GODI User's Manual

Gerd Stolpmann

10th October 2004

Chapter 1

Getting and Installing GODI

GODI is an O'Caml distribution that is compiled and installed from sources. This has a number of advantages:

- The software is up to date, because it is usually very simple for package maintainers to update to a new version.
- GODI supports a wider range of operating systems, not only a single one like other distribution efforts.
- GODI needs only very little infrastructure, especially centralised services. GODI needs only a software repository, and a distribution service for files, but not the really complicated things like compile farms etc.

Of course, the downside is that the GODI users must do more than for distributions where the software is available in binary form. First, the basic "C toolchain" to build C programs must be already installed on the system, and it must work. Second, the development parts of the system libraries must be installed on the system, e.g. C header files. The users must also recognize when additional external libraries are required for certain GODI packages. Third, the users' patience is sometimes stressed, as it takes a bit of time to build software. Fourth, the users should be prepared that sometimes things go wrong. The GODI developers cannot guarantee that the distributed build procedures work on every system. Sometimes they make too optimistic assumptions, for instance it is sometimes expected that the OS has features it actually does not have (e.g. that shell utilities have GNU extensions). Such errors happen from time to time, and the developers are glad when the users inform them about such mistakes. Of course, this would require that the users have some skills recognizing them. Summarised, the GODI users need some basic skills in system and software build management.

Resources

There are a number of on-line resources that are important for users:

- GODI homepage: http://www.ocaml-programming.de/godi/
- GODI mailing list: https://gps.dynxs.de/mailman/listinfo/godi-list
- GODI bug tracking system: https://gps.dynxs.de/tracker

1.1 Preparing your system

It is very likely that you must first install software because you can even start using GODI. This is the generic list for all OS:

- You need `gcc`, the GNU C compiler. Other compilers are not supported (but may work nevertheless). You need also the system header files, sometimes they are not installed by default (e.g. `glibc-dev` on many Linux distributions).
- You need GNU `make`.
- You need `gzip/gzcat`, `bzip2/bzcat`, and GNU `patch`
- It is an advantage to have GNU `nm` and GNU `objcopy` (enables `ocamlc -pack`), but it is not required
- You need the standard Unix shell and file manipulation tools. Sometimes `dc` (often bundled with `bc`), and `m4` are not installed by default.

Of course, these are only the requirements for the minimum GODI installation. Depending on which GODI packages are installed, further software may be needed.

There is a more detailed list in the README file contained in the GODI bootstrap tarball. It depends on the OS which of the tools come with the OS and which must be additionally installed.

1.2 Bootstrapping

The bootstrap procedure installs the minimum GODI system. Currently, the bootstrap procedure has two stages that must be performed one after the other. Stage 1 installs basic tools written in the C language that are required for the GODI package system. This means, after stage 1 the installed software is managed in the form of packages. We will discuss later what a package really is, for now just think a package as a group of files that is added to the system as a whole, and that can also be removed from the system as a whole.

Stage 2 installs the minimum O'Caml environment, and further parts of GODI.

In order to bootstrap, you need the bootstrap tarball, see
http://www.ocaml-programming.de/godi/
for where it can be downloaded. Basically, you execute the following steps:

1. Extract the bootstrap tarball:
   ```
   gzip -d godi-bootstrap-<VERSION>.tar.gz
   tar xf godi-bootstrap-<VERSION>.tar
   cd godi-bootstrap-<VERSION>
   ```
2. Enter stage 1:
   ```
   ./bootstrap --prefix <PREFIX>
   ```
3. Adjust `PATH`, optionally edit `<PREFIX>/etc/godi.conf`:
   ```
   PATH=<PREFIX>/bin:<PREFIX>/sbin:$PATH
   export PATH
   ```
4. Enter stage 2:
   ```
   ./bootstrap_stage2
   ```

For stage 2, you need an Internet connection.

The bootstrap procedure is discussed in more detail in the `README` file contained in the bootstrap tarball, especially what to do when things do not work as expected.

As `<PREFIX>`, you can choose any empty or not yet existing directory. It is not possible that `<PREFIX>` points to an already used directory like `/usr/local` as GODI needs its own private directory hierarchy.

It is not recommended to install GODI as super-user. Either install GODI privately under your own account, or create a special "godi" account for a shared installation.

Options

The bootstrap script (stage 1) has a number of command-line options:

- `--ftp=lukemftp`: Prefers the "lukemftp" utility to download files from the Internet
- `--ftp=tnftp`: Prefers the "tnftp" utility to download files from the Internet
- `--ftp=wget`: Prefers the "wget" utility to download files from the Internet. "wget" is not included in the bootstrap tarball like the other two utilities.
- `--append-path`: When looking up system utilities, first system-specific standard locations are tried, and after these, the directories enumerated in the `PATH` variable. This is the default.
- `--prepend-path`: When looking up system utilities, first the directories enumerated in the `PATH` variable are tried, and then system-specific standard locations.
- `--no-path`: When looking up system utilities, only the system-specific standard locations are tried.
- `--search-path <PATH>`: When looking up system utilities, only the directories enumerated in the `<PATH>` argument are tried.

```
+------------------------------- GODI Console -------------------------------+

      >>> Select Source Packages <<<

      FL NAME               INSTALLED  AVAILABLE  COMMENT
==========Packages available as source code:==================================
[  1]    apps-camlmix       1.1        1.1        Processes macros written in pu
[  2]    apps-cduce         0.2.1      0.2.1      XML-oriented functional langua
[  3]    apps-cduce-cvs                20040829   XML-oriented functional langua
[  4]    apps-godiva        0.9.2      0.9.2      High-level tool for simplifyin
[  5]    apps-headache      1.03       1.03       Tool for managing headers in s
[  6]    apps-ledit         1.11       1.11       Line editor wrapper
[  7]    apps-schoca                   0.2.0      Scheme interpreter written by
[  8]    apps-unison        2.10.2     2.10.2     File synchronizer
[  9]    base-curl          7.11.2#2   7.11.2#2   The version of CURL for GODI
[ 10]    base-expat         1.95.7#3   1.95.7#3   The version of expat for GODI
[ 11]    base-gdbm                     1.8.3#4    The GNU database manager
[ 12]    base-pcre          4.5#1      4.5#1      The version of PCRE for GODI
[ 13]    base-subversion-c$            1.0.6      The subversion client allows d
[ 14]    conf-curl          3          3          Configures which curl library
[ 15]    conf-expat         6          6          Configures which expat library
[ 16]    conf-freetype2     1#1        1#1        Configures which freetype2 lib
-----------------------------------------------------------------(more)----------
[p]rev [n]ext [u]pgrade all [s]tart/continue installation [h]elp e[x]it menu
>
```

Figure 1.1: Selecting source packages with `godi_console`

1.3 Installing packages with `godi_console`

After the bootstrap procedure has been finished successfully, a number of programs are installed in `<PREFIX>/bin` and `<PREFIX>/sbin`. The former directory contains applications that may be executed by everybody whereas the latter directory is reserved for administration programs. One of these is `godi_console` which serves as the central management tool. It has an interactive mode when called without arguments.

After starting `godi_console` type "2" to enter the menu "Select source packages". The list of available and installed packages appears (shown in figure 1.1). You can select (and deselect) packages by entering the number and typing "b" to build the package, "k" to keep the package as it is, or "r" to remove the package (in the corresponding submenu). Finally, press "s" to start the installation:

- GODI checks the package dependencies. Missing packages are implicitly selected for build. Furthermore, if a package is rebuilt, all installed packages are checked whether they are dependent on this package, and also rebuilt. The same "expansion" of the package plan is performed for the packages scheduled for removal.
- When the dependencies had to be corrected, the package list is displayed again, and the additionally affected packages are shown at the top of the list. In this case, type again "s" to restart the resolution of dependencies.
- Finally, the update plan is executed: Missing software is retrieved from the Internet. Old packages are removed. The new packages are extracted, built, and installed. These actions are always performed automatically, without any need to gear into the running process.

This means, godi_console guides you through the build and installation processes, without having to enter commands. We do not discuss godi_console here in detail, as it is equipped with self-explanatory help texts.

A note for advanced users: godi_console also features a command-line mode which is sometimes useful to do mass updates. There is a manual page for godi_console explaining this mode.

1.4 Configuring external libraries

Unfortunately, godi_console cannot build everything in an automatic way. Especially one point requires manual intervention from time to time: External C libraries.

Of course, it is required that these external libraries are installed before GODI can make use of them. For example, if you install the package godi-zlib it is a good idea to check whether the underlying C library libz is already present or not (don't forget to check that the C header files are also installed). Especially for non-free OS like Solaris there is no standard place to install such external libraries. Some admins put them under /usr/local, some under /opt, and a lot of further private locations are in use, too. For Linux and BSD, however, these libraries are often part of the OS, and can be found at a known place in /usr.

The good news is that GODI is very flexible regarding these locations. There are two ways of telling GODI where to find libraries: By changing the global configuration, and by setting package-specific parameters.

The global parameter in question is SEARCH_LIBS: For example, when libz is installed under /opt/phantasy such that the library libz.so is located in /opt/phantasy/lib and the C headers are in /opt/phantasy/include, one can tell GODI this place by setting

```
SEARCH_LIBS += /opt/phantasy
```

in the global configuration file <PREFIX>/etc/godi.conf. This parameter is respected by most packages that need external libraries (but not by all, as there are other, incompatible methods of looking up libraries, see below).

GODI already knows a number of standard locations for libraries, i.e. directories where certain OS install libraries by default. For example, NetBSD usually installs add-on libraries in /usr/pkg which is already part of the built-in knowledge.

You may have already noticed that there are conf-<NAME> packages for a number of external libraries <NAME>. For example, there is a conf-zlib package. The role of these packages is to find and store the configuration where external libraries are expected to be found by GODI. By default, the conf packages iterate over the directories enumerated by SEARCH_LIBS, and look for the needed libararies in these places. Furthermore, a small test program is tried to build, just to see whether the found libraries really work.

Sometimes the library cannot be found, or additional compiler or linker flags must be set. The conf packages allow you to set the individual configuration parameters. Of course, the GODI user must already know which parameter must be set to which value – in other words, this is tweaking for experts. In `godi_console`, one can set the configuration parameters by going to the configuration screens of the conf packages. For example, `conf-zlib` has two such parameters:

`GODI_ZLIB_INCDIR`: The directory where the C header file `zlib.h` can be found

`GODI_ZLIB_LIBDIR`: The directory where the library file `libz.so` can be found

Some conf packages also allow you to set the flags for compiling and linking directly rather than setting directories. In any case, the individual configuration parameters override the search strategy followed by default.

Nowadays, libraries are more and more shipped with special configuration scripts. These scripts simply output the required compiler and linker flags. When available, GODI prefers these scripts, and sees them as a trusted source whose knowledge can be expected to be right. For example, the freetype library has such a script, `freetype-config`, and one can call it by "`freetype-config --cflags`" to get the compiler flags, and by "`freetype-config --libs`" to get the linker flags. The corresponding GODI configuration package is `conf-freetype2`. Instead of searching the library directly, it just looks for where this configuration script is installed. Normally, the locations in `SEARCH_LIBS` are checked (by looking into the `bin` subdirectories), and the directories in `PATH` are checked. If the script cannot be found, it is still possible to set the location directly with a configuration parameter:

`GODI_FREETYPE2_CONFIG`: The absolute path to the `freetype-config` script

When such a script is used, the configuration package does not support to specify the directories of the library directly, or to set the flags.

As pointed out, the output of the configuration scripts is simply trusted. If it happens that the obtained flags do not work, the script is wrong, and it is not possible to use the library from GODI.

Many OS now use the ELF file format for libraries (e.g. Linux, BSD, Solaris). ELF allows several ways of finding libraries at runtime, i.e. when the program using the library is started:

- The `LD_LIBRARY_PATH` variable may list directories where libraries are installed
- There is the `RPATH` (runtime path) entry in the executable using the library
- There is often a global configuration file (e.g. `ld.so.conf` for Linux) listing the default library directories of the OS

For a number of reasons, GODI never uses the `LD_LIBRARY_PATH` feature. This variable is more an ad-hoc solution to get misconfigured libraries working, but not the appropriate means for a permanent and professional environment like GODI.

GODI uses one of the other two options: First, it is checked whether the library can be found by keeping the default settings, and only if this does not work, the `RPATH` feature is enabled. Note that this automatism is not applied when a configuration script is used; in this case it is expected that this script already knows which way is the right one to find the library at runtime.

Important note: Currently, this approach is not put through at all places. This means that there are usually more RPATH settings than needed. This is rarely problematic, but I already had the case that a certain library is available in two versions, and the RPATH setting was wrong. In particular, this library was libGL.so, and one version (in /usr/lib) was the MESA software 3D rendering version, and the other version (in /usr/X11R6/lib) was the hardware 3D rendering version. Because of implementation errors, stub libraries were installed with RPATHs for /usr/lib (which is totally useless), and my programs suddenly loaded the GL library for software rendering. The workaround in such cases is to byte-compile with -custom (or to use the ocamlopt compiler), and to fix the RPATH with -cclib -Wl,-R/preferred/path. This GODI problem will not be fixed soon.

1.5 Using libraries from base packages

Usually, GODI does not include add-on C libraries; it is expected that these are already available by the OS, or that the sysadmin have already installed them. Sometimes, however, GODI needs certain versions of the libraries, and it would be painful to require that the sysadmin updates the OS or other parts of the system only to make GODI happy. In these cases, GODI includes the libraries to install in the "base" series of packages, e.g. `base-pcre` includes the preferred version of the PCRE library.

These packages are not used by default, however. It is first checked whether the library version provided by the OS or the version found somewhere on the system is acceptable. If not, the conf package fails, but prints a hint that enabling the base package would be a simple solution for the problem. By setting a configuration parameter, the GODI user can do this. For example, the `conf-pcre` package can be made using the `base-pcre` package by setting

```
        GODI_BASEPKG_PCRE=yes
```

The rest is again fully automatic.

Chapter 2

Using GODI

In this chapter, I would like to give some hints for O'Caml beginners, and explain where to find what in the GODI environment.

2.1 Starting the O'Caml toploop

The O'Caml bytecode compiler can be called as a so-called toploop: The user can enter declarations and expressions, and these are immediately compiled to bytecode, and immediately executed. The toploop is very handy to for coding attempts, and also a debugging aid for larger programs (because one can load already compiled bytecode into the toploop, too). One can invoke the toploop with the command `ocaml`:

```
$ ocaml
Objective Caml version 3.08.1
#
```

For example, define the faculty function as (note the `;;` at the end of the line, the semicolons indicate the end of the user input):

```
# let rec fac n = if n <= 1 then 1 else n * fac(n-1);;
```

The toploop answers with:

```
val fac :  int -> int = <fun>
```

This means that `fac` is a function taking integers as input, and returning integers as results. Call the function as

```
# fac 10;;
```

and you get the result 3628800. Of course, we cannot give here an introduction into the O'Caml language, so we stop here. It is recommended to install the package `godi-ocaml-manual` which includes both introductory and reference documentation of the O'Caml language. The installed manual can be found in the directory `<PREFIX>/doc/godi-ocaml-manual`.

You may have noticed that the toploop does not include a line editor. To get around this limitation, install the package `apps-ledit`, and call the toploop by

```
$ ledit ocaml
```

This enables a number of keys (cursor keys, delete key, etc.).

2.2 A simple IDE: ocamlbrowser

The O'Caml core distribution includes a simple IDE that allows you to explore libraries, to edit O'Caml sources, and to run the toploop: ocamlbrowser. As this program uses the Tk library for the GUI operations, it is not installed by the GODI bootstrap procedure (so it is not necessary to deal with the complications of finding external libraries already at this early stage). ocamlbrowser is contained in the `godi-ocaml-labltk` package.

The program `ocamlbrowser` is invoked without argument, and pops up a new window with three columns. In the leftmost column, the modules of the standard library are listed. If you click at a module, the middle column shows the definitions of the module. The rightmost column is only used when nested modules occur (e.g. `MoreLabels.Set`).

If you click at a definition (v=value, t=type, cn=exception, m=module, ...) the contents of the definition are shown below the three columns. You can also view the interface and the implementation files, if available, by pressing the buttons "Intf" and "Impl", respectively.

The toploop is invoked with the menu entry "File → Shell...". It works like the ordinary toploop but also does syntax highlighting.

The editor is invoked with the menu entry "File → Editor...". It performs syntax highlighting, and you can even typecheck your definitions ("Compiler → Typecheck"). As a special feature, you can query the types of subexpressions after a typecheck pass: Just put the cursor near the interesting symbol, or mark the expression, and press the right mouse key.

By selecting "Edit → To shell", the current definition (or the marked region) is copied over to the shell window (if open).

Although ocamlbrowser has some interesting features, it is still too limited in order to be useable as a developement environment.

2.3 Using emacs/xemacs

GODI does currently not include packages with the required emacs Lisp definitions.

2.4 The O'Caml compilers and tools

The O'Caml core distribution includes:

- `ocaml`: The toploop (see above)
- `ocamlc`: The bytecode compiler, available for all platforms. For example, to compile the file `sample.ml` to the program `sample`, call it as
  `ocamlc -o sample sample.ml`
- `ocamlopt`: The native code compiler, available for some platforms. The command-line options are almost the same as for `ocamlc`.
- `ocamlc.opt` and `ocamlopt.opt`: These are versions of `ocamlc` and `ocamlopt` that are compiled with the native code compiler, and are much faster than `ocamlc` and `ocamlopt`. The function is exactly the same.
- `ocamlcp` and `ocamlprof`: The bytecode compiler with profiling instrumentation, and the corresponding analysis tool
- `ocamlmktop`: A special version of the bytecode compiler to create toploops with custom functionality
- `ocamldep`: The dependency generator
- `ocamllex(.opt)` and `ocamlyacc`: Lexer and parser generators
- `ocamldebug`: The replay debugger
- `ocamlmklib`: A tool to create stub libraries
- `ocamlrun`: The bytecode interpreter. There is normally no need to call it directly.
- `camlp4`, `camlp4o(.opt)`, `camlp4r(.opt)`, `mkcamlp4`, `ocpp`: The configurable preprocessor
- `ocamldoc`: The documentation generator

These tools are all described in the O'Caml manual. The native-code compiler is not available for all platforms, as well as the `.opt` versions of the tools.

The GODI version of `ocamlmklib` is a wrapper script around the real `ocamlmklib.bin` tool that adds a number of default options that should be present in a GODI environment.

In addition to these official tools, a number of less official tools are also installed:

- `addlabels` and `scrapelabels`: These tools rewrite O'Caml programs from the old, unlabeled style to the new, labeled style, and vice versa.
- `objinfo`: Outputs valuable information about bytecode files (cmo and cma), for example which module versions must be loaded as prerequisites
- `dumpapprox`: Outputs valuable information about native-code files (cmx), for instance whether a function is enabled for inlining.

2.5 Using add-on libraries with findlib/ocamlfind

Unfortunately, the O'Caml core distribution does not include aids to manage libraries. These are handled in a rather low-level way, as the user of the libraries must know the directories where these are installed, and the dependencies between the libraries. This style is similar to the way libraries are handled in the C language.

The findlib library (and the command-line frontend `ocamlfind`) try to bridge the gap between the users' needs and this way of treating libraries. GODI equips all libraries with the necessary meta information findlib needs to process the libraries, and to make them available in a user-friendlier manner.

In the toploop, findlib can be enabled by the directive

```
#use “topfind”;;
```

This installs a number of additional toploop directives:

- `#require “<LIBNAME>”`: Loads the library `<LIBNAME>` into the toploop (unless it is already loaded), including all required prerequisites
- `#list:` Lists the available libraries
- `#camlp4o`: Enables the camlp4 preprocessor with standard syntax. This should be the first directive after loading topfind.
- `#camlp4r`: Enables the camlp4 preprocessor with revised syntax. This should be the first directive after loading topfind.

For example, a single

```
#require “pxp”;;
```

loads the XML parser PXP into the toploop, including all predecessor libraries PXP is dependent on.

Unfortunately, some platforms cannot load libraries into the toploop that depend on external C libraries. For example, Cygwin and NetBSD are such platforms. The workaround is to create a custom toploop that statically links the needed libraries. For example, to create a custom toploop with support for PXP, run the command

```
$ocamlfind ocamlmktop -o mytop -package pxp,findlib -linkpkg
```

which creates a toploop program called `mytop` which can be used instead of `ocaml`. Note that findlib must always be mentioned as package. The toploop `mytop` has already built-in support for findlib, so you need not to “`#use`” findlib at the beginning of every session. The “`#require`” directive for PXP is still necessary, however.

In order to call the standalone compilers ocamlc and ocamlopt, the tool `ocamlfind` should be used. This is a wrapper program around the compilers that adds a number of additional command-line options. For example, to compile the module `sample.ml` that calls functions of PXP, use

```
$ ocamlfind ocamlc -c sample.ml -package pxp
```

When linking executables, the option `-linkpkg` must be passed to indicate that the libraries must be linked, too:

```
$ ocamlfind ocamlc -o sample sample.cmo -package pxp -linkpkg
```

The tool `ocamlfind` can also be used as wrapper for `ocamlcp`, `ocamlopt`, `ocamlmktop`, `ocamldep`, `ocamldoc`, and `ocamlbrowser`. The latter is very convenient to browse the interfaces of add-on libraries, e.g. to view the definitions of PXP, run

```
$ ocamlfind ocamlbrowser -package pxp-engine
```

(Note that we refer to `pxp-engine`, and not `pxp`, as the latter is only an empty pseudo package, and the ocamlbrowser call does not resolve dependencies.)

A special feature of findlib is that it simplifies the usage of camlp4 enormously, the grammar-level preprocessor for O'Caml. In order to enable camlp4, pass the `-syntax` option:

```
$ ocamlfind ocamlc ...  -syntax camlp4o
```

This enables camlp4 with standard syntax. Replace `camlp4r` for `camlp4o` to get the revised syntax. The interesting feature of `ocamlfind` is that one can easily specify camlp4 extensions. For example, to get the xstrp4 extension, just use

```
$ ocamlfind ocamlc ...  -syntax camlp4o -package xstrp4
```

i.e. add such extensions simply to the list of included packages.

The findlib library is available as GODI package `godi-findlib`. It is installed as part of the bootstrap procedure. There is, however, a small but useful option that is not enabled by default, and requires a recompilation of findlib: The Makefile wizard. This is a GUI to create findlib-aware Makefiles with a few clicks.

To get it: Start `godi_console`, select the `godi-findlib` package, and enter the configuration menu. Set the parameter

```
GODI_FINDLIB_TOOLBOX = yes
```

and rebuild findlib. You will also need `godi-ocaml-labltk` (and thus tcl/tk) in order to build this special version of findlib (which is the reason why this option is disabled by default). The result is that the "Makefile wizard" is included in the findlib package. Call the wizard with

```
$ ocamlfind findlib/make_wizard
```

There would be a lot more to say about findlib. As part of the package, the findlib manual is also installed, so please look there for more information.

A final note on the syntax "-I +pkgname" the O'Caml compilers implement themselves. It was introduced as simple mechanism to locate add-on libraries. For GODI, this kind of referring to libraries should be regarded as deprecated legacy mechanism. The point is that this syntax is much less flexible than findlib, and that it is also dictates the directory where the library must be installed (which is not acceptable).

2.6 Finding package documentation

The documentation for package `P` can be found in `<PREFIX>/doc/P`. The preferred format is HTML.

There is also a small CGI that can display library interfaces: `godi-findlib-browser`. It can be found in `<PREFIX>/doc/cgi-bin/browser.cgi` after installation. One can easily view all interfaces (but w/o formatting), and there is also a full-text search option. In order to activate this CGI, you need a properly configured web server. For Apache, the widely used web server, it is usually sufficient to include the directives

```
ScriptAlias /godi-bin <PREFIX>/doc/cgi-bin
<directory <PREFIX>/doc/cgi-bin>
AllowOverride None
Options ExecCGI
Order allow,deny
Allow from all
</directory>
```

into the configuration file `httpd.conf`, and to restart the web server. The CGI becomes visible under the URL `http://servername/godi-bin/browser.cgi`. There are also other ways of configuring Apache for this purpose.

2.7 Key packages

Some packages play a special role in the package system:

- The O'Caml core distribution is the union of the packages:
  `godi-ocaml`: Compilers and runtime environment
  `godi-ocaml-src`: Sources of the core distribution
  `godi-ocaml-dbm`: NDBM access (module `Dbm`)
  `godi-ocaml-graphics`: Simple graphics (module `Graphics`)
  `godi-ocaml-labltk`: GUIs with Tk

  This means that the O'Caml source tarball, as it can be obtained from the INRIA FTP server, is split up into five packages. The reason is that external libraries are needed

for some parts of the O'Caml core, and this can be easier handled when packaged separately. In addition to this, there is also the O'Caml manual:
`godi-ocaml-manual`

You can select all of these packages by
`godi-ocaml-all`
which exists for convenience only (meta package).

The package `godi-ocaml-src` is very special, because it contains the already configured but not yet compiled source tree of O'Caml. It is the logical predecessor of the other O'Caml core packages, which all extract and build parts of this tree. The idea of this package is also to support patching the O'Caml compiler. There is one disadvantage of this construction: In order to force a rebuild of the whole O'Caml system when newer sources are available, one must select *both* `godi-ocaml` and `godi-ocaml-src`. If only the former were rebuild, the old, wrong sources would be taken, and if only the latter were rebuild, only the sources would be updated without compiling them.

- The software GODI consists of is itself packaged:
  `godi-core-digest`: Helper program to compute file digests
  `godi-core-ftp`: Helper program to download files from the Internet
  `godi-core-make`: The BSD "make" utility
  `godi-core-pax`: Archive tools (tar, pax) with special features for the package system
  `godi-core-pkgtools`: The utilitites to add and delete binary packages
  `godi-core-mk`: The build framework (a set of "make" rules) to compile and install packages from sources
  `godi-tools`: The higher-level tools, currently only `godi_console`

  Note that some of these packages have a special status, as they are part of the GODI system, and not every operation is permitted. For example, it is not allowed to delete `godi-core-pkgtools`, because there would not be any way to recover from the resulting situation. Furthermore, you will notice that `godi_console` lists some of them under the section "Installed packages not available as source code". This only means that there are no newer sources to update them. Actually, the bootstrap tarball contains the source code these packages were intially built from.

- GODIVA is an associated project to simplify the creation of GODI packages. In order to create GODI packages with GODIVA, you need `apps-godiva`. It is not needed to process the generated packages, though.

- As already pointed out, findlib is supported by all libraries in GODI, and thus `godi-findlib` is also a key package.

Chapter 3

The Architecture of GODI

In this chapter, we look at the various parts of GODI, and how they are related. It should become clearer how GODI works, and what one can expect from it.

3.1 GODI servers and clients

In principle, GODI is a kind of client/server system. This aspect is usually overlooked, although it is one of the most important properties. The GODI system, as it was installed in chapter 1, consists only of the client part that controls the locally installed packages. In addition to this, there is also a GODI server that provides the necessary information which packages exist and what the packages contain. The GODI client (e.g. `godi_console`) may contact the server to update the list of packages. In `godi_console` you can trigger this by selecting the menu item "Update the list of available packages".

The GODI server is mainly a Subversion repository where the files are stored that make up the various packages. This repository is maintained by the GODI developers in a collaborative effort. You can view this repository under the URL:

https://gps.dynxs.de/svn

The `godi-build` directory contains the packages, whereas `godi-bootstrap` contains the base software including the bootstrap script (from which it derives its name). (The other directories contain software not related to GODI, although some of the libraries are available as GODI packages.)

More precisely, the `godi-build` directory on the GODI server only provides the *build instructions*, i.e. the set of rules that control the build and installation procedures. The GODI server does not store the *distribution files*, i.e. the tarballs made and distributed by the authors of the software packages. These are downloaded from the primary http or ftp servers. (To be even more exact, the GODI server keeps copies of the distribution files, but these are only used when the primary servers are not reachable.) For example, the package `godi-ocamlnet` in version 0.98 has the following build instructions and distribution files:

- http://www.ocaml-programming.de/godi-build/3.08/godi-ocamlnet-0.98.build.tgz is the URL where the build instructions can be obtained. These are created by the GODI developers. This tarball contains the files:

`DESCR`: Just a text file with a description of the package

`distinfo`: Contains checksum for distribution files

`Makefile`: The rules to build and install the package

`PLIST.godi`: The package list, i.e. it is described which files are part of the package

- http://aleron.dl.sourceforge.net/sourceforge/ocamlnet/ocamlnet-0.98.tar.gz is the URL where the single distribution file can be obtained. This is the file distributed by the author of the software.

The build instructions also contain dependency information. For this reason, it is necessary that the build instructions of all packages must be available, and because of this, `godi_console` updates them in a single step. The distribution files, on the contrary, are only downloaded when the corresponding package is built.

3.2 Local directory layout

The GODI directory hierarchy follows Unix conventions with some additions:

- `<PREFIX>/bin`: Binaries
- `<PREFIX>/sbin`: Binaries for GODI administration
- `<PREFIX>/etc`: Configuration files. Especially, you find here `godi.conf`, the global configuration file for GODI, and `ld.conf`, the global configuration file for dynamic stub libraries
- `<PREFIX>/man`: Manual pages
- `<PREFIX>/share`: Platform-independent files (in subdirectories)
- `<PREFIX>/lib`: C libraries and, in subdirectories, platform-dependent files
- `<PREFIX>/lib/godi`: Generated configuration files
- `<PREFIX>/lib/ocaml/std-lib`: O'Caml standard library
- `<PREFIX>/lib/ocaml/compiler-lib`: Additional O'Caml interface files
- `<PREFIX>/lib/ocaml/pkg-lib`: Add-on O'Caml libraries
- `<PREFIX>/lib/ocaml/site-lib`: User-installed (non-packaged) add-on O'Caml libraries
- `<PREFIX>/build`: GODI build system
- `<PREFIX>/build/buildfiles`: Contains the build.tgz files with the build instructions (for archive purposes only)
- `<PREFIX>/build/distfiles`: Contains distribution files

- `<PREFIX>/build/packages`: Contains binaray packages (in `All`)
- `<PREFIX>/build/mk`: Global GODI build rules
- `<PREFIX>/build/<CATEGORY>/<PACKAGE>`: These directories contain the unpacked build.tgz files

3.3 Packages

We already said that a package is a group of files that is installed as a whole. This is a simplified definition, and when one looks in detail at the package concept, it becomes clear that the same package may occur in three ways:

- As "source package": The build instructions plus the distribution files may be viewed as source packages. It is important to remember, however, that there is no single file bundle containing the files to build the software (like SRPMs). The concept is rather that the package metainformation are obtained from the GODI server in the form of the mentioned build instructions, and that the raw sources of the software are directly retrieved from the original file servers where the software authors distribute them (the distribution files).
- As installed package: These are the installed files. GODI remembers which files belong to which packages, and stores these data in the *package database* which can be usually found at `<PREFIX>/db`.
- As binary package: This is an archive file containing the files to install (together with a subset of the metainformation). These archive files can be found under `<PREFIX>/build/packages` and they are automatically created when a package is built from sources.

The metainformation includes:

- The package name: The name is derived from the original name under which the author distributes the software. GODI prefixes this name with a category indicator ("base", "conf", "apps", "godi"):

base:	This is software outside of GODI's scope, but required for GODI. Often, the base software is part of the OS. In the case it is not available, or only in the wrong version, the "base" packages may be used as replacement.
conf:	The configuration packages represent the knowledge about software outside of GODI that is used by GODI. For example, the configuration packages for external libraries remember where these libraries are located, and how these must be linked to programs created with GODI. The source package usually only consists of build instructions, but not of distribution files. The instructions often includes a script that systematically guesses facts about the external software to configure, and checks these assumptions by tests (autoconfiguration). See also section 1.4 for an explanation from

the user's point of view. The result of the script is usually stored in a file, and this file is the only content of the installed/binary package. For example, the package `conf-zlib` remembers the configuration parameters in `<PREFIX>/lib/godi/conf-zlib.mk`. See the section 6.1 for a detailed discussion.

apps:	These are applications (end-user software).
godi:	This is software to build applications, i.e. libraries, meta-programming (generators, compilers, ...), and development tools. Sometimes, software falling into the "godi" category has also parts that could be seen as applications. In such cases, "godi" is preferred over "apps". Libraries sometimes also have a runtime part, and this means they are needed to run the applications. In the O'Caml world, this does not happen very often, because most libraries are statically linked. Nevertheless, it might be necessary to create another category for run-time files, but this makes only sense when the run-time part of libraries is separated from the build-time part.

- The version of the package: The version string has two parts. The first part is the "dotted" sequence of decimal numbers one usually associates with a version string. It is the version the author of the software announces (but see below). The second part is the package revision number. Sometimes the first attempt of the package has errors, and to distinguish improved versions of the package from the previous ones, the revision number is incremented. The revision number is a natural number. In the version string, the revision number is separated from the primary version by the keyword "godi". For example, in "1.2godi2", the primary version, as announced by the author, is "1.2", and the GODI revision number is 2. When the "godi" suffix is missing, the revision number is 0 by definition. Sometimes, the character "#" is used as separator instead of "godi", but this is only an abbreviation and does not have any meaning.

  Usually, the primary version number consists of natural numbers separated by dots. The syntax of the version numbers allows a few further elements. GODI restricts the syntax because it must always be possible to compare two version numbers, and to decide which one must be sorted before the other (linear ordering). Of course, this can only be ensured when it is known how the version numbers are constructed.

  In addition to the dotted decimals, one can also include letters. These are compared lexicographically, e.g. "1.1ab" < "1.1ac". One can also use the characters "+" and "_" as separators, but they have lower weight than the dot, e.g. "foo3.07+1" < "foo3.07.1". There are a few further keywords: "test", "alpha", "beta", "pre", "rc", "pl" which are also recognised as separators. The first five of these have the special property that they *decrease* the weight, for example "1.1test1" < "1.1". The separator "test" decreases the most, the separator "rc" (release candidate) decreases the least. For instance, "1.1test1" < "1.1beta1". The separator "pl" (patch level) has again positive weight, but less than all other separators. Other characters than the mentioned ones are not allowed in version strings.

  Of course, the authors of software do not always use versioning schemes that are compatible with the one GODI applies. In this case, the packager should try to port the original version numbers as closely as possible to the GODI scheme. It is essential, however, that the order of the numbers is correctly represented, otherwise it might happen that a newer version of the software is available, but GODI does not recognise that. An example illustrates that: Some authors make both development snapshots and regular releases available. For the former, date strings are used, e.g. foo-20040921. For the latter, classic dotted numbers are used, e.g. foo-3.2. GODI can process both formats, and the sorting order for each format is properly represented. It is not possible, however, to mix both formats. Because "20040921" is just a big number, it is higher than "3.2", even if it was released before "3.2". The moral of the story: Use either date strings, or dotted numbers, but do not alternate between them.

A final clarification about the revision numbers: They distinguish between several editions of the source packages. They do not distinguish between different versions of binary packages that are made from the same sources and the same build instructions, but with a different equipment of predecessor packages. For example, if foo-4.4 is once built with bar-1 and once built with bar-2 as predecessors, the *same* version string (and thus the same file name) will be used for both binary packages that result from the build. (Maybe we will have a mechanism to handle this some day.)

- Source: The package has fields that describe where the distribution files can be downloaded, and where more information can be obtained.

- Description: The package has a short, one-line description ("comment"), and a longer description that may even consist of several paragraphs.

- Dependencies: The package may require that other packages are already installed. This is called a package dependency. There are two kinds of dependencies (for the moment, further types are discussed in godi-list): Build dependencies demand that the predecessor packages must be installed at build time, and strict (runtime) dependencies express the requirement that the predecessor packages must be installed both at build and at run time.

  Furthermore, dependencies are handled differently for source packages, and for installed/binary packages. One difference is clear: For the latter type of package, there are no build dependencies, because they are already built. The other differences have to do with the handling of the transitivity of the dependency relation, and the meaning of version conditions.

  For source packages, it is not necessary to state indirect predecessors. For example, if foo requires bar, and bar requires baz, GODI concludes that foo also requires baz indirectly. GODI finds this out automatically. In contrast to this, installed and binary packages must list all predecessors explicitly, even indirect ones, so bar has to demand baz. Fortunately, GODI users never have to resolve such dependencies, as this is done internally by GODI, so this detail users rarely see. (The transitive closure is taken for several reasons. During compilation of software it may happen that indirect predecessors influence the current build. The cross-module inlining feature of O'Caml is an example for this. Of course, such effects must be represented by the dependency relation. Furthermore, the closure eases the distribution of binary packages.)

  Dependencies may carry a version condition, for example foo may require bar in version $\geq$ 3.2. For source packages, these conditions are just handled as constraints. For installed and binary packages, however, these conditions are transformed into exact version requirements. For example, if the user happens to have version 3.3 of bar installed, this is acceptable when foo is built, because $3.3 \geq 3.2$. The resulting binary package lists the dependency bar == 3.3, i.e. the actually found version is taken as fixed version. The reason for this is that O'Caml libraries (and most dependencies are about libraries) are very sensitive to changes, and it is unlikely that any other version works than the one found at build time.

- Maintainer: The person who is responsible for maintaining the package as part of GODI.

3.4 Libraries

As already pointed out, the O'Caml libraries always support findlib in the GODI system. This is not a very hard requirement, and it is usually simple to even add such support to libraries where the author does not do this. Findlib bases only on a few concepts, and complicated situations cannot arise.

The key ideas are that libraries are stored in known directories (directory convention), and that there is a file with metainformation about libraries (called "META"). The directory convention is as follows (here shown in the way GODI realises it):

- `<PREFIX>/lib/ocaml/pkg-lib/<NAME>`: This directory contains all code files for the library `<NAME>`, except DLLs. By "code files" we mean compiled interfaces (suffix cmi), compiled modules (suffixes cmo, cmx, o), and library archives (suffixes cma, cmxa, a). It is also a good idea to put the source interfaces here (suffix mli), for better documentation.

  Files of other kind can go elsewhere, e.g. into `<PREFIX>/lib/<NAME>`, this is out of the scope of findlib.

  The code directory must not have subdirectories.

- `<PREFIX>/lib/ocaml/pkg-lib/stublibs:` This directory contains the DLLs for all findlib libraries that are installed below `pkg-lib`. The DLLs are simple to recognise, because their name begins with the prefix "dll", and has an OS-dependent suffix (e.g. ".so" for Linux, ".dylib" for MacOS, etc.). Findlib also puts for every DLL a second file into this directory, with the suffix ".owner", e.g. "dllfoo.so.owner" for the DLL "dllfoo.so". This file indicates to which library the DLL belongs (the name is stored in the file).

  This `stublibs` directory is already configured in `ld.conf`, the DLL configuration file of O'Caml, so one need not to care about this detail.

  If the OS does not support DLLs, the directory remains empty.

- `<PREFIX>/lib/ocaml/site-lib/<NAME>`: This is another directory for the library `<NAME>`. In the site-lib hierarchy user additions are stored, whereas the pkg-lib hierarchy is reserved for libraries installed by GODI packages.

By a trick it is achieved that the command "ocamlfind install ..." automatically installs the library into `site-lib` when it is executed outside of a GODI build, but into `pkg-lib` when executed within a GODI build. So the libraries automatically end in `site-lib` when GODI users build and install libraries manually.

- `<PREFIX>/lib/ocaml/site-lib/stublibs`: The DLL directory for libraries in the `site-lib` hierarchy.

Of course, it is possible that the same library is installed under both `pkg-lib` and `site-lib`. In this case, `site-lib` has precedence. Anyway, it is a bad idea to do so, because this may break GODI's build system. In general, it is ok to have libraries in `site-lib` that depend on libraries in `pkg-lib`, but not vice versa.

In addition to the directory convention, findlib manages libraries also by storing metainformation about the libraries. These are put into files with the name `META`, and the `META` files are contained in the code directories of the libraries. `META` has usually only a few lines, e.g.

```
description = "The library foo"
version = "1.0"
archive(byte) = "foo.cma"
archive(native) = "foo.cmxa"
```

and is seldom more complicated.

Findlib can also express dependencies (library X depends on library Y = library X uses features of Y). This mechanism is different from the GODI package dependencies, and there is no strict need that the dependencies of findlib and GODI correspond to each other, although this is usually the case for obvious reasons.

O'Caml libraries linked with external C libraries are a special case. In principal, there is an O'Caml part, and a C part. As shown in section 1.4, it is required that the GODI user can configure where the C library is located. The configuration data are contained in the configuration package `conf-foo` for the external C library `foo`. The whole story is as follows:

- The O'Caml library (e.g. `foo.cma`) is linked with a small stub library (e.g. `libfoo_stubs`), which is also written in C, and whose purpose is to translate the O'Caml conventions for data and function representation into the C conventions, and vice versa.
- The stub library is linked with the external C library.

The O'Caml library and the stub library are part of the GODI package `godi-foo`. The location of the external C library is specified by `conf-foo`. It is now possible that the C library is located outside of GODI, or that the C library is available as GODI package, too. In the latter case, the package is called `base-foo`, and it is required that the C library is installed in the directory `<PREFIX>/lib`.

Note that although the information where the external C library resides is specified in `conf-foo`, these locations are also entered into the O'Caml library as part of the build process, so that they are also available in `godi-foo`. At runtime, `conf-foo` is no longer needed. Furthermore, if the user wants to change the configuration, it is not only required to build `conf-foo` again, but also `godi-foo`, and GODI does not remind you of that.

Some platforms allow that C libraries are dynamically loaded into the running bytecode interpreter, but in general this cannot be assumed. Of course, the C libraries must be available as DLLs (or DSOs in Unix terms) in order to be dynamically loadable. When the platform does not support this technique, however, there is no advantage to have C libraries in DLL form, as O'Caml links statically anyway.

Linking with C libraries is really complicated, and there are a number of details that must be handled differently for the various platforms. It is currently not clear which facts about linking are important for GODI users, and which are really technical details only experts need to know.

Chapter 4

Managing a GODI Installation

4.1 What can be done with `godi_console`

The following tasks can be easily carried out with the help of godi_console:

- Updating the source packages: `godi_console` retrieves a new package list from the GODI server, and gets the updated build instructions for all packages with a newer version string.

  Instructions:

  1. Select: Main menu → Update the list of available packages
  2. Wait until the system responds with a success message
  3. Press "x" to exit from the dialogue

- Building additional packages from source: By selecting a package for build, `godi_console` downloads the distribution files, compiles and installs the package. Furthermore, a binary package is created.

  If necessary, `godi_console` also builds predecessor packages automatically, for both build and strict dependencies.

  Instructions:

  1. Select: Main menu → Select source packages
  2. Scroll up/down by pressing "p" and "n" (or use PageUp/PageDown keys). Finally enter the number of the package to build, and press Enter
  3. The detailed description of the package appears. Press "b" to select it for build. Press "x" to exit from the dialogue
  4. It may happen that now the configuration dialogue appears. In this case, you can set configuration parameters by entering their number, and changing their value. If done, press again "x"
  5. The package list is again shown. Press "s" to start the installation process.

6. It may happen that further (predecessor) packages are also selected for build, and the package list is only updated to reflect this. Press again "s" in this case.
7. GODI now asks whether it is ok to start the installation. Press "o" to confirm this, or "x" to cancel.
8. The installation process begins and runs fully automatic. (Actually, there is one exception from the latter: When godi_console updates itself, the user must confirm this step, because it is very critical.)
9. When the installation process prints a success message, it is done. Press "x" to exit.

- Updating packages by rebuilding them from source: By selecting an already installed package for build, `godi_console` performs all necessary steps to upgrade the package to the new version. First, the old version of the package is removed, and then, the new distribution files are downloaded, built, and installed.

  The dependencies are checked in two directions: Missing predecessor packages are installed. This can happen when the new version requires additional prerequisites. Furthermore, the successor packages are also handled in a special way, because the strict successors are updated, too, or at least rebuilt if no newer version is available.

  Instructions: Updating packages works like building packages for the first time.

- Removing installed packages: By selecting an installed package for removal, `godi_console` deletes the package and all strict successors from the system.

  Instructions:

  1. Select: Main menu → Select source packages
  2. Scroll up/down by pressing "p" and "n" (or use PageUp/PageDown keys). Finally enter the number of the package to remove, and press Enter
  3. The detailed description of the package appears. Press "r" to select it for removal. Press "x" to exit from the dialogue
  4. The package list is again shown. Press "s" to start the installation process.
  5. It may happen that further (successor) packages are also selected for removal, and the package list is only updated to reflect this. Press again "s" in this case.
  6. GODI now asks whether it is ok to start the installation. Press "o" to confirm this, or "x" to cancel.
  7. The installation process begins and runs fully automatic.
  8. When the installation process prints a success message, it is done. Press "x" to exit.

It is recommended to check whether enough disk space is available before installing packages. One can get into problematic situations when the disk becomes full in the wrong moment, and it is difficult to recover from this. Furthermore, it is a bad idea to stop `godi_console` in the wrong moment (CTRL-C) because of the same reasons. The critical step begins when `godi_console` prints that it is installing a package ("===> Installing for package"), and ends after the installation has been registered ("===> Registering installation for package").

4.2 Installed packages

The command godi_info may be used to get detailed information about installed packages:

- `godi_info <NAME>`: Prints the comment, the description, strict dependencies (both predecessors and successors are output), and the homepage of the software.
- `godi_info -L <NAME>`: Prints the files the package consists of.

Note that `godi_info -F` does not work (looking up the package by file name), because the needed reverse index is not available.

The command `godi_delete` may be used to remove a package from a system (you can also do this with `godi_console`). Give the `-r` option to delete the package and all successor packages.

4.3 Binary packages

As mentioned earlier in this manual, the binary package file is put into the directory `<PREFIX>/build/pac` after the build of a package has succeeded. This means that this directory always contains a complete copy of the current installation plus a lot of history information.

When a package is updated to a new version, the binary package file of the old version is not deleted. In principle, this allows you to go back, and to restore the old package. However, the history function is limited by the fact that often package files are overwritten when a package is rebuilt for a different equipment of predecessor packages, but for the same version of the software. For example, consider that there is a package foo in version 1, and a package bar in version 1. Furthermore, bar is dependent on foo. When a new version of foo is released, e.g. version 2, and the GODI user updates the package, *both* packages are built again, foo in version 2, and bar in version 1. The result is that a new binary package file is created for foo, namely `foo-2.tgz`, but that the old package file for bar is overwritten, because it is still called `bar-1.tgz`.

The main purpose of the binary packages is to simplify the distribution of software in LANs. One can copy the packages from one system to another, and install them, which is a lot simpler than to build the software on every system from source.

Tasks related to binary packages:

- Getting information: The `godi_info` command can also deal with package files. Just pass the name of the file as argument, and the output of the command refers to the file instead of the installed package. Example:

  ```
  godi_info <PREFIX>/build/packages/All/godi-ocaml-3.08.1.tgz
  ```

- Installing an additional package: This is performed by `godi_add`. By default, it is not allowed to overwrite the package if it is already installed. Example:

  ```
  godi_add <PREFIX>/build/packages/All/godi-ocaml-3.08.1.tgz
  ```

You can also mention several package names on the command line to install several packages at once. When predecessor packages are missing, they are searched in the same directory the original package is taken from. When the predecessor packages exist, but in the wrong version, the installation fails.

- Replacing a package: This can be done with `godi_add`, too, when the `-u` option is given. In principle, it is not necessary to delete the successor packages first, as the package can be in-line replaced by a different version if the version conditions of the dependent packages permit it. However, this is normally not the case.

  Predecessor packages are not updated, however, even if this first enabled the installation of the original package. `godi_add` is simply not intelligent enough for this operation.

  The package replacement even works if an older version is to be installed (downgrade).

- Cleaning the package directory: Of course, you can delete the files in `<PREFIX>/build/packages/A` when you are sure you do not need them anymore. Note that there also symlinks in the neighbour directories that should be deleted, too.

4.4 Restoring old packages

In principle, one can restore binary packages (i.e. install the old binary package file again), and one can recover source packages (i.e. go back to an old version of the source package, and build it again).

4.4.1 Restoring an old binary package

In principle, you can restore an old binary package by calling `godi_add -u` for it:

```
cd <PREFIX>/build/packages/All
godi_add -u <NAME>.tgz
```

Sometimes this does not work, however, because the old package is not compatible with the current set of predecessor or successor packages (`godi_add` reports about a conflict). You can try to replace the problematic packages by historic versions, too, but this is a very complex task, and it is even unclear whether it is possible at all. As explained in section 4.3, package files may be overwritten in certain circumstances, and this means that the required old files might be lost.

In general, it is better to build the old version of the package from source, see the next section for instructions.

4.4.2 Building an old version of a package

This way of restoration is usually possible. In `<PREFIX>/build/buildfiles`, GODI stores the build instructions of all package versions that were ever downloaded. In `<PREFIX>/build/dist`
GODI stores the distribution files of all package versions that were ever built.

Follow these steps to install the package corresponding to the build instructions in `<CAT>-<NAME>-<VERSI`

1. Change the directory:

   `cd <PREFIX>/build/<CAT>`

2. Delete the current build instructions (if present):

   `rm -rf <NAME>`

3. Extract the build.tgz archive:

   `godi_tar xzf ../buildfiles/<CAT>-<NAME>-<VERSION>.build.tgz`

4. Start `godi_console` and build the package again (see above). It might happen that the distribution files do not exist locally in the `distfiles` directory, because the package was never built before. In this case, `godi_console` tries to download them from the Internet. There is no guarantee, however, that historic distribution files remain available forever.

   Of course, godi_console may want to rebuild other packages, too. This is just the same mechanism as building new packages. It may happen, however, that the already installed packages are too modern for the installed package. This is not detected by GODI in advance because the information is not available. The most likely symptom is that the build fails with a compiler error.

In order to keep this possibility, it is strongly discouraged to delete the files in the `buildfiles` and `distfiles` directories.

4.5 Distribution upgrades

There are usually several "release lines" of GODI, i.e. several completely independent series of packages. When you perform the bootstrap procedure, one of the release lines is automatically selected and installed (there is always one release line reflecting the most up-to-date state). Up to now, the following types of release lines are in use:

- For every major version of the O'Caml compiler a new release line is started. For example, there are releases for the O'Caml 3.07 and the O'Caml 3.08 systems. Minor updates (bug fixes) of the compiler are usually handled within the existing release line, e.g. the 3.07 compiler was updated to 3.07pl2 after a while in the existing 3.07 line.

  The reason for this is that the version of the O'Caml compiler is a major determinant of all software covered by GODI. Often, software must be explicitly ported to a new O'Caml compiler release, because the new version introduces almost always incompatibilities with the previous version.

- For certain development purposes, special release lines are started. These are often experimental, and include features that are not yet ready for the stable GODI system.

Currently, the existing release lines of the first type are “3.07” and “3.08”. The release lines of the second type are announced in godi-list.

The release line GODI uses is determined by the variable `GODI_SECTION` that can be set in the file `<PREFIX>/etc/godi.conf`. After changing this variable, please note that GODI does not really detect that a different distribution is selected, it just retrieves the package from the newly selected release. This means:

- If the new release has packages with newer versions, these are recognised as being new, and the build instructions are downloaded when the package list is updated.
- If the new release has packages with older versions, these are recognised as being older than the available ones, and the build instructions of the new release are ignored.

The consequence is that the change of the release line works only correctly when this implies an upgrade, i.e. the packages of the new release are newer. If you really want to change to an older release, you must first delete all extracted build instructions:

```
cd <PREFIX>/build
rm -rf apps/* base/* conf/* godi/*
```

After that, at least the build instructions of the older release are downloaded, and you can start to build packages. You might still encounter strange effects, though, because package downgrades are a somewhat hairy operation, as necessary information about compatibility between old and new packages are missing.

Chapter 5

Packaging Software

This chapter explains the core method of packaging software. GODIVA is another method with a simpler package specification file. GODIVA processes this file, and generates build instructions according to the core method, so GODIVA should be regarded as a higher layer that works on top of the core layer.

The question is whether to use GODIVA, or to manually apply the core method. There is no simple answer, but the following criterions may help:

- GODIVA requires that the software to be packaged follows certain rules, e.g. that there are certain "make" targets, that $PREFIX is honoured, etc. Normally, the build scripts coming with a software distribution do not match exactly with the requirements (although they come quite close). If you package your own software, this is not a big problem, just change the build scripts such that the GODIVA requirements are fulfilled. If you package third-party software, it may become necessary to apply patches such that GODIVA and the build scripts play nicely together.
- GODIVA does not support configuration options.
- GODIVA does not support the inclusion of library flags from conf-* packages. This makes it a bad choice for software that provides bindings for C libraries.

Some of the limitations can be worked around by patching the result of the generated build instructions, however.

If you think you may want to try GODIVA, just install the apps-godiva package, and read more on the homepage: http://projects.phauna.org/godiva/. The following information may be still of interest, however.

5.1 The build directory

The build instructions for a package are stored in the build directory:

```
<PREFIX>/build/<CAT>/<CAT>-<NAME>
```

You find here the following files and directories:

- `Makefile` (mandatory): This `Makefile` controls the overall build process. It does not directly invoke compilers, but it calls the `Makefile` of the software to build as sub process. This `Makefile` has a certain structure, see below.
- `distinfo` (mandatory): This file contains checksums of the distribution files, and of patches.
- `DESCR` (mandatory): The long description of the package. Free ASCII text.
- `BUILDMSG` (optional): This message describes configuration options. Free ASCII text.
- `MESSAGE` (optional): A message to be displayed when the package is finally installed (e.g. from a binary archive). Free ASCII text.
- `PLIST` or `PLIST.godi` (mandatory): The file describes which files are installed by the package, and are owned by the package. The name "PLIST.godi" should no longer be used. `PLIST` can include a number of directives, and is explained below in detail.
- `CONFOPTS` (optional): The list of configuration options (just one option per line).
- `patches` (optional): This directory contains patches to be applied to the unpacked software before the build starts. Patches are automatically applied by the build framework. Note, however, that patches are only applied when they also occur in distinfo, otherwise an error is indicated.
- `files` (optional): Additional files to be added to the software. There is no automatism, the commands performing the addition must be programmed in `Makefile`.

Usually, one needs only `Makefile`, `DESCR`, and `PLIST` (`distinfo` is generated, see below).

There may be the directory `work` as well. It contains the unpacked software, and this is the place where the build process really happens. By deleting `work`, one can reset the build process to the beginning. Note that it is possible to configure a different place for `work` (e.g. somewhere in /tmp).

5.2 Stages

The build process is structured into several stages. A certain stage can only be reached from the immediately preceding stage (but there is some shortcut logic, see below). The stages are:

- fetch: This stage ensures that the distribution files are in <PREFIX>/build/distfiles. If not, the files are downloaded.
- extract: Unpacks the distribution files into `work`.
- patch: Applies the patches to `work`

- configure: Configures the software (usually by calling a configure script)
- build: Compiles the software
- install: Installs the software
- package: Creates the binary package from the installed image

Normally, the build process just "climbs" the stages one after the other. The build process remembers the already reached stages by placing invisible files into "work", e.g. "work/.install_done" indicates that build has completed the install stage.

When developing a package, it is possible to go to a certain stage, and to check whether everything has been done right (in godi_console, the stages are simply iterated, and does not have the chance to stop and to check what is going on). This is done by calling `godi_make`, and passing the name of the stage, e.g.

```
godi_make configure
```

continues the build process until at least the configure stage is reached.

There are defined actions that are carried out for every stage. These actions can be configured by setting variables in Makefile (explained below). In addition to this, one can define pre and post actions for every stage by adding rules to Makefile, e.g.

```
pre-configure:
        <commands>
```

causes that these commands are executed before the predefined actions of the configure stage. In the same way, a `post-configure` rule would be executed after the predefined actions of the configure stage.

5.3 The Makefile

The minimum Makefile looks as follows:

```
.include "../../mk/bsd.prefs.mk"
VERSION= ...
PKGNAME= ...-${VERSION}
PKGREVISION= ...
DISTNAME=...
DISTFILES=....tar.gz
CATEGORIES=...
MASTER_SITES= ...
MAINTAINER=...
HOMEPAGE=...
COMMENT=...
.include "../../mk/bsd.pkg.mk"
```

Of course, this is only rarely enough, but often only a few additions are needed. We will discuss possible additions below. The ".include" directives are mandatory, and load the rest of the build framework. The variables have this meaning:

- `VERSION`: This is the version string of the package, without revision suffix ("godi" plus revision number)
- `PKGNAME`: The package name, including the version string at the end (as defined by `VERSION`).
- `PKGREVISION`: The revision number. This is a natural number ≥ 0. If omitted, the revision number is assumed to be 0 which is the same as not specifying a number.
- `DISTNAME`: The name of the directory below `work` into which the distribution files are unpacked, i.e. the name of the topmost directory of the distribution tarball. This is usually the name of the package, optionally including the version string. If omitted, `DISTNAME` is tried to be derived from `DISTFILES`.
- `DISTFILES`: The names of the distribution files. This can be any number of files (even zero); the names are separated by spaces. These files are downloaded from `MASTER_SITES`, stored into the distfiles directory, and finally unpacked in `work`.
- `CATEGORIES`: This should be the category prefix of `PKGNAME`, i.e. apps, conf, godi, or base. In the future it will be possible to assign a package to several categories, and the categorization will be visible in the user interface.
- `MASTER_SITES`: A list of URLs where the `DISTFILES` can be downloaded. The URLs are tried in turn, so you can mention mirror sites in addition to the primary site. The URLs must end with a slash (or more precisely, with the separator character to which the file name can be appended). Currently, only "http" and "ftp" URLs are allowed. There is special support for common download sites like Sourceforge, e.g. one can define
  `MASTER_SITES=${MASTER_SITE_SOURCEFORGE:=path/}`
  where path is the part to append to the master URL. (This strange syntax is needed because `MASTER_SITE_SOURCEFORGE` expands to a list of URLs!)

  The GODI backup URL is implicitly added to `MASTER_SITES`, and when neither of the URLs work, the GODI backup server is tried in a final attempt.
- `MAINTAINER`: The name of the package maintainer, including email address
- `HOMEPAGE`: The homepage of the packaged software
- `COMMENT`: The short, one-line description of the package

With only these variables, the stage actions are defined as follows:

- fetch: The `DISTFILES` are downloaded from the `MASTER_SITES`.
- extract: The `DISTFILES` are unpacked into `work`.

- patch: Applies the patches to `work`
- configure: Does nothing, the default is that there is no configuration script
- build: Changes to the subdirectory of work containing the unpacked sources, and tries to build the target "all"
- install: Changes to the subdirectory of work containing the unpacked sources, and tries to build the target "install"
- package: Creates the binary package from the installed image

We will now discuss how to define a number of frequent variations of this default. As most configurations can be done by setting further variables, the reference document of the variables is quite important. This is makevar-ref.txt (XXX where installed).

5.3.1 Customising the "extract" stage

There are not really many ways for this type of customisation. If only some `DISTFILES` need to be unpacked, one can set

- `EXTRACT_ONLY`: Enumerates the file names to extract. This should be a subset of `DISTFILES`.

5.3.2 Customising the "patch" stage

Patches need not to be declared in Makefile (only in distfiles). The patches must be put into the patches directory. Usually, the file names of patches have the convention

`patch-`<letter><letter>`-`<info>

where the two letters define the order in which the patches are applied. The info suffix is just a descriptive string, e.g. one can mention the patched file (if it is only one file).

Patches are applied relative to the toplevel directory of the unpacked sources. The patch format should be "unified" patches. It is not allowed that files are created or deleted by patching; in the first case one must copy the files from the `files` directory to the source tree, and in the latter case one must delete the files by a script.

How to create patches

In this example we create a patch for the file `foo.txt` which is part of the `bar-3.14` package.

1. Copy the original version of the file:
```
cd work/bar-3.14
cp foo.txt foo.txt.orig
```

2. Modify `foo.txt` as required
3. Create the diff file (current directory is still `work/bar-3.14`):
   `diff -au foo.txt.orig foo.txt >../../patches/patch-aa-foo.txt`

If the file to patch is located in a subdirectory, do not change to this subdirectory! Create the difference always from the toplevel directory of the unpacked sources.

Finally, you should also update `distfiles`, see below how to do this.

5.3.3 Customising the "configure" stage

As mentioned, the default is not to configure the software. The following variables enable this:

- `HAS_CONFIGURE=yes`: Enables to call a configuration script.
- `CONFIGURE_SCRIPT`: The name of the script to call. This defaults to "configure".
- `CONFIGURE_ARGS`: The arguments to be passed to the script. By default empty. Note that the arguments are in shell syntax, i.e. separated by spaces, and if necessary, quoted. E.g.

  ```
  CONFIGURE_ARGS="word1 word2" word3
  ```

  passes two arguments to the script where the first one is composed of two words.

  If you just want to pass Makefile variables, the "Q" modifier is useful. It quotes automatically, e.g.

  ```
  CONFIGURE_ARGS=-prefix ${LOCALBASE:Q}
  ```

  so `LOCALBASE` is here always an argument of its own.

  As with many other "plural" variables that have a list of arguments as values, it is common to use the "+=" operator to add arguments, e.g.

  ```
  CONFIGURE_ARGS+=-prefix ${LOCALBASE:Q}
  CONFIGURE_ARGS+=-with-foo
  ```

  This operator appends the new value to the already existing list, and ensures that there is a space character as separator.
- `CONFIGURE_ENV`: Environment variables to be passed to the script. A number of variables are already passed by default (see reference). The syntax is simply:

  ```
  CONFIGURE_ENV+=var=value
  ```

  where `var` is the name of the variable, and `value` the new value.
- `CONFIGURE_DIRS`: Lists the directories where to invoke the configure script. By default, this is only done in the toplevel directory of the source tree.

5.3.4 Customising the "build" stage

One can set a number of variables, and there are also some typical "code snippets".

- `USE_GMAKE=yes`: Effects that GNU make is used to build the software. By default, `godi_make` is used (which is a BSD-type make utility).
- `MAKEFILE`: The name of the invoked Makefile of the source tree. Defaults to "Makefile".
- `ALL_TARGET`: The target(s) to pass to "make" to build the software. Defaults to "all". See below for a discussion.
- `MAKE_FLAGS`: Further arguments to pass to "make". Especially, it is possible to override variables of the invoked Makefile by passing "name=value" arguments. The arguments are again in shell syntax.

  Pitfall: There is also a variable `MAKEFLAGS`, without underscore. This variable has a different meaning, as it is used to pass flags to recursive invocations of godi_make. Don't touch it!
- `MAKE_ENV`: Environment variables to be passed to "make". A number of variables are already passed by default (see reference). The syntax is simply:

  ```
  MAKE_ENV+=var=value
  ```

  where `var` is the name of the variable, and `value` the new value.
- `BUILD_DIRS`: Lists the directories where "make" is invoked to build the software. By default, it is only invoked in the toplevel directory of the source tree. By setting this variable to the empty string, the default action for the "build" stage is completely disabled.

Often, O'Caml software must be built with "make all" to get the bytecode version, and "make opt" compiles to the native code version (if supported). This is usually expressed by this code snippet

```
.if ${GODI_HAVE_OCAMLOPT} == "yes"
ALL_TARGET= all opt
.else
ALL_TARGET= all
.endif
```

(or some variation). The variable `GODI_HAVE_OCAMLOPT` expands to `yes` when the ocamlopt compiler is available, and to `no` otherwise. (Note that there are some more variables that describe the properties of the O'Caml core, see the variable reference.)

When findlib is used by the software, a special configuration must be enforced. This configuration sets a different lookup path for libraries such that the GODI-managed libraries have precedence. To get it, put this line into Makefile:

```
MAKE_ENV+= ${BUILD_OCAMLFIND_ENV}
```

If forgotten, it may happen that the build fails because a required library is not found, or in the wrong version.

5.3.5 Customising the "install" stage

Many of the "build" variables are also applied in the "install" stage, namely `USE_GMAKE`, `MAKEFILE`, `MAKE_FLAGS`, `MAKE_ENV`. The following variables are specific for "install":

- `INSTALL_TARGET`: The target(s) to pass to "make" to install the software. Defaults to "install".
- `INSTALL_DIRS`: Lists the directories where "make" is invoked to install the software. By default, it is invoked in the same directories as for the "build" stage. By setting this variable to the empty string, the default action for the "install" stage is completely disabled.

It is quite common to add post-install actions to Makefile, because often documentation files are not installed by the installation procedure defined by the packaged software. An example for such an action is:

```
post-install:
    ${MKDIR} ${PREFIX}/doc/godi-getopt
    ${CP} ${WRKSRC}/README ${WRKSRC}/COPYING \
      ${PREFIX}/doc/godi-getopt
```

There are a number of further variables here:

- `PREFIX`: This is the base directory where packages are installed. The variable `LOCALBASE` has usually the same value, but for a number of reasons `PREFIX` should be preferred when files are installed, and `LOCALBASE` should be preferred when files are looked up (owned by other packages that are already installed)
- `WRKSRC`: This is the toplevel directory of the source tree, as absolute path.
- `MKDIR`, `CP`: These are *defined* commands. Of course, one could also directly call `mkdir` and `cp`, but there are systems where there several versions of various commands (e.g. a BSD version and a System V version), and the GODI framework has selected one that can be called through the mentioned variables.

5.3.6 Dependencies

In Makefile one can also declare dependencies on other packages. The variables are:

- `DEPENDS`: Lists strict dependencies for build and runtime
- `BUILD_DEPENDS`: Lists build-time-only dependencies

The syntax for the expressions one can use in these variables:

- <BASENAME> <OPERATOR> <VERSION> `:` <PATH>

where <BASENAME> is the name of the package without version string, <OPERATOR> is one of ==, >=, <=, >, <, `!=`, and <VERSION> is the version string the operator refers to. Note that package revisions in <VERSION> (i.e. any "godi" suffixes) are ignored, and one cannot enforce a certain revision.

The <PATH> is (almost) a legacy component of the dependency expressions, i.e. it is ignored by godi_console, but there are still some scripts that need it. The <PATH> must be set to

`../../`<CATEGORY>`/`<BASENAME>

where <CATEGORY> is the category prefix of <BASENAME> (apps, godi, etc.). The <PATH> is the relative path where to find the build directory of the package one refers to.

Examples for package dependencies:

- `DEPENDS+=godi-ocaml>=3.07:../../godi/godi-ocaml`
- `BUILD_DEPENDS+=conf-zlib>=0:../../conf/conf-zlib`

The comparison ">=0" should be read as "any version is accepted". There is also the legacy syntax "-*" or even "-[0-9]*" for the same purpose, e.g. "conf-zlib-*", but it should no longer be used in new packages.

Selecting the kind of dependency

`BUILD_DEPENDS` should only be used when it can be ensured that the referred package is only needed at build time, *and in all other cases* ***`DEPENDS`*** *must be used.* Examples when build-time dependencies are sufficient:

- A generator or other build-time tool is required, and the other package includes it. For instance, godi-findlib is usually a build-time dependency because of this rule.
- The other package configures the build. All conf-* packages can be referred to by build-time dependencies.

- An application program is built, and all executables are either created with `ocamlc -custom`, or `ocamlopt`. In this case, the used libraries are not necessary at runtime, and the corresponding packages can be listed in `BUILD_DEPENDS`.

  Note, however, that there are exceptions from this rule. For example, godi-ocamlnet also includes runtime files that are required even when the application is completely statically linked.

In general, when a library is linked into a program, or a library is the antecedent for another library, the dependency is of the runtime type.

Magic dependencies

There are dependencies that need normally not to be listed, because it is impossible to even start the build process without a certain minimum equipment. These dependencies include:

- godi-tools: This is the package including godi_console.
- godi-core-mk: This package contains the build framework.

One can, however, demand certain minimum versions for these packages. This is sometimes useful when one needs a special feature that was only recently introduced into the packaging system.

5.3.7 Legacy expressions

In the past, it was necessary to put a number of expressions into Makefile to get a certain behaviour, but due to improvements this is no longer required:

- `GODI_PLIST`: When set to "yes", the file PLIST.godi is used instead of PLIST
- `.include "../../mk/godi.pkg.mk"`: This is now an empty file
- `PATH:=${LOCALBASE}/bin:${PATH}`: This is now always done

If you find that in existing packages, don't copy it to new packages.

5.4 Packing lists

The file PLIST describes the installed files, so GODI knows which files are owned by the package, and the files are removed when the package is deleted.

In general, the "install" stage must already have arranged that the files are installed in the right directories. Often, this is done by passing PREFIX to the configure script, but there is no general technique how this can be achieved. In doubt, read the documentation coming with the software, and, of course, the Makefiles and other build scripts included in the software distribution.

The PLIST file is needed first when the "install" stage is entered. Before this stage, the PLIST file can be omitted.

If you don't know the software you are packaging, it is sometimes difficult to find out which files are actually installed. As mentioned, the PLIST file is needed *before* the installation is done (at the beginning of the "install" stage), so you cannot just install and see what has been installed. See below for a discussion how to cope with this problem.

In the simplest form, PLIST just lists the files that have been installed, plus includes a number of directives how to deal with directories. For example:

```
bin/foo
```

This one-liner means that the package consists only of the executable `bin/foo`. Filenames are relative to the installation prefix of GODI, and it is an error to use absolute path names.

Because `bin` is a shared directory, no special handling of it is required.

For private directories, however, one should add `@dirrm` directives to PLIST. For example, this package installs files into `lib/foo`, and because this directory is considered to be the private property of the package, it must be removed when the package is deleted. This is achieved by the `@dirrm` directive:

```
lib/foo/bar.txt
lib/foo/baz.a
@dirrm lib/foo
```

As `lib` is again shared, it is not necessary to add another `@dirrm` directive for it. These directives should come after the files contained in the directories, and when private directories occur within private directories, the `@dirrm` directives must be in the order such that the inner directory comes first. (As you guess it, at package deletion time the PLIST is just interpreted line by line from top to bottom, and the removal actions are performed in this order.)

Note that the handling of directories is going to be changed. In the future, it will no longer be necessary to add @dirrm statements to PLIST. In a development version of godi_console, this revised handling is already implemented, but it will take some time to release it.

As it is quite error-prone to enumerate files, there are a number of abbreviations and special notations. We explain here only the most useful ones, for a complete reference see plist-ref.txt (XXX where installed?).

- `@deepdir` <dir>: The mentioned directory and its contents (including subdirectories) are declared to be owned by the package.
- `@findlib` <name>: The named findlib library is declared to be owned by the package. When the library includes DLLs, these are ignored, however.

- `@dllfindlib` <name>: The named findlib library and the included DLLs are declared to be owned by the package. (Note: The ownership of the DLLs is only recognised when there are .owner files. This is the case when the library was installed with ocamlfind, but is usually not the case when another method was used.)
- `@optional` <directive>: The files covered by the <directive> are only installed in certain configurations, and may be missing in other configurations. E.g. `@optional @findlib` <name> means that the library is only optionally installed. (Note: Currently, `@optional` is broken when <directive> is a plain file. Use `@glob` instead, without glob metacharacter.)

In most cases, these four directives are sufficient.

There is still the problem how to figure out which files are installed. A simple method:

1. Begin with an empty PLIST, and do `godi_make install`
2. Get the installed files by calling `godi_make print-installed`. Note, however, that there is no guarantee that this list is complete, the "print-installed" script checks the timestamps of all files in the GODI installation. This may go wrong, especially on systems where the "find" command does not support the comparison of the ctime field of the timestamp (which is less problematic than the mtime field).
3. Write the PLIST according to the output of "print-installed". One should keep in mind that some files are only installed for certain system configurations, so just copying the list may not be enough.
4. Now remove the package, and the files (so the package database is clean again):
   `godi_delete` <packagename>
   `godi_make print-installed | ( cd <PREFIX>; xargs rm -f )`
5. Remove these files, and install the package again:
   `rm work/.install_done work/.PLIST`

5.5 An Example

Here the Makefile of godi-xstr (slightly updated):

```
.include "../../mk/bsd.prefs.mk"
VERSION=        0.2.1
PKGNAME=        godi-xstr-${VERSION}
DISTNAME=       xstr
DISTFILES=      xstr-${VERSION}.tar.gz
CATEGORIES=     godi
MASTER_SITES=   http://ocaml-programming.de/packages/
MAINTAINER=     gerd@gerd-stolpmann.de
```

```
HOMEPAGE=       http://ocaml-programming.de
COMMENT=        additional string functions

DEPENDS+=       godi-ocaml>=3.06:../../godi/godi-ocaml
BUILD_DEPENDS+= godi-findlib>=0.8.1:../../godi/godi-findlib

MAKE_ENV+=      ${BUILD_OCAMLFIND_ENV}

USE_GMAKE=      yes

ALL_TARGET=     all
.if ${GODI_HAVE_OCAMLOPT} == "yes"
ALL_TARGET+=    opt
.endif

post-install:
        ${MKDIR} ${LOCALBASE}/doc/godi-xstr
.       for F in README LICENSE
            ${CP} ${WRKSRC}/${F} ${LOCALBASE}/doc/godi-xstr
.       endfor

.include "../../mk/bsd.pkg.mk"
```

The corresponding PLIST file:

```
@findlib xstr
doc/godi-xstr/README
doc/godi-xstr/LICENSE
@dirrm doc/godi-xstr
```

5.6 Further targets for godi_make

The following targets have relevance:

- clean: Deletes the "work" directory, and resets the build process to a clean, initial state.
- print-installed: Prints a list of files that have been installed (created or modified) since the last time the package was unpacked. The underlying method is not fully reliable.
- makesum: Creates the distinfo file, and puts entries for all DISTFILES into it.
- makepatchsum: Adds checksums for patches to an already existing distinfo file.

5.7 Testing packages

One of the basic properties every package must have is that one can delete it. So the most primitive test checks this:

1. Install the package with `godi_make install`

2. Delete the package with `godi_delete`, check that there are no error messages.
3. Check that `godi_make print-installed` does not output anything!

5.8 The package repository

The package repository is realised with Subversion. Everybody can check it out:

> svn checkout https://gps.dynxs.de/svn/godi-build

Of course, you need an account to modify any of the files. If you had one, you could do the following:

- Copy your new package into the checked-out directory hierarchy. Normally, the right directory is godi-build/trunk plus the relative path of the package directory.
- Add the package to the repository: "svn add" calls, and commit it: "svn commit".
- Ask other developers to check the package out, and to test it
- Release the package: Add the right line to godi-build/pack/release.<SECTION>.map, invoke the web application called "Release Tool", and do the release.

I have prepared more detailed instructions how to perform these actions for everybody who wants to have an account on the GODI server.

5.9 Check list

- Can the package be cleanly deleted?
- Does the package also work for other types of systems than your own one? Not every system supports DLLs, for example. Not every system has the ocamlopt compiler.
- Are all dependencies declared?
- Are the descriptions in DESCR, and optionally BUILDMSG up to date?
- Is the documentation installed? As bare minimum, there should be a file clarifying the license conditions.
- Are the examples installed? These are part of the documentation.
- Are all references to external C libraries covered by conf packages? There should also be conf packages for other unusual equipment that is not part of GODI.

(to be extended...)

Chapter 6

External Dependencies

The autoconf nightmare.

6.1 Configuration packages

6.2 The policy for libary lookup

6.3 Using `godi_script` to create configuration packages