Building Infrastructure for DROPS (BID) Specification

Jork Löser Ronald Aigner
drops@os.inf.tu-dresden.de

April 11, 2012

Abstract

To ease the writing of DROPS components, this document describes the unified infrastructure of the DROPS source tree. It also describes what files and macros should be used to create the Makefiles needed to compile own components.

Acknowledgements

This document is based on the work of Lukas Grützmacher, Michel Hohmuth, Jork Löser and Lars Reuther done in 2000. Since then, Frank Mehnert proposed a lot of functionality which is included in this document as well.

Availability

This document is also available in HTML format. A short abstract is available as abstract.ps and abstract.pdf. You can download this documentation in its various formats at http://os.inf.tu-dresden.de/l4env/.

Contents

1 Source-Tree

Directory/File	Comment
`l4/`	base-directory, contains *no* `Makeconfs`
`l4/kernel/`	kernel sources
`l4/pkg/`	contains the packages
`(disappeared)`	was: l4/Makeconf
`l4/mk/`	contains role-files (see Section 2)
`l4/pkg/PKGANME/`	DROPS-components (packages)
`l4/pkg/PKGNAME/include/`	headers to be exported by that package
`l4/pkg/PKGNAME/idl/`	IDL-definition
`l4/pkg/PKGNAME/{lib, server}/{lib, src, include}`	sources
`l4/pkg/PKGNAME/etc/, examples/, man/, doc/, README, TODO, INSTALL, COPYING`	preferred names for according files
`l4/pkg/PKGNAME/Makefile`	package-Makefile, there is no symlink anymore
`l4/{bin, lib}/{x86_586/,x86_-586/v2/,ia64/,...} and l4/{include, idl}`	Build-Installation path for the packages. These directories contain stripped versions of compiled binaries.

1.1 MAINTAINER files and L4Check

To maintain at least a low level of code consistency in our DROPS project, we designed the L4Check [1] tool. It runs every night and checkouts the various packages from CVS and builds them. If the build-process fails, an email is automatically generated and sent to the maintainer of the failing package. To define the maintainer of a package (to be more precise, of a subdirectory), a so-called *maintainer* file is used. This is a file named `MAINTAINER` containing lines of the following syntax:

`mailaddr` ⟨*mail address*⟩ `[,` ⟨*mail address*⟩ [, ...]]

If an error occurs within the directory subtree containing the maintainer file, a mail indicating the error will be sent to the specified mail addresses. You can add maintainers for a specific subdirectory tree by placing an additional maintainer file in the root of that tree. The “sub-maintainers” will receive a mail in the case of an error in their subtree. Additionally, the maintainers of the upper directories will receive an according mail.

1.2 Package Flags

broken If a package becomes broken for some reason, place a file named `broken` in the top-level directory of that package and it won’t be built.

obsolete If a package becomes outdated for some reason, place a file named `obsolete` in the top-level directory of that package and it won’t be built.

2 Directory Roles

Each directory in the source-tree has a distinct task, i.e. each directory builds special kinds of targets. This distinct task is expressed by a role, where role is one of the following:

subdir:	be a container for other directories and recursively build them
prog:	build executable binaries
lib:	build libraries
idl:	translate IDL definitions into code
include:	hold include-files which have to be installed
doc:	build documentation for the package
ptest:	test the package using Fiasco-UX

To define the role of a directory, the Makefile of that directory has to include an according Makefile-include, the *role-file*. The role-files are templates that define rules and variables to build the proper targets. Make-variables are used to control the behaviour of the role-files. Each Makefile must include exactly one role-file.

To define the role of a directory, use the following code in your Makefile:

subdir:	`include $(L4DIR)/mk/subdir.mk`
prog:	`include $(L4DIR)/mk/prog.mk`
lib:	`include $(L4DIR)/mk/lib.mk`
idl:	`include $(L4DIR)/mk/idl.mk`
include:	`include $(L4DIR)/mk/include.mk`
doc:	`include $(L4DIR)/mk/doc.mk`
ptest:	`include $(L4DIR)/mk/runux.mk`

Makeconf.local's

The Make-variables to control the role-files are defined in various files: The Makefile, a local `Makeconf.local`, a packet-local `Makeconf.local`, a global `Makeconf.local` and a global `Makeconf.bid.local`. The local `Makeconf.local` resides within the same directory as the Makefile, the packet-local `Makeconf.local` is located in the top-level directory of a package, and both global `Makeconf.local` and `Makeconf.bid.local` reside in `$(L4DIR)`.

The gobal `$(L4DIR)/Makeconf.bid.local` is created by the DROPS Configuration Tool (see Section 3) when called in `$(L4DIR)`. This file is mandatory, i.e., you have to call `make config` and configure your DROPS tree prior to compilation. In contrast to this, the `Makeconf.local`s are optional, they are included if they are available. Create them with a text editor.

The `Makeconf*local`'s should never be checked into the CVS, as they are meant to define user-specific things. Additional configuration files can be created using the DROPS Configuration Tool.

The role-files include the configuration files in the following order: global Makeconf.bid.local, global Makeconf.local, packet-local, local.

3 DROPS Configuration Tool

Most role-files provide support for an interactive configuration tool, which is derived from the *config* and *menuconfig* tools of Linux. It differs in that it can be configured using environment variables. Some minor extensions are build in too.

The configuration tool is run if the user runs "`make config`", "`make txtconfig`" or "`make oldconfig`". The following parameters control the configuration tool.

`DROPSCONF`	if non-empty, the configuration tool will be run for the makefile-targets {**old,txt,**}**config::**. If neither {**old,txt,**}**config**, **scrub**, **clean** nor **cleanall** is within the make targets, the file `$DROPSCONF·CONFIG` is included using the -include directive. If this parameter is empty, no commands are run and no file is included. However, you can specify additional commands to be run in your own Makefile. Modifies **help::**, **clean::** and **cleanall::** rules. Default: empty, do not run the configuration tool on "`make config`".
`DROPSCONF_DEFCONFIG`	the default configuration file, which is used as the initial configuration. Default: `defconfig`
`DROPSCONF_CONFIG_IN`	the configuration definition file. Default: `config.in`
`DROPSCONF_CONFIG`	the configuration file, which holds the configuration. This file is automatically build from `$DROPSCONF_DEFCONFIG` when needed. Default: `.config`
`DROPSCONF_CONFIG_H`	the configuration header file, which is included by the source files. Default: `config.h`
`DROPSCONF_CONFIG_MK`	the make include file, which is included by the makefiles. Default: `Makeconf.bid.local`
`DROPSCONF_DONTINC_MK`	If this option is non-empty, inclusion of the make include file is supressed. Default: empty
`DROPSCONF_HELPFILE`	the file containing the option help texts. Default: `config.help`
`DROPSCONF_MACRO`	the macro which is defined when DROPSCONF_CONFIG_H is included. Default: `CONFIG_H_INCLUDED`
`DROPSCONF_TITLE`	the main title in the configuration tool. Default: "DROPS Configuration Tool"
`DROPSCONF_TOOL`	the path to the menu-driven configuration tool used for the **config::** target. Default: `$(L4DIR)/tool/config/Menuconfig`
`DROPSCONF_TOOL_TXT`	the path to the configuration tool used for the **txtconfig::** target. Default: `$(L4DIR)/tool/config/Configure`
`DROPSCONF_TOOL_OLD`	the path to the configuration tool used for the **oldconfig::** target. Default: `$(L4DIR)/tool/config/Configure -d`
`DROPSCONF_UNDEF`	If nonempty, disabled boolean options won't be defined in `$(DROPSCONF_CONFIG_H)` at all. If empty, disabled options will be defined as `0`. Enabled boolean options are defined as `1` always. Default: empty.

The configuration tool creates three files: The configuration file (`$DROPSCONF_CONFIG`), the configuration include file (`$DROPSCONF_CONFIG_H`) and the make include file (`$DROPSCONF_CONFIG_MK`). The first contains the configuration in a format suitable for the config tool. The second is a standard include-file for C files. The last contains the configuration in a format to be included in Makefiles. Note, that it is automatically included in your Makefile when one of the role-files is included and if your Makefile supports configuration. The role-files ensure that the configuration file is build appropriately. If you want to include the configuration file in other Makefiles, make sure the configuration file is built.

When using the configuration tool, the special target **DROPSCONF_CONFIG_MK_POST_HOOK::** is called after generating `$(DROPSCONF_CONFIG_MK)`. You can add your own commands here, the configuration options will already be effecive during execution.

4 Build Directories

BID does use build-directories to contain all generated files. So your source directory will not be modified. If you try to configure or build your L4 system without specifying a build-directory an error is issued. Simply name an build-directory (or output directory) when calling make with the variable `O` set:

```
˜> make O=/path/to/your/build/dir
```

To create this directory and set up a basic Makefile infrastructure in this directory you have to call first:

```
˜> make O=/path/to/your/build/dir config
```

4.1 Building External Sources

If you do have a source directory outside your regular L4 setup and want to build this into your build directory you may do so using the `SRC_BASE` environment variable. To determine the target location in the build directory BID requires a reference point. This reference point is denoted by the `SRC_BASE` variable. An example:

```
L4DIR=$(HOME)/src/l4
O=$(HOME)/build-dirs/build-v2
PWD=$(HOME)/test/pkg/my-pkg
```

I you want `my-pkg` to appear in the `pkg` sub-directory in the build-directory, you would have to set `SRC_-BASE` to `$(HOME)/test`. Thus your make invocation would look like this:

```
˜/test/pkg/my-pkg > make O=$(HOME)/build-dirs/build-v2 \
L4DIR=$(HOME)/src/l4 SRC_BASE=../..
```

4.2 Parallel Build

While talking about building your source we might as well mention parallel builds here. If you simply run `make -j` in your package directory, you will notice that make will drop several dependencies (see Section 5) and then start a parallel build. This is actually not the intended behaviour, because of course we would like to keep the dependencies intact. Thus we must only enable parallel build where it is appropriate, for instance, in one sub-directory containing only source files. Therefore, we use the environment variable `PL` to specify the level of parallelism. You may invoke `make` like this:

```
˜/src/l4/pkg > make O=$(HOME)/build-dirs/build-v2 PL=4
```

5 Dependencies

5.1 Source-code dependencies

Dependencies are built automatically. When compiling source-files into intermediate object files, each source-file is accompanied by an according dependency file. If the source-file is named `x`, then the dependency file is named `.x.d`. Typically these dependency files contain header files. For binary linking the same rules apply. Dependency files for binaries typically contain libraries. Dependency files are deleted on "`make cleanall`".

For generating of dependency files, we use two methods: If the host system provides and uses the dynamic linker *ld.so*, we use *libgendep* [3] to build the dependency files. libgendep uses the `LD_PRELOAD` method to overload the `open()`- and `fopen()`-functions of libc during compilation or linking. All files opened read-only are added to the list of dependencies for the current target. This is a flexible solution, as it can be used on a broad range of compilers, linkers and other tools. To enable this tool, set `HAVE_LDSO` (Section 8.1.3) to nonempty in your local configuration files.

If the host system does not provide *ld.so*, we fall back to the second method of dependency file generation: For compilation of C and C++ files, we use *gcc* to generate dependency files and postprocess those. For linking, we use a heuristic that mimics the behavior of the linker *ld*. Please note, that this heuristic does not reflect the exact behavior of the linker for efficiency reasons and has several problems (see below). For IDL compilation with *dice*, we use *dice* to generate according dependency files and postprocess those.

With both methods, dependency files are created automatically and they are used automatically. We avoid a typical error when dealing with dependency files: If a dependency file poses a dependency of a source-file `src.c` to an include file `inc.h`, and `inc.h` is deleted with the appropriate modification in `src.c`, the build process may fail: The make process notifies the dependency because of the unmodified dependency file, but make does not know how to build `inc.h` (albeit it is not needed anymore). Hence, make throws an error. We circumvent this by adding a rule "`inc.h:`" to the dependency file. The result is, that make rebuilds `src.c` if `inc.h` disappears — exactly what we want.

However, the fall-back method for libraries has a problem with invalid symbolic links: If a symbolic but invalid link is found within the library directories, this is used for the library dependencies. This might be the wrong choice (as the actual library might appear later in the search path), resulting in an abortion of the compilation process.

5.2 Configuration File Dependencies

The dependency generation also covers configuration and Makefiles. This is, each object file and target is made dependent on the Makefile and on the various `Makeconf.*local`'s present in the file-system.

Information about if a configuration file is existing or not is stored in a file named `.general.d`, and this file is made dependent of the existing files. The role-files then ensure that all objects files and targets depend on this `.general.d`.

5.3 Package Dependencies

To resolve inter-package dependencies basically two mechanisms are used. The first one is an explicit notation in the `l4/pkg/Makefile`: each package is separated into three parts: `headers`, `lib`, and `bin`. The headers parts consists of the `idl` and `include` subdirectories of the package. The lib part contains the `lib` subdirectory and the bin part does contain the `server` and `examples` subdirectories. Usually the L4 packages are build in the order of first building all `headers` sections, then all `lib` sections and finally all `bin` sections. The packages belonging to the L4 Environment are here preferred above the rest: so first all of the above steps for the L4 Environment, later the same steps for the rest.

You will find several of explicit dependencies between the parts of different packages specified in `l4/pkg/Makefile`. These are respected by make during the build process and will ensure the correct order of building the packages. However, this mechanism fails if a package is not present in your source tree. Then make will silently ignore that dependency and you will probably see errors of missing include files during compile or unresolved symbols when linking binaries. This is where the second mechanism comes into play.

In your Makefiles you may specify the Variable `DEPENDS_PKGS` with a list of other packages that have to be present for this package to compile. Whenever the build systems finds this variable it will check in the `l4/pkg` directory for package directories with that names. If a package from the list is missing in the directory, an error is printed and the make run is aborted. Thus, you can specify inter-package dependencies.

6 Systems, Architectures and Modes

This section describes how to build for multiple architectures and for multiple target systems.

6.1 Definitions of Terms

When building for multiple environments, we distinguish between the *system* and a *mode*:

System specifies the system environment of the target machine including the running μ-kernel. System itself splits up into three parts:

```
<SYSTEM> = <ARCH> "_"<CPU> [ "-"<L4API> ]
```

`<ARCH>` specifies the architecture of the target processor. `<CPU>` specifies a model of the processor in more detail. It can be used for certain optimizations. `<L4API>` specifies the binding of the μ-kernel. See Table 4 for a list of currently defined architectures, CPU types and L4 APIs.

The L4API part is optional. If it is omitted, the target is assumed to be independent of a specific L4 binding. Each target can be compiled for multiple systems at the same time.

Mode covers the target execution environment respectively the C-libraries binaries should be linked against. Examples are *l4env*, *loader* or *host*. See Section 6.5 for a list of suggestions. Each target can be built/compiled for only one mode.

ARCH	<u>x86</u>	arm	amd64	ppc32
CPU	<u>586</u>, 686, K6, K7	<u>sa</u>, pxa, int	<u>k8</u>	<u>750</u>
L4API	<u>l4v2</u>, l4x2, l4secv2emu	<u>l4x0</u>	<u>l4v2</u>, l4x2	<u>l4x2</u>

Table 4: Valid values for the components of a system description. Defaults are underlined.

In contrast to the mode, the system is visible within the directory structure: The system of a (compiled/linked/built/installed) file is encoded in the directory part of the name of that file. For example, if a library is compiled for system *x86_586-l4v2*, its name is `libfoo.a`, it will be installed to `$(L4DIR)/lib/x86_586/l4v2/libfoo.a`. Another library `libbar.a` compiled for system *x86_586*[1], it will be installed to `$(L4DIR)/lib/x86_586/libbar.a`. This is the case for includes and binaries analogically. So far for the installed files.

The compilation process must respect multiple systems too: We allow to build multiple systems at the same time from the same source-files. Therefore, the maintainer of a package specifies the possible systems[2]. In a global configuration file (maintained by the user that compiles its tree), all systems that should be compiled are specified.

[1] hence, it is independent of an L4 binding
[2] read below about the granularity

```
pkg/
 +- foo/
     +- include/
     +- lib/
     |   +- src/
     |       +- Makefile1
     |       +- Make.rules
     |       +- foo.c
     |       +- OBJ-x86_586-l4v2/
     |       |   +- Makefile2
     |       |   +- foo.o
     |       |   +- libfoo.a
     |       +- OBJ-ia64-l4v2/
     |           +- Makefile3
     |           +- foo.o
     |           +- libfoo.a
     +- server/
```

Figure 1: Structure of an example package

6.2 File System Representation

Let us look now at a package subdirectory, containing the source-file `pkg/foo/lib/src/foo.c`. We compile for systems *x86_586-l4v2* and *ia64-l4v2*. To distinguish between both object files, we place them in different directories: `OBJ-<system>`: `.../src/OBJ-x86_586-l4v2/foo.o` and `.../src/OBJ-ia64-l4v2/foo.o`. As we have a smooth migration from one target system to multiple systems in mind, we compile *always* into subdirectories, even if we compile for only one system. Thus, we have the structure as depicted in Figure 1 (please, ignore the Makefiles for now).

6.3 Compilation for Multiple Systems

Within the Makefile in the `.../src/` directory, `L4DIR` and `PKGDIR` and the supported systems are specified. Also, either `prog.mk` or `lib.mk` is included. During the make process, BID creates the system-specific subdirectories with Makefiles therein. Those Makefiles define the variables `L4DIR`, `PKGDIR`, `SYSTEM`, `ARCH`, `CPU` and `L4API`[3]. During compilation of C and C++ files, the preprocessor macros SYSTEM_$(SYSTEM), ARCH_$(ARCH), CPUTYPE_$(CPU) and L4API_$(L4API) will be defined. The generated Makefiles also include either the original Makefile in the upper directory or alternatively a separate file `Make.rules` if it exists within the upper directory.

Having that separate Make.rules file might be useful, as otherwise all definitions have to be done in that one Makefile. Note, that this Makefile is interpreted in the src/ directory and in the system-specific subdirectories. To prevent trouble, you can write all the declarations meant for the src/ directory in the Makefile (this are L4DIR, PKGDIR and the systems normally). All declarations for generating the targets and controlling the actual build process for one system in Make.rules. Make.rules will only be interpreted within the subdirectories.

If immediate evaluation of variables defined in the role-files is needed (e.g., to define rules), `$(L4DIR)/mk/*.mk` can be included in `Make.rules`. It is ensured, that the same role-file will only

[3]Note that `L4API` can be empty.

be included once. If no role-file is included explicitly, it will be automatically included after including Make.rules.

6.4 Specifying the Systems to build

So far, we described how to build multiple systems. No, we describe how to specify the systems a packet can build.

Each source directory of a package specifies the target systems this directory can be built for. The target systems are defined as a list within the parameter `SYSTEMS` in the Makefile of that source directory. The systems listed in `SYSTEMS` may have a CPU component, but it can also be empty. The L4API in `SYSTEMS` is optional too.

The user that compiles its L4-tree specifies the systems he wants to build. Therefore he sets the variable `BUILD_SYSTEMS`, e.g. using the global Makeconf.bid.local (see Section 3). The systems listed in `BUILD_SYSTEMS` always have an L4API component. The user may specify a specific CPU for optimization purposes within the CPU component. He may even list the same CPU architecture/L4API combination multiple times, each time with a different CPU type, e.g., `x86_386-v2 x86_586-v2 x86_K7-v2`.

Within a source directory, the matching systems are built: A system matches, if it can be found in both variables `SYSTEMS` and `BUILD_SYSTEMS`. If that directory requires no specific L4 binding (no L4API part within `SYSTEMS`), the L4API parts in `BUILD_SYSTEMS` are ignored. The CPU parts in `BUILD_-SYSTEMS` are always ignored for matching, but if a match is found, they become part of the matching system. More formal:

> From each item *sys* specified in `BUILD_SYSTEMS` the CPU component is removed. Then, each (shortened) item *sys* is matched against the list in `SYSTEMS`. If an element *i* of `SYSTEMS` can be found as a substring within *sys*, *i* is added to the intersection list. If the original value of *sys* had a CPU component, this CPU component is added to *i* as well.

Some packages built code for multiple systems and use an IDL compiler to generate the communication stubs. The systems these packets can be built for depend mainly on the IDL compiler. To avoid to change all the Makefile of those packages if the IDL compiler supports more systems, the variable `IDL_-SYSTEMS` contains all the systems our default IDL compiler *dice*[2] can generate code for. A Makefile can use `SYSTEMS=$(IDL_SYSTEMS)` to automatically build for all systems that are supported by dice.

Note that when compiling files in the `OBJ-*` subdirectories, the contents of `$(SYSTEM)` can not necessarily be found in the $(SYSTEMS) variable of the original Makefile, e.g. due to the added `$(ARCH)` component. To help writing system-specific makefiles, the make variable `$(OSYSTEM)` is defined by the generated makefile. It contains that part of the `$(SYSTEMS)` variable in the Makefile that corresponds to the current `$(SYSTEM)`. Note,

6.5 Modes

The following modes are defined:

tiny	Minimal environment. Includes basic libc functionality from one of the C libraries selected and utility functions.
sigma0	Minimal L4 environment. Includes basic libc functionality like tiny mode but additionally links *log* and *names* libraries.

l4env	Default mode to build L4Env applications. Sets appropriate include and library paths. Also links against the L4Env libraries. I/O output is printed to log.
l4env_minimal	l4env mode with just write(1, ...) as I/O backend.
l4env_base	l4env mode with the libc backends: basic_io, basic_mmap, mmap_util, syslog, simple_sleep, time, file_table.
libc	Rather internal mode to compile C libraries
host	Build an application for the host operating system.
l4linux	Build hybrid applications for L^{4}Linux, which are normal Linux applications, but are also linked against L4Env libraries.

Deprecated modes: l4linux_kern, l4env_oskit06, l4env_oskit10, l4env_freebsd, loader, sigma0_oskit06, oskit10_sigma0,

The mode specifies which standard C library and header files are being used, which objects for startup-code (CRT0), end-/marker-code (CRTN) are being linked and which linker scripts are used for all this. See Tables 6, 7, 8, 9, 10, 11, and 12 for what is determined by each mode. See Section 8.2 and Section 8.3 for the meaning of the variables.

Mode	`$LIBCINCDIR`
l4env	`-nostdinc -DUSE_DIETLIBC=y -I{ $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc $(GCCINCDIR) }`
l4env_minimal	`-nostdinc -DUSE_DIETLIBC=y -I{ $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc $(GCCINCDIR) }`
l4env_base	`-nostdinc -I{ $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc $(GCCINCDIR) }`
tiny	`-nostdinc -DUSE_DIETLIBC=y -I{ $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc $(GCCINCDIR) }`
sigma0	`-nostdinc -DUSE_DIETLIBC=y -I{ $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc $(GCCINCDIR) }`
libc	`-nostdinc -I{ $(GCCINCDIR) $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc }`
host	-
l4linux	`-I$(GCCINCDIR)`

Table 6: Directories for include file search depending on the mode for *dietlibc*.

Examples Examples for the various modes can be found within the following packages:

sigma0 *names*: library

host *names*: lister example

l4env *con*: library and server

6.6 Defining new architectures and modes

Defining additional architecture/mode pairs for a specific architecture is done by setting appropriate Make-variables. To define mode `mode` for architecture `arch`, set `BID_SUPPORTED_arch_mode` to “y” and

Mode	`$LIBCINCDIR`
l4env	`-nostdinc -I{ $(L4DIR)/include/{x86, arm, amd64}/uclibc $(L4DIR)/include/uclibc $(L4DIR)/include/uclibc++ $(DROPS_STDDIR)/include/{x86, arm, amd64}/uclibc $(DROPS_STDDIR)/include/uclibc $(DROPS_STDDIR)/include/uclibc++ $(GCCINCDIR) }`
l4env_minimal	`-nostdinc -I{ $(L4DIR)/include/{x86, arm, amd64}/uclibc $(L4DIR)/include/uclibc $(L4DIR)/include/uclibc++ $(DROPS_STDDIR)/include/{x86, arm, amd64}/uclibc $(DROPS_STDDIR)/include/uclibc $(DROPS_STDDIR)/include/uclibc++ $(GCCINCDIR) }`
l4env_base	`-nostdinc -I{ $(L4DIR)/include/{x86, arm, amd64}/uclibc $(L4DIR)/include/uclibc $(L4DIR)/include/uclibc++ $(DROPS_STDDIR)/include/{x86, arm, amd64}/uclibc $(DROPS_STDDIR)/include/uclibc $(DROPS_STDDIR)/include/uclibc++ $(GCCINCDIR) }`
tiny	`-nostdinc -I{ $(L4DIR)/include/{x86, arm, amd64}/uclibc $(L4DIR)/include/uclibc $(L4DIR)/include/uclibc++ $(DROPS_STDDIR)/include/{x86, arm, amd64}/uclibc $(DROPS_STDDIR)/include/uclibc $(DROPS_STDDIR)/include/uclibc++ $(GCCINCDIR) }`
sigma0	`-nostdinc -I{ $(L4DIR)/include/{x86, arm, amd64}/uclibc $(L4DIR)/include/uclibc $(L4DIR)/include/uclibc++ $(DROPS_STDDIR)/include/{x86, arm, amd64}/uclibc $(DROPS_STDDIR)/include/uclibc $(DROPS_STDDIR)/include/uclibc++ $(GCCINCDIR) }`
libc	`-nostdinc -I{ $(GCCINCDIR) $(L4DIR)/include/dietlibc $(DROPS_STDDIR)/include/dietlibc }`
host	–
l4linux	`-I$(GCCINCDIR)`

Table 7: Directories for include file search depending on the mode for *uclibc*.

Mode	`$LIBCLIBDIR`
l4env	–
l4env_minimal	–
l4env_base	–
tiny	–
sigma0	–
libc	–
host	–
l4linux	–

Table 8: Directories for library path search depending on the mode.

Mode	**$LIBCLIBS**
l4env	-nostdlib $(GCCLDNOSTDLIB) $(DIETLIBC_IMPLEMENTATION) -ldietlibc_support $(MALLOC_BACKEND) -lc_be_mmap -lc_be_mmap_util -lc_be_l4env_start_stop -lc_be_minimal_log_io -ldiet_be_simple_sleep -ll4env -llogserver_capsule -ll4rm -ldm_generic -ldm_mem -lthread -lgeneric_ts $(DIETLIBC_IMPLEMENTATION) $(GCCLIB) $(DIETLIBC_IMPLEMENTATION) -ll4sys
l4env_minimal	-nostdlib $(GCCLDNOSTDLIB) $(MALLOC_BACKEND) $(DIETLIBC_IMPLEMENTATION) -lc_be_l4env_start_stop -lgeneric_ts -lc_be_minimal_log_io -lc_be_mmap_util $(MALLOC_BACKEND) -lc_be_mmap -lc_be_mmap_util -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(DIETLIBC_IMPLEMENTATION) $(GCCLIB) $(DIETLIBC_IMPLEMENTATION) -lc_be_l4env_start_stop -ll4sys
l4env_base	-nostdlib $(GCCLDNOSTDLIB) -u printf -lc_be_io.o $(MALLOC_BACKEND) -lc_be_time -lrtc -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(DIETLIBC_IMPLEMENTATION) $(GCCLIB) $(DIETLIBC_IMPLEMENTATION) $(MALLOC_BACKEND) -ll4rm -ldm_mem -ldm_generic -lc_be_time -lc_be_mmap -lc_be_mmap_util -lc_be_l4env_start_stop -lgeneric_ts -lc_be_syslog -lc_be_file-table -ldiet_be_simple_sleep -ll4vfs_common_io -ll4vfs_basic_io -ll4vfs_connection -ll4vfs_basic_name_server -ll4vfs_name_server -ll4vfs_name_space_provider -ll4vfs_extendable
tiny	-nostdlib $(GCCLDNOSTDLIB) -ldiet_c -ldietlibc_support -ldiet_be_minimal_io -ldiet_be_l4_start_stop -ldiet_be_sigma0_mem -ldiet_c -ll4util -lsigma0 -ldiet_c $(GCCLIB) -ldiet_c -ll4sys
sigma0	-nostdlib $(GCCLDNOSTDLIB) -ldiet_c -ldietlibc_support -ldiet_be_minimal_io -ldiet_be_l4_start_stop -ldiet_be_sigma0_mem -ldiet_c -llogserver -lnames -lsigma0 -ll4util_root -ll4util $(ROOTLIB) -ldiet_c $(GCCLIB) -ldiet_c -ll4sys
libc	-nostdlib $(GCCLDNOSTDLIB) -ldiet_c $(LIBCBACKEND_LIB) $(GCCLIB) -ldiet_c
host	-
l4linux	-ldm_generic -ldm_mem -lnames -ll4util_root -ll4util $(ROOTLIB) -lloaderif -ll4env -ll4env_err -lslab -ll4sys

Table 9: Standard libc-libraries of *dietlibc* depending on the mode.

Mode	**`$LIBCLIBS`**
l4env	`-nostdlib $(GCCLDNOSTDLIB) $(UCLIBC_IMPLEMENTATION) -luclibc_support $(MALLOC_BACKEND) -lc_be_mmap -lc_be_mmap_util -lc_be_l4env_start_stop -lc_be_minimal_log_io -luc_be_simple_sleep -ll4env -llogserver_capsule -ll4rm -ldm_generic -ldm_mem -lthread -lgeneric_ts $(UCLIBC_IMPLEMENTATION) $(GCCLIB) $(UCLIBC_IMPLEMENTATION) -ll4sys`
l4env_minimal	`-nostdlib $(GCCLDNOSTDLIB) $(MALLOC_BACKEND) $(UCLIBC_IMPLEMENTATION) -lc_be_l4env_start_stop -lgeneric_ts -lc_be_minimal_log_io -lc_be_mmap_util $(MALLOC_BACKEND) -lc_be_mmap -lc_be_mmap_util -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(UCLIBC_IMPLEMENTATION) $(GCCLIB) $(UCLIBC_IMPLEMENTATION) -lc_be_l4env_start_stop -luclibc_support -ll4sys`
l4env_base	`-nostdlib $(GCCLDNOSTDLIB) -u printf -lc_be_io.o $(MALLOC_BACKEND) -lc_be_time -lrtc -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(UCLIBC_IMPLEMENTATION) $(GCCLIB) $(UCLIBC_IMPLEMENTATION) $(MALLOC_BACKEND) -ll4rm -ldm_mem -ldm_generic -lc_be_time -lc_be_mmap -lc_be_mmap_util -lc_be_l4env_start_stop -lgeneric_ts -lc_be_syslog -lc_be_file-table -luc_be_simple_sleep -ll4vfs_common_io -ll4vfs_basic_io -ll4vfs_connection -ll4vfs_basic_name_server -ll4vfs_name_server -ll4vfs_name_space_provider -ll4vfs_extendable -ll4sys`
tiny	`-nostdlib $(GCCLDNOSTDLIB) -luc_c -luc_be_minimal_io -luc_be_l4_start_stop -luc_be_sigma0_mem -luc_c -luclibc_support -ll4util -lsigma0 -luc_c $(GCCLIB) -luc_c -ll4sys`
sigma0	`-nostdlib $(GCCLDNOSTDLIB) -luc_c -luc_be_minimal_io -luc_be_l4_start_stop -luc_be_sigma0_mem -luc_c -luclibc_support -llogserver -lnames -lsigma0 -ll4util_root -ll4util $(ROOTLIB) -luc_c $(GCCLIB) -luc_c -ll4sys`
libc	`-nostdlib $(GCCLDNOSTDLIB) -ldiet_c $(LIBCBACKEND_LIB) $(GCCLIB) -ldiet_c`
host	-
l4linux	`-ldm_generic -ldm_mem -lnames -ll4util_root -ll4util $(ROOTLIB) -lloaderif -ll4env -ll4env_err -lslab -ll4sys`

Table 10: Standard libc-libraries of *uclibc* depending on the mode.

Mode	`$L4LIBS`
l4env	`-static -lgeneric_ts -ll4env -ll4rm -ldm_generic -ldm_mem -lthread -lsemaphore -llogserver_capsule -lnames -ll4util_root -ll4util -lsigma0 $(ROOTLIB) -ll4env -ll4env_err -ll4rm -ldm_generic -ldm_mem -lthread -lslab -ll4sys`
l4env_minimal	`-static -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(ROOTLIB) -ll4sys`
l4env_base	`-static -ll4rm -ldm_mem -ldm_generic -lthread -lsemaphore -ll4env -ll4env_err -lslab -llogserver_capsule -ll4rm -lthread -ldm_generic -lnames -ll4util_root -ll4util -lsigma0 $(ROOTLIB) -ll4sys`
tiny	`-static -lmain -ll4util -ll4sys`
sigma0	`-static -lmain -lnames -llogserver -ll4util_root -ll4util -ll4sys`
libc	–
host	–
l4linux	–

Table 11: L4 libraries to be linked depending on the mode.

Mode	`$CRT0`	`$LDSCRIPT`
l4env	`crt0.o`	`main_stat.ld`
l4env_minimal	`crt0.o`	`main_stat.ld`
l4env_base	`crt0.o`	`main_stat.ld`
tiny	`crt0.o`	`main_stat.ld`
sigma0	`crt0.o`	`main_stat.ld`
libc	`crt0.o`	`main_stat.ld`
host	–	–
l4linux	–	–

Table 12: Startup code (CRT0) and linker scripts depending on the mode.

define the variables `LIBCINCDIR_arch_mode`, `LIBCLIBDIR_arch_mode`, `LIBCLIBS_arch_mode`, `L4LIBS_arch_mode`, `CRT0_arch_mode`, `CRTN_arch_mode` and `LDSCRIPT_arch_mode`.

Also, you can set `CARCHFLAGS_<arch>_<cpu>` to specify cpu-specific compiler options, e.g. `CARCHFLAGS_x86_p3=-march=p3` if -march=p3 is the compiler option to generate p3-optimized code.

7 Relocated binaries and the STATIC file

This is the classical approach of building an executable binary: Link it starting at a specific address. However, we offer some magic here too. To determine the final start-address of your binary, `prog.mk` uses a kind of database. It determines the start-addresses of each binary using a central place, the file `$(L4DIR)/pkg/STATIC`. Each line of this file contains the start-address and the name of a binary. `prog.mk` looks up the name of the binary in this file, and if it is found, the corresponding start-address is used. If the name of the binary is not found in STATIC, the start-address specified in `DEFAULT_RELOC++` is used.

8 The Role-Files

Abbreviations

Within the next sections, the following abbreviations are used in the description of role-file parameters:

XXX+	The parameter `XXX` can be specified in multiple forms. To set the parameter for all targets (defined in the parameter `$(TARGET)` then), just specify **`XXX`**. To set the parameter for a special target `target` out of `$(TARGET)`, specify **`XXX_target`**.
XXX++	The parameter `XXX` can be specified in multiple forms. To set the parameter for all targets (defined in the parameter `$(TARGET)` then) and all systems (defined in the parameter `$(SYSTEM)` then), just specify **`XXX`**. To set the parameter for a special target `target` for all given systems, specify **`XXX_target`**. To set the parameter for a special system `sys` for all given targets, specify **`XXX_sys`**. To set the parameter for a special target `target` for a special system `sys`, specify **`XXX_target_sys`**.

8.1 Common Role-File Targets and Parameters

8.1.1 Make-Targets

All role-files support the following phony targets, which can be build using `make <target>`.

config:: Runs the menu-driven configuration utility. See Section 3 for details.

txtconfig:: Runs the configuration utility. For directories, a recursive invocation is done. See Section 3 for details.

oldconfig:: (Re)creates a configuration header file based on a prior configuration or defaults. For directories, a recursive invocation is done. See Section 3 for details.

all: The default target. It depends on the role, what exactly is done.

install:: Installs the generated files into the installation-tree `$(DROPS_STDDIR)`.

clean:: Deletes all intermediate files generated during compilation. This does *not* delete the generated targets such as binaries or libs.

cleanall:: Deletes all generated and backup files. Invokes **clean::**.

help:: Displays a short overview of the make-targets available with this role.

FORCE: This target is not intended to build, but to depend on. If something depends on FORCE, it is built.

8.1.2 Required Parameters

All role-files require the following variables to be set:

`L4DIR`	The base directory of the L4 tree.
`PKGDIR`	The base directory of the package.

8.1.3 Optional Parameters

Most role files use the following variables:

`SUBDIRS`	used by `prog.mk`, `lib.mk`, `idl.mk`, `doc.mk`
	A list of subdirectories that will be treated specially: For each element of the list, a target will be created. If this target it called, the **all**-rule in the according directory will be called. Furthermore, **scrub**, **clean** and **cleanall** are made recursively for the specified directories. Default: unset.
`DEPEND_VERBOSE`	used by all roles
	If set to '`@`', no commands for dependency-generation will be shown. Default: unset.
`DROPS_STDDIR`	used by all roles
	The base-directory of the install-tree. Default: `/home/drops`
`HAVE_LDSO`	used by `prog.mk`, `lib.mk`, `idl.mk`
	If enabled (nonempty), dependency generation uses descent algorithms (see Section 5 for details). Only disable this option if your host system does not provide ld.so, set this option empty then. Default: y (enabled).
`SHOWMESSAGES`	used by all roles
	If set to "true" or "y", a short textual description for every compilation step is printed. Default: true.
`VERBOSE`	used by all roles
	If set to '`@`', the commands invoked by make will not be shown. Default: unset.

8.1.4 Provided Variables

The following variables are *provided* by the role files.

`PKGNAME`	The last part of the directory name specified in `$(PKGDIR)`.

8.2 Role-File prog.mk

8.2.1 Purpose

The purpose of the prog role is to build executable binaries for different systems. The executable binaries are the targets.

8.2.2 Make-Targets

all: Phony target. Build all targets and install them into the local install-tree then. The local install-tree is `$(L4DIR)/bin`. To install, `strip --strip-unneeded` is used.

relink: Phony target. Relink and reinstall the targets, in the case your dependencies are somehow wrong.

8.2.3 Provided Variables

The following variables are provided by the prog.mk role-file:

`GCCINCDIR`	The path where the gcc-specific include-files, such as `stdarg.h`. Never (really, NEVER) try to include a `stdarg.h` from a C library!).
`GCCLIB`	The name of the compiler's companion library (libgcc).

8.2.4 Required Parameters

TARGET	The list of the targets.
Or:	
`TARGET_<system>`	The list of the targets for a special system. Specify the system(s) in `SYSTEMS`.

8.2.5 Optional Parameters

`ASFLAGS++`	standard Makefile variable, used additionally for compiling assembler files. Default: empty.
`BID_STRIP_PROGS`	If set to "y", strip binaries on installation into local or global bin-directories using `$(STRIP) --strip-unneeded`. Default: empty.
`BUILD_PROFILE++`	If nonempty, additional profile-versions of the targets are built. Therefore, each generated .o-file has an according .pr.o-file, which is compiled using the options `-pg -DPROFILE`. Each target is accompanied by an additional file with the suffix .pr. These additional files contain the profile code and are linked against profile libraries. See Section 8.3 on how to build profile libraries. Default: empty
`CLIENTIDL++`	IDL client source files. The .c and .cc-files compiled from the given IDLs are added to SRC_C++ and SRC_CC++ respectively. Default: unset.
`CFLAGS++`	standard Makefile variable, used additionally for the compiler. Default: adds $(OPTS) $(WARNINGS).

next page ...

. . . continued from last page

CPPFLAGS++	standard Makefile variables, used additionally for preprocessor. Default: adds $(DEFINES) -I{$(PRIVATE_INCDIR) $(IDL_PATH) $(GC-CINCDIR) $(L4INCDIR)} $(LIBCINCDIR). $(IDL_PATH) is added if either $(CLIENTIDL) or $(SERVERIDL) are set. With $(MODE)=host, $(L4INCDIR) is not added.
CRT0	CRT0 object to link as fist object to the binary. Default depends on the selected mode.
CRTN	CRTN object to link as last object to the binary. Default depends on the selected mode.
CXXFLAGS++	standard Makefile variable, used additionally for compiling C++ files. Default: adds $(OPTS) $(WARNINGS).
DEBUG	If set, adds `-DDEBUG` to $(DEFINES). See there for details.
DEFAULT_RELOC++	The default link-addresses for the targets. Depending on the system, a relocation address needs to be specified. This address is used, if the default-reloc mechanism (see 7) does not deliver an address. Default: unset.
DEFINES++	Container for additional defines for the preprocessor. Default: Adds $(ARCH), $(CPU) and $(L4API) as defines. See Section 6.3 for details. If $(DEBUG) is set, `-DDEBUG` is added.
IDL_PATH++	The path to IDL-files. This path is used to find the client and server idl-files. Default: `$(PKG_DIR)/idl/OBJ-$(SYSTEM)`.
IDL_PKGDIR	A list of package directories where IDL files are used from. This will eventually supercede `$(IDL_PATH)`. Default: `$(PKGDIR)`.
IDL_TYPE+	The type of the IDL-files. This type is used to determine the compiler to use. See Section 8.4 for details. Default: `dice`. Note: The '+' refers to the different IDL-files, not to the values of `$(TARGET)`.
INSTALL_TARGET+	The list of the targets that will be installed automatically or on `make install`. Default: The targets and the targets specific to the current system. Also see the `NOTARGETSTOINSTALL` parameter.
INSTALLDIR_BIN	The directory in the global install tree to install a binary to. Default: `$(DROPS_STDDIR)/bin`
INSTALLDIR_BIN_LOCAL	The directory in the local install tree to install a binary to. Default: `$(L4DIR)/bin`
INSTALLFILE_BIN	The command to install a binary to the global install tree. Default depends on `$(BID_STRIP_PROGS)`.
INSTALLFILE_BIN_LOCAL	The command to install a binary to the local install tree. Default depends on `$(BID_STRIP_PROGS)`.
KEEP_ON_CLEAN	A list of files that should not be removed on a “make clean”. Default: empty.

next page . . .

... continued from last page

`L4INCDIR++`	System-dependent list of include directories. Default:
	`$(L4DIR)/include/$(ARCH)/$(L4API)`
	`$(DROPS_STDDIR)/include/$(ARCH)/$(L4API)`
	`$(L4DIR)/include/$(ARCH)`
	`$(DROPS_STDDIR)/include/$(ARCH)`
	`$(L4DIR)/include`
	`$(DROPS_STDDIR)/include`

`L4LIBDIR++`	System-dependent list of library directories. Default:
	`$(L4DIR)/lib/$(ARCH)_$(CPU)/$(L4API)`
	`$(DROPS_STDDIR)/lib/$(ARCH)_$(CPU)/$(L4API)`
	`$(L4DIR)/lib/$(ARCH)_$(CPU)`
	`$(DROPS_STDDIR)/lib/$(ARCH)_$(CPU)`
	`$(L4DIR)/lib`
	`$(DROPS_STDDIR)/lib`

`L4LIBS`	Libraries and flags for the binary. Depends on the selected mode.
`LDFLAGS++`	standard Makefile variables, used as additional flags for the linker. Default for mode host: -L{$(PRIVATE_LIBDIR) $(LIBCLIBDIR)} $(LDSCRIPT) $(LIBS) $(L4LIBS) $(LIBCLIBS). Default otherwise: -L{$(PRIVATE_LIBDIR) $(L4LIBDIR) $(LIBCLIBDIR)} $(LDSCRIPT) $(LIBS) $(L4LIBS) $(LIBCLIBS).
`LDSCRIPT++`	The name of the standard linker script used to link the binary. Default depends on the mode.
`LIBCINCDIR`	Include path and flags to the libc includes. Default: Determined by the mode, points to dietlibc or uClibc normally.
`LIBCLIBDIR`	Library path and flags to libc. Default: Determined by the mode, points to dietlibc or uClibc normally.
`LIBCLIBS`	Libraries and flags for libc. Default: Determined by the mode, points to dietlibc or uClibc with support libraries normally.
LIBS++	libraries and objects to be linked to the targets. Contains additional PATHS (with -L) and libs (with -l). Default: unset.
MODE++	Specify, which target system a target has to be built for (see Section 6.5). Default: l4env
`NOTARGETSTOINSTALL`	If nonempty, the binaries will not be installed, neither in the local $(L4DIR)/bin nor in the global $(DROPS_STDDIR)/bin. Default: empty
`OPTS`	Optimization switches for the compiler. Default: `-g -O2` plus arch-dependent switches
PRIVATE_INCDIR++	Additional list of include-directories. Will be prefixed with -I. Default: unset
`PRIVATE_LIBDIR++`	Additional list of library-directories. Will be prefixed with -L. Default: unset
`SERVERIDL++`	IDL server source files. The .c and .cc-files compiled from the given IDLs are added to SRC_C++ and SRC_CC++ respectively. Default: unset.
SRC_{C,CC,S}++	The list of source {.c, .cc, .S}-files which have to be compiled and linked to the targets. Default: unset.

next page ...

... continued from last page

SYSTEMS	The systems the targets have to be build for. If multiple systems are given, each system is build into a separate directory. Default: x86-l4v2.
WARNINGS++	Warning switches for the compiler. Default: `-Wall -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations`

8.2.6 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

TARGET = ping
SRC_C  = ping.c

include $(L4DIR)/mk/prog.mk
```

8.2.7 C++ programs

If you do not need exception support, you can use the l4env mode, which comes with a much smaller libc. You have to define some callbacks then. Please see the exec server as an example. You should specify `CXXFLAGS` = -fno-exceptions -fno-rtti.

8.2.8 Implementation status

`BUILD_PROFILE` To make it work, additional libraries (libl4sys.pr) must be linked, which is not the case now. It is not clear, if all l4-libraries should be exchanged by profiling variants or not.

`MODE` is currently interpreted as MODE, not as MODE++.

8.3 Role-File lib.mk

8.3.1 Purpose

The purpose of the lib role is to build static and shared libraries for different systems. The libraries are the targets.

To build shared libraries, simply end the `TARGET` library name on .s.so. The additional ".s" later allows to explicitly specify linking against a shared library by specifying the different library base name to the linker. The use of `BUILD_SHARED` is deprecated.

To build libraries that can be used to build shared libraries later, set the `BUILD_PIC` option to nonempty. The resulting libraries can be linked into shared libraries then. They end on .p.a.

Both shared and pic-libraries use objects compiled with the PIC-option enabled. The corresponding object-files end on .s.o after compilation.

To build libraries with profiling support, set the `BUILD_PROFILE` to nonempty. Each generated .o-file will have an accompanying .pr.o-file, which is compiled using the options `-pg -DPROFILE`. Each target-lib is accompanied by an additional lib with the suffix .pr.a. These additional libs contain the profile code. See Section 8.2 on how to build profile executables.

If all `BUILD_SHARED`, `BUILD_PIC` and `BUILD_PROFILE` options are set, you end with not less than six libraries, and 4 object-files per source-file. Suffixes: .o, .pr.o, .s.o, .pr.s.o, .a, .pr.a, .p.a, .s.so, .pr.p.a, .pr.s.so.

Note, that prior to building libraries, the library files are deleted. This prevents old (obsolete) object-files to be within the libraries.

8.3.2 Make-Targets

all: Phony target. Build all libraries and install them into the local install-tree then. The local install-tree is `$(L4DIR)/lib`. To install, `ln -s` is used.

relink: Phony target. Relink and reinstall the libraries, in the case your dependencies are somehow wrong.

XXX_p.a: This target is a library containing position-independent code only. The according object-files end on _p.o and are compiled using appropriate compiler-calls analogously to the other .o-Files.

XXX_s.a: This target is a shared library. This difference in filenames allows to explicitly distinguish between linking of static versus shared libraries.

8.3.3 Provided Variables

The following variables are provided by the prog.mk role-file:

`GCCINCDIR`	The path where the gcc-specific include-files, such as `stdarg.h`. Never (really, NEVER) try to include a `stdarg.h` from a C library!).
`GCCLIB`	The name of the compiler's companion library (libgcc).

8.3.4 Required Parameters

TARGET	The list of the target libraries.
Or:	
`TARGET_<system>`	The list of the target libraries for special system. Specify the system(s) in `SYSTEMS`.

8.3.5 Optional Parameters

`ASFLAGS++`	standard Makefile variable, used additionally for compiling assembler files. Default: empty.
`BUILD_PROFILE++`	If nonempty, additional profile-versions of the target-libs are built. These libraries end on .pr.a. Default: empty
`BUILD_SHARED++`	If nonempty, the target-libs are built as shared libraries too. These libs end on .s.so. Default: empty
`BUILD_PIC++`	If nonempty, target-libs containing position-independent code (PIC) are built too. These libs end on .p.a. Default: empty
`CFLAGS++`	standard Makefile variable, used additionally for the compiler. Default: adds $(OPTS) $(WARNINGS).
`CLIENTIDL++`	IDL client source files. The .c and .cc-files compiled from the given IDLs are added to SRC_C++ and SRC_CC++ respectively. Default: unset.
`CPPFLAGS++`	standard Makefile variables, used additionally for preprocessor. Default: adds $(DEFINES) -I{$(PRIVATE_INCDIR) $(IDL_PATH) $(GCCINCDIR) $(L4INCDIR)} $(LIBCINCDIR). $(IDL_PATH) is added if either $(CLIENTIDL) or $(SERVERIDL) are set. With $(MODE)=host, $(L4INCDIR) is not added.
`CXXFLAGS++`	standard Makefile variable, used additionally for compiling C++ files. Default: adds $(OPTS) $(WARNINGS).
`DEBUG`	If set, adds `-DDEBUG` to $(DEFINES). See there for details.
`DEFINES++`	Container for additional defines for the preprocessor. Default: Adds $(ARCH), $(CPU) and $(L4API) as defines. See Section 6.3 for details.
`IDL_PATH++`	The path to IDL-files. This path is used to find the client and server idl-files. Default: `$(PKG_DIR)/idl/OBJ-$(SYSTEM)`.
`IDL_PKGDIR`	A list of package directories where IDL files are used from. This will eventually supercede `$(IDL_PATH)`. Default: `$(PKGDIR)`.
`IDL_TYPE+`	The type of the IDL-files. This type is used to determine the compiler to use. See Section 8.4 for details. Default: `dice`. Note: The '+' refers to the different IDL-files, not to the values of `$(TARGET)`.
`INSTALL_TARGET+`	The list of the targets that will be installed automatically or on `make install`. Default: The targets and the targets specific to the current system. Also see the `NOTARGETSTOINSTALL` parameter.
`INSTALLFILE_LIB`	The command to install a library to the global install tree. Default: `$(INSTALL) $(1) $(INSTALLDIR_LIB)/`

next page ...

. . . continued from last page

`INSTALLFILE_LIB_LOCAL`	The command to install a library to the local install tree. Default: `$(LN) -sf $(call absfilename, $(1)) $(INSTALLDIR_LIB_LOCAL)/$(1)`
`INSTALLDIR_LIB`	The directory in the global install tree to install a library to. Default: `$(DROPS_STDDIR)/lib`
`INSTALLDIR_LIB_LOCAL`	The directory in the local install tree to install a library to. Default: `$(L4DIR)/lib`
`CFLAGS++, CXXFLAGS++, ASFLAGS++, CPPFLAGS++`	standard Makefile variables, used as additional flags for the compiler, c++-compiler, assembler and preprocessor. Default: unset.
`KEEP_ON_CLEAN`	A list of files that should not be removed on a “make clean”. Default: empty.

`L4INCDIR++`	System-dependent list of include directories. Default:
	`$(L4DIR)/include/$(ARCH)_$(CPU)/$(L4API)`
	`$(DROPS_STDDIR)/include/$(ARCH)_$(CPU)/$(L4API)`
	`$(L4DIR)/include/$(ARCH)_$(CPU)`
	`$(DROPS_STDDIR)/include/$(ARCH)_$(CPU)`
	`$(L4DIR)/include`
	`$(DROPS_STDDIR)/include`

`L4LIBDIR++`	System-dependent list of library directories. Default:
	`$(L4DIR)/lib/$(ARCH)_$(CPU)/$(L4API)`
	`$(DROPS_STDDIR)/lib/$(ARCH)_$(CPU)/$(L4API)`
	`$(L4DIR)/lib/$(ARCH)_$(CPU)`
	`$(DROPS_STDDIR)/lib/$(ARCH)_$(CPU)`
	`$(L4DIR)/lib`
	`$(DROPS_STDDIR)/lib`

LIBS++	objects to be archived to the target libraries. Contains additional PATHS (with -L). Default: unset.
MODE++	specify, which target system a target has to be built for (see Section 6.5). Default: l4env
`NOTARGETSTOINSTALL`	If nonempty, the libraries will not be installed, neither in the local $(L4DIR)/lib nor in the global $(DROPS_STDDIR)/lib. Default: empty
`OPTS`	Optimization switches for the compiler. Default: `-g -O2` plus arch-dependent switches
PRIVATE_INCDIR++	Additional list of include-directories. Will be prefixed with -I. Default: unset
`PRIVATE_LIBDIR++`	Additional list of library-directories. Will be prefixed with -L. Default: unset
`SERVERIDL++`	IDL server source files. The .c and .cc-files compiled from the given IDLs are added to SRC_C++ and SRC_CC++ respectively. Default: unset.
SRC_{C,CC,S}++	The list of source {.c, .cc, .S}-files which have to be compiled and archived into the libraries. Default: unset.
`SYSTEMS`	The systems the target libraries have to be build for. If multiple systems are given, each system is built into a separate directory. Default: x86-l4v2.

next page . . .

. . . continued from last page

`WARNINGS++`	Warning switches for the compiler. Default: `-Wall -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations`

8.3.6 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

TARGET = libping.a
SRC_C  = ping.c
include $(L4DIR)/mk/lib.mk
```

8.3.7 C++ programs

If you do not need exception support, you can use the l4env mode, which comes with a much smaller libc. You have to define some callbacks then. Please see the exec server as an example.

8.4 Role-File idl.mk

8.4.1 Purpose

The purpose of the IDL role is to translate IDL definition files into appropriate .c- and .h-files. The .c- and .h-files are the targets. To compile the generated files, use directories with the prog or lib role, and import the generated files using IDL-related parameters. There are no special targets.

8.4.2 Make-Targets

all:: Phony target. Generate the .c- and .h-files.

install:: Phony target. Install into `$(DROPS_STDDIR)`.

8.4.3 Required Parameters

IDL	The list of the idl definition files.

8.4.4 Optional Parameters

`DEFINES++`	Container for additional defines for the preprocessor. Default: Adds $(ARCH), $(CPU) and $(L4API) as defines. See Section 6.3 for details.
IDL_EXPORT_SKELETON	Mask of IDL files, whose generated server-include files should be installed into `$(INSTALLDIR_IDL)`. Default: empty, nothing to install.
IDL_EXPORT_STUB	Mask of IDL files, whose generated client-include files should be installed into `$(INSTALLDIR_IDL)`. Default: %, install the client-headers of all IDL files in $(IDL).
`IDL_FLAGS`	Additional flags to be passed to the idl-compiler. adds `-P{ $(DEFINES) -I{ $(PRIVATE_INCDIR) $(GCCINCDIR) $(L4INCDIR)} $(LIBCINCDIR) }`. Also adds flags to use the appropriate back- and front-ends according to `$(L4API)` and `$(IDL_TYPE)`. With $(MODE)=host, $(L4INCDIR) is not added.
IDL_TYPE+	This specifies what kind of idl the idl-files are. This determines the compiler to use. Supported values are **`dice`** The idl(s) is in a modified DCE-style. Use dice to compile the idl-file(s). **`corba`** The idl(s) is in corba style. Use dice to compile the idl-file(s). Default is 'dice'.
`INSTALLDIR_IDL`	The directory to install the created include files to. Default: `$(PKGDIR)/include/ARCH-$(ARCH)/L4API-$(L4API)`

next page ...

. . . continued from last page

`L4INCDIR++`	System-dependent list of include directories. Default: `$(L4DIR)/include/$(ARCH)/$(L4API)` `$(DROPS_STDDIR)/include/$(ARCH)/$(L4API)` `$(L4DIR)/include/$(ARCH)` `$(DROPS_STDDIR)/include/$(ARCH)` `$(L4DIR)/include` `$(DROPS_STDDIR)/include`
`LIBCINCDIR`	Include path and flags to the libc includes. Default: Determined by the mode, points to dietlibc or uClibc normally.
`MODE++`	specify, which target system a target has to be built for (see Section 6.5). Default: host
`PRIVATE_INCDIR+`	List of directories to prepend to the include-directive of the idl-compiler.
`SYSTEMS`	The systems the .c and .h files have to be build for. Each system is build into a separate directory. Default: x86-l4v2.

8.4.5 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

IDL = ping.idl

include $(L4DIR)/mk/idl.mk
```

8.4.6 C++ programs

If the `IDL_FLAGS` parameter specifies the C++ languag mapping using -BmCPP, then C++ files are generated. They end on .cc and .hh and are added to SRC_CC variables of lib and prog make roles.

8.5 Role-File include.mk

8.5.1 Purpose

The purpose of the include role is to install all .h and .i under INCSRC_DIR into the local install-tree or into the drops install-tree. This role facilitates two goals with respect to managing include-files:

1. The path where the include-files are found is entirely defined by the package name and the target system of the include-files.
2. Once written, the make file need not be modified, even when new include files are added or old include files are removed from the package.

The target system of an include file is determined by putting it into a specific subdirectory. The role-file will then install it accordingly. By default, INCSRC_DIR points is set to the source directory. To install generated header files, set INCSRC_DIR to OBJ_DIR. There is currently no mechanism to exclude header files.

When generating header files, add them to the headers target (which all depends on).

The following table defines the subdirectories depending on the target systems.

if the include-file is designated for ...	put it into ... or subdirectories thereof
any system	./
CPU architecture `<arch>`	`ARCH-<arch>/`
CPU architecture `<arch>` and L4API `<api>`	`ARCH-<arch>/L4API-<api>/`

The installation paths are determined according to the following table:

Designated system	**Local and global installation paths**
no system dependency	`$(L4DIR)/include/l4/$(PKGNAME)/`, `$(DROPS_STDDIR)/include/l4/$(PKGNAME)/`
CPU architecture `<arch>`	`$(L4DIR)/include/<arch>/l4/$(PKGNAME)/`, `$(DROPS_STDDIR)/include/<arch>/l4/$(PKGNAME)/`
CPU architecture `<arch>` and L4API `<api>`	`$(L4DIR)/include/<arch>/<api>/l4/$(PKGNAME)/`, `$(DROPS_STDDIR)/include/<arch>/<api>/l4/$(PKGNAME)`

8.5.2 Make-Targets

all:: Phony target. Install all header-files found in this directory-tree into the local install-tree.

install:: Phony target. Install all header-files found in this directory-tree into the global install-tree.

next page ...

... continued from last page

8.5.3 Optional Parameters

`INSTALLDIR_INC`	The directory in the global install tree to install an include to. Default: `$(DROPS_STDDIR)/include/`
`INSTALLDIR_INC_LOCAL`	The directory in the local install tree to install an include to. Default: `$(L4DIR)/include/`
`INSTALL_INC_PREFIX`	The prefix of installed files. Think twice before changing this, as it pollutes the name space of include files. Default: `l4/$(PKGNAME)`
`PKGNAME`	The name of the package. Default value is determined from the name of the package directory.
`TARGET`	The list of file to install on **all::** or **install::**. Default: all `*.h`-files found in the subdirectory tree.

8.5.4 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..
include $(L4DIR)/mk/include.mk
```

8.5.5 How to use the Include Role

Note, that all include files found in the subdirectory tree are subject to installation.

Generally, you would insert an include file with no system dependency, hence directly into the `include/` directory. Admittedly, this may result in a wrong compilation if you use this include file with a system target it was not intended for. But, as no library is available for that system target, the linker should catch this wrong package use later.

If you realize later that you need different versions of the include file to support different systems, you can move the include files into appropriate subdirectories. The dependencies of the make system ensure that no old include files will be used: Files in system-dependent subdirectories will be found first when the preprocessor is looking for include files. Moreover, the symbolic links in the local installation directory `$(L4DIR)/include/` will become invalid, so that an undesired use of them is avoided safely.

8.6 Role-File subdir.mk

8.6.1 Purpose

The purpose of the subdir role is to be a container for other directories and recursively build them. Therefore, most make-targets are recursively forwarded to subdirectories.

8.6.2 Make-Targets

install:: Phony target. Builds the targets in the given subdirs in `$(TARGET)` using `make -C`. Calls `make install` in the subdirs then.

all::, clean::, cleanall::, install::, oldconfig::, scrub::, txtconfig:: Phony targets. Call `make {all, clean, cleanall, install, oldconfig, txtconfig, scrub}` in the subdirectories.

config:: Phony target. Does nothing.

8.6.3 Required Parameters

`PKGDIR`	If ".", additional dependencies are defined. If Makefiles in `idl` and `include` exist, `include` depends on `idl`. If Makefiles in `include` and `lib` resp. `server` exist, `lib` resp. `server` depend on `include`.

8.6.4 Optional Parameters

`TARGET`	List of subdirectories to build. The build-order is unspecified unless other make dependencies apply. Default: a list of selected subdirectories that contain a Makefile. If `PKGDIR` is "." the default selection contains `idl, include, src, lib, server, examples, doc`. If `PKGDIR` is not ".", `include` is not in the default list.
`MKFLAGS+`	This value is passed to make when building a subdir. Default: empty.

8.6.5 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

TARGET = example1 easy complex
include $(L4DIR)/mk/subdir.mk
```

8.7 Role-File doc.mk

Use the doc role to create doxygen documentation related to a package or to compile latex files.

8.7.1 Doxygen support

The doc role allows to generate multiple documentations using doxygen, e.g. an API documentation, an internal documentation and a documentation of the usage of your server. For each documentation, pick a unique name beginning with the name of the package you are writing the documentation for. Save the doxygen files using these names, with the `.cfg` extension added. Ensure all configurations build into a subdir with the name of that documentation. List the configuration files within the `SRC_DOX_*` variables. A `make all` will create the documentation and optionally install the html files into `$(L4DIR)/doc/html/`. With the make-file located in this directory, you can build an index to the documentation of all DROPS packages then.

The dependency mechanism detects all changes in your source files.

Doxygen support allows to group your documentation into one of four groups: user guides, reference manuals, internal documentation and the rest. This classification is used when generating the index page in `$(L4DIR)/doc/html/`.

8.7.2 Latex support

The doc role provides rules to compile LaTeX files into .ps and .pdf, and to convert xfig images to eps. List the .tex files you want to be compiled in the `SRC_TEX` variable. The first file in the `$(SRC_TEX)` is the *primary tex document*. The compiled version of the primary tex document is easily showed by calling the "showdvi" or the "showps" target. To rebuild the primary tex document, call "`make dvi`" or "`make ps`". A successful compilation triggers a redraw within the appropriate viewer.

8.7.3 Make-Targets

all:: Phony target. Generate documentation and install it locally.

install:: Phony target. Generate documentation and install it globally.

dvi::, ps::, pdf:: Phony targets. Compile the primary tex document into dvi and ps, respectively.

showdvi::, showps::, showpdf:: Phony targets. Invoke a viewer on the compiled version of the primary tex document.

clean:: Delete intermediate compilation files.

cleanall:: Delete all generated files.

next page ...

. . . continued from last page

8.7.4 Optional Parameters

`DOXY_FLAGS+`	Semicolon-separated list of additional flags for doxygen. Should be set in Makecond.local to enable/disable warnings and status output. Default: empty. Note: All semicolons are removed, no escaping!
`SRC_DOX`	List of *doxygen* configuration files that will be compiled using doxygen. The files should end on `.cfg` and should build into a directory with the same name and the `.cfg` removed.
`SRC_DOX_GUIDE`	As $(SRC_DOX). Documentation will be classified as user guide.
`SRC_DOX_INT`	As $(SRC_DOX). Documentation will be classified as internal.
`SRC_DOX_REF`	As $(SRC_DOX). Documentation will be classified as reference manual.
`SRC_TEX`	List of LaTeX files that will be compiled into postscript files.

8.7.5 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

SRC_DOX_REF = dm_phys.cfg
include $(L4DIR)/mk/doc.mk
```

8.8 Role-File runux.mk

8.8.1 Purpose

Use the ptest (runux) role to run 'automated' tests on your package.

This role runs Fiasco-UX with the specified server and test-application and compares the generated output with expected output. If the output does not match a message is displayed and the build process is interrupted.

You may also trigger this test run in the `l4/pkg` directory by issuing `make ptest`.

8.8.2 Make-Targets

all:: Phony target. Runs Fiasco UX with given arguments and compares output to expected output.

ptest:: Phony target. Same as all.

genexp:: Phony target. Generate the expected output.

plainrun:: Phony target. Simply runs Fiasco UX with given arguments. No watchdog is installed, no output filtering applied, and no output comparison.

clean:: Delete intermediate compilation files.

cleanall:: Delete all generated files (also the expected output file if generated by make).

8.8.3 Required Parameters

`TEST_SERVER`	The name of the binary to be run as the server to be tested. If this binary name contains no directory elements ('/'), then the binary is assumed to be in the architecture's and L4 API's binary directory. $(L4DIR)/bin/$(ARCH)/$(L4API) If there are directory elements, is used as is. If empty, a warning is issued and make terminates successfully.

8.8.4 Optional Parameters

`TEST_CLIENT`	The name of the binary to be run as the application testing the server. Same naming rules apply as with TEST_SERVER.
`FIASCOUX_<api>`	The Fiasco-UX binary to use for the specified $(L4API) Default: $(L4DIR)/kernel/fiasco/build-ux-$(L4API) (with `l4` removed from $(L4API)).
`BASE_SERVERS`	The binary names of the base servers required to run the test server. Default: `log names dm_phys` If this variable does not contain directory parts ('/'), then it is prefixed with the binary directory of user-land executables. Otherwise it is taken as is. The binaries are then prefixed with `-l` to run as modules of Fiasco-UX.

next page ...

. . . continued from last page

EXPECTED_OUT	The name of the file to contain the expected output of the test-run. This file is compared to the output generated in the test-run using `diff`. Default: `expected.txt`. If not specified, the file can be generated by running make with the target `genexp`. This will also create another file to indicate that the expected output file has been generated. It will then be deleted when running `make cleanall`. Note: Because the test is run in an OBJ directory, this should be relative to the OBJ directory (e.g., `../expected.txt`).
TIMEOUT	The timeout the watchdog waits for Fiasco-UX before killing it. This is the safe-guard for test-applications which do not shut down Fiasco-UX themselves or if Fiasco-UX jumps into the kernel-debugger. The time is specified in seconds. Default: 10.
DEBUG_PERL	If the Perl-script used to run Fiasco-UX with a watchdog is doing something wrong, you may set this variable to 1 to get some debugging output. Default: 0.
SYSTEMS	Specify the systems to run this test for. This determines the $(ARCH) and $(API) variable used to find the user-land binaries in the `l4/bin` directory.
EXPECT_FAIL	If set will accept failure of output comparison. If output matches expected output, an error will be issued. This option has no effect in generate mode.
FILTER_KEEP_ALL	If set, the output of Fiasco UX will not be filtered.
COMPARE_CMD	If set, this command is run instead of the default command. This command should only generate output if there is an error. If no output is generated by the command, success is assumed. In case of an error the last 1000 lines of the output are printed to stderr. Default: diff -u $(EXPECTED_OUT) $(TMP_OUT)
NO_FBUF_DEV	If set, Fiasco UX will not open a window to display graphical output. This is only relevant if your setup does graphical output and you would like to test it in batched mode (no DISPLAY).
USE_SYMBOLS	If set to 'y', Fiasco UX will load its Symbols. This helps when debugging. Default: 'y'.
USE_LINES	If set to 'y', Fiasco UX will load its Lines. This helps when debugging. Default: 'y'.

8.8.5 Example

```
PKGDIR = ..
L4DIR ?= $(PKGDIR)/../..

EXPECTED = ../expected.txt

TEST_SERVER = dice_hello_server
TEST_CLIENT = dice_hello_client

include $(L4DIR)/mk/runux.mk
```

The $(COMPARE_CMD) can be used to apply own filter rules as well. An example that sorts the output depending on the LOG tag is the following:

```
COMPARE_CMD = "cat $(TMP_OUT)|sort -t '|' -k1,1 -s >$(TMP_OUT) ; diff -u
$(EXPECTED_OUT) $(TMP_OUT)"
```

9 Reserved variables

When using BID, the following Make-variables are reserved besides the one already mentioned ('`*`' denotes 'any string'):

`BID*`, `CARCHFLAGS_*`, `currentdothfile`, `DEPEND_EXTEND_CMD`, `DEPEND_EXTEND_FUNC`, `DEPEND_FLAG`, `DEPS`, `DEPSNULL`, `DEPSVAR`, `DIR_FROM_SUB`, `FILTER_SYSTEM`, `IDL_INCLUDES`, `INSTALL_TARGET`, `INSTALL_TARGET_LOCAL`, `INSTALL_TARGET_GLOBAL`, `LIBDEPS`, `MAKEDEP`, `OSYSTEMS`, `ROLE`, `SYSTEM_TO_ARCH`, `SYSTEM_TO_CPU`, `SYSTEM_TO_L4API`, `TARGET_SYSTEMS`

A References

[1] L4Check. Documentation at `http://os.inf.tu-dresden.de/~l4check/`.

[2] DICE *User's Manual*, 2006. Available at `http://os.inf.tu-dresden.de/dice/manual.pdf`.

[3] Han-Wen Nienhuys. *gendep library*. Based on code from `http://www.xs4all.nl/~hanwen/public/software/README.html`.

B Configuration Language

This section describes the Configuration Language to be used to write the configuration definition files of DROPS packages.

As the drops configuration tool is inherited from the Linux Kernel Configuration Tool, the Configuration Language is the same. Hence, the text is that of the Linux-Kernel, version 2.4.18.

B.1 Introduction

Config Language is not 'bash'.

This document describes Config Language, the Linux Kernel Configuration Language. config.in and Config.in files are written in this language.

Although it looks, and usually acts, like a subset of the 'sh' language, Config Language has a restricted syntax and different semantics.

Here is a basic guideline for Config Language programming: use only the programming idioms that you see in existing Config.in files. People often draw on their shell programming experience to invent idioms that look reasonable to shell programmers, but silently fail in Config Language.

Config Language is not 'bash'.

B.2 Interpreters

Four different configuration programs read Config Language:

scripts/Configure	make config, make oldconfig
scripts/Menuconfig	make menuconfig
scripts/tkparse	make xconfig
mconfig	ftp.kernel.org/pub/linux/kernel/people/hch/mconfig/

'Configure' is a bash script which interprets Config.in files by sourcing them. Some of the Config Language commands are native bash commands; simple bash functions implement the rest of the commands.

'Menuconfig' is another bash script. It scans the input files with a small awk script, builds a shell function for each menu, sources the shell functions that it builds, and then executes the shell functions in a user-driven order. Menuconfig uses 'lxdialog', a back-end utility program, to perform actual screen output. 'lxdialog' is a C program which uses curses.

'scripts/tkparse' is a C program with an ad hoc parser which translates a Config Language script to a huge TCL/TK program. 'make xconfig' then hands this TCL/TK program to 'wish', which executes it.

'mconfig' is the next generation of Config Language interpreters. It is a C program with a bison parser which translates a Config Language script into an internal syntax tree and then hands the syntax tree to one of several user-interface front ends.

B.3 Statements

A Config Language script is a list of statements. There are 21 simple statements; an 'if' statement; menu blocks; and a 'source' statement.

A '\' at the end of a line marks a line continuation.

'#' usually introduces a comment, which continues to the end of the line. Lines of the form '# ... is not set', however, are not comments. They are semantically meaningful, and all four config interpreters implement this meaning.

Newlines are significant. You may not substitute semicolons for newlines. The 'if' statement does accept a semicolon in one position; you may use a newline in that position instead.

Here are the basic grammar elements.

- A */prompt/* is a single-quoted string or a double-quoted string. If the word is double-quoted, it may not have any $ substitutions.
- A */word/* is a single unquoted word, a single-quoted string, or a double-quoted string. If the word is unquoted or double quoted, then $-substitution will be performed on the word.
- A */symbol/* is a single unquoted word. A symbol must have a name of the form `CONFIG_*`. scripts/mkdep.c relies on this convention in order to generate dependencies on individual `CONFIG_*` symbols instead of making one massive dependency on include/linux/autoconf.h.
- A */dep/* is a dependency. Syntactically, it is a */word/*. At run time, a */dep/* must evaluate to "y", "m", "n", or "".
- An */expr/* is a bash-like expression using the operators '=', '!=', '-a', '-o', and '!'.

Here are all the statements:

- Text statements:
  `mainmenu_name` */prompt/*
  `comment` */prompt/*
  `text` */prompt/*
- Ask statements:
  `bool` */prompt/* */symbol/*
  `hex` */prompt/* */symbol/* */word/*
  `int` */prompt/* */symbol/* */word/*
  `string` */prompt/* */symbol/* */word/*
  `tristate` */prompt/* */symbol/*
- Define statements:
  `define_bool` */symbol/* */word/*
  `define_hex` */symbol/* */word/*
  `define_int` */symbol/* */word/*
  `define_string` */symbol/* */word/*
  `define_tristate` */symbol/* */word/*

- Dependent statements:

  `dep_bool` */prompt/ /symbol/ /dep/* ...
  `dep_mbool` */prompt/ /symbol/ /dep/* ...
  `dep_hex` */prompt/ /symbol/ /word/ /dep/* ...
  `dep_int` */prompt/ /symbol/ /word/ /dep/* ...
  `dep_string` */prompt/ /symbol/ /word/ /dep/* ...
  `dep_tristate` */prompt/ /symbol/ /dep/* ...

- Unset statement:

  `unset` */symbol/* ...

- Choice statements:

  `choice` */prompt/ /word/ /word/*
  `nchoice` */prompt/ /symbol/ /prompt/ /symbol/* ...

- If statements:

  `if` [*/expr/*] ; `then`
      /statement/
      ...
  `fi`
  `if` [*/expr/*] ; then
      /statement/
      ...
  `else`
      /statement/
      ...
  `fi`

- Menu block:

  `mainmenu_option next_comment`
  `comment` */prompt/*
      /statement/
      ...
  `endmenu`

- Source statement:

  `source` */word/*

B.3.1 `mainmenu_name` */prompt/*

This verb is a lot less important than it looks. It specifies the top-level name of this Config Language file.

Configure:	ignores this line
Menuconfig:	ignores this line
Xconfig:	uses */prompt/* for the label window.
mconfig:	ignores this line (mconfig does a better job without it).

Example:

```
# arch/sparc/config.in
mainmenu_name "Linux/SPARC Kernel Configuration"
```

B.3.2 `comment` */prompt/*

This verb displays its prompt to the user during the configuration process and also echoes it to the output files during output. Note that the prompt, like all prompts, is a quoted string with no dollar substitution.

The `comment` verb is not a Config Language comment. It causes the user interface to display text, and it causes output to appear in the output files.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/net/Config.in
comment 'CCP compressors for PPP are only built as modules.'
```

B.3.3 `text` */prompt/*

This verb displays the prompt to the user with no adornment whatsoever. It does not echo the prompt to the output file. mconfig uses this verb internally for its help facility.

Configure:	not implemented
Menuconfig:	not implemented
Xconfig:	not implemented
mconfig:	implemented

Example:

```
# mconfig internal help text
text 'Here are all the mconfig command line options.'
```

B.3.4 `bool` */prompt/* */symbol/*

This verb displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*. The legal input values are ”n” and ”y”.

Note that the `bool` verb does not have a default value. People keep trying to write Config Language scripts with a default value for `bool`, but **all** of the existing language interpreters discard additional values. Feel free to submit a multi-interpreter patch to linux-kbuild if you want to implement this as an enhancement.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# arch/i386/config.in
bool 'Symmetric multi-processing support' CONFIG_SMP
```

B.3.5 `hex` */prompt/ /symbol/ /word/*

This verb displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*. Any hexadecimal number is a legal input value. */word/* is the default value.

The `hex` verb does not accept range parameters.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/sound/Config.in
hex 'I/O base for SB Check from manual of the card' \
  CONFIG_SB_BASE 220
```

B.3.6 `int` */prompt/ /symbol/ /word/*

This verb displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*. */word/* is the default value. Any decimal number is a legal input value.

The `int` verb does not accept range parameters.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/char/Config.in
int 'Maximum number of Unix98 PTYs in use (0-2048)' \
      CONFIG_UNIX98_PTY_COUNT 256
```

B.3.7 `string` */prompt/ /symbol/ /word/*

This verb displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*. */word/* is the default value. Legal input values are any ASCII string, except for the characters '"' and '\'. Configure will trap an input string of "?" to display help.

The default value is mandatory.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/sound/Config.in
string '  Full pathname of DSPxxx.LD firmware file' \
      CONFIG_PSS_BOOT_FILE /etc/sound/dsp001.ld
```

B.3.8 `tristate` */prompt/ /symbol/*

This verb displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*. Legal values are ”n”, ”m”, or ”y”.

The value ”m” stands for ”module”; it indicates that */symbol/* should be built as a kernel module. The value ”m” is legal only if the symbol CONFIG_MODULES currently has the value ”y”.

The `tristate` verb does not have a default value.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# fs/Config.in
tristate 'NFS filesystem support' CONFIG_NFS_FS
```

B.3.9 `define_bool` */symbol/ /word/*

This verb the value of */word/* to */symbol/*. Legal values are ”n” or ”y”.

For compatibility reasons, the value of ”m” is also legal, because it will be a while before `define_tristate` is implemented everywhere.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# arch/alpha/config.in
if [ "$CONFIG_ALPHA_GENERIC" = "y" ]
then
  define_bool CONFIG_PCI y
  define_bool CONFIG_ALPHA_NEED_ROUNDING_EMULATION y
fi
```

B.3.10 `define_hex` */symbol/ /word/*

This verb assigns the value of */word/* to */symbol/*. Any hexadecimal number is a legal value.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# Not from the corpus
bool 'Specify custom serial port' CONFIG_SERIAL_PORT_CUSTOM
if [ "$CONFIG_SERIAL_PORT_CUSTOM" = "y" ]; then
  hex 'Serial port number' CONFIG_SERIAL_PORT
else
  define_hex CONFIG_SERIAL_PORT 0x3F8
fi
```

B.3.11 `define_int` */symbol/ /word/*

This verb assigns */symbol/* the value */word/*. Any decimal number is a legal value.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/char/ftape/Config.in
define_int CONFIG_FT_ALPHA_CLOCK 0
```

B.3.12 `define_string` */symbol/ /word/*

This verb assigns the value of */word/* to */symbol/*. Legal input values are any ASCII string, except for the characters '"' and '\'.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# Not from the corpus
define_string CONFIG_VERSION "2.2.0"
```

B.3.13 `define_tristate` */symbol/ /word/*

This verb assigns the value of */word/* to */symbol/*. Legal input values are "n", "m", and "y".

As soon as this verb is implemented in all interpreters, please use it instead of `define_bool` to define tristate values. This aids in static type checking.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/video/Config.in
if [ "$CONFIG_FB_AMIGA" = "y" ]; then
   define_tristate CONFIG_FBCON_AFB y
   define_tristate CONFIG_FBCON_ILBM y
  else
   if [ "$CONFIG_FB_AMIGA" = "m" ]; then
      define_tristate CONFIG_FBCON_AFB m
      define_tristate CONFIG_FBCON_ILBM m
   fi
fi
```

B.3.14 `dep_bool` */prompt/ /symbol/ /dep/* ...

This verb evaluates all of the dependencies in the dependency list. Any dependency which has a value of ”y” does not restrict the input range. Any dependency which has an empty value is ignored. Any dependency which has a value of ”n”, or which has some other value, (like ”m”) restricts the input range to ”n”. Quoting dependencies is not allowed. Using dependencies with an empty value possible is not recommended. See also `dep_mbool` below.

If the input range is restricted to the single choice ”n”, `dep_bool` silently assigns ”n” to */symbol/*. If the input range has more than one choice, dep_bool displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*.

Configure:	implemented
Menuconfig:	implemented
XConfig:	implemented
mconfig:	implemented

Example:

```
# drivers/net/Config.in
dep_bool 'Aironet 4500/4800 PCI support 'CONFIG_AIRONET4500_PCI \
  $CONFIG_PCI
```

Known bugs:

- Xconfig does not write ”# foo is not set” to .config (as well as ”#undef foo” to autoconf.h) if command is disabled by its dependencies.

B.3.15 `dep_mbool` */prompt/ /symbol/ /dep/* ...

This verb evaluates all of the dependencies in the dependency list. Any dependency which has a value of ”y” or ”m” does not restrict the input range. Any dependency which has an empty value is ignored. Any dependency which has a value of ”n”, or which has some other value, restricts the input range to ”n”. Quoting dependencies is not allowed. Using dependencies with an empty value possible is not recommended.

If the input range is restricted to the single choice ”n”, dep_bool silently assigns ”n” to */symbol/*. If the input range has more than one choice, `dep_bool` displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*.

Notice that the only difference between `dep_bool` and `dep_mbool` is in the way of treating the ”m” value as a dependency.

Configure:	implemented
Menuconfig:	implemented
XConfig:	implemented
mconfig:	implemented

Example:

```
# Not from the corpus
dep_mbool 'Packet socket: mmapped IO' CONFIG_PACKET_MMAP \
  $CONFIG_PACKET
```

Known bugs:

- Xconfig does not write ”# foo is not set” to .config (as well as ”#undef foo” to autoconf.h) if command is disabled by its dependencies.

B.3.16 `dep_hex` */prompt/ /symbol/ /word/ /dep/* **...**
 dep_int */prompt/ /symbol/ /word/ /dep/* **...**
 dep_string */prompt/ /symbol/ /word/ /dep/* **...**

I am still thinking about the semantics of these verbs.

Configure:	not implemented
Menuconfig:	not implemented
XConfig:	not implemented
mconfig:	not implemented

B.3.17 `dep_tristate` */prompt/ /symbol/ /dep/* **...**

This verb evaluates all of the dependencies in the dependency list. Any dependency which has a value of ”y” does not restrict the input range. Any dependency which has a value of ”m” restricts the input range to ”m” or ”n”. Any dependency which has an empty value is ignored. Any dependency which has a value of ”n”, or which has some other value, restricts the input range to ”n”. Quoting dependencies is not allowed. Using dependencies with an empty value possible is not recommended.

If the input range is restricted to the single choice ”n”, dep_tristate silently assigns ”n” to */symbol/*. If the input range has more than one choice, `dep_tristate` displays */prompt/* to the user, accepts a value from the user, and assigns that value to */symbol/*.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# drivers/char/Config.in
dep_tristate 'Parallel printer support' CONFIG_PRINTER $CONFIG_PARPORT
```

Known bugs:

- Xconfig does not write ”# foo is not set” to .config (as well as ”#undef foo” to autoconf.h) if command is disabled by its dependencies.

B.3.18 `unset` */symbol/* **...**

This verb assigns the value ”” to */symbol/*, but does not cause */symbol/* to appear in the output. The existence of this verb is a hack; it covers up deeper problems with variable semantics in a random-execution language.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented (with bugs)
mconfig:	implemented

Example:

```
# arch/mips/config.in
unset CONFIG_PCI
unset CONFIG_MIPS_JAZZ
unset CONFIG_VIDEO_G364
```

B.3.19 `choice` */prompt/ /word/ /word/*

This verb implements a choice list or ”radio button list” selection. It displays */prompt/* to the user, as well as a group of sub-prompts which have corresponding symbols.

When the user selects a value, the choice verb sets the corresponding symbol to ”y” and sets all the other symbols in the choice list to ”n”.

The second argument is a single-quoted or double-quoted word that describes a series of sub-prompts and symbol names. The interpreter breaks up the word at white space boundaries into a list of sub-words. The first sub-word is the first prompt; the second sub-word is the first symbol. The third sub-word is the second prompt; the fourth sub-word is the second symbol. And so on, for all the sub-words.

The third word is a literal word. Its value must be a unique abbreviation for exactly one of the prompts. The symbol corresponding to this prompt is the default enabled symbol.

Note that because of the syntax of the `choice` verb, the sub-prompts may not have spaces in them.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

Example:

```
# arch/i386/config.in
choice '  PCI access mode'          \
  "BIOS      CONFIG_PCI_GOBIOS      \
   Direct    CONFIG_PCI_GODIRECT    \
   Any       CONFIG_PCI_GOANY"      Any
```

B.3.20 nchoice */prompt/ /symbol/ /prompt/ /symbol/...*

This verb has the same semantics as the choice verb, but with a sensible syntax.

The first */prompt/* is the master prompt for the entire choice list.

The first */symbol/* is the default symbol to enable (notice that this is a symbol, not a unique prompt abbreviation).

The subsequent */prompt/* and */symbol/* pairs are the prompts and symbols for the choice list.

Configure:	not implemented
Menuconfig:	not implemented
XConfig:	not implemented
mconfig:	implemented

B.3.21 `if [` */expr/* `] ; then`

This is a conditional statement, with an optional `else` clause. You may substitute a newline for the semicolon if you choose.

/expr/ may contain the following atoms and operators. Note that, unlike shell, you must use double quotes around every atom.

/atom/: "…" a literal "$…" a variable

/expr/: */atom/* = */atom/* true if atoms have identical value */atom/* != */atom/* true if atoms have different value

/expr/: */expr/* -o */expr/* true if either expression is true */expr/* -a */expr/* true if both expressions are true ! */expr/* true if expression is not true

Note that a naked */atom/* is not a valid */expr/*. If you try to use it as such:

```
# Do not do this.
if [ "$CONFIG_EXPERIMENTAL" ]; then
  bool 'Bogus experimental feature' CONFIG_BOGUS
fi
```

…then you will be surprised, because `CONFIG_EXPERIMENTAL` never has a value of the empty string! It is always "y" or "n", and both of these are treated as true (non-empty) by the bash-based interpreters Configure and Menuconfig.

Configure:	implemented
Menuconfig:	implemented
XConfig:	implemented, with bugs
mconfig:	implemented

Xconfig has some known bugs, and probably some unknown bugs too:

- literals with an empty "" value are not properly handled.

B.3.22 `mainmenu_option next_comment`

This verb introduces a new menu. The next statement must have a comment verb. The */prompt/* of that comment verb becomes the title of the menu. (I have no idea why the original designer didn't create a 'menu …' verb).

Statements outside the scope of any menu are in the implicit top menu. The title of the top menu comes from a variety of sources, depending on the interpreter.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

B.3.23 `endmenu`

This verb closes the scope of a menu.

Configure:	implemented
Menuconfig:	implemented
Xconfig:	implemented
mconfig:	implemented

B.3.24 `source` */word/*

This verb interprets the literal */word/* as a filename, and interpolates the contents of that file. The word must be a single unquoted literal word.

Some interpreters interpret this verb at run time; some interpreters interpret it at parse time.

Inclusion is textual inclusion, like the C preprocessor #include facility. The `source` verb does not imply a submenu or any kind of block nesting.

Configure:	implemented (run time)
Menuconfig:	implemented (parse time)
Xconfig:	implemented (parse time)
mconfig:	implemented (parse time)