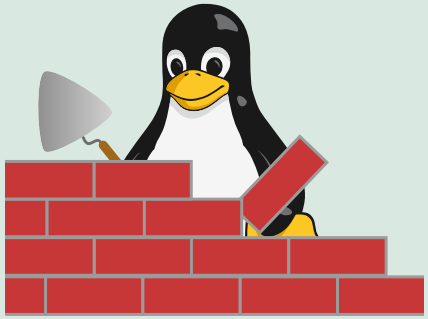
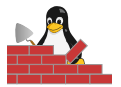
LINES – Linux for Embedded Systems
Lecture 4 - How to add my own application to Buildroot?



The Tux image by lewing@isc.tamu.edu Larry Ewing and The GIMP

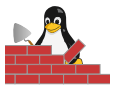
Wojciech M. Zabołotny, Ph.D., D.Sc.

Faculty of Electronics and Information Technology WUT, room. 225,
wzab@ise.pw.edu.pl (please add [LINES] in the subject)



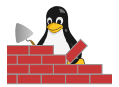
Adding of my own application

- Buildroot in version 2021.02 offers almost 2500 packages.
- Even though this is really a huge collection, finally we face a situation, where we need to add a new application. It can be your own application, or the one published by someone else.
- The optimal solution of course is preparing the new Buildroot package with the needed application
- However, on the initial stage of the development we may need to compile and test our application independently from the whole image. In this case, compilation “outside the BR tree” may be the best solution.

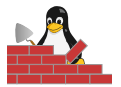


Adding of my own application (2)

- Methods of adding of our application depend on its implementation
- The easiest case is the application written in a script language (Python, Lua, Perl)
- In this case all you need to do is:
  - Build the BR image with the appropriate interpreter and libraries
  - Put the scripts with your application in the appropriate directory (how to do it ???)
- If the application is written in a compiled language (C or C++) then we must to compile it... (in fact cross-compile it)

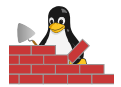


An example – Python application



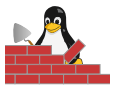
Python application – possible problems

- Sometimes the Python application may use a library written in C or C++
- Such a library must be compiled just as an application written in C or C++.
- Of course we may create a separate BR package for such library
- There is a special support for Python modules distributed by PyPI. It is described in BR Manual, section “Generating a python-package from a PyPI repository”.



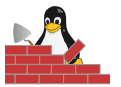
Compilation of our own application

- If we have sources of our application and we want to compile it, we should read point Using the generated toolchain outside Buildroot of the BR documentation
- If the application uses a typical “makefile”, we can:
  - Set the environment variable `PATH` so that it contains directory with our cross-compiler
  - Call “make” with appropriate values of the following environment variables: `ARCH` and `CROSS_COMPILE` (e.g. for the emulated virt 64 board: `ARCH=aarch64`, `CROSS_COMPILE=aarch64-none-linux-gnu-`)
- Of course the application must be transferred to our target machine. You can:
  - Put it into the „overlay” directory, rebuild the BR image and reboot target system.
  - You may transfer it using the scp (this approach allows to test consecutive version of the application without restarting the target system)
  - You may transfer it using the wget (from the mini HTTP server started with `python3 -m http.server`. In that case, remember to restore the “execute” right of the downloaded binary.



An example: worms application

- Lets analyse compilation of an application for BR, using the simple worms game as an example. The sources are available on the website: http://www.paulgriffiths.net/program/c/curworm.php
- We place the source files: helper.c helper.h main.c worms.c worms.h in a local directory
- We create the simple Makefile...



Compilation of worms

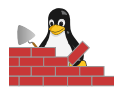
- Makefile:

```
CC=$(CROSS_COMPILE)gcc
OBJS := helper.o main.o worms.o
worms: $(OBJS)
        $(CC) -o worms $(CFLAGS) $(LDFLAGS) $(OBJS) -l ncurses
$(OBJS) : %.o : %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

- You must do: `export BRPATH=/home/dev/BR/buildroot-2021.2`
- Then call the building script:

```
(
export PATH=$BRPATH/output/host/usr/bin:$PATH
make ARCH=arm CROSS_COMPILE=\
   arm-none-linux-gnu- worms
)
```

- To verify: Use the `app_worms/worms_src_independent` directory from the `ex_w4` archive.

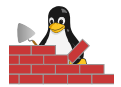


Compiling of our own kernel module

- If we want to compile independently our own kernel module, in its directory we must create the “Kbuild” file with the list of files to be compiled:
  `obj-m := my_module1.o`
- Create also the compiling script:

```
#!/bin/bash
# Set the variables according to your Buildroot configuration
BRPATH=/home/dev/buildroot-2020.02
CROSS_ARCH=aarch64 #BR2_ARCH
CROSS_PREFIX=aarch64-none-linux-gnu- #BR2_TOOLCHAIN_EXTERNAL_PREFIX
KERNEL_SRC=$BRPATH/output/build/linux-
(
  PATH=$BRPATH/output/host/usr/bin:$PATH
  echo $KPATH
  echo $PWD
  make ARCH=$CROSS_ARCH CROSS_COMPILE=$CROSS_PREFIX -C $KERNEL_SRC \
     modules M=$PWD
)
```

- To verify: Use the `kernel_modules/mod1_independent` directory from the `ex_w4` archive.



The Makefile file for a kernel module

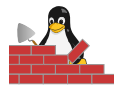
- Makefile:

```
ifneq ($(KERNELRELEASE),)
 obj-m := my_module1.o
else
   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
   PWD := $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean

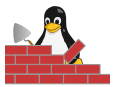
endif
```



The Makefile file for a kernel module

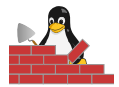
- A special version of the Makefile, where the module is created from multiple files and there is a name conflict:

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
 obj-m += my_fir.o
 my_fir-objs := ./src/my_fir.o ./src/fir_calc.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 PWD := $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
modules_install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
endif
```



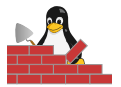
Preparation of the package with our application

- Buildroot offers different macros for creation of the package, depending on the compilation tools we use:
  - generic-package for software using non-standard tools or simple makefiles
  - autotools-package (if we use autotools)
  - cmake-package (if we use cmake)
- There are also special macros for applications written in script languages
  - python-package (for Python applications)
  - luarocks-package (for Lua applications, conforming to LuaRocks standard)
- For each macro we have a “tutorial” section (for “quick start”) and a “reference” section with detailed description.



Preparation of the package – basics

- Let's assume, that our package is named "mypkg"
- We need to create the directory `BR/package/mypkg` with two files:
  - `Config.in` – describing the configuration of our package (formatting rules)
  - `mypkg.mk` – describing how the sources should be downloaded and compiled
  - `mypkg.hash` – an optional file, with hashes of the source archive,
- Please note that the name of the package is used in the names of directory, file, and inside the created files (in the latter case sometimes in upper case and sometimes in lower case).

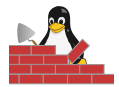


The minimal Config.in file

- Config.in:

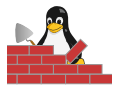
```
config BR2_PACKAGE_MYPKG
        bool "mypkg"
        help
            Here we should explain function of
            the mpypkg package.
            All lines must be indented by TAB
            and two spaces.
            After an empty line we add the URL
            of the associated project website.

            http://foosoftware.org/libfoo/
```



The Config.in file - dependencies

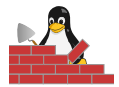
- The `Config.in` file describes dependencies:
  - `select` – describes that the package depends on certain libraries
    Warning! It causes selection of the library, but does not check if the prerequisites are met. Results may be different to predict... It should be explained in the description.
  - `depends on` – Used if the user should be aware that selection of the package requires previous selection of certain options, or selection of other packages that significantly affect the size or the performance of the system.
    Warning! You should add the conditional comment explaining why the package is not displayed until the “depends on” condition is met.



The dependency section of jack2 Config.in file

- Config.in:

```
config BR2_PACKAGE_JACK2
        bool "jack2"
        depends on BR2_TOOLCHAIN_HAS_THREADS # alsa-lib
        depends on BR2_USE_MMU # fork()
        depends on BR2_INSTALL_LIBSTDCPP
        depends on !BR2_STATIC_LIBS
        depends on BR2_TOOLCHAIN_HAS_SYNC_4
        select BR2_PACKAGE_LIBSAMPLERATE
        select BR2_PACKAGE_LIBSNDFILE
        select BR2_PACKAGE_ALSA_LIB
        select BR2_PACKAGE_ALSA_LIB_HWDEP
        select BR2_PACKAGE_ALSA_LIB_SEQ
        select BR2_PACKAGE_ALSA_LIB_RAWMIDI
        # Ensure we get at least one:
        select BR2_PACKAGE_JACK2_LEGACY if !BR2_PACKAGE_JACK2_DBUS
```



The essential part jack2.mk file

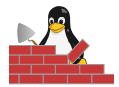
- jack2.mk:

```
JACK2_VERSION = 1.9.17
JACK2_SITE = $(call github,jackaudio,jack2,v$(JACK2_VERSION))
JACK2_LICENSE = GPL-2.0+ (jack server), LGPL-2.1+ (jack library)
JACK2_LICENSE_FILES = COPYING
JACK2_CPE_ID_VENDOR = jackaudio
JACK2_DEPENDENCIES = libsamplerate libsndfile alsa-lib
JACK2_INSTALL_STAGING = YES

JACK2_CONF_OPTS = --alsa

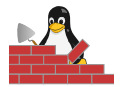
ifeq ($(BR2_PACKAGE_OPUS),y)
JACK2_DEPENDENCIES += opus
endif

ifeq ($(BR2_PACKAGE_READLINE),y)
JACK2_DEPENDENCIES += readline
endif
[...]
$(eval $(waf-package))
```



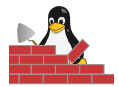
Preparation of my own package – downloading of sources

- We must inform the Buildroot from where it should download the application.
- We describe separately: the localization and the downloading method.
- The localisation (`MYPKG_SITE`): URL (`http`, `ftp`, `scp`, `git`, `bzr`, `hg`) or local path,
- The downloading method (`MYPKG_SITE_METHOD`): (`wget`, `scp`, `svn`, `cvs`, `git`, `hg`, `bzr`, `file`, `local`).



The relationship between the localization and the retrieval method

Method	Localization
wget	http://, https://, ftp://
scp	scp://
svn	svn://, http://
cvs	cvs://
git	git:// , http://, https://
hg	http:// , https://
bzr	bzr://
file	Local path to the (compressed) tar archive
local	Local path to the directory containing the sources



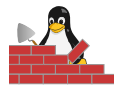
The package with worms application

- The Config.in file - the 1st version:

```
config BR2_PACKAGE_WORMS
        bool "worms"
        depends on BR2_PACKAGE_NCURSES
        help
            Game worms using ncurses

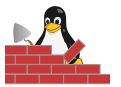
            http://www.paulgriffiths.net/program/c/curworm.php
```

- This approach is used in the `app_worms/worms_mak` directory in the `ex_w4` archive. It should be used with `worms_src_mak` directory.
- Use directories `worms_mak_comment` with `worms_src_mak` or `worms_mak_select` with `worms_src_mak` to see how the dependencies work.



The makefile for worms package

- ```
  OBJS := helper.o main.o worms.o
  worms: $(OBJS)
  	$(CC) -o worms $(CFLAGS) $(LDFLAGS) $(OBJS) -l ncurses
  $(OBJS) : %.o : %.c
  	$(CC) -c $(CFLAGS) $< -o $@
  ```
- Please don't copy makefiles from slides! Most likely the PDF conversion damages the "TAB" characters that must be used to indent commands, and Makefiles won't work. Such copied Makefiles require manual correction before use!


  ```

The “mk” for worms package

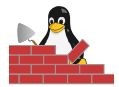
- The file worms.mk – the 1st version prepared for the standard Makefile

```
################################################################
# worms
################################################################
WORMS_VERSION = 1.0
WORMS_SITE = $(TOPDIR)/../app_worms/worms_src_mak
WORMS_SITE_METHOD = local
WORMS_DEPENDENCIES = ncurses

define WORMS_BUILD_CMDS
   $(MAKE) $(TARGET_CONFIGURE_OPTS) worms -C $(@D)
endef

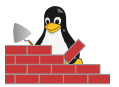
define WORMS_INSTALL_TARGET_CMDS
   $(INSTALL) -D -m 0755 $(@D)/worms $(TARGET_DIR)/usr/bin
endef

WORMS_LICENSE = Proprietary
$(eval $(generic-package))
```



Adding our own package

- How to make our package visible?
  - We must edit the file: `BRPATH/package/Config.in` adding in the appropriate section the line: `source "package/worms/Config.in"`
- Why sometimes our package is still not displayed?
  - It depends on NCURSES
    - We can add the appropriate comment.
    - We can change the dependency type to `selects`, (fortunately in this case it does not generate the additional dependencies, as it was in jack2)



Package with the worms application

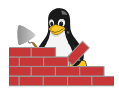
- The file worms.mk – cmake based version

```
################################################################################
#
# worms
#
################################################################################

WORMS_VERSION = 1.0
WORMS_SITE = $(TOPDIR)/../app_worms/worms_src_cmake
WORMS_SITE_METHOD = local
WORMS_LICENSE = Proprietary
WORMS_DEPENDENCIES = ncurses

$(eval $(cmake-package))
```

- This approach is used in the `app_worms/worms_cmake` directory in the `ex_w4` archive. It should be used with `worms_src_cmake` directory.



The files CMakeLists.txt

- In the main directory of sources

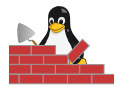
```
# The name of our project is "WORMS". CMakeLists files in this
# project can refer to the root source directory of the project
# as ${WORMS_SOURCE_DIR} and to the root binary directory
# of the project as ${WORMS_BINARY_DIR}.
cmake_minimum_required (VERSION 2.6)
project (WORMS)

add_subdirectory(src)
```

- In the `src` subdirectory

```
add_executable (worms main.c worms.c helper.c)
install (TARGETS worms DESTINATION bin)

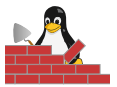
# Link the executable to the ncurses library.
target_link_libraries (worms ncurses)
```



How to avoid littering the official source repository during program development?

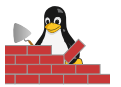
- If the sources of our program are downloaded from a public repository, we may not want to push there all the changes made during the development of the program.
- In this case, the option `package_OVERRIDE_SRCDIR` may help. We usually place it in the `local.mk` file in BR.
- The `package_OVERRIDE_SRCDIR` option can be set for several packages developed in parallel:

```
PKG1_OVERRIDE_SRCDIR = /opt/src/pkg1
PKG2_OVERRIDE_SRCDIR = /opt/src/pkg2
```



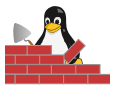
How to publish our package?

- If we have successfully packaged an interesting application and we suppose that many users may be interested in it, we can publish our package.
- To do it, we must have a clone of the Buildroot git repository.
- The procedure of publishing of our Buildroot corrections/extensions is described in the BR documentation.



What with packages, which extend the kernel?

- In the latest versions of the BR the documentation describes the kernel-module helper.
- Good examples may be the following packages:
  - owl-linux
  - lttng-modules



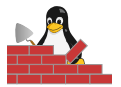
The Config.in file for a kernel module

```
config BR2_PACKAGE_WZMOD1_MODULES
        bool "wzmod1-modules"
        depends on BR2_LINUX_KERNEL
        help
            my first module

            http://www.nowhere

comment "wzmod1 needs a Linux kernel to be built"
        depends on !BR2_LINUX_KERNEL
```

- This approach is used in the `kernel_modules/wzmod1(or 2)` directory in the `ex_w4` archive. It should be used with the `wzmod1(or 2)-modules` directory.

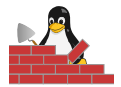


The wzmod1.mk file for a kernel module

```
##############################################################
#
# WZMOD1-modules
#
##############################################################

WZMOD1_MODULES_VERSION = 1.0
WZMOD1_MODULES_SITE    = $(TOPDIR)/../kernel_modules/wzmod1
WZMOD1_MODULES_SITE_METHOD = local
WZMOD1_MODULES_LICENSE = LGPLv2.1/GPLv2

$(eval $(kernel-module))
$(eval $(generic-package))
```

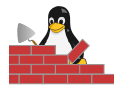


The Makefile for a kernel module

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
 obj-m := my_module1.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 PWD := $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```



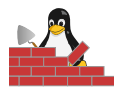
How to avoid repeated BR modifications?

- If we want to use our packages with different versions of Buildroot, or if we want to avoid
  - repeated recreating of our packages subdirectories in the `BRPATH/package`
  - modifications of the `BRPATH/package/Config.in`
- we may make use of the BR2_EXTERNAL functionality.
- To verify: run `make menuconfig BR2_EXTERNAL=path/to/br2_ext` (of course you should replace the path with your own path to the directory `br2_ext` taken from the `ex_w4` archive).
  The `BR2_EXTERNAL` should be used only once. It remains valid until you "connect" another external directories.

  You may "connect" multiple external directories simultaenously. For example:

  `make menuconfig BR2_EXTERNAL=path/to/br2_ext1:path/to/br2_ext2`

  (please see the `br2_ext_split` directory in `ex_w4`).

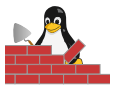


How to debug our application?

- Using the `gdb` debugger
- Compilation of the `gdb` debugger:
  - `Target packages → Debugging,... → gdb → gdbserver`
  - `Toolchain → Build cross gdb for the host`
  - We should also select:

    `Build options → build packages with debugging symbols`
- On the target system we start our application via the `gdb server`:
  - `gdbserver host:7654 MPATH/myapp`
- On the workstation:
  - `BRPATH/output/host/usr/bin/aarch64-none-linux-gnu-gdb MPATH/myapp`

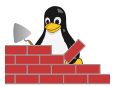
    (The gdb path depends on the toolchain)
- and then in the debugger session:
  - `set sysroot BRPATH/output/staging` *(here are the programs and libraries with symbols)*
  - `(gdb) target remote xxx.yyy.zzz.vvv:7654`



Work with the debugger cd.

- The gdb debugger may also use the additional serial port instead of the network:
  `gdbserver /dev/ttyAMA0 MPATH/myapp`
- We can also used a TUI- or GUI-equipped debugger instead of "raw" gdb:
  `BR2_PACKAGE_HOST_GDB` in `make menuconfig` or

```
ddd --debugger \
    BRPATH/output/host/usr/bin/arm-none-linux-gnu-gdb MPATH/myapp
```



Thank you for your attention!