



# RubyBuildTools: Webinterface

## Sitemap

- [Installer.rb](#)
- [URLs.html](#)
- [Generate PDF](#)
- [Use multiple yaml files](#)
- [Lokale GobolinuxPage](#)
- [Lokale Projects Page](#)
- [Homepage for Ruby Build Tools](#)
- [Google Project page for the Ruby Build Tools](#)
- [Task Taste for the Ruby Build Tools](#)

## The RubyBuildTools Tutorial

### Project :: Ruby Build Tools

Welcome to this Tutorial (last update: **08.07.2017**).

This tutorial has a fairly simple goal, which is to try to teach other human beings how to use the **RubyBuildTools (RBT)**.

It also has a secondary goal, which is to provide some reasoning and motivation for having created this project.

Reworded differently, this tutorial should provide you with an understanding how the RBT project may be useful for you.

This document here is essentially an introduction to the possibilities of what can be done with the **RBT-scripts**. It will also show a lot of specific examples, usually for use in the commandline. Ideally though, we can also make use of RBT through a GUI interface or the WWW.

You should be able to see which instructions are for the commandline, by simply looking for lines that are padded to the right and having a thin, colourized border.

Here is one simple example for this:

```
| rbt --version?
```

The above would, on the commandline, tell you which version of the RBT scripts you are currently using. At the time of writing this tutorial, the current version was **0.1.26**.

In the event that you do happen to find a bug, it would be appreciated if said bug could be reported on the github-issue tracker at

<https://github.com/shevegen/rbt/issues>

or via email to me at [shevegen@gmail.com](mailto:shevegen@gmail.com) directly.

All bug reports will additionally be stored in a local file, a "todo" file if you so will. This makes it easier for me to work through different issues one at a time - and sometimes postpone if things may be too hard to solve or take too long to implement.

One important aspect of this tutorial is, regardless over whether you are a "newbie" or a "knowbie", to understand how things work, the reasoning behind them, how they can be used and how they can be applied to help you in your endeavours.

The RBT Project appreciates having knowledgable users - there should not be any artificial separation between "developers" and "users", because in the end, knowledge should be made available for everyone, at their own discretion.

❗ In the event that you think that the tutorial is missing out some important information, report this as well please, so that this part of the Tutorial can be improved.

Now, without further ado, let's get started.



## Installing the RBT gem

You may have to first download and install one gem, the configuration gem.

```
| wget http://shevegen.square7.ch/configuration-1.0.18.gem
```

```
| gem install ./configuration-1.0.18.gem
```

Next, try to install the rbt gem:

```
gem install rbt
```



## Introduction

One of the main design decisions behind the **Ruby Build Tools (RBT)** Project is to remain **extremely flexible**.

That means that there will be no "one way to use them".

There will be multiple ways and you can pick whatever way suits you the most. Different paradigms may coexist with one another. For instance, there is a class that allows one to compile from a single file, with just the remote URLs listed in that file.

The RBT scripts will strive to remain logical, and consistent.

Complexity is ideally kept to a minimum, giving explanations whenever possible, so that the user is in full control over every decision in the end.

One example for this can be noticed for commandline options:

- If a long option is called "**--skip-extracting**" or "**skip\_extracting**", then a shortcut like "**se**" will - and should - work exactly as the longer version. This is valid for every commandline option - there are many aliases to them. So, if you see a long commandline option, you can also try a shorter variant of that - and it should work. You can also omit all "\_" - they are just used to make it easier to read the specific command at hand. **skipextracting** would work just the same.

The whole RBT project consists of various scripts, scripts which are the culmination of a very interesting endeavour into the [Gobolinux world](#).

This endeavour is explained in the following box "*History of the RBT Tools*", which you can skip if you are not interested in it.

Click here to toggle the visibility of the next paragraph.

### ☰ The History of the Ruby Build Tools [RBT] Project

When I started to write these scripts, possibly back in 2005 or 2006, I wanted to have them all written in **ruby** (my favourite programming language at that time, and still as of today), and write them on my own - so I actually would understand what they were doing in detail.

Back then - and I still am of this opinion even today - I thought that shell scripts ultimately tend to grow in complexity way faster and steeper than other programming languages would do. They also lean towards becoming really ugly to further extend with more code, at which point sooner or later maintaining them is no more real fun anymore. And when it is no fun to maintain something, you start to lose interest and eventually cease maintaining it or adding anything new to it.

In Ruby the situation is a **lot** better, as you can (re)arrange functionality freely into classes and modules, extend these later on easily, and use one of the various GUI bindings to present the user with an easy-to-use graphical solution.

In my opinion, Ruby is a much better language than shell scripts and the logic found in such scripts. The GNU Sourcimage project is trying to use only bash scripts, and they don't really evolve quite much from my point of view. Users prefer distributions like Archlinux for a reason.

Big software projects will hardly ever be restricted to shell scripts - it is only the Linux world which still embraces this archaic concept of using ugly shell scripts.

Also, well written Ruby code tends to be much nicer to read than the equivalent code in a shell script. Python code may be just as useful when compared to Shell Syntax, but since Ruby is my main language, I chose to use Ruby rather than Python.

In summary - I think **Ruby** provides you with more options than shell scripts will ever be able to give to you in the long run.

Let us reflect an important difference between a "traditional", and a "Gobolinux-like OS", because it is important to know about this difference with regards to using the **RBT scripts**.

• A traditional system conforms (more or less) **completely** to the FHS standard, and thus uses /usr, /usr/lib and all the directories the FHS mandates one to have.

This also means that the files of any given program will be spread around multiple locations and directories (/usr/bin, /usr/lib, /etc and so on). Most distributions conform to this setup, although sometimes they break the FHS standard (as did Sabayon on May 2009 when their LiveCD used /usr/qt/3)

In my opinion this approach is slightly problematic and not very elegant.

You can not easily uninstall programs again unless you use a package manager to help you (or somehow hope that you can do "make uninstall").

• A GoboLinux like system uses a slightly different approach - the AppDir approach. Every program is (or should be) self-contained in a single directory and will then get symlinked into other locations, into directories like /System, /Programs and so on. In a way this is similar to AppDirs, but it is also rare to see in the UNIX world in general; I think only PC-BSD follows a similar scheme, Whitix also follows a somewhat related model, and Mac OS-X also has a very similar scheme but focuses on a graphical approach with their AppDirs.

The **biggest advantage** is that this scheme is a LOT simpler to use, while remaining easy, pure and very clean.

If you want to remove a program, you just remove the **AppDir** directory. That's it basically.

No need for a package manager to "sanitize" your system again. (Additionally, GoboLinux uses "legacy" symlinks to stay completely compatible with the FHS. These symlinks are by default hidden from the user and can be unhidden using "gobohide --unhide target".

It is often amusing to read posts of people claiming that GoboLinux is not following the FHS - but every FHS-mapped dir will in fact work just as well on GoboLinux, "cd /usr/bin", "cd /opt" and so forth - so in fact the statement that GoboLinux is fully FHS compliant is 100% true.

The FHS does not forbid the use of symlinks, it merely mandates that these directories need to be available at that position, which they are.)

More explanations will follow in detail lateron, but for now, just remember the **basic distinction** between a traditional, and a Gobolinux-like System (Gobolinux was a pioneer in this area) lies in the **Filesystem layout**.

## Scope of the RubyBuildTools and Platforms

Ultimately, the **RBT tools** will not be restricted to only run on Linux, they will work on different platforms. However, the first concern will be to have them work under Linux. Wherever Ruby works, the RBT scripts should work on that platform as well.

For now please, use a terminal/shell and lets start.

Type or copy/paste the following (do not forget to press the **enter key**):

```
| rbt htop
```

If everything works, then you should be able to see some information printed on to the screen. The RBT scripts are compiling the application **htop** right now!

Once the scripts have finished compilation, you will be notified as to where you can find the compiled files.

! **Note that since as of November 2013, you can also pass in a full URL as argument.**

! **That way, you can download from a remote site and immediately continue from there.**

Examples for the latter way:

```
| rbt http://ftp.gnome.org/pub/GNOME/sources/jhbuild/3.10/jhbuild-3.10.0.tar.xz
```

```
| rbt https://downloads.sourceforge.net/project/open-cobol/open-cobol/1.1/open-cobol-1.1.tar.gz
```

This will generate the necessary .yml files, download and repackage (repackaging is optional). Then you can decide whether you wish to continue or not.

On a default, out-of-the-box system, htop can now be found under **/Programs/Htop/Current**, or rather \$PROGRAMS variable rather than hardcoded /Programs. Note that you can change this in the configuration settings too.

Lets go there to have a look:

```
| cd $PROGRAMS/Htop/Current; ll
```

You may notice the directory structure. It is how Gobolinux has layout their directory structure of directories within /Program as well, but there are some differences, which we will cover later.

Typically, you will find a **bin/** directory and a **share/** directory there. Inside bin/ lies the single binary called "htop".

You may also see a file called **CompileTimeInformation** which contains information about the compilation process.

Anyway, the most important point to note here is this:

→ *All of htop is self-contained in this directory.*

This in fact is the biggest difference to FHS-based systems. Others call this approach (more or less) AppDirs or similar.

Personally I prefer the word "self-contained" since it is more direct to the point that everything that belongs to the package directly, is aggregated inside one directory, and we wont confuse it with Mac OS X Appdirs for example.

If you want to, for instance, uninstalle htop again, simply remove the directory:

```
| rm -rf /Programs/Htop/
```

or just the specific version:

```
| rm -rf /Programs/Htop/0.8.1
```

❗ Note that I recommend to set the variable \$PROGRAMS so instead of hardcoding the above, rewrite it like this:

```
| rm $PROGRAMS/Htop/7.0
```

To get the value of \$PROGRAMS, simply do:

```
| echo $PROGRAMS
```

The removing of a dir should trigger an event, called **on\_remove\_directory**, and this will automatically invoke a few cleanup scripts.

This however, only works on Distributions specifically set up to deal with the RBT Scripts or if you use something like the Diamond Shell.

On most Linux Distributions you would find these things under **/usr/local** normally (if you compiled on your own **without** a --prefix option).

Other people like to compile it into their home directory, like **\$HOME/directory\_name/program\_version**.

Now, the problem with **/usr/local** is that it is an additional layer of complexity, with all the disadvantages the FHS already has. Additionally, **/usr/local** is typically exempt from the package manager of the Distribution in question - which means you are, once again, on your own.

The /usr hierarchy **\*must\*** be managed with a package manager/system manager, because the files were already spread around.

The FHS is a rather messy solution which causes more problems than it solves.

Only with the help of package and system manager can users manage their FHS-based systems - without that, it would quickly become an unmaintainable chaos.

Personally, I do not encourage you to do use the FHS, but there is one clear advantage:

- If you use the FHS approach, you will have less problems this way because it is the "standard" way to do things on Linux systems. The Gobolinux approach is a lot cleaner though.



It is rather tedious to persuade other people in the Linux world that they should not rely on inflexible hardcoded paths. Most individuals with an opinion are so **very** stubborn, dead-set on the "established" way to do something, unable to get their brains to think differently and consider other options as well.

It is especially annoying when you are confronted "upstream". A waste of time to argue.



Some people are so infected by this FHS doctrine that they hate other ideas, such as self-contained application directories. In doing so they fail to see that this solution is a LOT better and more elegant in the long run than the ugliness of the FHS-imposed, arbitrary rule-set.

(What is even worse, the LSB builds and **extends** upon the FHS.)

Back on track though - lets say you like the FHS approach (after all, Linux is or should be all about choice) and want to compile htop into a directory like /usr.  
No problem!

There are several ways to do this - one that I like is to modify the specific .yaml file in question.

Simply open up your local **cookbook.yaml** version - or the individual .yaml file in question, in case you follow the "one program per file" approach - like so:

```
| bluefish /System/Settings/cookbook.yaml &
```

```
| $EDITOR $COOKBOOK_DIRECTORY/ALL_COOKBOOKS/cookbook.yaml &
```

(Of course, use another editor like nano or vim if you like. For simplicity, the rest of the instruction set will refer to bluefish as the main graphical editor, just use the editor you prefer.)

This file, **cookbook.yaml**, is quite large. In November 2011 I had about 2600 registered programs there in all my cookbook files. Dont worry about reading everything in those files though ;- ) just jump straight to the entry called "htop:" (by pressing CTRL+F then typing htop:) and press enter.

You should now be at the entry for htop. Have a brief glance to see what is there.

(At the time of writing this tutorial, htop could be found in the file **cookbook.yaml**.)

Look at the important option " prefix:"

You will see a f next to it, which simply means "false".

As you may have guessed, a "t" would indicate a value of "true". RBT always tries to allow shortcuts when it is possible, but you can be explicit as well if you wouldd like to.

For now dont think too much about the meaning of traditional itself, just change the "f" letter to a "t".

This has changed the "traditional" option to be true, and true in this context roughly means "yes, we want to compile in a FSH way", thus the prefix target would become /usr.

If this convention confuses you, you could use an explicit setting such as this too:

```
| prefix: /usr
```

Though it is a bit longer to type.

**!** Please note that you can overrule this option on the commandline, via **--prefix=**, like so:

```
| rbt htop --prefix=$J
```

```
| rbt htop --prefix=/usr/local
```

```
| rbt htop --prefix=$HOME
```

**!** Note that the last option there, the **--prefix=\$HOME**, also is available via a short-cut flag:

```
| rbt htop --into-home-dir
```

```
| rbt htop --home-dir
```

**!** And you can also compile into your home dir, in an AppDir variant, by using this:

```
| rbt htop --appdir-into-home-dir
```

The above is equivalent to:

```
| --prefix=$HOME/Programs/Htop/1.0.1
```

```
| --prefix=/root/Programs/Htop/1.0.1
```

You can also omit the **prefix:** option entirely, it will **default** to `prefix=/usr/` in that case. But you could also put in `/opt` here or even `/opt/PROGRAM_NAME` - the last bit is a "macro" which will simply be expanded to the real program name.

I myself tend to use this here a lot in the yaml files:

```
| prefix: f
```

Which translates to "compile in a Gobolinux like fashion". In other words, a **standalone AppDir structure**.

Once you have done this change, save the file, and go to your shell again.

Then type this in your shell (and press the enter key):

```
| rbt htop
```

You will see that htop will be compiled, but this time it will use that other prefix - the `/usr` prefix.

Which means, after (a successful) compilation, you will find the htop binary at **`/usr/bin/htop`**.

*(Htop was created by Hisham who also started Gobolinux together with Andr ©, by the way. Htop is a really cute small program.)*

By changing a single letter in the `cookbook.yml` we have instructed our compile tools to build to another prefix. You can set other prefixes to this "traditional: value" setting, for example `/pkg` if you want to compile into the directory `/pkg` and so on.

**! You can also pass arbitrary -- configure options.**

For instance, you can do this:

```
| rbt utillinux ntrad --without-ncurses
```

This would attempt to compile the program called utillinux, use an AppDir prefix, and also pass in `--without-ncurses` as commandline option.

Let's review the concepts learned so far, in a nutshell:

■ Basically, all you had to do was to change one letter in the 'recipes collection', in order to compile it to another prefix.

This is one of the **principles of the RubyBuild Tools**. Things should be simple, clean and as logical as possible - for the **end user**.

IF we need complexity for some reason, then we need to be able to manage it with ease and explain why we have done so. It would be not much fun to be buried under a load of complexity.

If you have a proposal for a simplification, voice it.

Make sure you thought about it for a bit before issuing though, it makes our life easier if you are completely convinced that this can be useful to have.

One final word about this "traditional:" setting for now:

You can of course overrule this setting on commandline just as well.

So ultimately, you do not have to edit any of the various `cookbook.yml` files. The commandline must allow the user to overrule any setting, at runtime.

Or even try to ignore it completely and just provide a URL to a tarball of some sorts, to download and compile something.

To show you how to overrule the setting in the cookbook files, look at the following examples:

```
| rbt htop traditional
```

```
| rbt htop non_traditional
```

Of course, "compile htop ntrad" works too as does "compile htop t".

**!** If you try to pass both options (*non\_traditional* and *traditional*) then we will display a warning, and favour only one of those options given, which right now is the last option given. In general, options that come last are allowed to overrule earlier options given. We however display a warning in this case because it does not seem to make a lot of sense.

Please note - if you do not have a cookbook.yml file yet or if you want to generate a new one, you can do so by issuing:

```
| rbt generate_new
```

This will try to generate a new cookbookfile into /Depot/Temp and if this does not work, it defaults to the current working directory.

Lets look back at [cookbook.yml](#) now and contemplate on the size of it. It is indeed quite large... There may be some people who will say that having one big file like cookbook.yml is a bad thing.

I agree partially, and this is the reason why you can use as many .yml files as you want.

The RBT Scripts should allow you to freely build and install applications, and thus they should not restrict you. You can use multiple .yaml files or you could create and use standalone recipes: Enter The Recipes World!

More on Recipes later - stay tuned.

## Program Versions

You can overrule the version specified in the cookbook yaml file from the commandline. To do this, look at the following two examples:

```
| rbt nano --version=2.1.9
```

```
| rbt nano --version=2.2.2
```

```
| rbt nano --version=2.4.3
```

```
| rbt gtk+ --version=2.24.29
```

**!** Note though that this has a chance to fail right now, i.e. if the remote mirror does not have this specific version, or if you do not have it available locally.

You can also use a slight shortcut:

```
| rbt nano --v=2.4.3
```

You can also use a more complete copy-paste version setting (like you copy, then paste the full name), like this:

```
| rbt nano --version=pango-1.24.2.tar.bz2
```

The .tar.bz2 part, and the part before that, will simply be ignored.

**!** Also note that you can retrieve a listing of all locally available versions.

Use the following to get at this information:

```
| rbt gcc available_versions?
```

```
| rbt gcc ?
```

The result would be something like:

```
=====
- (1) gcc-3.4.6
- (2) gcc-4.0.4
- (3) gcc-4.1.2
- (4) gcc-4.2.4
- (5) gcc-4.3.3
- (6) gcc-4.4.2
- (7) gcc-4.4.3
- (8) gcc-4.5.0
=====
```

You can then compile one of these version, by doing i.e. this here:

```
| compile gcc v4
```

```
| compile gcc 4
```

You can omit the v too. The v is just a mnemonic shortcut for "version", but the position number alone suffices anyway:

```
| rbt gcc 3
```

This also works with negative numbers, like:

```
| rbt perl -3
```

```
| rbt perl v-3
```

The above would compile the Perl version that is the -3 most recent. In other words, we start to count from the current version, and step back 3 times from that. (This will only work if you have the perl versions available locally).

If you give a bogus entry like v-66126363 and that version could not be found, we will instead treat it as input that you meant to refer to the very last version anyway.

A positional index with a word works for the first and last entry too. The convenient keywords for this are called "first" and "last". So rather than type 1 or 8, you can do this:

```
| rbt v first
```

```
| rbt v last
```

The last version can be useful.

Aliases to these do exist:

```
| rbt pcre vlatest
```

```
| rbt pcre vnewest
```

```
| rbt pcre latest
```

```
| rbt pcre newest
```

You can also query for the current RBT Versions.

To do this, do either of the following:

```
| rbt htop --version
```

```
| rbt VERSION?
```



```
| rbt --version?
```

Since as of December 2015, you can also compile a random version.

This requires of you to have the other versions available locally though.

Usage example:

```
| rbt gcc --use-random-version
```

The scripts are written in ruby, but for several reasons one might want to prefer using shell scripts instead.

You can generate the small shell scripts by issuing something like:

```
| rbt x to_bash
```

Or shorter:

```
| ry x tobash
```

**! The name "x" can be any other char or word, it does not matter. The command invocation is simply to start generating the shell scripts. This behaviour may change in future versions, if we decide to move the functionality to another class.**

A single file will be created as a result, it can normally be found under

```
/System/Settings/Scripts/bash_programs_rc
```

Now, this is pretty handy, but what if you do **NOT** want to compile at all?

No problem. Use a package.

Example:

```
| rbt python install_package
```


```
| rbt fox package
```

This will install python from a binary package (and download it from a remote site if needed, i.e. if it can not be found locally.)

There are many more options available to the scripts that handle these things but for a starter, this should be it.

But what if you **dont** have this package on your computer?

By default the RBT Scripts will assume that you already downloaded this package and put it in some location, like **/Depot/Packages**.

But you can download  a package with the scripts too:

```
| rbt python download
```

This would (try to) fetch the latest (source package) of python and download it to your machine.

**!** Note that this command may fail due to several reason, such as that the remote server is not available or that the location has moved to another website and so on.

As of September 2014 we also added a configuration option to yaml files called:

**do\_not\_download**

Currently compiling programs

You can query which program(s) are currently compiled by issuing:

```
| rbt currently_compiled?
```

Porg

You can use porg to install a program.

To query whether porg is in use, you can issue this command:

```
| rbt --use_porg?
```

Porg will keep track of program files that are installed.

You can also disable porg on the commandline via:

```
| rbt --disable-porg
```

```
| rbt --disable_porg
```

```
| rbt --disableporg
```

Or just directly edit the respective .yml file.

Strip

You can use strip to reduce the size of the binaries.

This will be done via the configuration option called use\_strip.

If this configuration option is set to true, then we will run "strip --strip-unneeded" by default.

## 🌐 Installed files

You can ask RBT which files belong to a specific program.

Do this like so:

```
| rbt htop --installed-files?
```

```
| rbt htop --installed-files
```

## 🌐 M4 Macros

We can symlink .m4 Macros into /usr/share/aclocal.

The RBT project allows you to also generate some HTML pages, similar to how the LFS (Linux from scratch) project. The central ruby script that does that is called **central\_information\_agency.rb**.

In order to generate a bunch of HTML pages, do this:

```
| cia lfs
```

or

```
| cia html_pages
```

You can specify another source directory (base\_dir) by doing something like this:

```
| rbt mantis --source_dir=/Depot/j
```

You can batch-compile like this:

```
| rbt all proto
```

This will compile all proto tags.

## 🌐 UpdateEntry

You can use UpdateEntry to update an entry in one of your cookbooks.

UpdateEntry will store the last downloaded program in a specific file. You can then make use of this from RBT, by doing something like this here:

```
| rbt lastdownload
```

## 🌐 Binary and Library Checking

You can also compile a program if you only know the name of a binary. For example, to compile the package "xine-ui", you could supply the name of one of its **binaries**, like so:

```
| rbt xine-check
```

xine-check is a binary of xine-ui. The above will actually compile **xine-ui**.

Another example:

```
| rbt basename
```

**basename** belongs to the package **coreutils**, and thus we will attempt to compile coreutils.

❗ **Note that you can change this in the config file, look for the option ``check_for_binary_names``.**

One last example - say, you know the binary name which is called ``startx``. You attempt to compile it, and we will set it to the proper package name:

```
| rbt startx
```

startx belongs to xinit and the above is the same as:

```
| rbt xinit
```

And the same mechanism works for library names just as well (if they were registered in RegisteredLibraries.yml that is).

So for instance:

```
| rbt libqui.so
```

The above would actually try to compile QT4 and would thus be the same as if you would have entered:

```
| rbt qt4
```

In the end, this is just a slight convenience to make your life easier.

The default behaviour for the scripts is to treat the given input as alias to a real program name, and if this alias is not found we check for binary names, then for library names.

■ Sometimes, this is not desired. Consider that you input "ld". Now let us also assume that you already have an entry called "ldmud" in your scripts. But "ld" is also the name of the binary "ld" in the binutils package.

To force that we seek for the binary "ld" and not for "ldmud" you could do this:

```
| rbt ld treat_as_binary_name
```

Same works for library names as well:

```
| rbt ld treat_as_library_name
```

## 🌐 **ldconfig**

Note that the current behaviour is to (silently) run ldconfig after compilation has finished. This can be modified in the config file, see the option there called ``run_ldconfig``.

You can also disable the running of ldconfig from the commandline for the program you wish to install.

In order to do this, do:

```
| rbt htop no_ldconfig # Main command
```

```
| rbt htop noldconfig # Alias to the above ^^
```

```
| rbt htop nold
```

## 🌐 **Convert to FHS**

If you ever want to convert a given package to FHS style, use this:

```
| rbt php5 convert
```

```
| rbt php5 cpackage
```

```
| rbt php5 convert_package
```

```
| rbt nimrod cpa
```

This is, right now, using the files under /Programs - or more accurately, it uses the setting specified in your configuration yaml file, the value of "programs\_dir" - , and will then copy these into the /usr prefix.

To get a listing of available programs, you can do:

```
| rbt php available_programs?
```

```
| rbt php available_cookbooks?
```

This will iterate over all programs specified at programs\_dir.

## 🌐 **Installing older versions**

What if you want to install an older version of python?

We briefly addressed this elsewhere on this page already, but we will now go into more detail.



The **cookbook.yml** normally "supports" only the version you specified in "archive\_name:"

For example, if you jump to the "python:" entry in the cookbook.yml file, you will see that its version is 2.5.1.

🚫 In later revisions python apps are gathered inside python\_cookbook.yml instead of the general cookbook.yml file.

You can change this permanently by writing another name/version there.

Because editing of various cookbook.yml files can be a little annoying, and because flexibility is a good thing, there exists another way to specify a different version manually on the commandline:

```
| compile python --version=2.4.1
```

This works with every version, even "newer" ones.

The name of the version should however be part of the filename.

For more information about this, and an even **easier way**, jump [here](#) to.

## 🌐 Viewing LFS and BLFS URLs

The Linux From Scratch (LFS) Project has instructions in how to compile programs from source.

These are often so useful that we have added a BLFS URL to some programs.

You can try to show this URL by doing this:

```
| blfs curl
```


This would show the BLFS URL of curl.

As of July 2017, you can also directly paste the content of the remote BLFS webpage onto the terminal, if there is a remote BLFS page associated with a given program.

The way to do this is:

```
| ry gcc --paste-blfs-page
```

## 🌐 Viewing all packages

To  view a list of all packages on your system, do this:

```
| rbt php packages?
```

```
| rbt python packages?
```

The result would be something like this:

```
| ["/Depot/Packages/Python-2.5.tar.bz2"]
```

## 🌐 Package up the RBT Scripts

You can package all the latest RBT Scripts like so:

```
| rbt x package_cookbooks
```

This will create a directory in /Depot/Archives and package the RBT Scripts into that directory.

## 🌐 Dump all Recipe files

To dump all recipe files (Gobolinux files, ebuild files, .rb recipe files) do this:

```
| rbt x dumpall
```

## 🌐 Suppress Commandline Output

To suppress configure spamming useless information to you, and to also suppress other things, do this:

```
| rbt htop v2 suppress-output
```

```
| rbt htop v2 soutput
```

```
| rbt htop v2 sout
```

```
| rbt htop sout
```

You can also make all output completely disappear:

```
| rbt htop --be-completely-quiet
```

This will suppress all output.

## 🌐 Generate and use shell completion

You can generate a bash shell completion file from within RBT.

In order to do so, try this:

```
| rbt x generate_shell_completion
```

This will write a simple shell script which you can source if you want to use ry to compile programs.

❗ **This also works for Zsh by the way.**

The file in question is called **generate\_shell\_completion.rb**.

If called standalone from the commandline directly, it will generate .sh shell files for both Bash and Zsh.

## 🌐 Find out local location of tarballs

To find out the local location of a tarball use this:

```
| rbt gcc local_url
```

```
| rbt gcc lurl
```

The result will be something like this:

```
| => The local file for gcc can be found at:
```

```
| /Users/x/SRC/GCC/gcc-4.6.1.tar.bz2
```

## 🌐 Build inside a subdir

Sometimes you dont want to build in the current dir, but want to use a subdirectory instead. (For example, glibc requires you to build inside a subdirectory).

To do this from the commandline, simply do one of these things here:

```
| rbt php --subdir=PHP_BUILD_DIRECTORY
```

The above would build in the directory called "PHP\_BUILD\_DIRECTORY". If this directory does not exist (which is the case usually), then we will create it.

The default name for the subdirectory is **BUILD/** and you can do default to this directory:

```
| rbt pango -b
```

This will create a **BUILD/** directory and change the working directory into it while compiling.

In practise, defaulting to the BUILD/ directory is the easiest solution if you want to compile in your own directory.

❗ Please also note that you can specify build directories in the cookbook file as well - look at the description of the variables called ``use_build_directory``, and ``use_this_build_directory``.

Alternative syntaxes exist of course. If you want to use the default build directory for a project you compile, you can pass the option `usebuilddir` like so.

```
| rbt automake usebuilddir
| rbt htop usebuilddir
| rbt php use_build_dir
```

You can also specify a specific build directory to use.

One way that this works is via this:

```
| rbt htop --use-this-build-dir=/Depot/j
```

❗ We will append a trailing / if it is a directory and if it does not have a trailing /, so the above is the same as passing `/Depot/j/`.

## 🌐 Register all programs

It may be that you want to register all programs in the yaml Database. To do this, run this command:

```
| rbt --register_all_programs
| rbt --rap
```

## 🌐 Makefiles

Most open source projects on Linux that require you to compile C or C++ code, come with a file called a Makefile.

The command "make" can be used to interpret the rules stored in said file.

Sometimes no such file is provided though and instead, some other variant is used. For instance, a generic file called **Makefile.linux**.

This file is a template file that can be used to generate a Makefile file.

The common syntax for this would be:

```
| make -f Makefile.linux
```

However had, the RBT scripts will automatically check whether such a file exists, in the event that no Makefile exists. If this is the case, then the make command will be modified to first create this Makefile.

## 🌐 Find out how many programs are registered

To find out how many programs are registered, do one of this:

```
| rbt --n_programs?
| rbt n_registered?
| rbt registered?
```

It will then feedback something such as:

! Cookbooks::ReportTheRegisteredPrograms: 2619 programs are registered in our cookbooks as of 30.03.2014.

! Cookbooks::ReportTheRegisteredPrograms: 2832 programs are registered in our cookbooks as of 18.02.2016.

! Cookbooks::ReportTheRegisteredPrograms: 2862 programs are registered in our cookbooks as of 18.08.2016.

## 🌐 Extracting Archives

You can extract a tarball and finish the program. In order to do this, try:

```
| rbt php eonly
```

This would extract the latest PHP tarball and then exit the program.

Also note something else, related to Extracting. You can use the argument `--extract_to` (or its alias `--homedir` or `--temp_dir`) to specify another **extract target** to use. You can even use shell environment variables here.

Usage Examples:

```
| rbt htop --extract_to=pwd
```

```
| rbt htop --extract-to=$J # This makes use of the shell environment $J
```

```
| rbt htop --temp_dir=/tmp
```

```
| rbt htop --homedir=pwd
```

## 🌐 View urls of certain programs

To view the URLs of certain programs, you can do this:

```
| rinfo
```

```
| rinfo video
```

```
| rbt libxrandr url
```

```
| url libxrandr
```

Also note that you can set the program name by using `USE_URL` - we will then use the URL from `url1`

The `USE_URL` is automatically assumed if you only provide a remote URL.

## 🌐 GuessCompile

Via `GuessCompile` you can try to make a simple compilation.

This must be explained in more detail ...

First, you store all URL entries into a Database. By default this Database will be stored in the file `program_urls_only.yml` and can be found in the `YAML` subdirectory of the `RBT` project.

For instance, on my machine this is under:

```
| /Users/x/DATA/PROGRAMMING_LANGUAGES/RUBY/SRC/RUBY_BUILD_TOOLS/lib/rbt/YAML/program_urls_only.yml
```

The next step is to try and compile the latest entry of a specific program, like so:

```
| GuessCompile bash
```



You can also get a list of registered programs there via:

```
| GuessCompile ?
```

! Also keep in mind that as of January 2014 we will try to ensure that all entries in a cookbook-yaml file are valid. If they are not, we may report to the user or even refuse to compile unless this error was corrected.

## 🌐 Use the first URL instead of program name:

You can overrule the program name and specifically force the usage of the first URL entry.

You do this like so:

```
| rbt ffmpeg use_url
| rbt libgtkhtml use_url
```

Do note though that for this to work, it requires a proper url1 entry in the cookbook file.

## 🌐 Use the first URL instead of program name:

You can output all URLs like so:

```
| rbt --output_all_urls
| rbt all_urls?
```

! Note that this will also save this result in a file.

## 🌐 Use a specific compile version

You can overrule the program\_version? that will be used for app-dir like compilation like this:

```
| rbt glib --compile-version=2.16.33_ROEBE
```

The above command will try to compile glib, but will use a specific version that you defined here. Typically it should be only numbers, but you can also use non-numbers.

The above would translate into this prefix:

```
/Programs/Glib/2.16.33_ROEBE
```

## 🌐 md5sum

Verify the md5sum:

```
| rbt gcc4 md5
| rbt libogg md5
```

! Note that you can disable the md5sum calculation, by using the switch --no-md5sum:

```
| rbt htop --no-md5sum
```

## 🌐 Check locally

You can check for the latest packages locally via

```
| rbt htop latest
```

## 🌐 Feedback about a program

To get feedback about i.e. php, do:

```
| rbt php feedback_all
```

```
| rbt ruby fall
```

You can also make use of SanitizeCookbookDataset.

```
| rbt php --scookie
```

## 🌐 Use multiple yaml files

You can use more than one cookbook yaml file. Just specify this in the config file, at the entry "**cookbook\_directory**".

The first entry should always point to your most important cookbook file though, the one you use most of the time, or is the largest cookbook file.

You can of course just use one big cookbook file.

Additionally, you can use a "one-file-per-program" approach.

I myself transitioned into this approach several years ago. It simply was easier to update information stored in individual directories.

You can use a special "macro", which will load all files of a given directory.

Example:

```
| - /some/directory/YAML/
```

The above would load all files inside that directory.

**! Yes, this was changed some time ago, I think in 2009. RBT will automatically try to load all .yaml files from that directory.**

## 🌐 Querying whether we have a binary file

If you want to find out if we have a registered program, whose binary name is "file", do this:

```
| rbt file binary?
```

In other words, you can find out if we have such a binary name.

Different programs will have different binary names, and sometimes it is important to find out which binary belongs to what package.

## 🌐 Query Inclusions

To find out whether all your files are included in the cookbook file, do this:

```
| rbt htop inclusions?
```

It will report which files were not included.

This is mostly helpful when you want to cleanup unneeded packages.

## 🌐 Static Compilation

You can compile an application statically by passing **static** to **compile.rb**.

Example:

```
| rbt make static
```

❗ **Note that this will not work for every program.**

In the event that you, on the commandline, try to enable static compilation, and the yaml file specifies otherwise, such as via a configure switch like `--disable-static`, we will ignore the latter option and allow the user to specifically overrule this setting. RBT always assumes that the user knows what he is doing.

Note that you can also zero these flags via a commandline option:

```
| rbt strace --no-compile-time-flags
```

```
| rbt strace --plain
```

This option means that all CFLAGS etc... will be ignored altogether; that is, they will set to an empty again.

## 🌐 Searching for something

To search for something, do this:

```
| rbt ruby search
```

To find a specific list of packages, do this:

```
| rbt pyt*
```

The `*` will tell us that we intend to find all packages starting with `pyt`.

❗ **Be careful though if you use a shell like Bash or Zsh - the `*` can be expanded by the shell.**

## 🌐 Using another prefix

You can specify a different prefix to use when compiling a program.

To do this, do:

```
| rbt gimp --prefix=$J
```

```
| rbt gimp --prefix=/opt
```

Some nifty shortcuts exist. For example, to compile gimp into a standalone directory, into `/Programs/Gimp/LatestVersion`, you can do:

```
| rbt gimp ntrad
```

```
| rbt gimp non_traditional
```

❗ **Please also note that you can use the Option `noprefix` to avoid using any specific prefix. (This is the same as passing `--prefix=/usr/local` though on GNU systems.)**

If you have the need to find out the prefix that will be used, do this:

```
| rbt htop prefix?
```

## 🌐 Specify another program to compile

You can specify another program to compile.

For this there are currently two ways possible:

```
| rbt x --this_file=/path/to/mousetweaks-3.1.1.tar.bz2
| $RBT/Compile.rb /path/to/mousetweaks-3.1.1.tar.bz2
```

## 🌐 Handling Dependencies

To get with dependencies do:

```
| rbt php with-deps
| rbt gcc3 wdeps
| rbt feh wdeps
```

Dependencies can be specified as **ranged**, which means that they will be valid only under some conditions.

For example, consider the following entry which you can find in some entries of the cookbook:

```
| glib2 >= 2.17.6
```

This entry means that the program requires glib2 of at least version 2.17.6 or higher.

The Ruby Build Tools will notify the user if the dependency does not match. The default compilation mode will be to interrupt compilation when a dependency is not met.

## 🌐 Merge Cookbook Files

If you do have multiple cookbookfiles (or if you want to merge your existing cookbooks into one new file), you can do this:

```
| rbt php merge
```

By default this command will merge into the file DEPOT\_DIR+merged.yml

## 🌐 Debug

If you want to debug the RBT Scripts, you can set the **@debug** value on the fly via:

```
| rbt x debug
```

This will toggle the debug value simply. Set this to true if you want to debug. If set to true then we will feedback extra information.

You can also query the debug value by doing this:

```
| rbt x debug?
```

## 🌐 Applying a common operation

You can apply a common operation to multiple programs at the same time. First, let us assume that you want to compile all versions of GTK:

```
| rbt gtk all_versions
```

This would compile all the versions of GTK you have downloaded.

## 🌐 Stow

If you want to use stow, simply set the configuration option for Stow to true (in the **configuration file**)

You can query whether you have stow enabled, by doing:

```
| rbt --use-stow?
```

## 🌐 Included program

You can find out whether a specific program is included or not by doing this:

```
| rbt htop include?
```

## 🌐 Whereis

If you forgot where you defined your program entry, you can use **SourceDefinition.rb** to locate it. This is aliased to the action `where_is`.

So, try this:

```
| wis gcrypt
```

This will tell you where `gcrypt` is installed.

## 🌐 Symlinking all programs

If you want to (re)symlink all programs in the main directory called **/Programs/**, just do this any of these:

```
| rbt x symlinkall
```

```
| rbt x traverse
```

```
| rbt --resymlink
```

```
| rbt --symlink_every_program
```

The last option may quite possibly be the easiest one to remember.

**! A note about symlinks. The GoboLinux way is to keep the Version as a Symlink, under /Programs/Name/Version. RBT also uses this scheme when compiling in a gobolinux-like (== App)way.**

Say that you have successfully compiled `htop` version 2.0.2. Then, a symlink will also be set, called **Current**. This one could then be found at:

```
/Programs/Htop/Current
```

Which would point at:

```
/Programs/Htop/2.0.2
```

## 🌐 Sleep before continuing

You can use a **delay** before compiling (or doing another action).

This works like so:

```
| rbt htop sleep5
```

```
| rbt htop sleep55
```

That is technically the same as doing

```
| sleep 5; ry foo
```

or

```
| sleep 55; ry foo
```

This would mean that you would want to wait for 5 or 55 seconds before starting your action.

## ✦ Downloading Source Files

As explained elsewhere in this document, you can download the source code of different programs via RBT. So for example, say that you wish to download the sourcecode of python from its official homepage, then you can do this:



```
| rbt python download
```

This would attempt to download python, as specified per the `python.yml` file.

An alternative syntax (*=shortcuts*) exists as well, e. g.:

```
| rbt python --downloads
```

```
| rbt python down
```

```
| rbt python wget
```

```
| rbt python w
```

It is up to your personal preference what you wish to use here.

I prefer to use "wget", but "download" is probably a better, more general description of the process in question.

Please also note that you can specify different versions here.

This can be done like in this way:

```
| rbt kernel --version=2.6.30.1 download
```

This would attempt to download the kernel version 2.6.30.1

Note though that this has a chance for failure, and thus might fail, e.g. if the ftp site changed, so it is no foolproof mechanism.

Also note that there is a configuration option which allows you to automatically attempt to download a missing file.

## ✦ Subversion

You can try to download SVN sources, by issuing this command:

```
| ry dosbox svn
```

**!** Note that this depends on the setting `svn_url` in the respective cookbook file. The `svn_url` setting must be specified, as otherwise RBT can not download the code via svn.

## ✦ Compiling several programs in one go (Chain-Compiling)

Compiling several programs one-after-the-other is what the RBT tools will refer to as *chain-compiling*.

It is handled by the ruby script called `chain_compile.rb`, **class RBT::ChainCompile**. The code can be found within the file called `$RBT/utility_scripts/chain_compile.rb`

This script allows you to compile several programs in one go, thus making your life easier if you need to compile several programs quickly.

There are basically **two ways** to trigger this.

The first one uses `Installer.rb`, i.e. the file we aliased to ``compile`` earlier on.

The syntax for this is:

```
| rbt sdl wrap # This is no longer working. I removed this feature for now. If you  
| need this feature, use "chained sdl" instead.
```

```
| rbt sdl chained
```

Because the above idiom may be commonly used, you may omit the last argument if you prepend a `@` to the name.

For example:

```
| rbt @sdl
```

In this regard, `@` acts as a shortcut for wrap/chained mode.

However, there were a few design problems, and for now this is disabled - use "chained sdl" instead. One day I may reenable this feature.

The other way uses chained directly. Example:

```
| chained sdl
```

Personally I prefer the latter version "`chained sdl`", mostly because I feel it is cleaner to use another command for compiling multiple programs in one go.

❗ In this example `chained` is aliased to `ChainedCompiling.rb`.

In later versions I switched to "chained" command instead of chained.

Examples with some modular xorg chains follow:

```
| chained xorg_protos
```

```
| chained xorg_apps
```

The above commands would compile several xorg related programs.

This is also doable from within rbt itself, as of June 2017. Issue this command:

```
| rbt protos
```

To find out which chain is used, simply look at the `cookbook_chained_programs.yml` file.

If you only want to compile a single program from within that chain for some reason, you can use specific positions.

For example:

```
| chained xorg_apps 3
```

would compile the third program of `xorg_apps`. (At the time of this writing, this program would be **beforelight**)

The default is also to report the size of the tarballs used in this chained compilation process - this can be altered via the config file. (At March 2009 this was not a very sophisticated solution - it needs to be heavily improved.)

The info for compiling several programs at once is written (and can thus be changed) inside `cookbook_chained_programs.yml`

```
| cat $COOKBOOK_DIRECTORY/cookbook_chained_programs.yml
```

If you want to copy only the headers of a program, do this:

```
| rbt gmp only_headers
```

## 🔗 Do not show programs names

By default, the RBT Scripts will show the name of the Ruby script that is running an execution. You can disable this by doing:

```
| rbt htop dont_show_names
```

```
| rbt htop dontshownames
```

## 🔗 The Scripts and their (Infra)Structure

The scripts are written in 🐘 **ruby**, and although I have tried to keep them rather simple, at times they can be a bit complex (and some code parts are quite dated, and thus may not be easy to understand - not even for myself.) I do however often reconsider design decisions, and start rewrites every now and then to clean up the existing infrastructure.

If you want to explore these scripts, you should first look at

`Query.rb`

This file consists of one class, which has only one basic job:

Load the data from the `cookbook.yml` file by using `DatabaseLoader.rb` and answer queries the user gives us.

**DatabaseLoader.rb** will load the yaml dataset, and then you can actually use `Query.rb` to query whether a package is included or not.

That is basically all these two files do.

`DatabaseLoader.rb` is much larger because it ensures that the dataset is also correct.

Now, with this information we come to the most important script of the Ruby Build Tools - the **Installer.rb**

This is a bit complex. What you must keep in mind is that this file tries to fulfill many different user wishes at once: A user may want to install, uninstall, compile, copy, get the URL of a program, create .deb packages, create dependency graphs, create shell scripts, ebuilds, Gobolinux recipes, md5chksum it and and and.

Now add all these tasks together in one program, and the result will be quite large (though the code should be rather straight-forward). Maybe I will split it up in several new files, but for now this approach must suffice.

There are helper files available too.

The main purpose of `Installer.rb` is to be the **general wrapper** around these various tasks.

Basically, whatever you want to do, you should be able to do so via `Installer.rb`.

It is a basic duty of this script to invoke other scripts as needed as well, acting as THE general wrapper script

We now move to a more detailed section of how `Installer.rb` actually works.

### compile.rb

If you pass arguments to class `Compile`, then all "-" will be removed.

It makes it easier to label packages this way.

`Installer.rb` provides some logging capabilities:

The **general log file** will usually be stored here:

```
/InstalledSourceFiles.html
```

Additionally, a file called

```
/System/Information/LastCompiledProgram.html
```

will be created.

`Installer.rb` accepts the program name as command:

Typical Invocation examples are like so:

```
| rbt ebml
```

```
| rbt libxml
```

```
| rbt libogg
```



**!** If you omit the name of a program, the scripts will try to automatically guess a proper name using the script `GuessName`. It is however recommended to set the name manually, rather than guessing.

Also note, since December 2009 `GuessName` has been mildly deprecated. In Jul 2011 `GuessName` has been removed. The macro `USE_URL` (or shorter, `URL`) will be auto-used in case you forgot to set the program name.

To install in Gobolinux style (via the commandline, which will overrule any settings in the respective cookbook at hand), you can do something like this:

```
| rbt python gobo
| rbt gcc gobostyle
| rbt sed gstyle
| rbt gcc nt
```

I recommend you to try out `gobo`, it opens eyes. ;-)

In the above first example, Python would be compiled into this directory by default:

```
| /Programs/Python/VersionNumber
```

In effect, this will overrule any setting defined in the YAML file. The reason for this behaviour is that the user specifically wanted this behaviour.

Whatever the user wants to use, is the way-to-go for us.

If you want to skip extracting (i.e. because you already have it extracted) then do this:

```
| rbt php skip_extracting
```

or more formally:

```
| rbt gimp --skipextract
```

or for the lazy guys (includes me)

```
| rbt php se
```

You can also skip running any configure scripts, by doing this:

```
| rbt php skip_configure
| rbt php sc
```

And you can combine this too, like:

```
| rbt php skip_configure skip_extract
```

If you want to extract the package, and also run the configure script, but do not want to start compiling via `make` and `make install` yet, do this:

```
| rbt binutils dc
```

`dc` means "dont compile", and this works too:

```
| rbt binutils dont_compile
```

Please note that the last compiled program will be stored in a yml file. This will then allow you to easier access the last compiled program and a pointer to it will be available if you use `:last` or `@last` as commandline argument.

For example, you can do this:

```
| rbt :last url  
| rbt @last url
```

This allows you to recompile the program quickly.

```
| rbt @l
```

Yes, @l is an alias to @last.

Also note that we will store the configure options that are used to compile a program in a yaml file. Typically this is **last\_compiled\_configure\_options.yml**.

You can also query configure options for a given program.

To do this, try this:

```
| rbt gcc3 configure?  
| rbt htop configure_option?
```


Note that the script **ExclusiveLogic.rb** will check if a configure script makes sense. If you pass both `--enable-fading` and `--disable-fading` to the `Installer.rb` script, for instance, then `ExclusiveLogic.rb` will warn you of this mistake. This will allow you to fix the problem before you continue.

If you do not want to symlink after compilation has finished, add the option **dont\_symlink** (or shorter, `ds`):

```
| rbt gcc4 dont_symlink  
| rbt gcc4 ds
```

Of course you can freely combine those instructions together:

```
| rbt gcc4 dont_symlink skip_extracting
```

To create a new  package, you use this command:

```
| rbt fox cpackage  
| rbt htop cpackage  
| rbt fox cpackage archlinux
```

Default for this (if you omit i.e. `archlinux`) to create a `.deb` packages.

If you pass i.e. `archlinux` to it, we will create a package for Archlinux.

You can explicitly create a `.deb` Package as well by using something like:

```
| rbt gcc3 cdeb
```

If you want to create a slackware package for the editor "tea" instead, do this:

```
| rbt tea slackware
```

If you want to create a gentoo ebuild, you can do this:

```
| rbt gcc3 ebuild
```

## ! A word about conventions ...

After the `./configure` string, we will normally use `--prefix` first.

After `--prefix` come the other options.

You can also create dependency graphs, although this right now is not top notch state of the art ... ;-P

See Query for more information about this.

To search/search for certain packages or entries, do something such as:

```
| rbt ruby search
```

```
| rbt gcc s
```

If you instead have several chained programs such as `e17`, then you can do something like this:

```
| rbt e17 wrap url
```

```
| chained e17 url
```

For the crazy guys, you can do this:

```
| rbt foo all
```

This basically will compile EVERYTHING!

It is probably not a good idea to do this, but then again, whoever *wants* to do it can do so. Perhaps for test purposes... go on.

(This is no different than running each program entry in the `cookbook.yml` manually anyway, no magic at all.)

If you however want to compile a defined set of programs, there exists another way - chained programs in `cookbook.yml`

Via chained programs you simply specify a certain chain of compilation. I use this to compile `xfce` for example, or `kde`.

Via **generate\_custom\_yaml** you can dump specific programs into a new `yml` file.

Usage example:

```
| rbt gcc dump_yaml
```

```
| rbt gcc dyaml
```

```
| rbt x generate_custom_yaml
```

`Installer.rb` has different install procedures like `"python setup.py install"`, which are stored in a macro such as `PINSTALL1`

You can use this macro in **cookbook.yml**

Currently these Types are supported in the Ruby Build Tools scripts:

- **.rb Files (ruby)**
- **.py Files (python)**
- **Cmake Files**
- **GNU Autoconfigure Files**

You can copy the whole `yml` database by doing this:

```
| rbt x copy_yaml
```

You can disable the renaming of the KDE Konsole tab, even if the configuration setting specifies otherwise, on the commandline, by issuing something like the following:

```
| ry pango dontrenametab
```

```
| ry pango dont-rename-tab
```

```
| ry pango --dont-rename-tab
```

If you dont want to display colours, either disable them globally in the config.yml file (the entry is called `use\_colours\_output`) or pass --nocolor (or --nocol or --nocolour) on the cmdline like this:

```
| rbt libmemcached --nocolor
| rbt libmemcached --nocolor
| rbt libmemcached --nocolour
| rbt libmemcached --nocolours
| rbt libmemcached nocolours
```

**!** Note that you can also change the colours. For this, edit the file colours.yml

The above will, however had, only disable the colours for the current run.

You can find out whether you use colours specifically or not, by issuing this command:

```
| rbt --use-colours?
```

You can also permanently disable the colours by issuing this command:

```
| rbt --permanently-disable-colours
```

### Cookbook.yml

You can apply patches, like a linux kernel patch, by adding an **apply\_patch:** entry in the cookbook.yml file. Also note that you can use this entry in a conditinal. For instance:

```
| apply_patch: http://connie.slackware.com/~alien/slackbuilds/id3lib/build/id3lib-3.8.3_gcc4.diff if PROGRAM_VERSION == 5.14.1
```

The above line would only be applied if the program version would match. This allows us to apply patches in a conditional way.

Since 13.09.2007 you can also manipulate the cookbook.yml file via a .cgi Page.

You are currently viewing this page :)

Just look at the following formular.

#### Remove Program

Use this formular if you want to remove a program from your system.

Name of program which should be removed:

You can show the last configure option used by issuing:

```
| rbt last_options?
```

**!** Note that this requires you to have at least compiled one program, as otherwise we have not yet stored a configure option.

You can display the last n compiled programs.

This is done like so:

```
| rbt --last-compiled?=6
```

```
| rbt --last-4-compiled?
```

We will use a base directory for logging output, which will be used to store log files and similar files.

If you want to know where the log directory is on your system, then you can ask rbt, using this command:

```
| rbt rbt_log_dir?
```

## 🌐 Programs Settings

On traditional Linux structures, you store system-wide settings under /etc. We will pursue another approach.

Each directory will keep its own Settings directory (as it was done on GoboLinux as well).

If you want to backup all these stuff, you can use

```
ProgramsBackup.rb
```

## 🌐 Install only the bin/ subdirectory

You can also install only the bin/ subdirectory. Every other directory that belongs to this program will not be installed.

Example:

```
| rbt htop --only-binaries
```

## 🌐 Compile and upload the binaries

You can compile and then upload all binaries. This currently only works for my own FTP host.

Usage Example:

```
| rbt make ntrad upload-binaries
```

This would upload all binaries of make, from **/Programs/Current/Make/bin**.

**! Do also note, for my own system (or if you enable this by setting an environment variable called IS\_ROEBE to 1), we can upload the current ruby to a remote website via ftp. The following example shows how to invoke this:**

```
| ry --upload-ruby-binary
```

## 🌐 Convert Program

You can convert a program too.

To convert a program such as htop to non-traditional, do this:

```
| rbt htop to_programs
```

## 🌐 Gobolinux Recipes

In order to create Gobolinux Recipes for i.e. python, do this:

```
| rbt python goborecipe
```

Note that this will create the recipe in the current working directory.

## 🌐 The Highest n programs

You can show the largest local programs, a Top 50 for instance, by doing:

```
| rbt x highest
```

## 🌐 Slackware slack-desc files

You can create a slackware slack-desc file.

Do this:

```
| rbt ruby --generate-slack-desc-file
```

```
| rbt ruby --slack-desc
```

## 🌐 Standalone Yaml Files

To create standalone Yaml files, do this:

```
| rbt ruby create_standalone_yaml
```

## 🌐 /System/Tags

The /System/Tags directory is a special directory. If you set the configuration option called **use\_tags** in the file **configuration.yml** to **true**, then we will honour the **tags: variable** in the individual cookbook yaml file.

What does this actually mean? Let's look at a specific example.

Say you have an **editor** like vim, defined in a yaml file.

The yaml file option for vim includes a line such as:

```
tags: editor
```

Now we have tagged the program vim to be an editor. When vim is then compiled, we also symlink the vim binaries into:

```
| /System/Tags/Editors/vim
```

This is somewhat useful in that you can group different programs according to what feature they give you.

🚨 **You can also search for tags, via the class SearchForTags, or invoke it from the commandline:**

```
| stag mate # Search for all tags with mate
```

```
| stag mate WITH_URL # ^^ same as above but also output the URL.
```

If at any moment you wish to setup the tags again, do this:

```
| rbt --setup-tags
```

Note that you can also combine searching via tags.

For instance, consider:

```
| stag gnome python
```

This would yield all entries that combine both of these tags.

Let us look at another example.

```
| tags kde ftp
```

This will yield the following results:

```
| 1) kasablanca
```

```
| 2) kftpgrabber
```

**!** As of February 2017, we will exit when an invalid tag is in use. The idea behind this is to clean up all the various cookbook files properly, and specify the tags that are allowed.

Note that you can also display all valid tags, that is, the ones in /System/Tag, by issuing the following command:

```
| rbt --valid-sys-tags?
```

## 🌐 Commandline Options

As was already written somewhere else, you can pass in several commandline options to the Installer.rb script. You can also use commandline options that include -- and these options can overrule other configure options.

For instance, let's assume that the configure options in the yaml file include an option like --enable-gtk. You can then overrule this on the commandline by doing:

```
| rbt weechat --disable-gtk
```

This will overrule the setting in the cookbook.yml file and compile weechat with a disabled gtk.

## 🌐 Package Users

This concept is easy to understand:

Every package belongs to a certain user.

This way you can immediately identify which file belongs whereto.

By default this setting is **turned off** though.

## 🌐 PDF Generation Users

If you want to generate PDF files which you can print or distribute anyway you see fit, have a look at the PdfGenerator.rb script (Note though, right now the output is a bit ugly... I will look into this later.)

## 🌐 Simulate running

This option - also known as **no harm** Option, allows you to run the RBT Scripts in a simulation mode.

This works like so:

```
| rbt php simulate
```

## 🌐 View Available Licenses

To view available licenses, do this:

```
| rbt --view-available-licenses  
| rbt --licenses?
```

You can filter for specific licenses on a given program, such as by issuing this command:

```
| rbt gcc license lgpl
```

This would filter for all **LGPL license** (case-sensitiveness is ignored for finding results).

## 🌐 View Available Programs

To view which programs are available, do this:

```
| rbt x all_programs?
```

! As of JUL 2011, the above no longer works. `compile` is a direct alias to `Compile.rb` now. Try the following instead:

```
| rbt x all_programs?
```

## 🌐 View Configure Options

You can view the configure options that will be used for a program by doing this here:

```
| config? glibc  
| rbt glibc qco  
| rbt glibc query_configure_options
```

## 🌐 Changing Versions

You can always edit the .yml files in question using a simple text editor. But sometimes you may only want to update a specific entry without opening an editor, and you can do this by issuing the following command:

```
| newversion ruby-1.9.2
```

This would find the entry (i.e. at `ruby_cookbook.yml`), and change the version of ruby to be 1.9.2

The advantage is that you dont have to open an editor to change it.

## 🌐 Postinstall

The **Postinstall-Phase** will begin after compilation has finished.

Usually, via **postinstall actions**, you can do something like remove certain directories or files, create new files or new directories, handle symlinks and similar actions. It is quite rarely needed, but at least this functionality exists whenever you need it.

You can also disable this behaviour if you want to, at run-time.

Use this:

```
| rbt htop --no-postinstall
```



## • Autogen

You can specifically enable autogen. Example:

```
| rbt htop --use-autogen
```

## • Other Scripts

The script ScanArchive will scan through your main archive repository and checks if a found tarball there is registered in your cookbook files.

## • Shebang Fixing

You can fix all Shebangs of the RBT Scripts by issuing:

```
| rbt x shebang
```

## • Configuration Settings

You can quickly overrule a specific configuration setting by doing something like this:

```
| rbt rbt.use_ccache
```

The above will try to use ccache. If ccache was not found, the scripts will not try to make use of it.

**! Also note that the configuration will be optional - if you do not have a configuration file, we will assume some default values.**

If you need to find out the configuration directory of your running DiamondShell, do this:

```
| config_dir?
```

This will give you the main directory for all the yaml configuration files.

If you wish to view the whole RBT Configuration, consider issuing the following command:

```
| rbt config?
```

## • CMake

RBT supports cmake files.

Here come some conventions.

Right now, if a CMake\*.txt file was not found, then we will not run cmake. Instead, we will try to run GNU configure instead.

**! Note that you can force the use of cmake from the commandline by passing --cmake, -cmake or cmake to bin/rbt.**

## • Legalese / Disclaimer

Now to some boring ☹️legalese - otherwise evil people may decide to sue me, right?)

So ladies and gentlemen, without further ado ... here we go ...

- The source code is **provided as is**. No explicit guarantees are given of any kind. We do however try to fix all possible bugs, and we are committed to what we wrote. Chances are rather high that it will run on your system, and

even quicker if you provide feedback.)

- Once you DOWNLOAD these scripts, they in fact help YOU to automate tasks which YOU could do from a shell `_alone_`.

It is NOT that WE do it for YOU, you are doing so.

You are in the whole process (and control), it is thus your own risk to use them. This is also the reason why there are so many settings - we want people to change the scripts to their likings as much as possible and as much as it makes sense.

It is entirely in your responsibility to ensure that your police state does allow what you do - we have no control over your actions and we do not encourage you to fight against your police state or do any other activities that should not be done ... remember, obedience to the law means freedom, cuz otherwise the armed and armoured guys come in - and get you to jail. ;-)

However we try to ensure our best that these scripts do not teleport your cats to Texas, nor make polar bears appear at your local friseur in Goteborg or bring Ninjas to your aunt's place... Still, use at your own risk.

This section is more or less needed so noone goes and claims that what these scripts are doing is harmful, or is making any **derivative work** upon others, or that they **endorse illegal activity**.

We dont even care about this crap. Lawyers that fight for their money purse are annoying.

Once you use these scripts THEY ARE YOURS.

**THEY ARE NOT A DERIVATIVE WORK.**

They are scripts which `_YOU_` control and in essence `_YOU_` use. We only have in common that we helped write these scripts.

They are `_NOT_` under GPL. They are not even under a BSD licence. If, you need a license, you may license them under the freest possible ruby license.

Once you have these scripts and use them, they BECOME YOUR PROPERTY and you are FREE TO DO WHAT YOU WANT WITH THEM with **one** exception - you MUST clearly and openly state that you wrote these scripts IF you publish them.

This means you may not claim that we are the author of your scripts if you decide to relicense them to the public in any way. (If you dont release them to others, it doesnt matter anyway. You can do whatever the hell you want to do with them anyway.)

This is the only conditional clause.

The reason is simple. We dont care if you make money with it, we dont actually care WHAT you do with them but if you use them, **leave us totally alone**.

It must be clear to you that the scripts are YOUR scripts at the moment you download them. We do not own the datastream.

In a way you immediately become a full sublicensee.

**! If the laws of your state do not allow this, then YOU MAY NOT DOWNLOAD/USE THESE SCRIPTS!**

If you dont want to do this, or are in doubt of your state law, then do **NOT** download these scripts.

## 🌐 Conventions

This is the reference for important conventions used as part of the ruby build tools project.

- Conventions about naming entries (how to properly name them):

**/Users/x/SRC/PHP/phpwiki-1.3.12.tar.bz2**

`@program_path` = /Users/x/SRC/PHP/phpwiki-1.3.12.tar.bz2

`@archive_type` = .tar.bz2

`@program_full_name` = phpwiki-1.3.12.tar.bz2

`@base_dir` = /Users/x/SRC/+`@base_dir` # specify the base directory of the program in question

`@short_name` = phpwiki

`@program_name` = `@short_name`+`@program_version` Example: phpwiki-1.3.12

`@program_version` = 1.3.12

Since Feb 2012 we follow a new convention when it comes to options like `--no-harm`. All options that include `--` will become an integrated part of the main case/menu data structure.

### ■ Interesting / Useful Links

A list of Things Todo specifically for the Scripts

GoboLinux

RBT Project (remote)

## ■ Contributing to this project

This project is appreciative about contributions of all sorts, especially in regards to these topics:

- Ideas (especially about new/missing features)
- Code contribution of ALL sorts. Please try to adhere to the coding standard, but it is not so important, as I will make your ruby code adhere to the standards anyway, so just send it to me.
- Coverage (so that more people know about it)

The code base is far from perfect, so anything that can be done here to seriously improve it is important, however right now adding more features is more important.

## ■ Pseudo Macros

We use some Pseudo-Macros in the RBT Project.

This section here details them:

Will be replaced with the name of the program, for instance: "htop" Will be replaced with the actual version of the program, example: "1.0.0"

```
PROGRAM_NAME  
PROGRAM_VERSION
```

❗ **Note that you also compile a random program.**

This can be done by issuing this command:

```
| compile RANDOM  
| compile random
```

## ■ Future Outlook

The future is hard to predict, but I do have some ideas on my mind. Not everything seems to be possible in pure Ruby, but who knows - maybe we will include some C.

I will also include other fancy ideas. Here goes:

- Ability to compile a project without being dependent on hardcoded Makefiles, cmake, or whatever.
- Ability to snapshot these compilations at any time, and being able to resume the compile process lateron.
- Ability to generate Makefiles etc... for projects.
- We should realize a generic plugin architecture, so that build types like cmake or scons work easily.

## Show Todo List

### Release History and Version for the RBT Versions

**Version** When

1.0.0 In the future, when we are 1:1 compatible with the GoboLinux Scripts.

## ■ Intramap

- /system/tags
- applying a common operation
- autogen
- binary and library checking
- build inside a subdir
- changing versions
- check locally
- cmake

- [commandline options](#)
- [compile and upload the binaries](#)
- [compiling several programs in one go \(chain-compiling\)](#)
- [configuration settings](#)
- [conventions](#)
- [convert program](#)
- [convert to fhs](#)
- [copy only the headers](#)
- [debug](#)
- [do not show programs names](#)
- [downloading source files](#)
- [dump all recipe files](#)
- [extracting archives](#)
- [feedback about a program](#)
- [find out how many programs are registered](#)
- [find out local location of tarballs](#)
- [generate and use shell completion](#)
- [gobolinux recipes](#)
- [guesscompile](#)
- [handling dependencies](#)
- [included program](#)
- [install only the bin/ subdirectory](#)
- [installed files](#)
- [installing older versions](#)
- [ldconfig](#)
- [legalese / disclaimer](#)
- [m4 macros](#)
- [makefiles](#)
- [md5sum](#)
- [merge cookbook files](#)
- [other scripts](#)
- [package up the rbt scripts](#)
- [package users](#)
- [pdf generation users](#)
- [postinstall](#)
- [programs settings](#)
- [query inclusions](#)
- [querying whether we have a binary file](#)
- [register all programs](#)
- [searching for something](#)
- [shebang fixing](#)
- [simulate running](#)
- [slackware slack-desc files](#)
- [sleep before continuing](#)
- [specify another program to compile](#)
- [standalone yaml files](#)
- [static compilation](#)
- [stow](#)
- [subversion](#)
- [suppress commandline output](#)
- [symlinking all programs](#)
- [the highest n programs](#)
- [the scripts and their \(infra\)structure](#)
- [updateentry](#)
- [use a specific compile version](#)
- [use multiple yaml files](#)
- [use the first url instead of program name:](#)
- [use the first url instead of program name:](#)
- [using another prefix](#)
- [view available licenses](#)
- [view available programs](#)
- [view configure options](#)
- [view urls of certain programs](#)
- [viewing all packages](#)
- [viewing lfs and blfs urls](#)
- [whereis](#)

Append this Program to *cookbook.yaml*

Name of Program to add:

Base Dir:

Description:

URL:

Submit Data

[ReloadThisPage](#)

Listing some RBT files now:

*shebang\_fixer.rb*