# The Room List

Christian von Schultz

2004–12–08

## Contents

---

The room list is the file that coordinates everything the program does. It tells the program what rooms to visit, and how those rooms are created. It also contains information about "room guards" and "time guards", and the amount of time before the meteor hits the moon base.

# 1 Where to find the room list

The application first and foremost looks at its command line arguments. An argument specifying the location of the room list might look like this: "`--roomlist=roomlist.txt`", meaning that the program should look for a file called "`roomlist.txt`" in the current directory. If you want the program to remember where to find the room list, specify "`--generate-config`". The program's configuration will be updated, and you will no longer need to use the command line arguments.

Where the configuration is stored, varies between different operating systems. On UNIX and GNU/Linux systems, it will be a file in the user's home directory called "`.moonrc`". On Windows systems it will probably be the registry. The exact details of the file format is unknown to the program, and managed by the **wxConfig** class. On my GNU/Linux system, the `.moonrc` lines telling the program where to find the roomlist are:

```
[RoomList]
filename=/home/christian/development/moon/roomlist.txt
```

```
start(
        module = "initial",
        intro = "intro",
        first_room = "addition1",
        timeout = "timeout",
        time = 3600000,
        time_guards = ["thirty_minutes", "one_minute"],
        room_guards = ["data_control", "full_addition1"],
        start_timer = "later",
        theme = "moon_theme"
    ).
intro(
        module = "intro",
        first_room = "addition1",
        start_timer = "later"
        html_file = "intro.en.html",
        html_file_sv = "intro.sv.html",
        button_text = "First room in the maze",
        button_text_sv = "Första rummet i hinderbanan",
        button_help = "Start playing the game",
        button_help_sv = "Börja spela spelet"
    ).
```

Table 1: Part of a room list file

# 2 The file format of the room list

The room list uses a subset of Prolog-like syntax. The file comprises a series of *clauses*. Each clause is an object with a *functor* or object name, followed by a list of attribute-value pairs enclosed in parentheses, and finished with a full stop. Each attribute value may be a string within double quotes, an integer, a real number, or a list. See table 1.

## 2.1 Encoding and character set

The room list file will always be encoded in UTF-8. You must not, however, assume that the entire Unicode character set is available. The attributes controlling user-visible strings (or files containing user-visible strings) will be

automatically transcoded into the current character set. That means that you should only use characters that are available in every encoding in use for a particular language.

If you want two different files or strings for the same language, you can specify the *encoding number*. You should look up how `wxFontEncoding` is defined in the current version of wxWindows. Find the encoding in the `enum` list, and find out which number you want (each item in an `enum` list has the value of the previous entry plus one). If that number turns out to be negative, add its absolute value to the value of `wxFONTENCODING_MAX`. For ISO-8859-1 this number will be 1. Then, the attribute naming the string or file in this encoding will be e.g. "`name_sv_SE_1`". (Note that you can also specify the language and country for user-visible attributes: just precede the corresponding code with an underscore). You can alter the precedence of the different "`name`" attributes by omitting the last underscore, for the exact precedence of the different "`name`" attributes, see the definition of `GetLocalizedAttribute()` in `application.cc`.

Please note that this only tells the encoding we transcode into. The files or attribute values should themselves be encoded in UTF-8, but restricted to the character set of the target encoding. HTML files are an exception: they are encoded in any encoding understood by **wxHtmlWindow**. The encoding is set in meta tags.

Attributes that are not user-visible should only use 7-bit ASCII characters.

## 2.2 Special attributes and clauses

### 2.2.1 The "`module`" attribute

All clauses should (unless otherwise noted) contain the attribute "`module`". This is the name of the module to load. If you are using a GNU/Linux system this will typically be a *shared library* with the extension "`.so`", but without the common "`lib`" prefix. If you are using Windows, it will be a *Dynamically Linked Library* with the extension "`.dll`". Since the room list should be usable on all platforms, the extension is always omitted in the room list.

A module will either contain an object that puts some contents into the main window (a `WindowContents` object), or a so called "*guard*" — a special object that waits for some event to occur and then performs some actions. See the section on module types, page 9.

### 2.2.2 The "start" clause

When the program is first started it looks in the room list for a clause with the functor "start" (as in table 1). If it does not find "start", the program will not start. The "start" clause supports several special attributes:

**module** This is the name of a module containing a `WindowContents` object (see page 9), that is invoked when the program is started.

**timeout** This is a so called "*door*". A door is an attribute whose value is a functor (the name of a different clause). When the user runs out of time, the system "goes through this door", i.e. does whatever is specified in the clause that this attribute refers to.

**time** This is an integer, and is interpreted as the amount of time the user is allowed before we automatically go through the `timeout` door. The `time` is measured in milliseconds.

**time_guards** This is a list of functors, and specifies that the corresponding clauses should be loaded. These clauses should refer to `TimeGuard` objects. (See page 9.)

**room_guards** This is a also list of functors, and specifies that the corresponding clauses should be loaded. However, these clauses should refer to `RoomGuard` objects. (See page 9.)

### 2.2.3 The "restore" clause

The "`restore`" clause should not be present in any room list. It is, however, present in the files saved by the program, which use the same format as the room list. If the room list would contain a "`restore`" clause, users would be able to load files which are not supposed to be loaded by this program. (The same code is used for both the room list and the saved files.)

### 2.2.4 The "data_control" clause

Reserved for use by the program. It contains the data of the `RGDataControl`, an object which can store various information that several other objects might need. Any object can add or read data from the `RGDataControl`, so the actual contents of this clause depends on what the other objects have done so far.

You must have a "`data_control`" clause, and it should look like this:

```
data_control(
        data_labels = [ ]
    ).
```

This clause should be referred to by the "`room_guards`" attribute of the "`start`" clause.

### 2.2.5 The "`start_timer`" attribute

Any clause (except the theme, `TimeGuard` and `RoomGuard` clauses) can contain the attribute "`start_timer`". If this is set to "`later`" (as in table 1), this means that if the timer[1] hasn't been started yet, we don't start it until later. If this attribute is absent, or set to anything other than "`later`", the timer will be started (if it isn't already running).

### 2.2.6 The "`stop_timer`" attribute

Any `WindowContents` clause can contain the attribute "`stop_timer`". If this is set to "`yes`", it means that we will stop the timer (if it's running). Any other value will be ignored. This will typically be used on the clause describing the last room that the user visits, saying either "you succeeded" or something less encouraging, but it could also be used to temporarily turn off the timer during one room, and start it again when we go through a door.

### 2.2.7 The "`html_file`" attribute

Some modules want to display an HTML file before continuing; some modules have the single task of displaying this file. The file to be displayed is set with this attribute, and should be found in the same place as the modules themselves.

Modules supporting this attribute often has a button which requires the "`button_text`" and "`button_help`" attributes.

This attribute supports several languages using the "`html_file_xx`" syntax.

### 2.2.8 The "`button_text`" and "`button_help`" attributes

The button text is the text on the button. The button help is the help text displayed in the status bar and a tool tip when the mouse pointer hovers over the button. The actions performed by the button are defined by the module providing it.

These attributes support several languages using the "`button_text_xx`" syntax.

[1]The timer is the object counting down until it's time to go through the "`timeout`" door of the "`start`" clause.

```
moon_theme(
        placement = "centered",
        background = [ 0, 0, 0 ],
        text_background = [ 176, 159, 139 ],
        text_foreground = [ 0, 0, 0 ],
        image = "Moon-galileo-color-400x395.jpg",
        big_image = "Moon-galileo-color-800x789.jpg",
    ).
```

Table 2: A theme clause

### 2.2.9 Theme clauses and the "theme" attribute

Some modules support graphical themes. The graphical themes are stored in theme clauses, whose functors are referenced by the "theme" attribute of other modules. For instance, table 1 shows a "theme" attribute set to "moon_theme". This would then refer to a theme clause, as in table 2.

The attributes supported by a theme clause are:

**image** The name of an image to use as the background of the room. It should be located in the same directory as the modules.

**placement** The placement of the image: either "tiled" or "centered". If centered, the image will appear in the centre of the window, otherwise it will be used as a wallpaper.

**big_image** If you want a bigger image for people with high resolutions, set it here.

**background** This is the background colour. I strongly recommend setting this attribute, since it will fill any space that is not covered by some image. It is given as RGB-values (red, green, blue) as shown in table 2.

**text_background** This is the background colour of the text. If you don't set it, it will be the same as the colour set by the background attribute.

**text_foreground** This is the colour of the actual text. It defaults to white, since the background defaults to black, but you should not rely on this behaviour: set the colours yourself. Not all modules honour the text_background and text_foreground attributes, they are merely a suggestion.

**TB_height** The hight of the top border, if any.

**BB_height** The hight of the bottom border, if any.

**LB_width** The width of the left border, if any.

**RB_width** The width of the right border, if any. If you set any borders, you should also set the corresponding images. If you have both horizontal and vertical borders, you should also set the corner images.

**TB_image** The image used in the top border.

**BB_image** The image used in the bottom border.

**LB_image** The image used in the left border.

**RB_image** The image used in the right border.

**TLC_image** The image used in the top left corner.

**BLC_image** The image used in the bottom left corner.

**TRC_image** The image used in the top right corner.

**BRC_image** The image used in the bottom right corner.

### 2.3 Other attributes

Any attributes not listed above, are module specific attributes. They may be supported by one module, but not by other modules. To see which additional attributes are supported, see the documentation of each module. The `WindowContents` modules will typically support one or more doors, leading to other `WindowContents` clauses. The `TimeGuard` modules will generally support the "`minutes`" attribute, specifying when the `TimeGuard` object should activate itself. "`minutes = 0`" would mean "when the user runs out of time".

## 3 Module Types

### 3.1 `WindowContents` modules

These modules contain `WindowContents` objects, which are responsible for communicating with the user and populating the main window. These are the only objects that are allowed to do anything with the main window, although other objects may pop up some dialog or message box.

There is only one `WindowContents` object active at a time. When it has finished, it can either quit the application or go through a door, specified in its clause.

Several of the `WindowContents` modules may be "templates" that act differently depending on their arguments (i.e. the attributes of the corresponding clause). Table 3 shows an example of this: the "`multsim`" module (which asks the user multiple questions simultaneously), can be used to do either addition or multiplication, depending on its arguments.

## 3.2 `TimeGuard` modules

A `TimeGuard` object does not control the main window. It is sent a message every minute. When the time is right it can then choose to act, for example displaying a dialogue that tells the user that only one minute is left. Many `TimeGuard` objects will support a "`minutes`" attribute in their clauses, which specifies the time when they should act. The time specified is the time the user has left when this object activates itself.

It is, however, possible that some `TimeGuard` modules do not support the "`minutes`" attribute. They might choose to do something every time they are invoked. To be certain, look at the documentation for the module you are thinking about using.

Any object can add a `TimeGuard` to the system, so it might make sense to define more `TimeGuard` clauses than listed in the "`time_guards`" attribute in the "`start`" clause, if you know of a module that can use them. Only the `TimeGuard` clauses listed in the "`time_guards`" attribute in the "`start`" clause will have their modules loaded at start time, though.

## 3.3 `RoomGuard` modules

A `RoomGuard` object, like a `TimeGuard` object, does not control the main window. These guards are sent a message every time we go through a door. They are given the new clause with all the information needed to instantiate a new room, and may change anything: the attributes, or even the module to load. So even if a `WindowContents` module wants a certain room, a `RoomGuard` could make sure something else is loaded.

Any object can add a `RoomGuard` to the system, so it might make sense to define more `RoomGuard` clauses than listed in the "`room_guards`" attribute in the "`start`" clause, if you know of a module that can use it. Only the `RoomGuard` clauses listed in the "`room_guards`" attribute in the "`start`" clause will have their modules loaded at start time, though.

```
addition1(
        module = "multsim",
        columns = 2,
        rows = 10,
        data_file = "addition1.txt",
        success = "multiplication1",
        theme = "test_theme",
        html_file = "addition1.en.html",
        html_file_sv = "addition1.sv.html",
        button_text = "Enter the room",
        button_text_sv = "Gå in i rummet",
        button_help = "Start solving the problems",
        button_help_sv = "Börja lösa uppgifterna"
    ).

multiplication1(
        module = "multsim",
        columns = 3,
        rows = 4,
        data_file = "multiplication1.txt",
        data_file44 = "multiplication1.ascii",
        success = "final",
        theme = "test_theme",
        html_file = "multiplication1.en.html",
        html_file_sv = "multiplication1.sv.html",
        button_text = "Enter the room",
        button_text_sv = "Gå in i rummet",
        button_help = "Start solving the problems",
        button_help_sv = "Börja lösa uppgifterna"
    ).
```

Table 3: The same module — several rooms

### 3.4 The difference between `WindowContents`, `TimeGuard` and `RoomGuard` clauses

Even though the clauses describe vastly different types of modules containing even more different types of objects, there is no straight forward way to look at a clause and say whether it's a `WindowContents`, a `TimeGuard` or a `RoomGuard` clause. You will have to look at the module: if the module to be loaded is a `TimeGuard` module, then you are looking at a `TimeGuard` clause. I have taken to giving all `TimeGuard` modules the prefix "`tg`" and all the `RoomGuard` modules the prefix "`rg`".

Although the clauses look very much the same, they are not interchangeable. If you try to load a `RoomGuard` module when a `WindowContents` module is expected, the program will complain that there's an error in the room list. It's up to you to ensure that your doors lead to clauses of appropriate types. What doors you can use, and whether any non-`WindowContents` clauses are expected, are detailed in the description of every module.

## 4 The `WindowContents` modules

### 4.1 The `initial` module

This module presents the user with two buttons: "intro" and "skip intro". If the user chooses "intro" then an introductory text or video is displayed, and if the user decides to "skip intro" we enter the first room with mathematical contents. This module supports graphical themes.

The module will typically be used in the `start` clause. Apart from the special attributes supported by the `start` clause (you will probably want `start_timer = "later"`), this module also supports the following doors:

**intro** If the user wants an intro, we go through this door, i.e. the clause with the given name is looked up in the room list and the corresponding module loaded.

**first_room** If the user wants to start the program straight away, we go through this door.

### 4.2 The `intro` module

This will display some introductory text (a HTML-page). There is a button that will make us go through the "`first_room`" door. Please use the

"`html_file`", "`button_text`" and "`button_help`" attributes to set the HTML-page and the label of the button. You will probably want `start_timer = "later"`. This module supports the following doors:

**first_room** When the user has read the text, we go through this door.

## 4.3 The `chooseone` module

This module will display a series of images and choices. For instance, it may show you an image containing a rectangle and a question ("What is this?" or similar), and a set of choices. If you choose the right answer, you might be presented with the next image or go through a door. Which images to display, which answers are supported and what they result in, is controlled in a separate control file. The module will probably support one or more doors; that is regulated in the control file. The syntax of the control files is documented elsewhere.

Before starting, however, an HTML file is displayed with a button to enter the room. These are specified in the usual way. This module also supports graphical themes, using the "`theme`" attribute and a corresponding clause.

In addition we support the following attribute:

**control_file** The name of the file controlling this room. This attribute supports the `control_file_xx` syntax for specifying a different file for the language `xx`.

## 4.4 The `multsim` module

This one will present the user with multiple questions simultaneously. The user can choose which one to answer first. If the user answers incorrectly, the program will simply wait for the user to enter another answer. This module supports one door:

**success** When the user has answered all questions, we go through this door.

The exact questions to be asked are read from a file. The name of the room, and the number of columns and rows of questions are specified using attributes. We also display a HTML file before entering the room. This module supports graphical themes and the following attributes:

**columns** The number of columns.

**rows** The number of rows. The number of questions will be `columns` × `rows`.

**data_file** The file from which to read the data. The file should contain one question and the corresponding answer per line, with question and answer separated by #. For example: `5 + 3 = # 8`

**html_file** The HTML file containing some introductory text.

**button_text** The label on the button displayed with the HTML file. When the button is clicked, the user will be presented with the questions set in the `data_file`.

**button_help** The help text of the button.

### 4.5 The `final` module

This module has no doors, and reaching the module will end the program, after having displayed a message to the user. You will probably want to set the attribute `stop_timer = "yes"`.

The following attributes are supported by this module:

**html_file** The HTML file containing some text, either congratulating the user for making it on time, or saying something less encouraging (if this room is entered because the time is up).

**button_text** The label on the button displayed with the HTML file. When the button is clicked, the program quits.

**button_help** The help text of the button.

## 5 The `TimeGuard` modules

There is one `TimeGuard` object that does not live in any module, but in the main application. It's the time guard that waits for a message from the timer that there is no time left, and when this happens, goes through the `timeout` door of the `start` clause.

The `TimeGuard` modules commonly have the prefix "tg".

### 5.1 The `tgdialogue` module

This `TimeGuard` module supports the `minutes` attribute. After the specified amount of time, a dialog pops up. The message displayed is controlled through the following attributes:

**title** The title of the dialogue.

```
notimer_addition1(
        module = "rgchange",
        change = "addition1",
        attributes = ["stop_timer"],
        stop_timer = "yes"
    ).
```

Table 4: Rgchange — adding the attribute `stop_timer` to the addition1 clause

**title_xy** The title in the language xy (e.g. "`sv`" for Swedish)

**text** The message displayed in the body of the dialogue.

**text_xy** The message in the language xy.

# 6 The `RoomGuard` modules

`RoomGuard` modules commonly have the prefix "rg".

## 6.1 The `rgchange` module

This module waits until a we want to load a `WindowContents` clause with a given functor, and replaces some attributes found in that clause. It supports the following attributes:

**change** The name of the clause we are waiting for and will be changing.

**attributes** A list of attributes to change. (Remember how lists are entered? See table 1.)

All attributes with their names entered in the `attributes` list should also be present. See table 4. (That particular example might not be all that useful: it would be simpler to add the `stop_timer` attribute to the `addition1` clause directly.)