<MHT>

A macro processor, template mechanism and CGI library
Version 1.2 March 2003

by Thomas Weckert
`info at weckert.org`

Table of Contents:

1 Introduction

This document describes the features of the MHT template mechanism and provides examples to introduce software developers and web producers that are interested in using MHT for HTML page creation.

1.1 What is MHT?

MHT is the shortcut for *Macro Hyper Text*, which should point out that MHT is a macro processor to render HTML, thus it can generate any form of text files. You should know that MHT comes from the very early days of internet programming in the mid-90‘s, when there was no PHP, no ASP and also no Java servlet programming yet developed.

By linking the MHT macro processor to your CGI programs, CGIs can generate the dynamic HTML output by processing templates. With the static binary of the MHT macro processor, static HTML pages can be rendered, controled by a set of MHT scripts. A set of "content" templates is compiled through a set of "style and layout" templates. The MHT template mechanism is also able to generate MHT templates itself to be used by CGI programs to generate the dynamic HTML output.

1.2 Why using MHT?

- A vast majority of internet sites utilize CGI in one way or another. Perl and PHP, inherently slow and simple, are not always the best choice. MHT is the right choice for simple web applications without an application server or Java servlet environment. Programs written in ANSI-C are in the same sense „platform independent“ as Java, except that you have to re-compile your program code on each new platform (and even Java isn’t too much „platform-independent“ at all).
- Embedded languages like PHP or ASP make it easy to stick some code into HTML, but make it too difficult to maintain the HTML layout and program code, because both the program code and the HTML are placed in the same files. MHT just sticks some macros into the HTML, and the functionality is implemented by a CGI in C.
- I don’t see that the time is over for small CGI programs written in C. Compiled binaries are a lot faster executed than interpreted Perl or ASP/PHP code. I also don’t want to process every HTML page through a PHP procerssor as many people do in their web sites.
- Additionally, some servers (notably Netscape and Apache) have C-APIs so that CGI programs can be written as extensions to the server, rather than as separate programs. This greatly improves performance, especially on high-traffic sites. If you really need a session based functionality, you can extend your C sources by using FastCGI.

1.3 Overview

1.3.1 The MHT macro processor

The nucleus of MHT is the MHT processor. It manages a hashtable where all the macros and their definitions are stored, together with a set of functions to process templates. The MHT macro processor currently supports these functions:

- Recursive expansion of macros
- Macros with parameters
- File-inclusion of templates
- Blocks to process portions of text in a template more than once
- Blocks with parameters
- A set of directives such as conditionals, file operations etc.
- Macro- functions to compare strings, to check substrings, to prove macro definitions, and to prove the existence of blocks
- Switches to control the global behavior of the MHT processor such as killing newlines, unnecessary whitespaces, conversion of German umlauts
- Max. 64 different file handles to write the output into text files

1.3.2 The mht2html compiler

`mht2html` is again the MHT processor, packaged into a program that can be called from the command line. Every text file is a legitimate, processable input file, no matter what characters it includes, as long as its directives are properly formed. `mht2html` reads each input file in its entirely, and then the input is processed by expanding macros and acting on directives. The output is written to a single output sink, which is `STDOUT` unless there is an output file handle specified. `mht2html` supports currently the following command line options:

- `-h` prints a help screen with an explanation of the command line options
- `-v` prints the release version and compile date of the MHT processor
- `-p` processes a MHT file specified by it‘s absolute path to `STDOUT`, for example: `mht2html -p /tmp/file.mht`

You should use `.mht` as a proper file extension for all your MHT input files.

1.3.3 The template mechanism

The basic idea of the MHT template mechanism is, to package the layout structure which is applied to every HTML page into a separate „MHT style“, which is a set of MHT templates. The advantage of doing this is that changes to the MHT style need only be made once in order to take effect on all pages after they are re-compiled. This makes it possible for example to produce:

- "Multi-lingual" websites, where content in different languages etc. is compiled by using the same MHT style,
- "Multi-sites", such as websites that offer each page in a frameless "plain" version for version 2.x browsers, a "text" version without images for fastest

download, and a "frame" version for 3.x browsers and above. The same content is then compiled through different MHT styles.
- "offline" versions of your website with correct adjusted image sources and href-links to open the HTML files from a local file-system.

1.3.4 The CGIMHT C-API

The MHT macro processor is also available as a library for Windows NT and Linux named CGIMHT. You can link the CGIMHT library to your CGI programs written in ANSI-C. This way your CGI programs can easily generate the dynamic HTML output by processing MHT templates. These templates could be written „by-hand“ as any other HTML page, or they can be generated by the MHT template mechanism for a larger website out of a given MHT style. The CGIMHT library includes also a set of functions to read in POST and GET CGI input values.

1.3.5 Download

The latest release of MHT is available at `http://www.weckert.org/mht/` as a gnuzipped TAR archive.

1.3.6 License

The MHT package is published under the terms of the GNU Public License, pls. see `http://www.gnu.org/copyleft/gpl.html`

Here are some further wishes:

- As courtesy and down to genuine personal interest, I'd like to know what people do with MHT. Please drop me a line if you achieve anything significant.
- Don’t hesitate to contact me for questions and comments, but I will not provide time consuming support to you.
- Please leave some credit to me in your projects if you use MHT.

2 The macro processor

2.1 Basics

In MHT a maro is a string like `<#title>` for example. You write text templates which contain any text such as HTML tags, with MHT macros spread inside. The MHT processor goes through the text, and whenever a macro is encountered in the source, it is expanded and replaced with its value, in the same manner as you would do a „Search and Replace“ in a word processor.

Macros are expanded by recursion, so if the expanded value of a macro contains again one or more macros, MHT tries to expand them immediately. If the MHT processor cannot find a definition for a macro, it is left unchanged inside the text.

MHT itself is controled by directives, which are preceded by a "#" (hash sign) character as the first character on the line. Lines beginning with a directive are ignored and filtered out of the output. MHT expects a directive after a hash sign at the beginning of a line. Lines are commented out by `#!`

MHT currently supports these directives (in alphabetical order): `begin, def, defex, echo, echoln, elif, else, end, endif, if, include, loop, mhtexit, mhtfile, mhtvar, pause, process, undef, undefblock, write, writeln`

The directives are reserved names and may not be used as names for macros etc.

2.2 Macros

2.2.1 Definition of macros

The `#def` directive defines a macro, its general form is:

```
1   #def name definition
```

The macro name may be any contiguous string without spaces, and the macro definition is everything after the macro until the end of the line, and may include spaces as well as other macros. The max. length of the macro name is 128 chars., the max. length of the macro definition is 1024 chars. (which should be fairly enough).

A simple macro definition to define the macro „name“ as „Steve Jobs“:

```
2   #def name Steve Jobs
```

To get the output, the macro is invoked by `<#name>`, for example...

```
3   My name is <#name>!
```

...will be expanded to:

```
1   My name is Steve Jobs!
```

If you want to try out the example above (you should do so!), copy and paste the lines 1-2 of the example without the beginning line numbers into a text file, save it as `example1.mht`, and invoke it by invoking:

```
mht2html -p /<path>/example1.mht
```

Macros are case sensitive! In example 2 in line 3 the macro `<#name>` is written with a beginning lower case character, and in line 4 with a beginning upper case character.

Case- sensitive macros:

```
1   #def name Bill Gates
2   #def Name Steve Jobs
3   My name is <#name>!
4   My name is <#Name>!
```

The output will be expanded to:

```
1   My name is Bill Gates!
2   My name is Steve Jobs!
```

What happens if you re-define a macro, that is also used in the definition of other macros as well? Look at the next example first:

Overwriting the definition of a macro:

```
1   #def macro1      a test
2   #def macro2      <#macro1>
```

```
3  This is <#macro2>
4  #def macro1     another test
5  This is <#macro2>
```

The output will be expanded to:

```
This is a test
This is another test
```

To understand the example above, you should know that in MHT a macro is **NOT** subsequently expanded to its definition, it is saved "as is". Not until `<#macro2>` is invoked in line 3, MHT will look up the current value of `<#macro1>`, and expand it. `macro2` is indeed defined as `<#macro1>` and not as the expanded value of `macro1`.

2.2.2 Definition of expanded macros

As describe before, the definition of a macro is saved „as- is". But sometimes it might be useful to expand the definition of a macro already when it is defined. So does `#defex`, it works quite the same as `#def`, but the definition is immediately expanded to its value.

Difference between `#def` and `#defex`:

```
#def macro1      a test
#defex macro2    <#macro1>
```

According to the previous section, now `macro2` is saved as „a test" and not as `<#macro1>`. The value of `macro2` is immediately expanded when the #defex directive is processed by the MHT processor. Please note that `#defex` is a directive for special cases, MHT won't work as you expect if you define your macros by using `#defex`.

2.2.3 Using macros with parameters

Macros may have parameters in their definition. Parameters are referred as <`#.%`n> for the n- th parameter of a macro. When a macro is invoked, its parameters are separated by a „`|`" (pipe sign) from the macro name. Macro parameters can include any text, including spaces, as well as other macros.

„Empty" macro parameters remain in the same manner unexpanded inside the output as macros which cannot be expanded due to a missing definition/value. It is allowed to cut- off the remaining „empty" parameters if you invoke a macro with less parameters than your macro expects.

The macro „igb0" below could be described as „image- GIF- border- 0", and produces HTML image tags for GIF images without a border:

```
#def pic_path   /pics/
#def igb0 <img src="<#pic_path><#.%1>.gif" width="<#.%2>" height="<#.%3>"
alt="<#.%4>">
<#igb0|headline|150|20|MHT is great!>
<#igb0|headline|150|20>
<#igb0|headline|||MHT is great!>
<#igb0>
```

The output will be expanded to:

```
<img src="/pics/headline.gif" width="150" height="20" alt="MHT is great!">
<img src="/pics/headline.gif" width="150" height="20" alt="<#.%4>">
```

```
<img src="/pics/headline.gif" width="<#.%2>" height="<#.%3>" alt="MHT is great!">
<img src="/pics/<#.%1>.gif" width="<#.%2>" height="<#.%3>" alt="<#.%4>">
```

To avoid unexpanded macro parameters like `<#.%1>` that result from missing parameters, you can encapsulate macro parameters inside a `<#ifequal>` macro. Inside this `<#ifequal>` macro, each parameter `<#.%n>` is compared with an empty string or the `<#null>` macro, to check whether the parameter exists, or not. `<#ifequal>` is described in a later section of this documentation. Please note that `<#ifdef>`, which is also described later, cannot be used to prove macro parameters!

2.2.4 Deleting macros

The `#undef` directive removes the definition of a macro so that it will not be expanded if it is encountered in the source. This is a „safe“ and maybe more intuitive method than overwriting the same macro again and again with different definitions.

Deleting macros:

```
#def macro1      another test!
Hi, this is <#macro1>
#undef macro1
Hi, this is <#macro1>
```

The output will be expanded to:

```
Hi, this is another test!
Hi, this is <#macro1>
```

2.2.5 Delayed macros and directives

What if you want to compile again MHT templates and not HTML files out of MHT templates? Sounds ridiculous? Remember that by using the CGIMHT library, CGI programs can process MHT templates to produce the dynamic HTML output. If you use your own MHT style to compile the HTML files of your web site, it is quite useful to compile also the CGIMHT templates for your CGI programs out of the same MHT style to get a 100% consistent layout.

The problem is, that all macros and directives inside the templates will be expanded and processed during compilation. In case you want to render a new MHT template instead of a HTML page, maybe some macros and directives should not be expanded and processed at „compile- time“, but rather at „run- time“ when the template is processed by a CGI program.

This is done simply by using two or more hash- signs „#“ when you use a macro or directive: each time the MHT processor finds two or more hash- signs at the beginning of a macro or a directive, the macro or directive is written unchanged to the output, with just one hash- sign being removed.

Using delayed macros:

```
#def macro1      This is a test!
##def macro2     <#macro1>
<#macro1> <###test|<##macro1>>
```

The output will be expanded to:

```
#def macro2      This is a test!
```

```
This is a test! <##test|<#macro1>>
```

2.2.6 Predefined macros

When the MHT processor is initialized, a set of macros is already for your use predefined:

- `<#short_date>` is the actual date in the format `mm/dd/yy`, `<#kurzes_datum>` is the corresponding German form of `<#short_date>` in the format `dd.mm.yy`.
- `<#long_date>` is the actual date in the format „weekday, month and day of month, year“, `<#langes_datum>` is again the corresponding German form of `<#long_date>` in the format „Wochentag, Tag des Monats Monat, Jahr“.
- `<#time>` is the current time in the form „hh:mm“
- `<#space>` is a whitespace character
- `<#crlf>` is a newline character \n
- `<#tab>` is a tabulator character \t
- `<#null>` is an empty string (in the sense of „NULL“)

Date and time macros in MHT:

```
<#short_date>
<#kurzes_datum>
<#long_date>
<#langes_datum>
<#time>
```

The output will be expanded to:

```
02/22/2001
22.02.2001
Thursday, February 22, 2001
Donnerstag, 22. Februar 2001
21:21
```

2.3 Blocks

2.3.1 Definition of blocks

A block is a named portion of text which should be either all printed or all skipped. Blocks can contain any text or HTML-code, as well as MHT macros. When you process a MHT template, everything outside a block is expanded and written only once to the output immediately, but blocks can be called over and over again.

“Empty” blocks, meaning that a block has no content but just a `#begin` and `#end` directive, are not saved by the MHT processor! Second, MHT has no namespaces, meaning that if you define macros inside a block, they can be exapanded and overridden anywhere in your MHT templates!

The `#process` directive invokes a block and inserts the content of the block with all its expanded macros in the place of the `#process` directive. The string after the `#process` directive (block name, block parameters) may contain macros which are expanded before the `#process` directive is executed.

Definition and use of blocks:

```
#begin block1
This is a block.
#end block1

#begin block2
This is another block!
#end block2

#process block1
#process block2
#process block1
```

The processed output of the example is:

```
This is a block.
This is another block!
This is a block.
```

2.3.2 Using blocks with parameters

Blocks can be called with parameters in two different methods:

Method 1:
The block parameters are separated by a pipe symbol „|“ and may contain spaces:

```
#process blockname|param-1|param-2|param-3|...|param-n
```

Method 2 (deprecated):
This method is only supported for compatibility issues to process older templates of previous MHT versions. The block parameters are separated by spaces, that’s why block parameters must **NOT** contain spaces:

```
#process blockname : param-1 param-2 param-3 ... param-n
```

You see the difference in the output of the example below, compare the lines 2 to 5, and 20 to 23. Each time the block was called with the same parameters, but first by method 1, the second time by method 2.

Invoking blocks with parameters:

```
#begin test
block parameter 1: <#test.%1>
block parameter 2: <#test.%2>
block parameter 3: <#test.%3>
block parameter 4: <#test.%4>
#end test

Test 1:
#process test|1st parameter|2nd parameter|3rd parameter|4th parameter

Test 2:
#process test|a||b

Test 3:
#process test|a

Test 4:
#process test : 1st parameter 2nd parameter 3rd parameter 4th parameter

Test 5:
#process test : a
```

The processed output of the example is:

```
Test 1:
block parameter 1: 1st parameter
block parameter 2: 2nd parameter
block parameter 3: 3rd parameter
block parameter 4: 4th parameter

Test 2:
block parameter 1: a
block parameter 2: <#test.%2>
block parameter 3: b
block parameter 4: <#test.%4>

Test 3:
block parameter 1: a
block parameter 2: <#test.%2>
block parameter 3: <#test.%3>
block parameter 4: <#test.%4>

Test 4:
block parameter 1: 1st
block parameter 2: parameter
block parameter 3: 2nd
block parameter 4: parameter

Test 5:
block parameter 1: a
block parameter 2: <#test.%2>
block parameter 3: <#test.%3>
block parameter 4: <#test.%4>
```

You should use `<#ifequal>` to prove block parameters in the same manner as you do with macro parameters to avoid unexpanded block parameters.

2.3.3 Deleting blocks

When you render your static HTML pages by using `mht2html` and the template mechanism, your content is saved in different templates, but in blocks which have the same names (automation!). If you process in the same build with `mht2html` blocks with different content, but with the same name, it is elegant to have a directive to „free“ a previously read in block completely. This is done by the `#undefblock` directive. To save memory if you have more than one block with the same name in your templates, the „old“ block is completely removed from the memory before the „new“ block is stored.

Using `undefblock` to delete a block:

```
#begin block1
This is a block.
#end block1

#process block1
#undefblock block1
#process block1
```

The processed output of the example is:

```
This is a block.
MHT-Error: Block not found!
```

Due to the fact that in line 6 the block „block1“ is deleted, a MHT error message is printed to `STDOUT`. By using the CGIMHT C-API from within a CGI program, `mht_process()` will return the corresponding error code (see the table of error codes/messages).

2.3.4 Looping blocks

The `#loop` directive processes a block n-times and expects the following parameters:

- The name of a block that should be processed,
- The name of a macro that could be used as a counter,
- A start and end index.

The string after the `#loop` directive (block name, loop parameters) may contain macros, that are expanded before the `#loop` directive is executed.

Looping a block:

```
#begin test
This is test no. <#i>!
#end test

#loop test|i|1|5
```

The loop in the example above will process the block `test` 5 times, each time the macro `i` will be registered with the current value of the counter:

```
This is test no. 1!
This is test no. 2!
This is test no. 3!
This is test no. 4!
This is test no. 5!
```

2.4 Conditionals

2.4.1 If, else and endif

The general form of conditionals is represented as:

```
#if {TRUE|1}
...
#else
...
#endif
```

This starts a run- time conditional "if" block, based on the conditional after `#if`. If the conditional is `FALSE` or `0`, all the lines up until the matching `#else` or `#endif` are filtered out of the output. If `TRUE` or `1`, all the lines up until the matching `#else` or `#endif` are processed, but if there is an `#else` then any lines between it and the `#endif` will be skipped. Conditional blocks can be nested, but all `#if`s must be paired with an equal number of matching `#endif`s within each individual template.

2.4.2 Elif

`#elif` allows to prove several conditionals in a row in the same manner as `#if`.

Using `#if-#elif-#else-#endif` conditionals:

```
#if FALSE
   This line is skipped from the output, because the condtional is false!
#elif 0
   This line is skipped from the output as well!
#elif TRUE
   #if 0
          This line is skipped from the output as well!
   #else
          This line is printed to the output.
   #endif
#elif 1
   These lines are skipped from the output, because
   there was already a true conditional!
#endif
```

The processed output of the example is:

```
This line is printed to the output.
```

2.5 Functions

2.5.1 Comparing strings with ifequal

The general form of `<#ifequal>` is:

```
<#ifequal|string 1|string 2|result 1|result 2>
```

`<#ifequal>` will compare `string 1` with `string 2`. If these two strings are equal, the entire macro is replaced by the string `result 1`, if not, the macro will be replaced by the string `result 2`. `string 1`, `string 2`, `result 1` and `result 2` can contain any text, spaces or macros. `<#ifequal>` will work if `string 1` or `string 2` are empty, `NULL` or the `<#null>` macro.

All parameters of `<#ifequal>` are fully expanded before `<#ifequal>` compares `string 1` and `string 2`. As with macros, you can cut- off missing parameters at the end instead of placing „empty“ parameter pipe symbols such as „`||>`“ in your template (see lines 8 to 11 in the output of the example below). Check especially line 6 in the MHT output, `string 1` and `string 2` really do have to be exactly equal, beginning and/or trailing spaces are not cut- off! Line 7 shows that you can also „compare“ empty strings.

Example 13: string comparison using

```
1  test 1: <#ifequal|A|A|right|WRONG>
2  test 2: <#ifequal|A|A|right>
3  test 3: <#ifequal|A|B|WRONG|right>
4  test 4: <#ifequal|A|A||WRONG>
5  test 5: <#ifequal|Bssdsdsdsds|B|WRONG|right>
6  test 6: <#ifequal|   A|              A|right|WRONG>
7  test 7: <#ifequal|||right|WRONG>
8  test 8: <#ifequal|||right>
9  test 9: <#ifequal||>
10 test 10: <#ifequal|>
11 test 11: <#ifequal>
```

The processed output of the example is:

```
test 1: right
test 2: right
test 3: right
test 4:
test 5: right
test 6: WRONG
test 7: right
test 8: right
test 9:
test 10:
test 11:
```

`<#ifequal>` is very useful when it is used in conditionals, so that the conditional doesn’t have to be „hard- wired“ in your template and can be switched during „run-time“.

Conditionals and `<#ifequal>` macros:

```
#def macro1            Steve Jobs
#def macro2            Bill Gates

#if <#ifequal|<#macro1>|<#macro2>|TRUE|FALSE>
This line is not written to the output, because macro1 doesn't equal macro2.
#else
#def macro2            Steve Jobs
#endif

#if <#ifequal|<#macro1>|<#macro2>|TRUE|FALSE>
This time macro1 equals macro2!
#endif
```

The processed output of the example is:

```
This time macro1 equals macro2!
```

2.5.2 Proving macro parameters with ifequal

To avoid unexpanded macro or block parameters when you invoke a macro with less parameters than ist definition expects, then wrap each parameter into an `<#ifequal>` macro, which checks the parameter against and empty string or the `<#null>` macro:

```
<#ifequal|<#.%1>|<#null>|There is no parameter!|There is a parameter!>
```

Please remember the igb0 example macro, which is now rewritten. Each parameter is now proved:

```
#def pic_path   /pics/
#def igb0 <img<#ifequal|<#.%1>||| src="<#pic_path><#.%1>.gif"><#ifequal|<#.%2>|<#null>|| width="<#.%2>"><#ifequal|<#.%3>|<#null>|| height="<#.%3>"><#ifequal|<#.%4>|<#null>|| alt="<#.%4>">>
<#igb0|headline|150|20|MHT is great!>
<#igb0|headline|150|20>
<#igb0|headline|||MHT is great!>
<#igb0>
```

The output looks now different:

```
<img src="/pics/headline.gif" width="150" height="20" alt="MHT is great!">
<img src="/pics/headline.gif" width="150" height="20">
<img src="/pics/headline.gif" alt="MHT is great!">
<img>
```

Because the definition for the macro igb0 is now pretty long, copy the example into a ASCII text editor where it will be much more readable than in this document.

2.5.3 Boolean operations with ifequal

`<#ifequal>` could easily be used to do boolean operations for example in conditionals. It's just a question about how you nest several `<#ifequal>` calls. The following two macros for `AND` and `OR` operations with two arguments give an example:

```
#def AND  <#ifequal|<#.%1>|TRUE|<#ifequal|<#.%2>|TRUE|TRUE|FALSE>|FALSE>
#def OR   <#ifequal|<#.%1>|TRUE|TRUE|<#ifequal|<#.%2>|TRUE|TRUE|FALSE>>
```

Each macro gets two arguments, which have to expand to either `TRUE` or `FALSE`:

```
<#AND|FALSE|FALSE>
<#AND|TRUE|FALSE>
<#AND|TRUE|TRUE>
<#AND|FALSE|TRUE>
```

The processed output of the example is:

```
FALSE
FALSE
TRUE
FALSE
```

Using the OR macro

```
<#OR|FALSE|FALSE>
<#OR|TRUE|FALSE>
<#OR|TRUE|TRUE>
<#OR|FALSE|TRUE>
```

will be expanded to:

```
FALSE
TRUE
TRUE
TRUE
```

Of course, instead of using `TRUE` and `FALSE` as arguments, you can use any macros that finally expand to either `TRUE` or `FALSE`.

2.5.4 Proving macro definitions with ifdef

The general form of `<#ifdef>` is:

```
<#ifdef|string 1|result 1|result 2>
```

`<#ifdef>` will prove if `string 1` was previously defined as a macro. If `string 1` is an existing macro, the entire `<#ifdef>` macro is replaced by the string `result 1`, if not, the macro will be replaced by the string `result 2`. `string 1`, `result 1` and `result 2` can contain any text, spaces or macros.

All parameters of `<#ifdef>` are fully expanded before `<#ifdef>` proves the definition of `string 1` as a macro. As with usual macro parameters, you can cut- off missing parameters at the end instead of placing „empty" parameter pipe symbols such as „`||>`" in your template (see lines 4 to 6 in the output of the example below).

Proving macro definitions:

```
#def macro1      test
macro1 is <#ifdef|macro1||not >defined!
macro2 is <#ifdef|macro2||not >defined!
#def macro2      blabla
macro2 is <#ifdef|macro2||not >defined!
test 1: <#ifdef|macro1>
test 2: <#ifdef|macro1||>
test 3: <#ifdef||1|2>
```

The output will be expanded to:

```
macro1 is defined!
macro2 is not defined!
macro2 is defined!
test 1:
test 2:
test 3:
```

One common error is, that when people want to check whether there exists a macro „test", they write something like:

```
#def test        This macro is exists!
<#ifdef|<#test>|TRUE|FALSE>
```

In this example, first `<#test>` would be expanded to „This macro is exists!", and then `<#ifdef>` would try to find a macro named „This macro is exists!". But such a macro doesn't exist in the example!

Remember, where ever MHT finds a macro, it tries to expand this macro, no matter where this macro appears in the text. But to check whether a macro is defined, you simply use the name of the macro as an argument for `<#ifdef>`. The example would be spelled correct:

```
<#ifdef|test|TRUE|FALSE>
```

That's why you can't use `<#ifdef>` to prove macro parameters: you would have to write something like `<#ifdef|.%1|...>`. The string „.%1" would then be misinterpreted by MHT as the name of a macro.

2.5.5 Proving block parameters with ifdef or ifequal

Whereas `<#ifdef>` cannot be used to prove macro parameters, you can both equally use `<#ifdef>` or `<#ifequal>` to prove block parameters. This is possible because while a block with parameters is processed, the MHT processor registers each block parameter as a temporary macro.

But be careful, where you would intentionally write something like:

```
#begin test
This is the first block parameter: <#test.%1>
<#ifdef|<#test.%1>|TRUE|FALSE>
#end test
```

The correct call of `<#ifdef>` is:

```
<#ifdef|test.%1|TRUE|FALSE>
```

The next example shows again the difference about how to prove block parameters by using `<#ifdef>` or `<#ifequal>`:

```
#begin test
<#ifdef|test.%1|TRUE|FALSE>
<#ifequal|<#test.%1>|<#null>|FALSE|TRUE>
#end test

#process test : hello
```

2.5.6 Proving blocks with ifblock

The general form of `<#ifblock>` is:

```
<#ifblock|string 1|result 1|result 2>
```

`<#ifblock>` will prove if `string 1` is the name of a block thats was previously read in. If `string 1` is the name of a existing block, the entire macro is replaced by the string `result 1`, if not, the macro will be replaced by the string `result 2`. `string 1`, `result 1` and `result 2` can contain any text, spaces or macros.

All parameters of `<#ifblock>` are fully expanded before `<#ifblock>` proves the definition of `string 1` as a block. As with macros, you can cut- off missing parameters at the end instead of placing „empty“ parameter pipe symbols such as „`||>`“ in your template.

Proving blocks:

```
<#ifblock|test|TRUE|FALSE>
<#ifblock||TRUE|FALSE>
<#ifblock|test|TRUE>
<#ifblock|test>

#begin test
Block test exists!
#end test

<#ifblock|test|TRUE|FALSE>
<#ifblock||TRUE|FALSE>
<#ifblock|test||FALSE>
<#ifblock|test>
```

`<#ifblock>` works quite the same as the other macro functions, so the expansion of the result above will not be discussed in detail here (try yourself instead and guess the result before!).

2.5.7 Checking substrings with isin

The general form of `<#isin>` is:

```
<#isin|string 1|string 2|result 1|result 2>
```

`<#isin>` will prove if `string 1` is a substring of `string 2`. If `string 1` is a substring of `string 2`, the entire macro is replaced by the string `result 1`, if not, the macro will be replaced by the string `result 2`. `string 1, string 2, result 1` and `result 2` can contain any text, spaces or macros.

All parameters of `<#isin>` are fully expanded before `<#isin>` checks whether `string 1` is a substring of `string 2`. As with macros, you can cut- off missing parameters at the end instead of placing „empty“ parameter pipe symbols such as „||>“ in your template (see lines 9 to 11 in the output of the example below).

Proving substrings:

```
#def macro1           car,plane,ship,train

Test 1: <#isin|plane|<#macro1>|right|wrong>
Test 1a: <#isin|plane|<#macro1>||wrong>
Test 1b: <#isin|plane|<#macro1>>

Test 2: <#isin|bike|<#macro1>|right|wrong>
Test 2a: <#isin|bike|<#macro1>|right>
Test 2b: <#isin|bike|<#macro1>>

Test 3: <#isin|||right|wrong>
Test 3a: <#isin||>
Test 3b: <#isin>
```

The output will be expanded to:

```
1  Test 1: right
2  Test 1a:
3  Test 1b:
4
5  Test 2: wrong
6  Test 2a:
7  Test 2b:
8
9  Test 3: right
10 Test 3a:
11 Test 3b:
```

2.6 Variables

MHT supports a few variables which are intended as a global switches to affect all the output without the need of special macros etc.. MHT variables are set by the `#mhtvar` directive, followed by the variable name and a boolean value, which is either `TRUE` or `FALSE`, or, `1` or `0`.

2.6.1 Removing whitespaces

MHT allows to remove all unnecessary white spaces in the output on automation. Removing white spaces will shrink the output, and so minimize download time.

- `#mhtvar killspace {TRUE|1}` all unnecessary white spaces will be removed in the output
- `#mhtvar killspace {FALSE|0}` the output is written „as is“ containing all empty lines, white spaces etc.

The default value is `FALSE`.

2.6.2 Converting German umlauts

In the beginning, MHT was a simple tool to replace German umlauts with their HTML equivalents. With the `convumlauts` variable, the conversion of German umlauts can be switched on or off in the output:

- `#mhtvar convumlauts {TRUE|1}` turns the conversion on
- `#mhtvar convumlauts {FALSE|0}` turns the conversion off

The default value is `FALSE`.

2.6.3 Writing output

You can even completely turn off any output of MHT by setting `writeoutput`. This is a rather seldom used feature and, in fact, only useful for the MHT template mechanism. If `writeoutput` is set on `FALSE`, you can still use `write` or `writeln` to write to the current output stream.

- `#mhtvar writeoutput {TRUE|1}` turns writing of the MHT on
- `#mhtvar writeoutput {FALSE|0}` turns writing of the MHT off

The default value is `TRUE`.

2.7 Writing outputfiles

2.7.1 Definition of output file types

Writing to output files is a bit more complex. It is complex, because you need the ability to write to different output files by compiling one template. This is needed to compile multi- lingual sites where you have each page in different languages. The idea is, to keep the content for all languages in the same template, and to use directives to tell MHT which content has to be written into which output file.

Writing to different output files:
```
1  Two file types will be defined and opened...
2
3  #mhtfile type ger
4  #mhtfile open ger c:\\temp\\gerfile.txt
5
6  #mhtfile type eng
7  #mhtfile open eng c:\\temp\\engfile.txt
8
9  This text is written to STDOUT because there is no filetype selected yet!
10
11 #file ger
12 This text is written to gerfile.txt
13
14 #file eng
15 This text is written to engfile.txt
16
17 #file all
18 This text is written to all open file handles.
19
20 #mhtfile close
21 All open file handles are closed! The output is now again written to
   STDOUT.
```

Line 1 ist still written to `STDOUT`. In line 3 a file type named „ger“ is defined. Any contiguous string without spaces is a valid name for a file type. For this file type is a output file opened in line 4. The output file should be specified by an absolute path. The parameters for the name of the file type or the path of the outputfile can be expanded out of macros.

If you opened an outpfile, but did not choose yet a certain file type to write into, all the output is still written to the default output sink, such as in line 9.

The outputfiles are selected by the `#file` directive, followed by the name of a file type that you defined before. Line 12 is written to the file gerfile.txt, line 15 is written to the file engfile.txt. The name „all“ is a reserved name for a file type. If you choose „all“ as a file type, all following lines are written to all open output files. The file type „all“ is reserved and may not be used as a valid name for a file type.

Not before you call `#mhtfile close` as in line 20, all the output is **ONLY** written to the outputfiles, either a single outputfile, or all outputfiles. Afterwards you can print to `STDOUT` as usual.

Set `writeoutput` to `FALSE` to disable any output between the opening of the file type(s) and the selection of a specific file type (lines 8-10 in the example above).

2.7.2 Writing to STDOUT

If you need to print to STDOUT after you have choosen a filetype, for example after line 11 in the example above, you can use the `#echo` or `#echoln` directive to print to `STDOUT`.

Printing to `STDOUT`:

```
#echoln This line is printed to STDOUT with a trailing new line character
#echo This line is printed to STDOUT without a new line character
```

2.7.3 Writing files with disabled output

If you set `writeoutput` to `FALSE`, `write` or `writeln` will nevertheless still write its output lines, similar to `echo` or `echoln`. This is useful if you wish to hide too many empty lines by setting `writeoutput` to `FALSE`, but still wish now and then to write to the output stream.

Disabling output:

```
#mhtvar writeoutput FALSE
#writeln This line is printed to the current output stream(s) with a trailing new line character
#write This line is printed to the current output stream(s) without a new line character
This line is not printed, because writeoutput is disabled.
#mhtvar writeoutput TRUE
This line is again printed to the output
#writeln Even #writeln will now still write its output!
```

2.8 Miscellaneous

2.8.1 Including templates

The `#include` directive includes the content of a file given by an absolute path. It is essentially an automated copy and paste, except that it does **NOT** alter any of the source files, it only changes what is written to the output stream. Files which are included can include other files, and so on. Circular inclusion (A includes B which includes A) will be quietly ignored!

The path and file name can be expanded out of macros. MHT has no namespaces, meaning that if you define macros inside a template A, they can be exapanded and overridden anywhere in your MHT templates!

A template including another template:

```
1   #include c:\\temp\\includefile.mht
2   #process block1
```

The source of includefile.mht:

```
1   This text comes from the included file!
2
3   #begin block1
4   This line comes from a block inside the included file.
5   #end block1
```

As with any MHT template, any text outside a block (lines 1-6) is processed immediately when the template is loaded. Thats why you should place everything into blocks in include files, to process the text when it is really needed, and not when it is loaded.

2.8.2 Interruptions

The `#pause` directive is only relevant to `mht2html`. It simply pauses `mht2html` wherever it is placed inside a template until the enter key is pressed, then `mht2html` will continue. This is quite useful to set „breakpoints“ inside MHT templates to control the values of macros etc. during the development process of a template. **NEVER** use `#pause` in templates that are processed by CGI programs!

2.8.3 Exit

The `#mhtexit` directive is only relevant to `mht2html` when you want to terminate MHT explicitly. **NEVER** use `#mhtexit` in templates that are processed by CGI programs!

3 The template mechanism

3.1 Overview

The template mechanism described here is just a description of the template mechanism that I have developed for my personal use. You can take it as a basis to customize it for your own needs, or to add new functionalities such as rendering WAP pages etc.. The template mechanism is currently able to render each HTML page in 12 different layouts/versions on automation out of the same content files:

Layouts:

- „frame": the page is rendered as a frameset. All necessary framesets and frames (navigation, sub-framesets etc.) to make the frameset complete are rendered on automation.
- „plain": the page is rendered into a single frameless HTML page. The content of the other frames is then rendered into HTML table cells instead of single frames.
- „text": this layout could be used for simple web clients such as lynx. All image tags are rendered as text links, there are no images in the HTML pages.

Versions:

- „final": all newline characters are removed from the resultant HTML pages, which will make the files smaller and so decreased download times. You should compile the pages as „final" for your production system.
- „work": compiles the pages the same as „final", except that all newline characters are not removed, which makes the HTML code much more readable. You should compile the pages as „work" for staging versions or development purposes.
- „offline": compiles the pages the same as „work", except that all hrefs and image tags in the resultant HTML pages are adjusted so that they can be opened from a local file system.

Further more:

- each page can be compiled into a CGIMHT template to be used by a CGI program
- each page/CGIMHT template can be rendered in up to 8 languages

Due to the large amount of layouts/versions, it is possible that dozens of HTML pages will be created by compiling just a single page, all on automation!

3.2 How to set up a new project

To begin with, setting up a new project with all style and layout definitions is pretty difficult. Therefor, adding content pages to an existing project will save you a lot of time! All content and layout templates of your website are packaged in a „project", and every project should have a unique name. Let's assume that the project described here is simply named „demo". To expose the project name in macros and files names, it is written in *cursive font*.

Since Unix gurus know their computer a lot better, I will give the following explanations only with Windows examples to help users to get their first MHT project running.

3.2.1 How to compile a MHT project

By using the template mechanism, a set of source MHT files is compiled into the resultant HTML pages. The source MHT files are split into these directories/groups:

- „script“ MHT files, these files control the page compilation process,
- „style“ MHT files, that keep all the layout definitions and commonly-used HTML sequences,
- „data“ MHT files, that keep the content (which will be rendered between the `<body>...</body>` tags),
- „user“ MHT files, where each user is able to define which MHT pages of a project he wants to compile (pretty useful if you work in a team on the same project!).

`mht2html` gets the info about where it can find these files by reading some environment variables:

- „MHTSCRIPTPATH“ is the directory of the „script“ MHT files,
- „MHTUSERPATH“ is the directory of the „user“ MHT files,
- „USERNAME“ (under Windows) or „USER“ (under Unix), which is the user’s login name.

Please note that these environment variables are case-sensitive! The MHT template mechanism won’t work if these variables are not set in your environment. These environment variables are registered as MHT macros when `mht2html` is invoked. The paths of the „style“ and „data“ files are defined as MHT macros in `main-`*demo*`.mht`, a file that keeps all the specific settings of your project (similar to a `MAKEFILE`). To make it easy, you should create a batch file or shell script for each of your MHT projects that sets these environment variables correct. The example project assumes that all MHT files are stored in `c:\mht\`. You should adjust this path to your local settings.

An example batch file (`compile_`*demo*`.bat`) to compile the demo project:

```
@echo off
SET MHTSCRIPTPATH=C:\\mht\\scripts\\
SET MHTUSERPATH=C:\\mht\\users\\
@c:\mht\bin\mht2html -p c:\\mht\\projects\\demo\\mht\\style\\main-demo.mht
@pause
```

`mht2html` is invoked by using –p with the absolute path to the project’s „main“ file. This file is described in a following section of the documentation.

3.2.2 Directory structure

The following graph gives a good overview about the recommended directory structure about how the files of a MHT project should be organized in the file system:

```
mht/
├───bin/
│       mht2html(.exe)
│       compile_demo(.bat)
│
├───projects/
│   └───demo/
│       ├───html/
│       │   └───final/
│       │       ├───cgibin/
│       │       ├───deutsch/
│       │       │   ├───frame/
│       │       │   ├───plain/
│       │       │   └───text/
│       │       └───...
│       └───mht/
│           ├───data/
│           │       data1.mht
│           │       ...
│           │
│           └───style/
│                   main-demo.mht
│                   ...
│
├───scripts/
│       compile.mht
│       ...
│
└───users/
        user1.mht
        ...
```

In the `mht/bin/` directory you should place the `mht2html` compiler along with all batch files/shell scripts to compile your projects.

The `mht/projects/` directory has a sub-directory for each MHT project. You should place at least the „style“ and „data“ MHT files of your projects in sub-directories there.

In the example directory tree above, it is assumed that the resultant HTML pages are compiled into a sub-directory of `mht/projects/demo/html/`. Each version is then compiled into a separate sub-directory, and again each layout is compiled into a sub-directory. The paths of these directories are defined in the „main“ file of your project. Of course, you can compile the HTML pages directly into the document root of your webserver(s).

The `mht/scripts/` directory keeps the MHT files which control the page compilation process. **Don’t edit these files unless you know exactly what you are doing!**

Every user who compiles pages of a MHT projects needs to have his own compile file in `mht/users/`, which has to be named `username.mht`, where „username“ is the users’s login name. Please note that the username is case-sensitive! If you are uncertain about the correct spelling of your username, then open a command prompt (under Windows), and type `set`. You will get a list of all environment variables, where you should also find a line beginning „`USERNAME=...`“. Under Unix you will get a list of all environment variables by typing `env`. You should then find a line „`USER=...`“.

3.2.3 The MHT scripts of the template mechanism

This section I will give you a short overview of the MHT files of the template mechanism in the `mht/scripts/` directory. **Don’t edit these files unless you know exactly what you are doing!** Any changes on these scripts will immediately affect how the HTML files will be rendered.

- compile.mht

This scripts controls the main flow to compile the pages. It has a block `compile-page` which gets just one parameter: the name of a MHT source file in the project’s `data/` directory.

- copyright.mht

Keeps only one block with a short copyright notice which will be printed into the footer of every HTML page. Instead of changing the content of this block, you should change the settings in your project style to use another customized copyright notice.

- defaults.mht

For every layout version which the template mechanism is able to render, a separate block exists to define which blocks are used to build a HTML page. The macros which are defined in these blocks are used like „pointers“:

```
#def plain-copyright                              copyright-default
```

The line above (inside the block `set-plain-defaults`) means, that to print the copyright notice in plain HTML documents the block `copyright-default` is invoked, which itself is defined in `copyright.mht`. Each time before a new page is rendered, the block `set-plain-`***`version`*** is invoked, to initialize the macros with default values of the blocks to be used. To use customized blocks, you override these macros in your project's style definition. The macros which you can override are exactly the macros inside the `set-plain-`***`version`*** etc. blocks.

- frame.mht, frameset.mht, plain.mht and text.mht

These scripts contain the blocks to render a specific layout version. They are invoked from blocks in `compile.mht`.

- init.mht

Init does all the required pre-processing to compile the pages, such as reading in all MHT scripts of the template mechanism, the project's style definitions, the user files to determine which pages should be compiled, and to set and check all required MHT macros.

- javascript.mht, macros.mht and metatags.mht

Here you should place all the project independent JavaScripts, macros and metatags for your pages. You would rather seldom edit these files.

- paths.mht and settings.mht

In `paths.mht`, all the path related macros are defined with default values, the same goes for `settings.mht`. Override these macros in your project's main file.

- streams.mht

This MHT script has all the functionality to define, open and close the file streams to write the rendered HTML into files inside the appropriate directories.

3.2.4 The main file to compile a project

Again, the `mht2html` compiler is invoked with the absolute path of the project's main file (`main-`***`demo`***`.mht`). The main file is the „root“ of the entire compilation process, here you define the „global“ configuration of your project. All the settings about how the pages should be rendered and where the resultant HTML pages should be written are defined there. The following tables give an overview, by which macros the settings of the project are adjusted.

Here you will find a description of each macro (out of `paths.mht` and `settings.mht`) which can be overriden in your project's main file

Macro	**Default value**
`COMPILE_FINAL`	`FALSE`
`COMPILE_WORK`	`FALSE`
`COMPILE_OFFLINE`	`TRUE`
`make_frame`	`TRUE`
`make_plain`	`FALSE`
`make_text`	`FALSE`
`max_file_streams`	`8`
`max_frames`	`8`

`COMPILE_FINAL`, `COMPILE_WORK` and `COMPILE_OFFLINE` are macros that determine which version should be compiled. When you compile a project, you set only one of these macros on `TRUE`, all the other macros on `FALSE`.

If `COMPILE_FINAL` is `TRUE`, mhtvar `killspace` is set on `TRUE`, which will remove all newline characters in the resultant HTML pages. This should help to minimize download times. `COMPILE_WORK` will compile the pages the same as `COMPILE_FINAL`, but with all newline characters inside.

By compiling a project with either `COMPILE_FINAL` or `COMPILE_WORK` set on `TRUE`, the HTML pages will be rendered with href-links and image-paths to be downloaded from a webserver. The settings of `COMPILE_OFFLINE` are used to compile HTML pages to be loaded from a local file-system.

The macros `make_frame`, `make_plain` and `make_text` tell the template mechanism, which layout version of the HTML pages should be rendered. The may all be set on `TRUE`, to compile in one process all layout versions. At least one of these macros has to be set on `TRUE`, otherwise no pages will be compiled. `make_text` will compile the HTML pages without any images. Instead of `<img>` tags, an href of the hidden image is rendered to load the image over a link.

`max_file_streams` and `max_frames` give default values for the maximum number of allowed file types (=language versions of the HTML pages!) and frames inside a frameset. The maximum number of allowed file types is limited by the MHT processor to 64 file handles. But nevertheless you should set `max_file_streams` on a useful value.

The default values for these macros are defined in `settings.mht` (in `mht/scripts/`). Have a look to this file to see which macros you can define/override, but do not edit these values there, override them in your project's main file instead!

Path and file settings:

Macro	**Default value**
`fileroot`	`<#null>`
`wwwroot`	`/`
`picpath`	`<#wwwroot>pics/`

framedir	frame/
plaindir	plain/
textdir	text/
cgidir	cgibin/
cgipath	<#wwwroot><#cgidir>/
javapath	<#wwwroot>applets/
html_postfix	.html
mht_postfix	.mht

These macros define the paths settings, both where the resultant HTML pages are written and for page navigation through links in the HTML pages.

`fileroot` is the absolute path where the compiled HTML pages should be written on the filesystem of your disc. You do have to define a correct value for this macro in any case!

`wwwroot` is a relative path from where the resultant HTML pages are referred by the given document root. Usually, `wwwroot` remains as `/`, then all the HTML pages are referred directly from the document root of your webserver. If the HTML pages should be downloaded via URLs such as `http://<domain>/<subdirectory>/index.html`, you do have to define `wwwroot` as the path of `<subdirectory>`.

`framedir`, `plaindir` and `textdir` are the names of the sub-directories, where the HTML pages of these layout versions are written. If one of these macros is defined as `<#null>`, the resultant HTML pages will be written directly into `wwwroot` (which is finally `fileroot` if `wwwroot` is also defined as `<#null>`).

The default values for these macros are defined in `paths.mht` (in `mht/scripts/`). Have a look to this file to see which macros you can define/override, but do not edit these values there, override them in your project's main file instead!

3.2.5 An example main file

Now we can discuss an example project main file in detail:

```
#mhtvar writeoutput                    FALSE


#begin main
   #def project                        demo

   #def COMPILE_WORK                   FALSE
   #def COMPILE_FINAL                  FALSE
   #def COMPILE_OFFLINE                TRUE

   #def make_frame                     TRUE
   #def make_plain                     TRUE
   #def make_text                      TRUE

   #def file1                          ger
   #def file2                          eng

   #def gerdir                         deutsch/
```

```
19     #def engdir                       english/
20
21     #def projectpath                  c:\\mht\\projects\\demo\\mht\\
22
23     #if <#COMPILE_OFFLINE>
24            #def wwwroot               c:\mht\projects\demo\html\offline\
25            #def fileroot              c:\mht\projects\demo\html\offline\
26     #elif <#COMPILE_WORK>
27            #def wwwroot               /
28            #def fileroot              c:\mht\projects\demo\html\work\
29     #elif <#COMPILE_FINAL>
30            #def wwwroot               /
31            #def fileroot              c:\mht\projects\demo\html\final\
32     #endif
33  #end main
34
35
36  #begin projectfiles
37     #process include-file : <#_STYLEPATH>compile-demo.mht
38     #process include-file : <#_STYLEPATH>style-layouts-demo.mht
39     #process include-file : <#_STYLEPATH>style-elements-demo.mht
40  #end projectfiles
41
42
43  #include c:\\mht\\scripts\\init.mht
```

- In line 1, writeoutput is set on `FALSE`. This has to be done to avoid unncessary empty lines in the output of the MHT processor during page compilation. If writeoutput is not set on `FALSE`, the output of the MHT scripts which control the page compilation process might look terrible in your console.
- All the poject settings are done in a block `main`, which is processed by the template mechanism.
- In line 5, the name of the project („demo“) is defined. The template mechanism will use this name to load project specific files.
- In line 7-9 follow three macros that define which version will be compiled the next time `mht2html` is invoked for this project. Actually, the pages will be compiled as a offline version. Only one of these macros may be set on `TRUE`.
- In line 11-13 is defined which layout versions should be rendered. At least one of these macros has to be set on `TRUE`. Otherwise no output will be created.
- Because there are no definitions for the directories of the layout versions (`framedir`, `plaindir`, `textdir`), the default values are used (see the table in the section before). If you wish to compile the different layout versions into other directories, you would have to override these macros.
- The resultant HTML pages should be rendered in two languages: german and english. Therefore two filetypes which named ger and eng are defined in line 15-16. Without the definition of any file type in the project’s main file, the template mechanism would register at least one „default“ file type named „ger“ (for „german“ HTML files). So even if you have just one language in your HTML files, it would make sense to define one file type for your certain language. It is assumed that you use common abbreviations for the language versions, such as „eng“, „fra“ etc..
- The directories where the files of these two filetypes are written are defined in line 18-19.

- In line 21 the `projectpath` is defined. This path is essential, since the template mechanism tries to load all project files from this root. It expects a `style/` and `data/` sub-directory there.
- In the conditional in line 23-32, the `wwwroot` and `fileroot` are correct defined, depending on which version should be compiled as defined in line 7-9.
- The block `projectfiles` in line 36-40 is a block which is processed by the template mechanism. For each file in the `style/` directory, except the main file, you have to add a line, so that the template mechanism can load these files during the compilation process. In these files, you should keep all the blocks of your style and layout defintions, along with macros and any other blocks which are project specific.
- Line 43 includes the init MHT file of the template mechanism, which includes the code to start the page compilation.

3.2.6 Style and layout definitions (Intro)

The template mechanism renders a HTML file out of several page parts, such as metatags, JavaScripts, the content itself or copyright notices in the page footer etc. During the compilation process, the template mechanism searches for macros, which tell the template mechanism which block to use next to render a page part. As mentioned before, in `mht/defaults.mht` are default values of blocks for each page part defined.

When the template mechanism compiles a page, first the blocks in `mht/defaults.mht` are pocessed, to set the macros on default values. Afterwards blocks in your project's style definition are processed, to override these default settings, to use template blocks out of your own project files. Each of these blocks has to be placed in a file which is included in the „projectfiles" block in your project's main file.

These template blocks are handled different by the template mechnism than the data files which keep the content of your HTML files. Because `writeoputput` was set on `FALSE` in the first line of the project's main file, every line in a block that is part of a style or layout defintion, has to start with a `#writeln` directive to print the line to the output stream.

Only when the content of a data file is rendered, `writeoutput` is set on `TRUE`, and afterwards again on `FALSE`. This is also the case in every javascript or metatags block to be rendered into the HTML files. Whenever you edit a block in which the lines start with a `#writeln` directive so that the output appears in the HTML files, you know for sure that you edit something which is part of your style and layout definition and not of your content!

3.2.7 Style definitions (Overview)

The style and layout definitions are split up into two files: the file `style-elements-`***`project`***`.mht` (in your project's `style/` directory) keeps the macros which define which blocks to use to render the HTML. Second, the file `style-layouts-`***`project`***`.mht` contains HTML templates with page parts of the resultant HTML files. The MHT template mechanism takes these style informations together with the HTML template page parts, to generate the HTML.

The frame version of each page in the demo project consists of a „top“ frameset, which contains a „head“ frame and below another „sub“ frameset. In the top left corner of the „top“ frameset is a single frame „logo“ placed, which sould keep a company logo etc. The „sub“ frameset contains two frames „nav“ (for the navigation“), and „body“, which is the document itself.

In our demo project, the file `style-elements-demo.mht` might look like this:

```
#begin page-elements-demo
   #def frameset1                  top
   #def frameset2                  sub
   #def frame1                     body
   #def frame2                     head
   #def frame3                     nav
   ...
#end page-elements-demo


#begin frameset-elements-demo
   #def top-frameset-metatags       top-frameset-metatags-demo
   #def frameset-head-elements      frameset-head-elements-demo
   ...
#end frameset-elements-demo


#begin frame-elements-demo
   #def body-frame-metatags         body-frame-metatags-demo
   #def frame-head-elements         frame-head-elements-demo
   ...
#end frame-elements-demo


#begin plain-elements-demo
   #def plain-metatags              metatags-default
   ...
#end plain-elements-demo
```

Important notice: the template mechanism expects blocks with exactly these names, only the name of the project/style (here: `demo`) is variable in the block and macro names!

3.2.8 Style definitions (Explanation)

As you can see, the block `page-elements-`***`demo`*** simply defines for each page part a macro with its „real name“. This block is required!

To define your page parts for a frameset, you always need to keep two rules in mind:

1. frameset1 is always the top most frameset,
2. frame1 is always the „body“, meaning the document with the content.

What follows are blocks which tell the MHT template mechanism which blocks should be used to build the HTML. For each layout version you may implement one block *`layout`*`-elements-`*`style`*, such as `frameset-elements-demo`, `frame-elements-`***`demo`***, `plain-elements-`***`demo`*** and `text-elements-`***`demo`***. These blocks are optional! If you don’t implement a block for a certain layout version, the default blocks would be used as defined in `mht/scripts/defaults.mht`.

The following table explains each macro which can be overriden for a certain layout version in your style definition. The „xxx“ should be replaced by „frameset“, „frame“, „plain“ or „text“ to set the style definition for a layout version. Please note again, the value of each macro mentioned in the table is the name of a block to be invoked by the MHT template mechanism during page compilation:

Macro	**Explanation**
xxx-page-start	The beginning of the HTML page, `<html>` and everything up to the `<head>` tag. The `<head>` tag is rendered by the template mechanism.
xxx-metatags	The block with the metatags.
xxx-javascript	Dito for the entire javascript in the page head.
xxx-head-elements	Any additional head elements, such as `<script>` tags to include other javascripts or CSS definitions.
xxx-body-start	The opening `<body>` tag.
xxx-body-end	The closing `</body>` tag.
xxx-copyright	A block with the copyright notice.
xxx-page-end	Everything up to the, including the closing `</html>` tag.

If you never want to render a page element into the HTML files of a certain layout version, you define the macro of this page part as `<#null>`. For example if you don’t want to use a copyright notice in all your „top“ framesets, your would define the copyright macro this way:

```
#begin frameset-elements-demo
   #def top-frameset-copyright            <#null>
   ...
#end frameset-elements-demo
```

To be able to render metatags, javascripts or additional head elements only into specific framesets, but not into the „body“ document, you can define in the frame and frameset „`-elements-`„ blocks three additional macros (`xxx` should be replaced by the name of a frameset as defined in `page-elements-`***`demo`***):

- `xxx-frameset-metatags`
- `xxx-frameset-head-elements`
- xxx-frameset-javascript

Example:

```
2  #begin frameset-elements-demo
3     #def top-frameset-metatags       top-frameset-metatags-demo
4     #def frameset-head-elements      frameset-head-elements-demo
5     ...
6  #end frameset-elements-demo
```

In line 2 we defined to render the metatags into all „top“ framesets by using the `block top-frameset-metatags-`***`demo`***. This block should be defined in the file `style/metatags-demo.mht` of your project. This is one of the three optional style macros mentioned above for framesets. The macro `frameset-head-elements` in line 3 is a macro which is already defined in `defaults.mht`, which is now with a new value overriden.

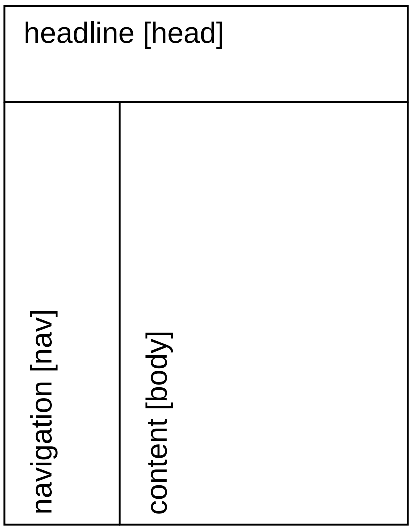
The same goes for the „frame-elements-xxx“ block for each frame. In case of a plain document, the body tag is hardcoded in the layout (discussed later), but in case of a frame, we need a 4th macro to set a customized `<body>`-tag. You cann add these macros in your style definition:

- `xxx-frame-metatags`
- xxx-frame-head-elements
- `xxx-frame-javascript`
- `xxx-frame-body-start` (This block includes only 1 line: the `<body>`–tag!)

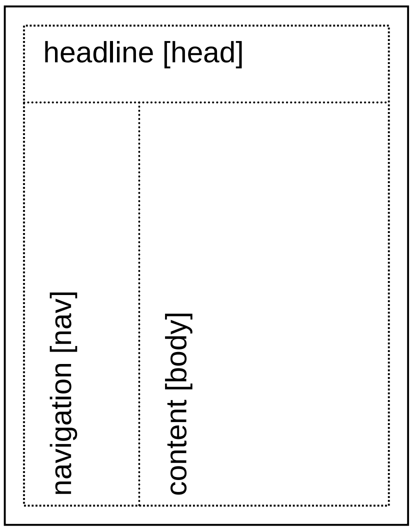
3.2.9 Layout definitions (Frame documents)

We have already defined which blocks should be used as templates to render the specific page parts of the HTML. Now we do have to explain how these blocks should look like, so that the MHT template mechanism can fill them with the right content. The layouts of the page parts are defined in `style-layouts-`***`demo`***`.mht` in your project's `style/` directory.

The following graph illustrates again the idea to render frame and plain documents out of the same source files:



„frame“ version: example layout of a page in a frameset.



„plain“ version: the same layout of the page in a table.

To make this work, the source files containing the content may not contain any `<body ...>` tags, but just the „real“ content between the opening and closing body tags. The body tags are part of the style definitons instead!

When a „frame“ page is compiled, the template mechanism will process for each frameset (as defined in the block `page-elements-`*`style`* in your style definition) a block `xxx-frameset-`*`style`*. In our demo project, we have to define two blocks `top-frameset-`*`demo`* and `sub-frameset-`*`demo`*, because we defined two framesets „top“ and „sub“. Instead of discussing a frameset in detail, I will just explain how to set up the frame tags in a frameset correct. You should refer to the demo project’s style-layout definition to see how the frameset and plain layout of the demo website is defined.

3.2.9.1 Frame tag for a non-content frame

A „non-content“ frame means, a frame tag for any frame which is not the content or „body“ frame itself. In our demo project, these frame are „head“ and „nav“. If you get back to the style definition, you will see that „frame2“ is the head frame:

```
<frame name="<#frame2>" src="<#<#_TYPE>path><#<#frame2>><#_HTML_POSTFIX>"
scrolling="no" noresize>
```

`<#<#_TYPE>path>` and `<#_HTML_POSTFIX>` are internal macros of the template mechanism that are build by recursion out of many other macros. You just insert the correct frame definition, such as frame2, frame3 etc. to load the correct frame.

3.2.9.2 Frame tag for a sub-frameset

In frameset1, the „top“ frameset, we need a fram tag to link to the „sub“ frameset:

```
<frame name="<#frameset2>"
src="<#<#_TYPE>path><#_NAME>_<#frameset2><#_HTML_POSTFIX>" border=0
frameborder=0 framespacing=0>
```

Again, you just insert the correct frameset definition such as frameset2, frameset3, to load the correct frameset. Any other macros inside the src attribute of the frame tag are internal macros of the template mechanism.

3.2.9.3 Frame tag for the content („body“) frame

The frame tag for the content frame is different, because in case of the content frame, the src attribute has to be rendered different:

```
<frame name="<#frame1>"
src="<#<#_TYPE>path><#_NAME>_<#frame1><#_HTML_POSTFIX>">
```

You should compile a document by using the frame layout and check the file names of the resultant HTML files, to see why. If the name of your page is „index“, and you have two framesets per frame page in your layout defined named „top“ and „sub“ as in our demo project, the template mechanism will generate three files: „index.html“ for the „top“ frameset, „index_sub.html“ for the „sub“ frameset, and „index_body.html“. Please note, that the names „_sub“ and „_body“ in the file names are taken from the style definition in the block `page-elements-`*`demo`*.

3.2.10 Layout definitions (Plain documents)

For each plain layout version, which is „plain“ or „text“, the template mechanism will process a block `xxx-layout-`*`demo`*, for example `plain-layout-`*`demo`*, to generate a plain document. The layout definition for plain documents is explained by the text layout of the demo project:

```
#begin text-layout-demo
#process head-doc

<hr>

#process nav-doc

<hr>

#process body-doc
#end text-layout-demo
```

In `page-elements-`***demo*** we defined that each page consists of four page elements: „body“ (frame1), „head“ (frame2) and „nav“ (frame3). To insert the content of the data file of a specific page elements, the template mechanism processes a block `xxx-doc`. For example, to render the content of the „body“ document into a layout, the template mechanism processes the block „body-doc“.

The layout for „text“ documents in the example above looks a bit over simplistic, but you can put there a sophisticated table layout as you like. Check the plain layout of the demo project for example (`plain-layout-`***demo***).

3.3 Adding new pages to a existing project

To begin with, creating new pages to be compiled with MHT is **a lot** easier than setting up a new layout structure. To add new pages to an existing project, you would first have to write the data file(s) for your new page(s). Then you add the new page to the right compile script, so that with the next compilation process it is also compiled with all the other pages.

3.3.1 Skeleton of a data file

Each data file for a new page is generally split into two parts: a „data“ part, which defines macros that give the template mechanism all the required informations about the name, title, style and other page elements of this page. The „doc“ part is the document itself, everything between the body- tags as if you would create a HTML page by hand.

You should try to use as many re-usable HTML fragments in your documents as possible, either as macros or blocks. You would place new HTML fragments which you could use in many other projects in the „macros.mht“, „javascript.mht“ or „metatags.mht“ scripts in the `mht/scripts/` directory. HTML fragments which are project specific, should be placed inside a MHT file which is included in the `projectfiles` block in the project’s „main.mht“ file.

The MHT file below is a rather simple, but functional example for a minimal data file:

```
#begin body-data
   #def name           test1
   #def title          MHT Demo Site: test1
   #def style          demo
   #def head           head_1
```

```
6      #def nav            nav_1
7   #end body-data
8
9
10  #begin body-doc
11
12  <h1>Hello World!</h1>
13
14  This is a example page to be compiled as a content page.
15
16  #end body-doc
```

Please note that the names of the blocks in the data file depend on which page element this data file describes. In the example above, the blocks are named „body-data“ and „body-doc“. Go back to the style definition example and see the lines 4-6, „body“ is a valid name for a page element. If you would write a new page for a navigation („nav“) page element, the block would be named „nav-data“ and „nav-doc“.

The same names for these page elements are used inside the „data“ block again, to tell the template meachanism which pages should be used for the other pages elements such as „head“ and „nav“ to compile a complete page or frameset. For example in line 6 in the example above is defined, that for the „nav“ page element the page „nav_1.mht“ is used (the „.mht“ is omitted!).

The macro „style“ sets which project style should be used to render the HTML pages. This makes it possible, to render the same content by using different styles. Finally, the macro „title“ is really self explanatory: is sets the title tag of the page. The name macro defines the filename of the resultant HTML page.

3.3.2 Skeleton of a data file with custom JavaScript

In the style definition we defined which JavaScripts should be rendered into certain page elements or not. If you want to render some JavaScript just into one page and nowhere else, you would have to set the definitions in the data block of the specific page by using the macro `page-javascript-special` as shown below:

```
#begin body-data
   #def name           test1
   #def title          MHT Demo Site: test1
   #def style          demo
   #def head           head_1
   #def nav            nav_1
   #def page-javascript-special     javascr-special
#end body-data


#begin body-doc

<h1>Hello World!</h1>

This is a example page to be compiled as a content page.

#end body-doc


#begin javascr-special
alert('Hello!');
#end javascr-special
```

When the template mechanism renders the JavaScripts in the head part, it checks whether it can find a macro `page-javascript-special`. In line 7 in the example above is defined, that inside the JavaScript part in the head of the resultant HTML page the content of the block `javascr-special` should be added.

Please note: the template mechanism supports custom JavaScripts currently only for page elements defined as „frame1“ in the style definition! If you wish to use custom JavaScripts per page also in other pages (such as frame2, frame3, etc. pp.), you would have to place them between `<script>`- tags inside the `xxx-doc` block of the page. It’s not an elegant method to place JavaScripts in the body of a HTML page, but this will just work fine until the template mechanism has been improved.

3.3.3 Skeleton of a multi-lingual data file

Remember, in line 15-16 in the project’s main file, we defined two file types for different languages, „ger“ to render german language versions, and „eng“ to render english language versions. You don’t have to write separate data files for each language versions, instead you switch between languages inside the same data file. This is done by invoking the `#file` directive with a valid file type as shown in line 11, 15 and 19 in the example below:

```
1  #begin body-data
2     #def name           test1
3     #def gertitle       MHT Demo Site: test1 (deutsch)
4     #def engtitle       MHT Demo Site: test1 (english)
5     #def style          demo
6  #end body-data
7
8
9  #begin body-doc
10
11 #file ger
12 <h1>Hallo Welt!</h1>
13
14 Dies ist ein Beispiel für eine Content-Seite.
15 #file eng
16 <h1>Hello World!</h1>
17
18 This is a example page to be compiled as a content page.
19 #file all
20
21 This line will be rendered into all language versions...
22 Diese Zeile wird in alle Sprachversionen eingefügt...
23
24 #end body-doc
```

In a multi-lingual project, the title macros in the data blocks of each page have to be adjusted to enable different title tags for each language version, otherwise the same title tag would be rendered into each language version. You set the title for a specific language as shown in line 3-4 in the example above.

3.3.4 Skeleton of a data file for a CGIMHT template

As mentioned before, one important feature of the template machanism is the ability to render again MHT templates for CGI programs by using the same style and data files.

A page for a CGIMHT template needs inside the data block one additional macro `cgi-name` for the file name of the resultant CGIMHT template. The file name of the CGIMHT template will be rendered as `<#cgi-name>-<#version>-<#lang>.mht`, as for example `test1-plain-ger.mht`:

```
#begin body-data
   #def name           test1
   #def cgi-name       test1
   #def title          MHT Demo Site: test1
   #def style          demo
#end body-data


#begin body-doc

<h1>Hello World!</h1>

#process cgi-page-split

This is a example page to be compiled both as a content and cgimht page.

#end body-doc


#begin body-cgi-doc

##begin test
This is a block to be processed by a CGI program...
<##date>
##end test

#end body-cgi-doc
```

Second, the content of the doc block needs to be split up into two blocks `page-head` and `page-foot`. This is done by processing the block cgi-page-split. The idea is, to process first the „static" block `page-head` block from your CGI, then any dynamic page content (lists, forms, etc.), and then again the „static" block `page-foot` to close the page.

Any additional page elements or blocks which should be read in and processed by the CGI program only have to be encapsulated inside a block `body-cgi-doc`. Again, you need to place the right name of the page element as defined inside the project's style definition in front of the block `xxx-cgi-doc`. Please refer back to the section „Delayed macros and directives" to understand why and when you would have to use delayed macros or not inside the block `xxx-cgi-doc`.

```
#begin page-head
<head>
...
</head>
<body ...>

<h1>Hello World!</h1>

#end page-head

#begin page-foot
```

```

This is a example page to be compiled both as a content and cgimht page.

</body>
</html>
#end page-foot


#begin test
This is a block to be processed by a CGI program...
<#date>
#end test
```

3.3.5 How to compile pages

Finally, each new page has to be added to your project's compile file (in your project's `mht/style/` directory). This file must contain one global block name `compile-xxx`, where `xxx` is the name of your project as defined in the projetc's main file. You are free to split your files up into several compile blocks, but make sure that each of these blocks is invoked by a `#process` directive from the global compile block.

```
1   #begin compile-demo
2      #process compile-cgi : 1.mht
3      #process compile-page : 1.mht
4   #end compile-demo
```

In the example above in line 2, a page is compiled as a CGIMHT template, and in line 3 as a „normal“ HTML page to be viewed in the browser. Whenever you want to compile a page as a CGIMHT template **and** as a „normal“ HTML page, you would have to compile it really two times!

Second, you need to adjust your own compile file. Usually you want to compile only specific pages, and not the entire project. That's why users may maintain their own compile file in `mht/users/`. There, you simply invoke there blocks from the project's compile file.

Please note, that the string „your login name“ really IS the name with which you log into your system. Check your environment (via „set“ or „env“) to see whether your environment includes your user name. MHT expects a environment variable „USER“ under Unix/Linux, and „USERNAME“ under Win32:

```
#begin compile-your login name
#if <#ifequal|<#project>|demo|TRUE|FALSE>
   #process compile-demo
#endif
#end compile-your login name
```

4 The CGIMHT library

4.1 Introduction

In this chapter I will give you an overview on how to implement CGI programs written in (ANSI-) C using CGIMHT. The common gateway interface, or CGI as people know it, is a standard by NCSA, see `http://hoohoo.ncsa.uiuc.edu/cgi/` for further documentation if you are not yet familiar with CGI programming.

As I mentioned before, I think it is a bad idea to mix up the program code and HTML in the same files. Programmers want to think about the way things work, and web designers want to think about the way things look. So it is the best to split the development process up into different „roles“. For web designers, a MHT template still looks and feels quite the same as a normal HTML page. For programmers, MHT makes it as easy as possible to put some data from the CGI into a HTML page by macro expansion.

4.1.1 A simple example

To give you and idea how this is done using CGIMHT, you see below the source of a very simple CGI program in (ANSI-) C. It defines (registers) only one macro `name` as „Steve“, opens a template, and processes the block `page_html` inside this template to produce the HTML output. We will discuss the details later, this example should give you just an overview to understand the basics.

The source code of a very simple CGI program using CGIMHT:

```
#include <stdio.h>
#include "mht.h"
#include "cgi.h"

int main( int argc, char **argv ) {
    mht_init();
    cgi_init();

    mht_register_macro("name","Steve");
    mht_quickopen(stdout,"template.mht");
    mht_process(stdout,"page_html");

    cgi_exit();
    mht_exit();

    return (0);
}
```

The content of „template.mht“ which is opend by the CGI:

```
#begin page_html
<html><body>
Hi, my name is <#name>!
</body></html>
#end page_html
```

Finally, the output sent to the browser when the CGI is invoked:

```
<html><body>
Hi, my name is Steve!
</body></html>
```

4.1.2 Compiling with Visual C++ under Win32

To compile 32-bit CGIMHT programs using Visual C++, you need to link the CGIMHT library to your binaries. You find this library in the `mht/lib/win32/` directory of the distribution. Second, you do have to extend the search path for standard include files in your compiler settings to enclose also the MHT include files which are placed in the `mht/include/` directory of the distribution. Please follow the steps below to adjust your VC++ configuration.

To create a new project:

- On the **File** menu, click **New**.
- Select the **Projects** tab.
- The AppWizards are displayed. Select *Win32 Console Application* from the list.
- Specify the location for your project in the **Location** edit box, and type the project name in the project **Name** text box.
- Click **OK**.
- The *Win32 Console Application – Step 1 of 1* dialog box is displayed. Accept the default selection, **An empty project**, and click **Finish**.

To adjust the compiler settings for an existing/new project to compile with CGIMHT:

- If you have more than one project in your workspace, select the project you'd like to build by selecting it from the available projects located in the project explorer window. The project explorer window is usually on the left. If you can't see it, select **View->Workspace**. Right-click and choose the *Set As Active Project* option. The chosen project should now be bold.
- Choose the desired „Active Configuration" as either *Win32 Release* or *Win32 Debug*. To do this, select **Build->Set Active Configuration**. The default is *Win32 Release*.
- Select **Project->Settings** from the menu. This brings up the "Project Settings" dialog box.

Adjust the module/library path settings:

- Select the **Link** tab.
- In the text box **Object/library modules:** add at the beginning "cgimht.lib". Make sure there is a space between this and the next library.
- In the **Category** pull-down menu, select **Input**
- In the **Additional module/library directories** text box, enter the absolute path where you saved `cgimht.lib`

Adjust the include file path settings:

- Select the **C/C++** tab.
- In the **Category** pull-down menu, select **Preprocessor**
- In the **Additional include directories** text box, enter the absolute path where you saved `mht.h` and `cgi.h`

4.1.3 Compiling with Cygwin under Win32

I now also use Cygwin to compile MHT for Win32. Cygwin (`http://www.cygwin.com/`) is a Unix like environment which is published under the terms of the GNU Public License. Please follow the steps as described in the next section, except that the Makefile is named "Makefile.win32".

4.1.4 Compiling with LCC-Win32 under Win32

Another free C compiler for Windows is LCC-Win32 (`http://www.cs.virginia.edu/~lcc-win32/`). I used it to build the static cgimht library and the mht2html binary of the version 1.2 release of MHT. LCC is really self explanatory, and I will not provide any further documentation on how to compile MHT with LCC here.

4.1.5 Compiling with egcs/gcc/make under Unix

To compile binaries under Unix/Linux using the CGIMHT library, you will have to comply the following requirements:

- The include files mht.h and cgi.h (in the `mht/include` directory of the distribution) need to be copied into the standard search path for include files, which is usually `/usr/local/include` or `/usr/include`
- the CGIMHT library libmht.a (in the `mht/lib/linux/` directory of the distribution) has to be copied into the standard search path for libraries, which is usually `/usr/local/lib` or `/usr/lib`

Under Sun Solaris, these paths might be placed somewhere in `/opt`. You need root privileges to copy files into these directories, so please refer to your system administrator to copy these files.

Below you find an example Makefile to compile a C source linked with the CGIMHT library (use this Makefile without the beginning line numbers):

```
1   # C compiler:
2   CC      = gcc#
3   # C compiler options:
4   CFLAGS  = -O2 -m486 -Wall#
5   # Libraries:
6   LIBS    = -lcgimht#
7   # Target directory for the binary, you might need to change it
8   CGIBIN = /home/httpd/www.weckert.org/cgi-bin/
9
10  all: contact
11
12  contact: contact.c
13      $(CC) $(CFLAGS) contact.c $(LIBS) -o $(CGIBIN)contact.cgi
```

4.2 Function reference

In the include files mht.h and cgi.h you will find the prototypes of the functions that you can use in the CGIMHT library. These are the prototypes of the same functions of the MHT processor that he calls when he processes a template, there are no „wrapper functions“ for external usage. It wouldn’t be a big deal to put all function prototypes into mht.h or cgi.h, but most functions don’t make sense to be used „outside“ the MHT processor from a CGI program.

4.2.1 Initialization and termination

4.2.1.1 void mht_init(void)

Purpose	Initializes the MHT processor, its data structures, and registers a set of predefined date and time macros. This function **MUST** be called before any MHT processing can be done!
Returns	No return value
Arguments	None

4.2.1.2 void cgi_init(void)

Purpose	Reads in the CGI input string of GET or POST requests, and registers each input value as a MHT macro. All the input strings are unescaped. Buffering of STDOUT is turned off. If an input values appears more than once in the query (checkboxes!) such as „&meat=bacon&meat=salami“, then all values are concatenated to one large value string separated by commas. In this example, the macro „meat“ would have the value „bacon,salami“. This function **MUST** be called before any CGI processing can be done!
Returns	No return value
Arguments	None

`cgi_init()` registers a set of useful CGI environment variables as MHT macros, which you can then use in your templates or C sources codes. Please note that the environment variables are OS and server software depended, not every operating system/server software registers writes all environment variables listed below!

DOCUMENT_ROOT	The root directory of your web server/domain.
HTTP_COOKIE	The visitor's cookie, if one is set.
HTTP_HOST	The hostname of your server.
HTTP_REFERER	The URL of the page that called the CGI.
HTTP_USER_AGENT	The browser type of the visitor.
HTTPS	"on" if the script is being called through a secure server.
PATH_INFO	Everything after the actual name of the CGI.
PATH_TRANSLATED	The PATH_INFO mapped onto DOCUMENT_ROOT.
QUERY_STRING	The query string in case the CGI was called via GET.
REMOTE_ADDR	The IP address of the visitor.
REMOTE_HOST	The hostname of the visitor (if your server has reverse-name-lookups on; otherwise this is the IP address again).
REMOTE_PORT	The port the visitor is connected to on the web server.
REMOTE_USER	The visitor's username (for .htaccess-protected pages).
REQUEST_METHOD	GET or POST
REQUEST_URI	The interpreted pathname of the requested document or CGI (relative to the document root).
SCRIPT_FILENAME	The full pathname of the current CGI.
SCRIPT_NAME	The interpreted pathname of the current CGI (relative to the document root).
SERVER_ADMIN	The email address for your server's webmaster.
SERVER_NAME	Your server's fully qualified domain name.

SERVER_PORT	The port number your server is listening on.
SERVER_SOFTWARE	The server software you're using.
SERVER_PROTOCOL	The protocol the web server supports.

This the example output of the macros described in the table above:

```
DOCUMENT_ROOT : /home/httpd/www.weckert.org
HTTP_HOST : www.weckert.org
HTTP_REFERER : http://www.weckert.org/index.html
HTTP_USER_AGENT : Mozilla/4.0 (compatible; MSIE 4.01; Windows NT)
QUERY_STRING : action=start&cgimht=contact&version=plain&lang=eng
REMOTE_ADDR : 192.168.1.1
REMOTE_PORT : 1184
REQUEST_METHOD : GET
REQUEST_URI : /cgi-
bin/contact.cgi?action=start&cgimht=contact&version=plain&lang=eng
SCRIPT_FILENAME : /home/httpd/www.weckert.org/cgi-bin/contact.cgi
SCRIPT_NAME : /cgi-bin/contact.cgi
SERVER_ADMIN : webmaster
SERVER_NAME : www.weckert.org
SERVER_PORT : 80
SERVER_SOFTWARE : Apache/1.3.11 (Unix)
SERVER_PROTOCOL : HTTP/1.1
```

4.2.1.3 void mht_exit(void)

Purpose	Terminates all allocated data structures by the MHT processor.
Returns	Nothing
Arguments	None

4.2.1.4 void cgi_exit(void)

Purpose	Terminates CGI processing, flushes STDOUT.
Returns	No return value
Arguments	None

4.2.2 Definition of macros

4.2.2.1 int mht_register_macro(char *name, char *definition)

Purpose	Registers a new macro or overwrites the value of an existing macro, equaivalent to the #def directive in templates. The definition string can contain macros. The max. length of the name is 128 chars., the max. length of the definition is 1024 chars.
Returns	0 : success >0 : an error occurred (see table of error codes)
Arguments	name : the name of the new macro definition : the definition of the new macro
Example	`mht_register_macro("today","This is a macro");`

4.2.3 Deleting macros

4.2.3.1 int mht_undef_macro(char *name)

Purpose	„Undefines“ (deletes) a macro, equivalent to the #undef directive un

	templates.
Returns	0 : success >0 : an error occurred (see table of error codes)
Arguments	name : the name of the new macro
Example	`mht_undef_macro("today");`

4.2.4 Proving macro definitions

4.2.4.1 int mht_search_macro(char *name, char **result)

Purpose	Checks whether a macro exists
Returns	0 : the macro does not exist, result points to NULL 1 : the macro exists, result points to the definition string of the macro
Arguments	name : the name of the macro that we search for result : will point on the definition string if the macro was found, or NULL otherwise

Example	`char **result = (char**)NULL;`
	
	`if ((mht_search_macro("today",&result))==0) {`
	`   /* The macro was not found! */`
	`   ...`
	`}`
	`else {`
	`   /* The macro was found! */`
	`   ...`
	`}`

4.2.5 Expanding macros

4.2.5.1 char *mht_expand(char *str)

Purpose	Expand all macros in a given string
Returns	A char pointer to the expanded string
Arguments	A string which should be expanded

Example	`char *dummy = (char*)NULL;`
	`char tmp_str[32];`
	
	`sprintf(tmp_str,"<#test>");`
	`mht_register_macro("test","My name is Steve");`
	`dummy = mht_expand(tmp_str);`

4.2.6 Opening templates

4.2.6.1 int mht_quickopen(FILE *out, char *fname)

Purpose	Will open a MHT template and read it in. All lines outside a block are processed immediately, the output for these lines is written to the output sink which is specified by the „out“ argument. All blocks in the template are saved for later processing.
Returns	0 : success >0 : an error occurred (see table of error codes)

Arguments	out : file handle. The output of the lines outside a block which are processed immediately when the template is openend is written to this file handle. Should be STDOUT. fname : absolute path and file name of the template that should be opened

Example	`int mht_error = 0;`
	
	`if ( (mht_error=mht_quickopen(stdout,cgimht_file))>0 ) {`
	`   /* The template was not opened correct! */`
	`   ...`
	`}`
	`else {`
	`   /* The template was opened correct! */`
	`   ...`
	`}`

4.2.7 Processing blocks

4.2.7.1 int mht_process(FILE *out, char *block_name)

Purpose	Processes a block that was read in from a templatevia quick_open() before.
Returns:	0 : success >0 : an error occurred (see table of error codes)
Arguments	out : file handle where the output is written block_name : the name of the block that should be processed

Example 1	Processing a HTML block to STDOUT:
	
	`int mht_error = 0;`
	
	`if ( (mht_error= mht_process(stdout,"page_confirm"))==0 ) {`
	`   /* The block was processed correct! */`
	`   ...`
	`}`
	`else {`
	`   /* The block was not correct! */`
	`   ...`
	`}`

Example 2	Sending e-mail via sendmail under Unix by processing a block from a MHT template:
	
	`FILE *mail = (FILE *)NULL;`
	
	`mail = popen("/usr/lib/sendmail -t","w");`
	`mht_process(mail, "page_mail");`
	`pclose(mail);`
	
	popen() and pclose() are defined as:
	`FILE *popen(const char *command, const char *type);`
	`int pclose(FILE *stream);`
	
	The block page_mail contains the entire header and body of the e-mail:
	`#begin page_mail`
	`#mhtvar convumlauts FALSE`
	`#mhtvar killspace FALSE`
	`Reply-to: <#email>`

```
From: <#email>
Subject: <#subject>
To: <#recipient>

Date: <#long_date>
Name: <#name>
Email: <#email>
Message:
<#message>
#mhtvar convumlauts TRUE
#mhtvar killspace TRUE
#end page_mail
```

4.2.8 Displaying runtime errors

4.2.8.1 void cgi_err_msg(char *err_msg)

Purpose	Use this function to send a complete error HTML page to the browser in case of an fatal error. The CGI and MHT environment is not terminated by this function.
Returns	No return value
Arguments	err_msg : the string of the error message that should be displayed in the error page. To make it more useful, err_msg should consist of two parts: a detailed error message unvisible inside a HTML comment, and a common error message that is visible in the browser.
Example	`cgi_err_msg("<!— Connect to the database failed! -->\nService currently unavailable!");`

4.2.9 Escaping/Unescaping strings

Some characters have special meanings in URLs and need to be escaped by converting them into their corresponding hexadecimal ASCII code (%##):

- All characters with ASCII values less than 32 decimal (0x20 hex) or greater than 127 decimal (0x7F hex)
- Any characters from the set: `!"#$%&'()+,/:;<=>?[\]^`{\}~`
- delete (backspace)
- spaces can also be escaped by replacing them with pluses (+)

4.2.9.1 char *cgi_escape_str(char *str)

Purpose	Escapes a given string
Returns	A pointer to the escaped string
Arguments	A pointer to the string that should be escaped.

4.2.9.2 void cgi_unescape_str(char *str)

Purpose	Unescapes a given string
Returns	No return value
Arguments	A pointer to the strings that should be unescaped

4.3 A CGI sample application

Below you find the complete source (Listing 1) in standard ANSI-C for a simple CGI to send an e-mail from a web formular, together with a MHT template that keeps the HTML and the mail body (Listing 2). It is a demo formular to order a pizza online, you can view it at `http://www.weckert.org/mht/`

It should compile out of the box on any Unix system such as Linux or BSD using gcc. If you want to compile this source on a win32 machine, you need to use CreateProcess() instead of popen() to send the e-mail via a mailer such as Blat. You can use this code as a general skeleton for your own CGIMHT programs. In the section „Compiling with gcc/make under Unix“ you will find a short Makefile to compile the sample application.

4.3.1 Overview of the application

To begin with, copy and paste the source code of the CGI (Listing 1) and compile it. The binary should be named `sample.cgi` (depends on what type of postfix you configured in your web server to handle CGI scripts). Place the binary together with the MHT template `sample-plain-eng.mht` (Listing 2) in the document root as shown below:

```
<your www document root>
      <your cgi-bin>
            sample.cgi
            cgimht/
                  sample-plain-eng.mht
```

The URL to start this CGI is from your browser is then:
`http://<your domain>/<cgi-bin>/sample.cgi?action=start&cgimht=contact&version=plain&lang=eng`

The CGI expects at least four input values to work correct:

- „action“ tells the CGI what to do next. „start“ prints the input form, with the current values in the formular inserted, if the form has already been sent, but failed. „send“ tells the CGI that the user filled out the formular. Then the input has to be checked, and if correct, a mail has to ben sent and a confirmation page has to be printed in the browser.
- „cgimht“ is the name of the MHT template that the CGI should open.
- „version“ is the layout version of the template, if you run your web site both in a frameless „plain“ version and a „frame“ version.
- „lang“ is the language of the MHT template that should be opened, if you offer your website in more than one language.

In this example, all MHT templates are named as *cgimht-version-lang*`.mht`.

4.3.2 Overview of the C source

The C source of this example is not too sophisticated, there may be parts that can be implemented easier.

- In line 9 is the directory defined where the MHT templates are stored. In the example, this is a directory „cgimht“ which is relative to the CGI binary’s path.

- The CGI and MHT environment is initialized in line 36-38. The input is read in and registered as MHT macros (form re-submission!).
- In line 40-68, it is proved whether the form was submitted with the input values „cgimht“, „version“, „lang“, and „action“. These values are at least required to make the CGI work correct. If one of those input values is missing, and error screen is printed to the browser, and any further exceution is stopped by calling `mht_exit()` and `cgi_exit()`.
- In line 72-79, the path for the MHT template is built together,a dn the template is opened.
- The required input values of the form input fields are proved in line 81-123. If one of those required values is missing, the name of the corresponding formular field is concatenated to a larger string that keeps the names of all the missing fields (lines 90-95 ex.) separated by commas.
- After the input is validated, and the MHT template opened and read in, the CGI can start to do ist job, which is declared by the „action“ input value.
- „start“ prints out the input form to the browser as in line 128-130.
- „send“ checks first, whether there were some required input fields of the form missing. If this is the case, the string containg the names of the missing form fields is registered as a macro (line 134), the input form is again printed to the browser (line 135), and any further execution of the CGI is terminated (line 136-138).
- If no required input value was missing, the CGI checks whether a macro „mail_command“ exists (line 141-146). This macro keeps the correct sendmail path, which might vary on different OS systems. The sendmail path is defined in the MHT template, and not „hard- wired“ in the C source.
- Then the CGI will open a new process to pipe the output of the „page_mail“ block which is processed to sendmail (line 148-150).
- Finally, in line 163-164, the MHT and CGI environments are destroyed.

4.3.3 Listing 1: the C source

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <ctype.h>
4   #include <string.h>
5
6   #include <mht.h>
7   #include <cgi.h>
8
9   #define CGIMHT_SRCPATH        "cgimht/"
10  #define MAX_LEN               256
11
12  int main( int argc, char **argv ) {
13      int
14              mht_error = 0;
15
16      char
17              tmp_str[MAX_LEN],
18              cgimht_file[MAX_LEN],
19              *action = (char*)NULL,
20              *version = (char*)NULL,
21              *cgimht = (char*)NULL,
22              *lang = (char*)NULL,
23              *name = (char*)NULL,
24              *email = (char*)NULL,
25              *size = (char*)NULL,
          *meat = (char*)NULL,
          *vegetables = (char*)NULL,
          required_values[MAX_LEN*4],
          *mail_command = (char*)NULL;

    FILE
          *mail = (FILE *)NULL;

    /*** Initialize the MHT and CGI environment ***/

    mht_init();
    cgi_init();
    fprintf(stdout,"Content-type: text/html\n\n");

    /*** Check if the CGI was called with an action, version and cgimht parameter ***/

    if ((mht_search_macro("cgimht",&cgimht))==0) {
          cgi_err_msg("No CGIMHT template specified!");
          mht_exit();
          cgi_exit();
          exit(1);
    }

    if ((mht_search_macro("version",&version))==0) {
          cgi_err_msg("The version (frame or plain) of the CGIMHT template was not specified!");
          mht_exit();
          cgi_exit();
          exit(1);
    }

    if ((mht_search_macro("action",&action))==0) {
          cgi_err_msg("No action specified!");
          mht_exit();
          cgi_exit();
          exit(1);
    }

    if ((mht_search_macro("lang",&lang))==0) {
          cgi_err_msg("No language specified!");
          mht_exit();
          cgi_exit();
          exit(1);
    }

    /*** Open and read in the MHT template ***/

    sprintf(cgimht_file,"%s%s-%s-%s.mht",CGIMHT_SRCPATH,cgimht,version,lang);
    if ( (mht_error=mht_quickopen(stdout,cgimht_file))>0 ) {
          sprintf(tmp_str,"Error reading CGIMHT file \"%s\"",cgimht_file);
          cgi_err_msg(tmp_str);
          mht_exit();
          cgi_exit();
          exit(1);
    }

    /*** Check all required input values ***/

    required_values[0] = '\0';

    if ((mht_search_macro("name",&name))==0) {
          strcat( required_values, "name" );
```

```
	}

	if ((mht_search_macro("email",&email))==0) {
		if (strlen(required_values)>0) {
			strcat( required_values, ", email" );
		}
		else {
			strcat( required_values, "email" );
		}
	}

	if ((mht_search_macro("size",&size))==0) {
		if (strlen(required_values)>0) {
			strcat( required_values, ", size" );
		}
		else {
			strcat( required_values, "size" );
		}
	}

	if ((mht_search_macro("meat",&meat))==0) {
		if (strlen(required_values)>0) {
			strcat( required_values, ", meat" );
		}
		else {
			strcat( required_values, "meat" );
		}
	}

	if ((mht_search_macro("vegetables",&vegetables))==0) {
		if (strlen(required_values)>0) {
			strcat( required_values, ", vegetables" );
		}
		else {
			strcat( required_values, "vegetables" );
		}
	}

	/*** Do the action ***/

	/* Put out the form the 1st time */
	if (strcmp("start",action)==0) {
		mht_process(stdout,"page_start");
	}
	else if (strcmp("send",action)==0) {
		/* Check if all required fields are filled out! */
		if (strlen(required_values)>0) {
			mht_register_macro("required_values",required_values);
			mht_process(stdout,"page_start");
			cgi_exit();
			mht_exit();
			exit(1);
		}

		if ((mht_search_macro("mail_command",&mail_command))==0) {
			cgi_err_msg("No mail_command specified!");
			cgi_exit();
			mht_exit();
			exit(1);
		}

		mail = popen(mail_command,"w");
		mht_process(mail, "page_mail");
		pclose(mail);
```

```
151          mht_process(stdout,"page_confirm");
152     }
153     else {
154          sprintf(tmp_str,"Unknown action \"%s\" specified!",action);
155          cgi_err_msg(tmp_str);
156          mht_exit();
157          cgi_exit();
158          exit(1);
159     }
160
161     /*** Exit the MHT and CGI environment ***/
162
163     cgi_exit();
164     mht_exit();
165
166     return (0);
167 }
```

4.3.4 Overview of the MHT template

The MHT template is split into three blocks:

- „page_start“ contains the HTML for the input page
- „page_confirm“ contains the HTML for the confirmation page that is sent to the browser after the user has successfully order a pizza
- „page_mail“ contains the mail header and body that is to the user as a confirmation email

It is recommended, that if you have a more complex HTML layout, to split up the layout of your pages into some more reusable blocks such as „page_head“, „page_foot“ for example. The complete layout is the built by placing `#process` directives in the right place.

The general issue of the MHT template is, what macros you have to use for automatic form re- submission. Remember that all input values are immediately registered as MHT macros when the are read into the CGI. If there is a required input value missing, you still got the remaining submitted values already as MHT macros. You then resend the input form to the browser to let the user to complete the form and re- submit it. You can put macros into the HTML of the form fields, which check whether a certain macro exists (the value was already submitted before), or not.

For text input fields you can use `<#ifdef>` as in line 32 and 36. For selections you have to use `<#ifequal>` to compare strings, because a selection can have different value (line 45-49). If a field is submitted with more than one value as with checkboxes, you can use `<#isin>` to prove substrings (line 60-62 and 65-67).

The lines 135-155 contain the entire header and body of the email that is sent to the user as a confirmation. Be sure so set killspace on „FALSE“!. Another very common error is, that the MHT template that keeps the email is not save in Unix mode.

The macro `<#SCRIPT_NAME>` is a macro that was registered from a environment variable when the CGI environment was initialized.

4.3.5 Listing 2: the MHT template

```
#begin page_start
<html>
<head>
<title>MHT pizza demo form</title>
</head>
<body bgcolor="#ffffff">

<h3>Order a pizza online!</h3>

<form action="<#SCRIPT_NAME>" method="get">
<input type="hidden" name="action" value="send">
<input type="hidden" name="cgimht" value="<#cgimht>">
<input type="hidden" name="version" value="<#version>">
<input type="hidden" name="lang" value="<#lang>">

<table border="0">
#if <#ifdef|required_values|TRUE|FALSE>
<tr>
<td colspan="2">
<font color="#ff0000">
<b>
Please fill out the following required fields:<br>
<#required_values>
</b>
</font>
</td>
</tr>
#endif
<tr><td colspan="2"><br><b>Personal data:</b></td></tr>
<tr>
<td>Your name:</td>
<td><input type="text" size="25" maxlength="50" name="name"<#ifdef|name|
value="<#name>">></td>
</tr>
<tr>
<td>Your e-mail address:</td>
<td><input type="text" size="25" maxlength="50" name="email"<#ifdef|email|
value="<#email>">></td>
</tr>
<tr><td colspan="2"><br><b>Select your pizza:</b></td></tr>
<tr>
<td valign="top">
Size:
</td>
<td>
<select name="size">
<option value=""<#ifequal|<#size>|<#null>| selected>>
<option value="small"<#ifequal|<#size>|small| selected>> small
<option value="medium"<#ifequal|<#size>|medium| selected>> medium
<option value="large"<#ifequal|<#size>|large| selected>> large
<option value="family"<#ifequal|<#size>|family| selected>> family
</select>
</td>
</tr>
<tr><td colspan="2"><br><b>What would you like on your pizza?</b></td></tr>
<tr>
<td valign="top">Toppings:</td>
<td>

<table border="0" cellpadding="2">
<tr>
```

```
<td><input type="checkbox" name="meat" value="bacon"<#isin|bacon|<#meat>| checked>>&nbsp;bacon</td>
<td><input type="checkbox" name="meat" value="pepperoni"<#isin|pepperoni|<#meat>| checked>>&nbsp;pepperoni</td>
<td><input type="checkbox" name="meat" value="salami"<#isin|salami|<#meat>| checked>>&nbsp;salami</td>
</tr>
<tr>
<td><input type="checkbox" name="vegetables" value="garlick"<#isin|garlick|<#vegetables>| checked>>&nbsp;garlick</td>
<td><input type="checkbox" name="vegetables" value="olives"<#isin|olives|<#vegetables>| checked>>&nbsp;olives</td>
<td><input type="checkbox" name="vegetables" value="mushrooms"<#isin|mushrooms|<#vegetables>| checked>>&nbsp;mushrooms</td>
</tr>
</table>

</td>
</tr>
<tr>
<td colspan="2" align="center">
<br>
<input type="submit" value="Submit">&nbsp;<input type="reset" value="Reset">
</td>
</tr>
</table>

</form>

</body>
</html>
#end page_start


#begin page_confirm
<html>
<head>
<title>MHT pizza demo form</title>
</head>
<body bgcolor="#ffffff">

<h3>Thank you for your order!</h3>

<table border="0">
<tr><td colspan="2"><br><b>Personal data:</b></td></tr>
<tr><td>Your name:</td><td><#name></td></tr>
<tr><td>Your e-mail address:</td><td><#email></td></tr>
<tr><td colspan="2"><br><b>Your pizza:</b></td></tr>
<tr><td valign="top">Size:</td><td><#size></td></tr>
<tr><td colspan="2"><br><b>You want on your pizza:</b></td></tr>
<tr>
<td valign="top">Toppings:</td>
<td>

<table border="0" cellpadding="2">
<tr><td><#meat></td></tr>
<tr><td><#vegetables></td></tr>
</table>

</td>
</tr>
<tr>
```

```
<td colspan="2" align="center">
<br>
<a href="<#SCRIPT_NAME>?action=start&version=plain&lang=eng&cgimht=sample">Order another pizza!</a>
</td>
</tr>
</table>

</body>
</html>
#end page_confirm


#if <#ifequal|<#SERVER_NAME>|endor.smart.script|TRUE|FALSE>
#def mail_command             /usr/sbin/sendmail -t
#else
#def mail_command             /usr/lib/sendmail -t
#endif


#begin page_mail
#mhtvar convumlauts FALSE
#mhtvar killspace FALSE
Reply-to: <#email>
From: <#email>
Subject: MHT pizza demo form
To: <#email>

Thank you for your order!

Date: <#long_date>
Your name: <#name>
Your email: <#email>

Size of your pizza: <#size>

Toppings:
Meat: <#meat>
Vegetables: <#vegetables>
#mhtvar convumlauts TRUE
#end page_mail
```

4.4 Error handling

4.4.1 How to get error messages in your C sources

If an error occurred while the MHT processor runs, MHT registers three macros:

- `mht_err_msg` with a clear text message of the error
- `mht_err_line` contains the line of the template where the MHT processor found the error
- `mht_err_code` with the error code, see the tables of all error codes

By using the MHT compiler `mht2html`, the error message will be printed to STDOUT automatically after the MHT processor has been stopped execution. An example error message:

```
MHT-Error: There is a #defex directive with a macro, but without a
definition found!
```

By using the CGIMHT C-API, you should always check whether the MHT functions return 0 (meaning the invoked function returned without an error, „no news are good news“), if not, an error occurred. The following code snippet would return the complete error message by doing a macro seach for `mht_err_msg`:

```
int mht_error = mht_quickopen(stdout,"test.mht");
if ( (mht_error!=0) && (mht_search_macro("mht_err_msg",&mht_err_msg)!=0) )
{
   fprintf(stdout,"MHT-Error: %s\n",mht_err_msg);
}
```

4.4.2 Table of error codes and messages

The table below gives a complete overview of all MHT error codes and their referring messages:

Err. Code	Error message
0	No errors!
1	Specified MHT (include) file not found!
2	#begin directive inside a block found, blocks may not be cascaded!
3	Outside a block opened filehandles must be closed before a #begin directive!
4	Inside a block opened filehandles must be closed before an #end directive!
5	The blocknames of the #begin and #end directive do not match!
6	#mhtvar was called without a value!
7	#mhtvar was called without a variable!
8	#mhtvar was called with an unknown variable!
9	#mhtfile was called without an operation!
10	#mhtfile was called with an unknown operation!
11	Cannot open a filehandle without a given filename!
12	Cannot open a filehandle, drive is full!
13	Cannot open a filehandle, drive is write protected!
14	Cannot open a filehandle, a general I/O error occured!
15	Another open filehandle must be closed before a new filehandle can be opened!
16	Empty #file directive without any arguments found!
17	Unknown action in a #mhtfile directive found! Valid actions are type or open.
18	Undefined file type in a #mhtfile or #file directive found!
19	You have too many cascaded #if directives!
20	Empty #if/#elif directive without any arguments!
21	The number of #if and #endif directives do not match!
22	Before an #else or #elif directive, a #if directive is needed!
23	The argument for #if/#elif should be {TRUE|FALSE} or {1|0}!
24	There is an #if directive which is not closed by #endif found!
25	Block not found!
26	Empty #process directive without any arguments found!
27	Empty #def directive without any arguments found!
28	There is a #def directive with a macro, but without a definition found!

29	Empty #defex directive without any arguments found!
30	There is a #defex directive with a macro, but without a definition found!
31	Empty #undef directive without any arguments found!
32	Empty #undefblock directive without any arguments found!
33	Empty #include directive without any arguments found!
34	Empty #mhtvar directive without any arguments found!
35	Block in #loop directive not found!
36	There is a #loop directive with insufficient parameters found!
37	There is a #loop directive with non-digit parameters found!
38	Too many recursive file inclusions!
39	#end directive without a opening #begin directive found!
40	Error registering macro, please check the length of your macro name and definition!

5 License

You can find the latest version of the GNU Public License online at
`http://www.gnu.org/copyleft/gpl.html`