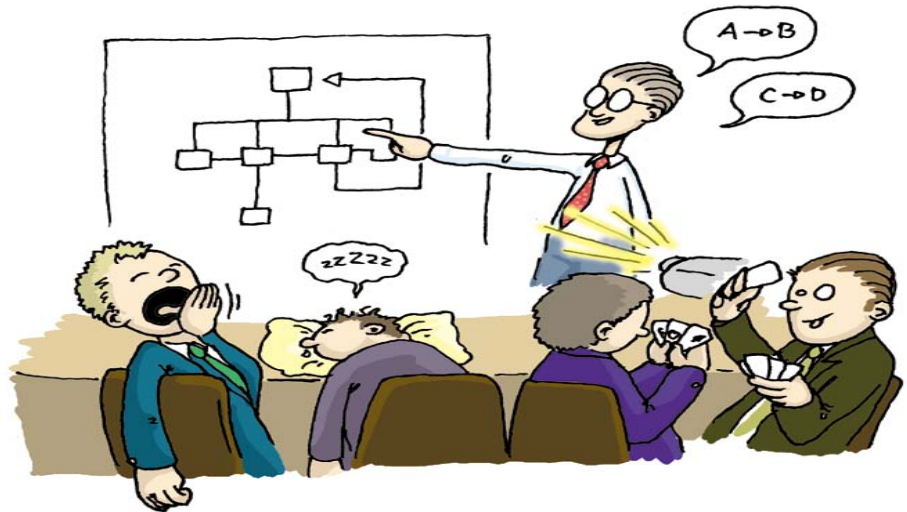
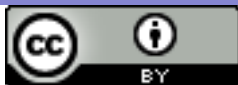


How to use package MultE





This Work is licensed under a Creative Commons Attribution
4.0 International License.

© Copyright © 2011-2016 János Végh
(Janos.Vegh@unideb.hu)

All rights reserved

Printed in the World, using recycled electrons

- 1 Sectioning document
- 2 Preparing program listings
- 3 Inserting figures

1 Sectioning document

■ Document units

- Frames
- Chapter
- Section and below

■ Dual language sources

■ Chapter illustration

Basically, the document is organized as 'beamer' needs it, but to print it in a book-like form, the sectioning must be changed, and also the package 'beamerarticle' must be used. In order to provide a uniform wrapper around sectioning, MultEdu introduces its own sectioning units.

These units actually correspond to the ones used in format 'book', and MultEdu transforms them properly when preparing slides.

Usage:

```
\MEframe[keys]{subtitle}{content}
```

Legal keys are

`shrink=true|false` and `plain=true|false`

By default, both are false.

Correspondingly, the biggest unit is the 'chapter'. (As mentioned, for slides it is transformed to 'section'.) Usage:

```
\MEchapter[short title]{long title}
```

The next, smaller unit is the 'section'. (As mentioned, for slides it is transformed to 'subsection'.) Usage:

```
\MEsection[short title]{long title}
```

In a similar way, there exists

```
\MEsubsection[short title]{long title} and
```

```
\MEsubsubsection[short title]{long title}; the latter  
one is transformed for slides to \paragraph
```


1 Sectioning document

■ Document units

■ Dual language sources

■ Switching between languages

■ Frames

■ Chapter

■ Section and below

■ Chapter illustration

It happens, that I teach the same course in my mother tongue for my domestic students, and in English, for foreign students. The course material is the same, and it must be developed in parallel. Obviously it is advantageous, if they are located in the same source file, side by side; so they can be developed in the same action. The `\UseSecondLanguage` macro supports this method.

The macros introduced above have a version with prefix 'MED' rather than 'ME only, which takes double argument sets (arguments for both languages). Depending on whether `\UseSecondLanguage` is defined, the first or the second argument set is used. In this way, just the 'src/Heading.tex' source file shall be edited, and the other language output will be prepared.

Usage:

`\UseSecondLanguage{YES}`

where the argument `{}` is not relevant, only if this macro is defined or not. Traditionally, it is defined in 'src/Heading.tex'. The two kinds of macros can be mixed, but only the 'D' macros are sensitive to changing the language.

In dual language documents, usually

```
\MEDframe[keys]{subtitle, first language}  
{content, first language } {subtitle, second  
language} {content, second language}
```

is used. I.e. the user provides titles and contents in both languages, and for preparing the output, selects one of them with `\UseSecondLanguage`.

Correspondingly, the biggest unit in a dual language document is the 'Dchapter'. (As mentioned, for slides it is transformed to 'Dsection'.) Usage:

```
\MEDchapter[short title1]{long title1}{short  
title2}{long title2}
```

which is transformed to

```
\MEchapter[short title1]{long title1} or
```

```
\MEchapter[short title2]{long title2} calls,  
depending on whether \UseSecondLanguage is or is not  
defined.
```

The usage of the lower units is absolutely analogous.

1 Sectioning document

- Document units
- Dual language sources
- Chapter illustration

Some book styles also allow presenting some illustration at the beginning of the chapters.

Usage:

`\MEchapterillustration{file}`

For slides, the illustration appears in a 'plain' style style. For books, the picture is placed at the beginning of the chapter. If the file name is empty, a 'fig/DefaultIllustration.png' file is searched. If the file not found, no illustration generated.

If macro `\DisableChapterIllustration` is defined, no picture generated.

- 1 Sectioning document
- 2 Preparing program listings
- 3 Inserting figures

When teaching programming, it is a frequent need to display program listings. Through using package 'listings', MultEdu can implement this in very good quality. For details not described here see documentation of package 'listings'. Notice that here the ratio of the listings within the text is unusually high, so it is very hard for the compiler to find good positioning. In the case of real texts, the page is much more aesthetic.

2 Preparing program listings

■ Setting appearance

- Displaying inline fragments
- Displaying program listings
- Decorations on listings
- Other related macros
- Extra program languages

Package 'listings' allows to set up the style of displaying program listings according to our taste (and the requirements). MultiEdu pre-sets some style and allows to modify it as much as you like.

Macro

`\MSESetStandardListingFormat` sets up a default appearance, and no programming language. Macro

`\MSESetListingFormat[options]{language}` sets the language, the same appearance as macro

`\MSESetStandardListingFormat`

and also allows to overwrite parameters of 'listings' through 'options'.

2 Preparing program listings

- Setting appearance
- **Displaying inline fragments**
- Displaying program listings
- Decorations on listings
- Other related macros
- Extra program languages

A typical task is to display a shorter fragment, like a line or a keyword. It is possible using `\lstinline|code|`.

The LaTeX commands appearing in this documentation are produced in such a way that at the beginning of the chapter commands

```
\MSESetListingFormat{TeX}
```

```
\lstset{basicstyle=
```

```
\ttfamily\color{black}\normalsize}
```

or

```
\MSESetListingFormat[basicstyle=
```

```
\ttfamily\color{black}\normalsize]{TeX}
```

are issued (otherwise the character size of the program text will be too small).

2 Preparing program listings

- Setting appearance
- Displaying inline fragments
- **Displaying program listings**
- Decorations on listings
- Other related macros
- Extra program languages

Program listings can be displayed using macro
`\MESourceFile[keys] {filename} {caption}`
`{label}{scale}`. Possible keys:
`wide,decorations`.

"Hello World" – a C++ way

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

The command used to display Listing was
`\MESourceFile[language={ [ISO]C++}]`
`{lst/HelloWorld.cpp} {A "Hello World"- C++`
`program} {lst:hello.cpp}{}`

Many times one needs wider program listings. In the case of the two-column printing, the listing shall fill the width of the two columns. In the case of one-column printing, the narrow list extend to 70% of the text width, while the wide lists The wide listings can be placed even hardly on the printed page (the first proper place, relative to the appearance of the macro is the top of the next page), and in addition, the orders of normal and wide listings cannot be changed. Because of this, the place where the listing appears, might be relatively far from the place of referencing it.

A "Hello World"- C++ program, wide

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

The command used to display Listing :

```
\MESourceFile[language={ [ISO]C++ },wide]
{lst/HelloWorld.cpp} {A "Hello World"- C++
program, wide} {lst:Whello.cpp}{} }
```

2 Preparing program listings

- Setting appearance
- Displaying inline fragments
- Displaying program listings
- **Decorations on listings**
 - Highlighting lines
 - Commenting highlighted lines
 - Commenting source lines
 - Numbered balls to listing
 - Figure to listing
- Other related macros
- Extra program languages

Different decorations can be placed on top of listings. To do so, one has to use the keyword `decorations`, and to insert as arguments the macros presented in this section.

The general form:

```
\MESourceFile[options, decorations={ list of  
decorations } ] {source file} {caption} {label}{}  
where the list of decorations may contain any of the  
decoration macros presented in the section. Any option, used  
by package 'listings' applies.
```

In a program listing, a range of source lines can be highlighted using macro

```
\MESourcelinesHighlight{BallonName} {SourceName}  
{FirstLine} {LastLine}.
```

Here `BallonName` is a new label, which denotes the newly created source line region highlighting box, `SourceName` is the label of the source file.

To highlight a program body in listing the macro

```
\MESourceFile[language={ [ISO]C++ }, decorations={
\MESourcelinesHighlight{HelloBalloon}
{1st:HLhello.cpp} {6}{8} } ]
{1st/HelloWorld.cpp} {"Hello World" -- a C++
way, kijel~0lt} {1st:HLhello.cpp}{}
```

shall be used.

"Hello World" – a C++ way, highlighted

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

The highlighting box can also be commented. Using macro `\MESourceBalloonComment[keys]{BallonName}{ShiftPosition}{Comment}{CommentShape}` allows to comment the balloon created previously. Here `BallonName` is the first argument of `\MESourcelinesHighlight`, `ShiftPosition` is the shift of the comment box, `Comment` is the comment text. Possible keys, with defaults are: `width[=3cm]` and `color[=deeppeach]`.

Listing is produced using macro

```
\MESourceFile[language={ [ISO] C++},wide,
decorations={
\MESourcelinesHighlight{HelloBalloon}
{lst:HLChello.cpp} {6}{8}
\MESourceBalloonComment{HelloCBalloon} {0cm,0cm}
{This is the body} {CommentShape} } ]
{lst/HelloWorld.cpp} {"Hello World" -- a C++
way, commenting highlighted} {lst:HLhello.cpp}{}
```

"Hello World" – a C++ way, commenting highlighted

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

This is the body

The individual source lines can also be commented, see

Listing. To produce it, the command was:

```
\MESourceFile[language={ [ISO] C++ },wide,
decorations={
\MESourcelineComment{lst:Chello.cpp} {6}
{0cm,0cm} {This is a comment} {CommentShape} }
]{lst/HelloWorld.cpp} {"Hello World" -- a C++
way, commenting source lines} {lst:Chello.cpp}{}
```

"Hello World" – a C++ way, commenting source lines

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

This is a comment

On the program listing numbered balls can also be located, for referencing the lines from the text. This can be done using macro

```
\MESourcelineListBalls[keys]{ListingLabel}{List  
of lines}
```


which puts a numbered ball at the end of the listed lines. Here `ListingLabel` is the label of the listing, `List of lines` is the list of sequence numbers of the lines to be marked.

Possible key, with defaults:

`color[=orange]` and `number[=1]`.

Notes:

- When making slides, the balls will be put to separated slides.
- The positioning using geometrical positions, does not consider 'firstline'.

The marked lines can then be referenced through the balls like '(Listing 33 )' is the return instruction'. It can be produced using

```
\MEBall{Listing~\ref{lst:LBhello.cpp}}{2}
```

To produce Listing, the macro

```
\MESourceFile[language={ [ISO]C++}, decorations={
\MESourcelineListBalls{1st:LBhello.cpp}{3,8,5} }
] {1st/HelloWorld.cpp} {"Hello World" -- a C++
way, with balls} {1st:LBhello.cpp}{}
```

has been used

"Hello World" – a C++ way, with balls

```
#include <iostream>
using namespace std;
int 
main ( int argc, char ** argv )
{
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

To produce Listing, the macro

```
\MESourceFile[language={ [ISO]C++}, decorations={  
\MESourcelineListBalls{1st:LBhello.cpp}{3,8,5} }  
] {1st/HelloWorld.cpp} {"Hello World" -- a C++  
way, with balls} {1st:LBhello.cpp}{}  
has been used
```

"Hello World" – a C++ way, with balls

```
#include <iostream>  
using namespace std;  
int  
main ( int argc, char ** argv )  
{  
    // print welcome message  
    cout << "Hello World" << endl;  
    return 0;   
}
```

To produce Listing, the macro

```
\MESourceFile[language={ [ISO]C++}, decorations={
\MESourcelineListBalls{1st:LBhello.cpp}{3,8,5} }
] {1st/HelloWorld.cpp} {"Hello World" -- a C++
way, with balls} {1st:LBhello.cpp}{}
```

has been used

"Hello World" – a C++ way, with balls

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{ 3
    // print welcome message
    cout << "Hello World" << endl;
    return 0;
}
```

Sometimes one might need to insert figures into the listing.

The macro is

```
\MESourcelineFigure[keys] {SourceLabel} {LineNo}  
{ShiftPosition} {GraphicsFile}.
```

Possible key is `width [=3cm]`

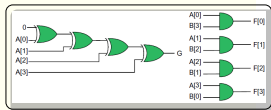
To produce Listing, macro

```
\MESourceFile[language={Verilog},wide,
decorations={ \MESourcelineFigure[width=5.2cm]
{lst:forloops.v}{8} {3.0,-.3} {fig/forloops} } ]
{lst/forloops.v} {Implementing \ctext{for} loop
with repeating HW} {lst:forloops.v}{}
was used.
```

Implementing for loop with repeating HW

```
// for == repeat HW
```

```
always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```



2 Preparing program listings

- Setting appearance
- Displaying inline fragments
- Displaying program listings
- Decorations on listings
- Other related macros
 - Comparing source files
- Extra program languages

Sometimes it is worth to compare source files, side by side.

The macro for this is

```
\MESourceFileCompare[keys]{source file1} {source  
file2} {caption} {label}
```

The command to produce Listing is

```
\MESourceFileCompare[language={ [ANSI] C}]
{lst/lower1.c} {lst/lower2.c} {Comparing two
routines for converting string to lower case}
{lst:lower12.c}
```

Comparing two routines for converting string to lower case

```
/* Convert string to lowercase:
   slow */
void lower1(char *s)
{
    int i;

    for (i = 0; i < strlen(s);
         i++)
        if (s[i] >= 'A' && s[i] <=
            'Z')
            s[i] -= ('A' - 'a');
}
```

```
/* Convert string to lowercase:
   faster */
void lower2(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)

        if (s[i] >= 'A' && s[i] <=
            'Z')
            s[i] -= ('A' - 'a');
}
```

It is also useful sometimes to show the source file with its output. The macro

```
\MESourceFileWithResult}[keys]{source file}
{result file} {caption} {label}
```

allows to do that.

For producing Listing the command

```
\MESourceFileWithResult[language=C++,wide,decorations=+
\MESourcelineListBalls{lst:calculatorwithresult}
{13,14,16,18,19} }]
{lst/expensive_calculator.cpp}
{lst/calculatorresult.txt} {The calculator
program with its result}
{lst:calculatorwithresult}
was used.
```

The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators
```

```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;
```

```
    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 <<
        endl;
```

```
    cout << "7 % 3 = " << 7 % 3 << endl;
```

```
    cout << "7 + 3 * 5 = " << 7 + 3 * 5 <<
        endl;
```

```
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5
        << endl;
```

```
    return 0;
```

```
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

1

The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;

    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 <<
        endl;
    cout << "7 % 3 = " << 7 % 3 << endl;

    cout << "7 + 3 * 5 = " << 7 + 3 * 5 <<
        endl;
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5
        << endl;

    return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;

    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 <<
        endl;

    cout << "7 % 3 = " << 7 % 3 << endl;

    cout << "7 + 3 * 5 = " << 7 + 3 * 5 <<
        endl;
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5
        << endl;

    return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

3

The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;

    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 << endl;

    cout << "7 % 3 = " << 7 % 3 << endl;

    cout << "7 + 3 * 5 = " << 7 + 3 * 5 << endl;
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5 << endl;

    return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;

    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 <<
        endl;

    cout << "7 % 3 = " << 7 % 3 << endl;

    cout << "7 + 3 * 5 = " << 7 + 3 * 5 <<
        endl;
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5
        << endl;

    return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

5

2 Preparing program listings

- Setting appearance
- Displaying inline fragments
- Displaying program listings
- Decorations on listings
- Other related macros
- Extra program languages

For my own goals, in addition to the programming languages defined in package 'listings', some further languages have been defined:

- diff
- [DIY]Assembler
- [ARM]Assembler
- [x64]Assembler
- [y86]Assembler

- 1 Sectioning document
- 2 Preparing program listings
- 3 Inserting figures**

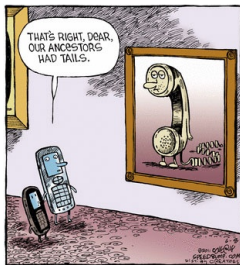
- 3 Inserting figures
 - Traditional figures

Traditional figures can be displayed using macro

`\MEfigure[keys]{image file} {caption} {label}`
`{copyright} {ScaleFactor}`. Possible keys:

`wide`.

©2011 <http://pinterest.com>



On slides, the single-width figures are placed in 'columns'

When new and old phones meet

The command used to display Figure was

`\MEfigure{fig/phone_ancestors} {When new and`
`old phones meet} {fig:phonenachestors} {2011`
`http://pinterest.com}{.8}`

