Expression	Meaning
Program File or <i>Main File</i>	The main mP file of the project: the one containing “Program <name>”. ¹
Project File	The file with the <code>.mpppj</code> extension, contains project data. ¹
Config file or <i>.cfg file</i>	The file with the <code>.cfg</code> extension, contains the PIC’s configuration. ¹
Source file or <i>.mpas file</i>	A file with the extension <code>.mpas</code> , containing pascal code, is either the program file or a unit.
Unit	Is a source file that has as 1 st line “Unit <name>”, containing pascal code.
Compiled unit or <i>.mcl file</i>	A file with the extension <code>.mcl</code> : is the output after compiling a Unit or the Program File .
Library	A compiled unit is present in the Library Manager .
CLC	The “Command Line Compiler”, started up from within the IDE with “Build”.
<i>.hex file</i>	The final file containing all code of the project in human readable format. Is the main input for the programmer.

¹ There is only one in every project.

Expression	Meaning
IDE	The “Integrated Development Environment”: editor, project manager, library manager, compiler starting, programmer starting, debugger, in circuit debugger, other tools starting, etc...
Project Manager	Part of the IDE, holding all filenames of project sources and other files, except those of libraries.
Library Manager	Part of the IDE, showing all libraries available for the current PIC and indicating which are used by the project.
Programmer	Stand alone Device/software that accepts a hex file and will program the pic (“flash” the code). Can be started up from within the IDE
Package Manager	Stand alone tool, used for handling “Packages” files (group of Libraries, extension: “.mpkg”). Also capable of installing packages in the .mlk files of processor’s.
MLK File	An XML file, processor type related, with extension “.mlk”, holding a.o. the library entries installed for a certain type of processor. To be found in the “Defs” directory.

4 Global Overview of the Development Process

4.1 Project activities

- Creating a new project with the IDE. The IDE will automatically enter the Program File name in the [Project Manager](#), and create a search path to its location (see [Search Paths](#)).
- Fill the project file with declarations and code (main routine, other routines).
- Create zero or more units, fill them with code and add their names into the [Project Manager](#) manually. The latter will create [Search Paths](#) to the units.
- Optionally: add units that were already existing (e.g. hardware driver units) to the [Project Manager](#). Again the latter will create the [Search Paths](#) to those units.
- Add zero or more [Uses Clauses](#) into the [Program file](#) or in units to enable them to use other units.
- “Check” zero or more libraries to use in the [Library Manager](#). The latter will add their path to the [Search Paths](#). The public part of all checked libraries will be visible to all source files in the project (as if all units had a “uses <library>” clause).
- “Build” the project in the IDE. This will start up the [Command Line Compiler](#) (CLC), which also does the linking activity. In its command line the [CLC](#) receives from the IDE
 - the [Search Paths](#) (see [Note1](#)),
 - the “.mcl” (compiled unit) filenames of the “checked” libraries in the [Library Manager](#)
 - the “.mpas” (source) filenames of the mP sources entered in the [Project Manager](#).
- The [CLC](#) will compile all [source files that need compilation](#) (i.e. the .mcl file is older than the .mpas file or the .mcl file does not exist or “always build all files” is checked in the IDE’s tools output), generating an .mcl file (compiled unit).
- If in the above step some units prove to contain [Uses Clauses](#), then the “used units” are compiled first.
- After all compilation of source files (.mpas) is done, the linking process starts.

- First the remaining “unresolved references” (references to e.g. library routines, or “external” references) are to be resolved (found and used). The attempt is made to solve those references by examining the checked libraries (the .mcl file parameters in the [CLC](#)’s command line) and the generated .mcl files out of the source file compilations. If some references can not be resolved (found and used) then an error is generated.
- The linker in the [CLC](#) will then links/relocates all (used) code together (give it a place in the PIC’s memory map) and generates a list file (.lst extension) and the .hex file (binary code in readable hexadecimal format for programming purposes). The latter is the final output of the compiling/linking process and is the main input for the programmer.
- After compiling/linking is done the programmer can also be started up from the IDE. It passes the .hex file and also here the needed parameters to e.g. the mikroProg Suite.

Note1: The [CLC](#) uses the [search paths](#) sequentially to find every file that is defined in its command line. If a file can not be found in one of the search paths an error is generated.

4.2 Global activities

- Optionally, the user can make a container of general purpose source files that can be used in different projects. Those could be called also “libraries”, but in this article they are not, because the term “Library” here is used only for those appearing in the [Library Manager](#).
- Additionally, the user can install libraries (or packages of libraries) for a number of processors. These libraries can, after installing them, used by the projects. See the [Library Manager](#) and the [Package Manager](#).

5 Project Components

A project consists of a number of components (files) of which some contain code (source or compiled) and others contain properties of the project (e.g. configuration, used libraries etc...) .

5.1 Code components

5.1.1 Source Files

All source files are of type ‘.mpas’ and are either

- the [main file](#) (the “[Program](#)” file) (the one containing “Program <name>”), or
- a unit (containing “Unit <name>”).

As you know a project can only contain 1 main file and zero or more units.

The name of the main file is the same one you created the project with.

Normally the source files of your project are present in the [Project Manager](#), section “Sources”.

5.1.2 Compiled unit files

These are files with the extension `‘.mcl’`.

Your project can also need units of which the source is not available. Normally this type of units are the so-called “libraries”, see below.

In some cases however compiled units are no “libraries” as defined below (they are not present in the [Library Manager](#). In this case the `.mcl` files your project uses should be present in [Project Manager](#), section “Binaries”.

5.1.3 Libraries

The definition for “Library” used here is: present in the [Library Manager](#).

This means they were installed with the [Package Manager](#) (or entered in the [MLK File](#) manually or with an other tool).

There are two kinds of libraries:

- The ones delivered with the compiler: they are visible in the [Library Manager](#) under the package name “mikroE”,
- Those that are installed with the [Package Manager](#): they are visible in the [Library Manager](#) under their respective package name.

Libraries do have at least an `‘.mcl’` file available, in some cases the `‘.mpas’` source file is available too.

5.2 Other components

5.2.1 The Project file

The extension of this file is:

- `‘.mpppi’` for mP PIC projects,
- `‘.mppds’` for mP dsPIC and PIC24 projects,
- `‘.mpp32’`, for mP PIC32 projects,
- `‘.mbppi’`, for mB PIC projects,
- `‘.mbpds’`, for mB dsPIC&PIC24 projects
- `‘.mbp32’`, for mB PIC32 projects
- `‘.mcpqi’`, for mC PIC projects
- `‘.mcpds’`, for mC dsPIC&PIC24 projects
- `‘.mcp32’` for mC PIC32 projects

It contains some important information about the project:

- The name of the main file,
- The name of other sources of the project,
- The libraries used,

- The Search Paths
- etc...

It can be found in the same directory as the main source file.

5.2.2 The Config file

This file has the extension '.cfg' and contains (as the title suggests) the "configuration" of the PIC:

- The settings for the configuration words
- The type of PIC
- The MCU clock speed of the PIC

It can be found in the same directory as the main source file.

5.2.3 Project Level Defines

Project level defines contain no code, only "defines" visible for every source file in the project. See Project Level Defines for more details.

6 Public and Private declarations

6.1 In Units



As you probably know, all declarations above the "implementation" statement in a unit are "public", that is: they are known to other units and to the [main file](#) provided they have a "uses" clause for the unit(s) used. The part below "implementation" is private, only accessible from within the unit itself.

6.2 In the Program (main) file



Also everything, except type definitions, of the [main file](#) is public, provided it is above the main "begin.. end." statement. Since other units in the project can not "use" the main file, the only way to use the public items of the [main file](#) in another unit is through "external" definitions. (see [External References](#)).

7 Search Paths



To bring the compilation and linking process to a good result all files mentioned above must be available to the compiler/linker (the IDE and the compiler started up from within), see section [Command Line Compiler](#).

The system to achieve this is this:

- A number of “Search Paths” are available, in which files are searched,
- The [names²](#) of the source (and/or binary) [files](#) and the names of the libraries are available,

As said before, those “Search Paths” are to be found in [The Project file](#). The search paths can be viewed in the IDE with menu “Project -> Edit Search Paths”. Here one can add or delete search paths manually.



The order of the search paths in the list is important: the paths first mentioned will be searched first! In doubt, always check the outcome of the compiler/linker (IDE’s “messages”) about the version of a file that was used if it is present in more than one location.

Search paths are added to the project file in a number of cases automatically:

- The [Standard Paths](#),
- When adding a file to the [Project Manager](#),
- when selecting a library in the [Library Manager](#).



Removing a file from the [Project Manager](#) or de-selecting a library in the [Library Manager](#) does not remove its corresponding path from the search path list!



³You can specify either absolute or relative path to the source files. If you specify a relative path, mikroPascal PRO for PIC will look for the file in following locations, in this particular order:

- the project folder (folder which contains the project file .mpppi),
- your custom search paths,
- mikroPascal PRO for PIC installation folder > Uses folder.

7.1 Standard Paths

A number of paths are always added to the Search Path list:

- The path to the “[Defs](#)” directory, where the [MLK File](#) can be found of the PIC used in the project, e.g. “... \mikroPascal PRO for PIC\Defs”. See (1) in Figure 1.
- The path to the [standard mE libraries](#) for the PIC used in the project, e.g. “... \mikroPascal PRO for PIC\Uses”. See (2) in Figure 1.
- The path to the project files itself (where the [main file](#), the project file and the config file reside). See (3) in Figure 1.

² Only the names of the files, not their full path.

³ Extract from the mP help file.

7.2 Additional Paths

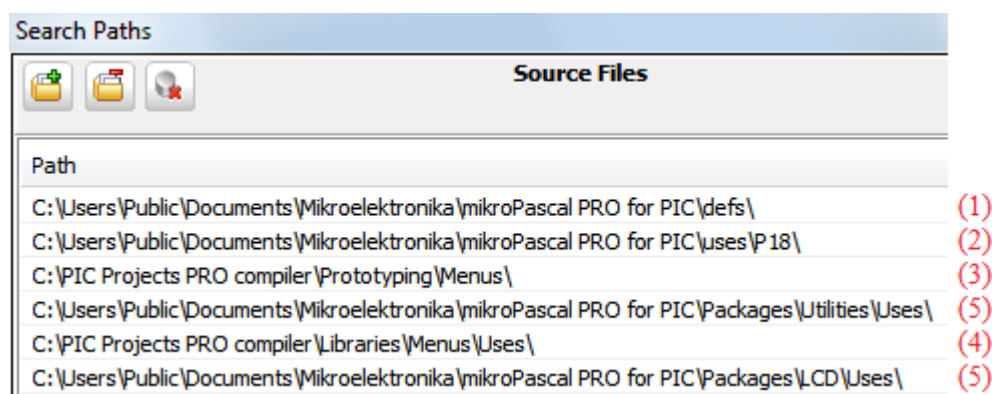
Besides the “standard” path above some additional paths can be available:

- Paths for files, present in the project manager, which are in another location than the [main file](#) (e.g. a container of project independent source files or .mcl files). See (4) in Figure 1.
- Paths to libraries which are not standard, e.g. the ones installed by the package manager.

This type of path looks like “... \mikroPascal PRO for PIC \Packages \<package name> \Uses”. See (5) in Figure 1.

An example:

Figure 1



8 Uses Clauses

A uses clause defines an other unit that is to be used by the one containing the clause. The latter has full access to the “public” part of the “used” unit: type definitions, variables, constants and routine prototypes.

Example (current unit wants to use 3 others):

```
Uses Unit1, Unit1, Unit3; // all “useses” should be in 1 statement
```



- [Uses Clauses](#) in the [main file](#) to files present in the [Project Manager](#)’s sources are not necessary. Apparently all units present there are accessible to the [main file](#). It is however not forbidden (and even recommended) to add also uses clauses in the Program File.

In all other units the “uses” clause is obligatory when another unit is to be used.

- Normally only units in the “Sources” part of the Project Manager are referenced to by “Uses” clauses. However, it is also possible to “use” libraries or other compiled units. The unit name in the uses clause has never an extension, so it does not matter. “Using” a library is the same

as “checking” it in the [Library Manager](#). Still: it is recommended not to refer to libraries in “uses” clauses but to use the [Library Manager](#) to add a library to the project. One remark though: if you define a library in a “uses” clause, then always the native filename (without extension) should be given, not the “alias”, see the [MLK File](#) and the [Library Manager](#) for details.

- The [Program File](#) can not be “used”, but see next section [External References](#).

9 External References

As said above, the public part of the [Program file](#) can not be accessed by another unit by means of a “uses” clause (one can not “use” the [program file](#)). Also another usage of the “uses” clause is forbidden: the “recursive” or “circular” usage.

In both cases above the problem can be solved by means of the usage of external declarations. One advantage of external declarations is that the using unit does not have to know the name of the unit where the referenced items are actually declared, one disadvantage is that an external definition is required for every single items to be accessed (as far as needed) in the accessed unit.

Example:

In the file where they are actually declared:

```
var Variable1: byte;
    Array1: array[10] of byte;
    String1: string[10];

const Constant1: byte = 20;
    Array2: array[20] of byte = (10,20,30);
    String2: string[10] = 'abcd';

procedure Proc1(Bte: byte): word;
```

In the file where they are referred to:

```
var Variable1: byte;           external;
    Array1: array[10] of byte; external;
    String1: string[10];       external;

const Constant1: byte;        external;
    Array2: array[20] of byte; external;
    String2: string[10];       external;

procedure Proc1(Bte: byte): word; external;
```

External definitions have to be declared in the public part of a unit.



Not possible is the following:

```
const Constant1: byte;                                external;
    Array2: array[Constant1] of byte; external;
    String2: string[10];                               external;
```

One can not use external constants in external constant definitions.

External references are only resolved at the very end of the make process.



Additionally: You can use other size values for arrays and strings than the actual ones in the “external” declarations without problem, the compiler will not complain. But: the “SizeOf” function will return the one declared in the “external” declaration, not the actual one.



In libraries that use external declarations (see also [Libraries](#)), this can happen frequently: the size of arrays and strings is a “dummy” size in the external declaration: the size can not be known at the library’s compilation time. The actual size is then to be found in a constant (or variable) also declared external in the library. See example above: “Constant1” holds the real size of Array2.

10 Command Line Compiler/Linker (CLC)

The command line compiler/linker is started up from within the IDE. The command line itself is generated by the IDE, and contains (among other things) the following information that is relevant for the compiling/linking process:

- The search paths (copy from those in the project file)
- A number of filenames of “.mcl” files. These are usually libraries checked in the [Library Manager](#)
- A number of filenames of “.mpas” files. These are the source files of your project. The last one of these is the name of the [main file](#).



The order of those parameters in the compiler’s command line are important, this is the order required for a successful compilation and linking:

- The search paths are sequentially searched for .mcl and .mpas files until one is found. That one is the one that will be used.
- The order of the “.mcl” file parameters: The order of these is defined by the order of the “libraries” defined in the [MLK File](#). If a library is used by another one it should be in the parameters before the using one.

Example of the compiler's command line:

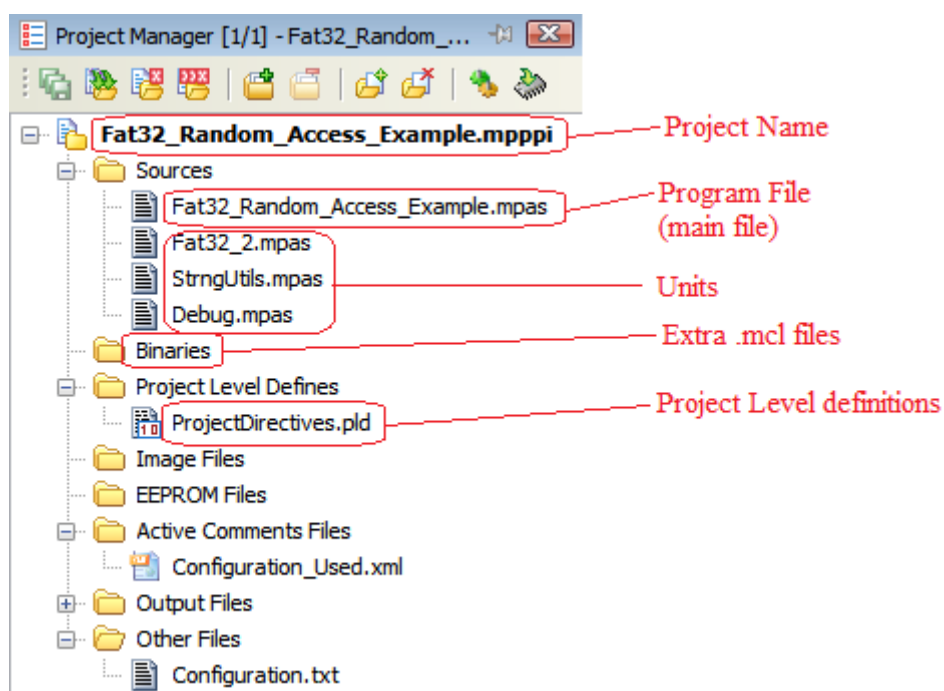
```
mPPic.exe -MSF -DBG -pP18F2620 -Y -DL -RA -O11111114 -fo40 -N"C:\PIC Projects PRO
compiler\Tests\Make_process\MakeProcessTests.mpppi" -
SP"C:\Users\Public\Documents\Mikroelektronika\mikroPascal PRO for PIC\defs\" -
SP"C:\Users\Public\Documents\Mikroelektronika\mikroPascal PRO for PIC\uses\P18\" -
SP"C:\PIC Projects PRO compiler\Tests\Make_process\" -
SP"C:\Users\Public\Documents\Mikroelektronika\mikroPascal PRO for
PIC\Packages\Utilities\Uses\" " __Lib_Math.mcl" " __Lib_MathDouble.mcl"
" __Lib_System.mcl" " __Lib_Delays.mcl" " __Lib_String.mcl" " __Lib_UART_c67.mcl"
"Debug.mcl" "MakeProcessTests.mpas" "Unit_One.mpas" "Unit_Two.mpas"
```

search paths: yellow

libraries: blue

Source files: green

11 Project Manager



11.1 Sources



The first file in this section should always be the Program File. This is how the IDE/compiler identifies it. If another source file is the first one then an error message “no main function” is generated.



It is important that all sources are entered here because it will make them “part of the project”:

- Their search path will be added if not already there
- They will be surely compiled if necessary or wanted, they will be entered in the compiler’s parameter list for compilation/ linking.

11.2 Binaries

Adding binaries to the [Project Manager](#) also makes them “part of the project” (see above for what this means).

11.3 Project Level Defines

Adding a project Level defines file (extension .pld) makes the definitions in there visible to all source files in the project.

Example of a .pld file content:

```
DEBUG
UART
VERSION1
```

is the same as

```
{ $DEFINE DEBUG }
{ $DEFINE UART }
{ $DEFINE VERSION1 }
```

in every source file of the project.

12 MLK File

The .MLK file plays an important role in library management: it defines

- which libraries are valid (installed) for each processor (there is a separate .MLK file for each processor)
- Which library can use other libraries: the order of library entries in the .MLK file define this: a library that is used another one must be defined first (before the “using” one). This order is never a problem for standard (mE) libraries, but can be a problem for User libraries. If libraries are installed by the [Package Manager](#) then their order in the .MLK file is defined by their “Dependency”: libraries depending on others will have an entry below the libraries they depend on.
- Under which name (the “alias”) the libraries will be visible in the library manager, and what the actual names of the libraries .mcl files are.

Libraries are either

- **Standard** (or “**regular**”) libraries (delivered with the compiler: the “mikroE package”)
- **User** libraries installed with the [Package Manager](#)
- User libraries installed manually or with another tool

Example (extract of the P18F2550.mlk file, where a number of library packages have been installed for):

```
<LIBRARIES> <--- the libraries section in the .MLK file
...
    <LIB>
        <ALIAS>C_Type</ALIAS>      <--- the name shown in the library manager
        <FILE>__Lib_CType</FILE>    <--- the actual .mcl filename
        <TYPE>REGULAR</TYPE>        <--- the type (regular = standard library)
    </LIB>
    <LIB>
        <ALIAS>String</ALIAS>
        <FILE>__Lib_String</FILE>
        <TYPE>REGULAR</TYPE>
    </LIB>
    <LIB>
        <ALIAS>C_Stdlib</ALIAS>
        <FILE>__Lib_CStdlib</FILE>
        <TYPE>REGULAR</TYPE>
    </LIB>
...
    <LIB>
        <ALIAS>LCD_4bits_p18</ALIAS> <--- the name shown in the
                                   Library Manager
        <FILE>LCD_4bits</FILE>      <--- the actual .mcl filename
        <TYPE>USER_LIB</TYPE>        <--- user defined (non standard) library
        <PACKAGE_NAME>LCD</PACKAGE_NAME> <--- the package name shown in the
                                   library Manager.
    </LIB>
    <LIB>
        <ALIAS>LCD1602_4bits_p18</ALIAS>
        <FILE>LCD1602_4bits</FILE>
        <TYPE>USER_LIB</TYPE>
        <PACKAGE_NAME>LCD</PACKAGE_NAME>
    </LIB>
...

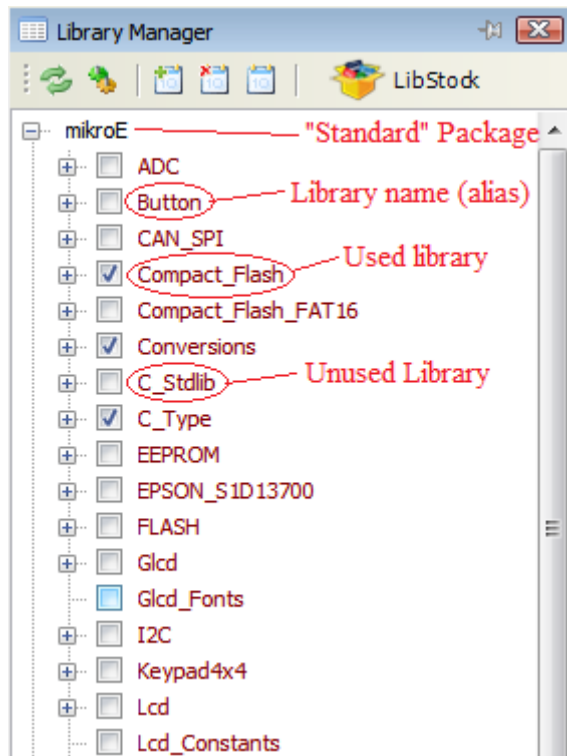
```

As you can see in the above extract, the library “String” can use the “C_Type” library (I do not know if this is actually the case), because the “C_Type” library entry is before the “String” library’s entry. Also the “package name” is absent in the libraries of type “REGULAR” (the “standard” libraries delivered with the compiler): it is always “mikroE”.

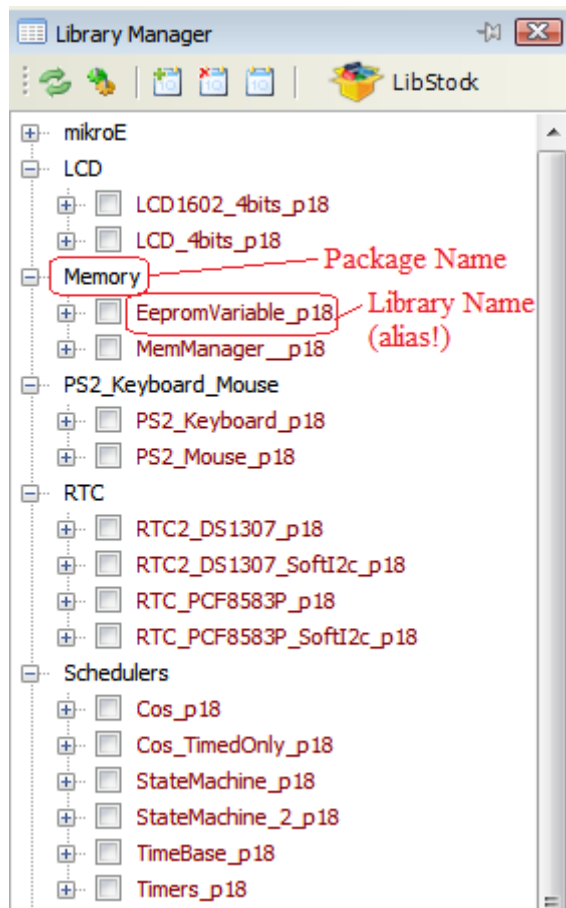
Adding library entries to the .MLK file manually is possible if one obeys the XML syntax (see above) and places the new entry in the correct place (below the libraries it uses – always below the “REGULAR” library types!

13 Library Manager

Standard (regular) Libraries:



User Libraries:

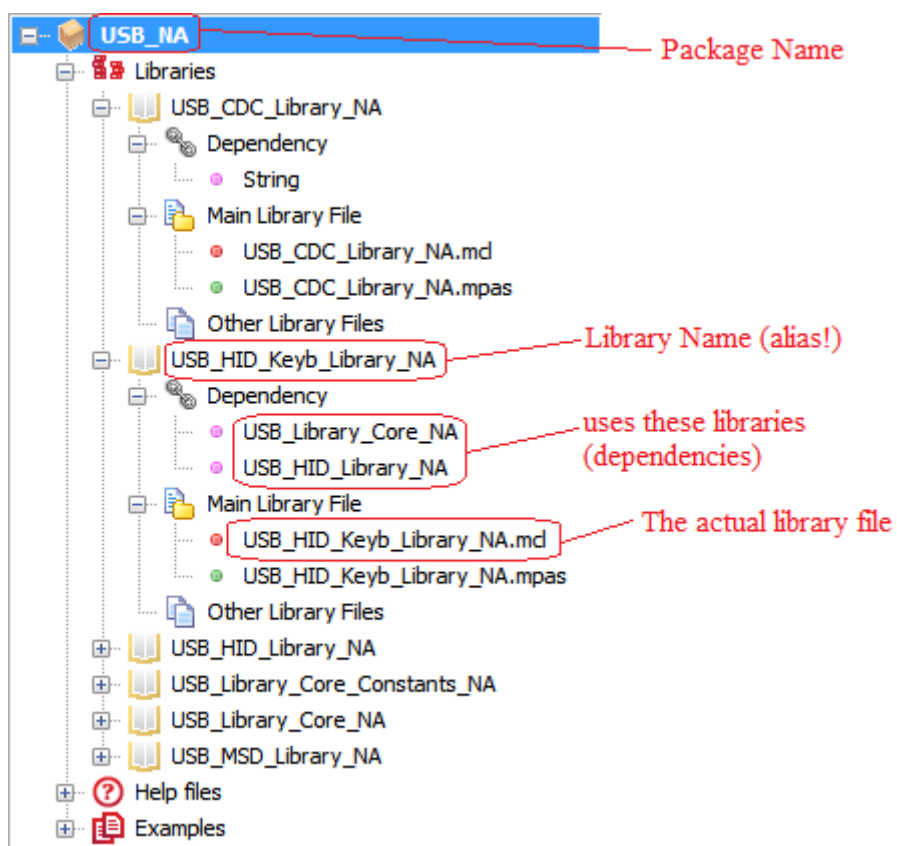


Not much more to say about the Library Manager that was not already explained on other sections: all libraries that are “checked” can be used by the [main file \(Program file\)](#) and [all units of the Project](#). As you can see in the screenshots above: the Package name and the Library names (aliases, not the real filenames, see the [MLK File](#) and the “Package Manager”) are shown in the [Library Manager](#).

14 Package Manager

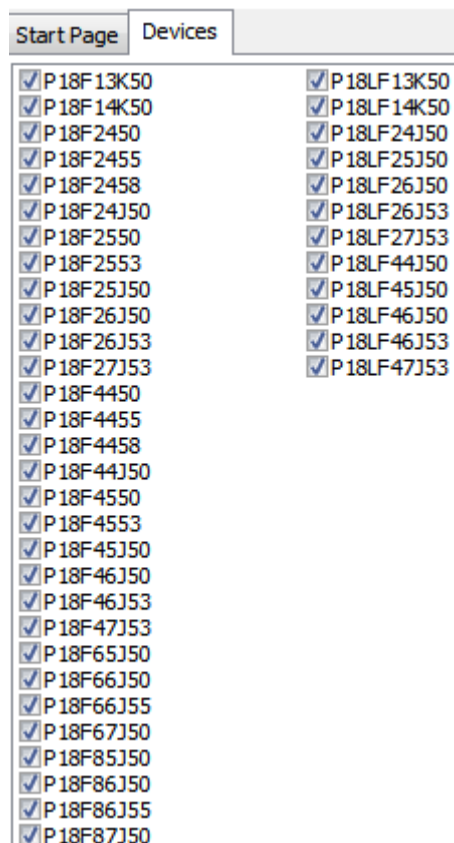
This tool is not embedded in the IDE, it can be started both separately of from within the IDE (Tools menu).

It is capable to build packages of libraries, together with their help files and usage examples. Also (and most important) it will install a package for the latter’s selected processors: it will insert the correct library entries on the correct place in the [MLK File](#) (provided the “dependencies” of each library is defined correctly).



As one can see in the screenshot above a package can contain several libraries, each with their name (the alias, which will be shown in the [Library Manager](#)), their dependencies (which will define their entry order in the [MLK File](#)) and the actual library .mcl filename. If a source file of the library is available it can also be entered here.

Below the “Devices” screen of the Package Manager is shown. It shows the devices for which the currently selected library should be installed (for which the library is meant).



15 Libraries

Units can be promoted to “Libraries” by (e.g.)

- compiling them,
- putting them (at least the .mcl file) in a package (if needed distributing the package) and
- installing the package with the [Package Manager](#).

The units to transformed into “libraries” should be compiled with

- “Build all files as Library” checked (in IDE’s Tools -> Options -> Output settings, Compiler)
- Build type “ICD debug” (in IDE’s “Project Settings”)⁴
- Debugger “mikro ICD” (in IDE’s “Project Settings”)⁴



Libraries can not “use” (with [Uses Clauses](#)) units from the project that uses the libraries. If a library needs something from its using project, then always [External References](#) to the items to access must be used. This is the only manner of correctly accessing Project items.

⁴ Only if the using project must be debugged with the in circuit debugger



Compiler directives (“{\$DEFINE}, “{\$IFDEF ...}” etc...) in libraries of which only the .mcl file is available do not work. The .mcl file holds the compiled contents as defined by those compiler directives at compile time.

A separate .mcl file should be provided as a separate library for each wanted value of the compiler directives.

[end of document]