AVR32004: AVR32 AP7 How to add a software package to Buildroot

Features

- **Adding a package to Buildroot**
- **Adding entries in the configuration system**
- **Creating a Buildroot package Makefile**
- **Creating an own package**

1 Introduction

This application note describes how a new “user-space” software package can be added to the Buildroot build system. In addition it documents how new software must be modified to fit into the Buildroot system. In order to understand and adopt the instructions in this document the reader should possess basic knowledge of following:

- The Buildroot build system
- Makefiles
- Kconfig

  Additional information to all the above mentioned tools is available in the references chapter at the end of this document.

32-bit AVR® Microcontrollers

Application Note

Rev. 32082B-AVR32-11/08

1 Adding a package to Buildroot

This part of the application note describes how user-space software can be added to the Buildroot system.

The Buildroot build system provided by Atmel already contains an example package which can be used as a template for your packages. This example package is located in the directory package/dummy. All further steps in this application note are based on this example package. If you are implementing your own package just follow the described steps and replace every “dummy” or “DUMMY” with your package name.

The first thing to do is to create a new directory in the package directory for the new software. After that you should copy the example files contained in package/dummy/ to your new directory. Now you have two new files in your new directory. These are Config.in and dummy.mk.

1.1 Adding the package to the configuration system

The Config.in file is needed by the Buildroot configuration system to make your package available upon configuration. The Buildroot configuration system is based on the Linux® kernel configuration system and uses therefore the same syntax. Documentation about the syntax is available in the kernel sources in the directory Documentation/kbuild. The content of the Config.in file from the dummy package looks as follows:

```
config BR2_PACKAGE_DUMMY
    bool "dummy"
    default n
    help
    This is a dummy package to show how to integrate a new
    package into Buildroot. The syntax for this file is the
    default kbuild from the Linux kernel, more information at
    http://git.kernel.org/?p=linux/kernel/git/torvalds/linux2.6.git;a=tree;f=Documentation/kbuild
```

BR2_PACKAGE_DUMMY is the configuration option for the package and the following lines define attributes for this option. This name is used by Buildroot internally as a reference to your package. Change this reference name to your own package by removing “DUMMY” and replacing it with your package name in capital letters instead.

The next line specifies the package name. This name will appear later in the Buildroot configuration menu as a selectable entry. Remove “dummy” and replace it with your package name.

The “n” after default makes this package not selected by default. This means that this package is not selected in the configuration menu if you create a new clean configuration. To enable the package by default replace the “n” with a “y”.

The text after the “help” attribute is the package description. This text will appear in the Buildroot configuration menu when you enter the help section of your package.

For a simple package, that does not depend on others or provides functionalities to others, it is enough to edit the above mentioned attributes. But if your package relies

on others or you want to enable another package upon selection of your package in the menu you have to add dependencies in the Config.in file.

To enable a package upon selection if your package add

```
select BR2_PACKAGE_PACKAGE_NAME
```

Replace “PACKAGE_NAME” with the package name you want to enable. For instance to enable the wget package use BR2_PACKAGE_WGET.

If your package can not be build without having already build another package you have to add a dependency. This avoids a broken build process beforehand. To add such a dependency add

```
depends on BR2_PACKAGE_PACKAGE_NAME
```

and replace as above described “PACKAGE_NAME”. More detailed information can be found in the kernel documentation.

After you have edited the Config.in file you have to make Buildroot aware of your new package. Therefore you have to edit the file package/Config.in. The position where you insert the reference to your packages decides where the entry in menu system is going to be later on. A good position is after the line:

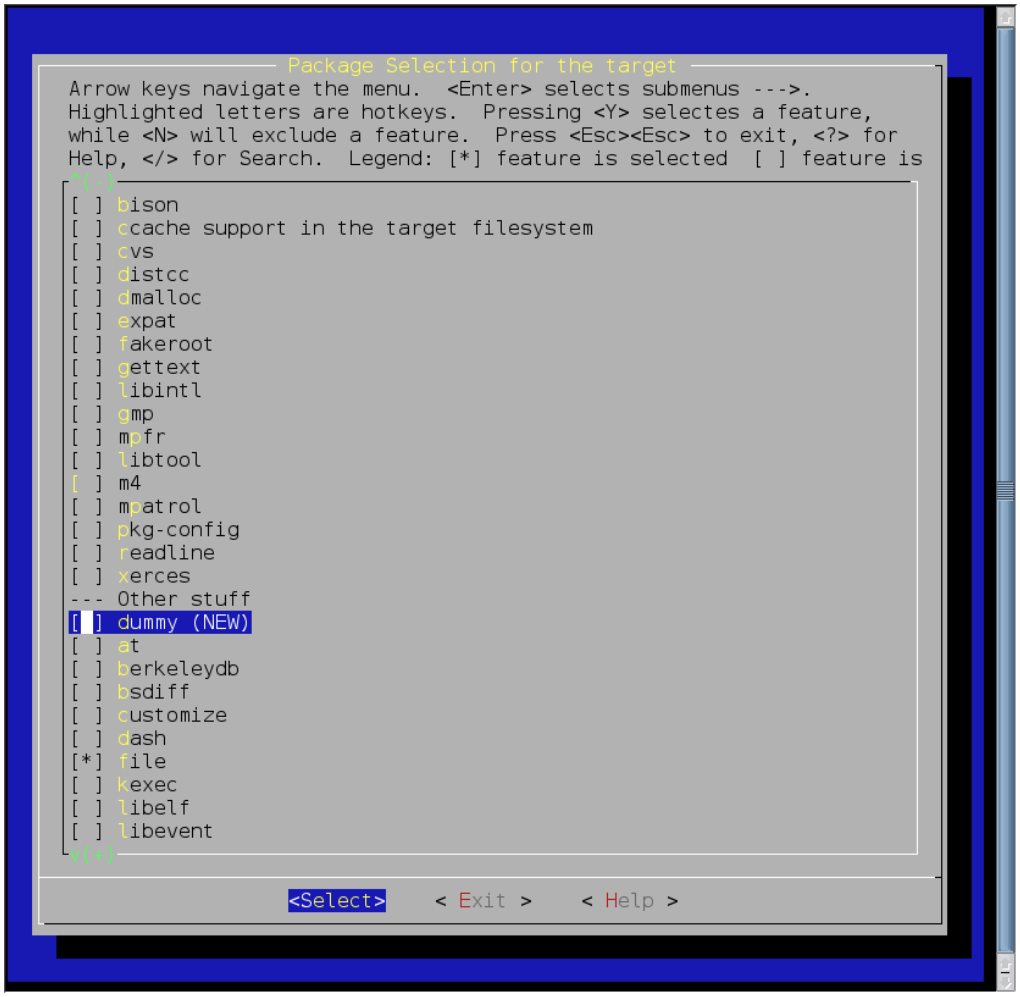
```
comment “Other stuff”
```

After the above line insert:

```
source “package/dummy/Config.in”
```

Replace “dummy” in the above path with the directory you have created for your package. The dummy package is not registered in this file as this would confuse a Buildroot user unnecessarily. Figure 2-1 shows the menu entry for the dummy package if you insert the line above unchanged.

Figure 2-1 Buildroot menu entry of the dummy package



1.2 Creating a Buildroot package Makefile

After adding your package to the Buildroot configuration system a Makefile is needed that contains rules for downloading, configuring, compiling and installing the software. The dummy.mk file which you have already copied to your own package directory can be used as a Makefile template. This Makefile example fits all simple applications which consist of a single binary. For other software such as libraries or more complex projects with multiple binaries, it must be adapted. Take a look at the other *.mk files in the package subdirectories for more examples on how to write a Buildroot package Makefile.

The following variables are defined on the top of the Makefile:

```
DUMMY_VERSION=1.2.3
DUMMY_SOURCE=dummy-$(DUMMY_VERSION).tar.bz2
DUMMY_SITE=http://www.example.net/dummy/source
DUMMY_DIR=$(BUILD_DIR)/dummy-$(DUMMY_VERSION)
DUMMY_CAT:=$(BZCAT)
DUMMY_BINARY:=dummy
DUMMY_TARGET_BINARY:=usr/bin/$(DUMMY_BINARY)
```

- **DUMMY_VERSION**: Version of the software package.
- **DUMMY_SOURCE**: File name of the tarball containing the dummy package on the website, ftp location or in the local download directory. The DUMMY_VERSION value is used to build the archive name.
- **DUMMY_SITE**: The HTTP or FTP site address from where the package is available for download. Must include the complete path to the directory where DUMMY_SOURCE is located. It is also possible to just place the archive in the download directory (src/dl/) and omit this entry as the package does not need to be downloaded any more.
- **DUMMY_DIR**: The directory into which the software will be decompressed. After extraction the package is configured and compiled there. Actually DUMMY_DIR is a subdirectory of the BUILDROOT internal BUILD_DIR (BUILD_DIR is build_avr32/ for the AVR32).
- **DUMMY_BINARY**: Name of the binary. This file will be copied to the target file system.
- **DUMMY_TARGET_BINARY**: Relative path to the binary on the target file system. In this location the compiled binary will be installed.
- **DUMMY_CAT**: Decompression tool to use in conjunction with the package archive. Valid values are $(BZCAT) for the bz2 algorithm, $(ZCAT) for the gz algorithm.

The rest of the Makefile are rules for downloading, unpacking, configuring and so forth. The first rule looks as follows:

```
$(DL_DIR)/$(DUMMY_SOURCE):
        $(WGET) -P $(DL_DIR) $(DUMMY_SITE)/$(DUMMY_SOURCE)
```

This rule downloads the package from the earlier specified website and stores it in the download directory (src/dl by default). If the package exists already in this directory the downloading rule will not be executed.

```
$(DUMMY_DIR)/.unpacked: $(DL_DIR)/$(DUMMY_SOURCE)
    $(DUMMY_CAT) $(DL_DIR)/$(DUMMY_SOURCE) | \
           tar -C $(BUILD_DIR)  $(TAR_OPTIONS) –
    toolchain/patch-kernel.sh $(DUMMY_DIR) package/dummy/ \
           dummy-$(DUMMY_VERSION)-\*.patch\*
    $(CONFIG_UPDATE) $(DUMMY_DIR)
    touch $@
```

This rule tests if the hidden file ".unpacked" exists in the dummy build directory. If this file does not exist the package is unpacked from the download directory to the build directory, all available patches are applied, config.guess and config.sub are updated and the file ".unpacked" is created. If you create this file in the dummy build directory Buildroot never tries to download, unpack and patch the sources. This is useful if you want to add your package locally. If you do not have to patch the package or you don't work with configure scripts you can remove these commands. Otherwise adapt these lines to your needs.

A lot of software makes use of configuration scripts to prepare itself for the build process. Especially large open source projects on the internet (e.g. GNU GCC) have many build options available and a lot of dependencies which need to be resolved before a build. The next rule addresses these configuration scripts.

```
$(DUMMY_DIR)/.configured: $(DUMMY_DIR)/.unpacked
```

```
(cd $(DUMMY_DIR); rm -rf config.cache; \
        $(TARGET_CONFIGURE_OPTS) \
        $(TARGET_CONFIGURE_ARGS) \
        ./configure \
        --target=$(GNU_TARGET_NAME) \
        --host=$(GNU_TARGET_NAME) \
        --build=$(GNU_HOST_NAME) \
        --prefix=/usr \
        --sysconfdir=/etc \
        $(DISABLE_NLS) \
        $(DISABLE_LARGEFILE) \
)
touch $@
```

The example above includes the most commonly used configuration options. Depending on your package you might have to add others. By running "./configure --help" in the package build directory all available configuration options are listed.

You can remove most of the above lines if your software has no configuration script. In this case the rule could look like this:

```
$(DUMMY_DIR)/.configured: $(DUMMY_DIR)/.unpacked
    touch $@
```

This will only create the file ".configured" in the source build directory to indicate that this package is configured.

The last rule needed to build a package is the call to the actual Makefile of the sources. This is done by the following lines:

```
$(DUMMY_DIR)/$(DUMMY_BINARY): $(DUMMY_DIR)/.configured
        $(MAKE) -C $(DUMMY_DIR)
```

If needed, you can pass values to the Makefile by adding them to the command line. An example of a Makefile that can be used with Buildroot is described in chapter 2.

After the build the package needs to be installed on the target file system. Following rule copies the dummy binary to the target file system and strips any unneeded symbols from it to reduce its size.

```
(TARGET_DIR)/$(DUMMY_TARGET_BINARY): $(DUMMY_DIR)/$(DUMMY_BINARY)
        $(INSTALL) -D $(DUMMY_DIR)/$(DUMMY_BINARY) $@
        $(STRIPCMD) $@
```

If you have to copy more files add respective copy commands here. Another approach is to let the sources Makefile handle the installation process. This approach is useful when the package is configured by scripts and has its own installation rules. In this case you have to call the installation routine from here and let the package handle this process itself. The following lines call the installation routine in the sources Makefile:

```
$(TARGET_DIR)/$(DUMMY_TARGET_BINARY): $(DUMMY_DIR)/$(DUMMY_BINARY)
        $(MAKE) DESTDIR=$(TARGET_DIR) -C $(DUMMY_DIR) install
```

Additional lines may be needed here to strip the binaries or remove any unneeded parts, such as man pages, from the target file system.

All software needed by your package should be available before the compilation or configuration of your package. In order to make Buildroot aware of these

dependencies you have to specify them. The uclibc library is for instance a software part on which your package most likely depends on. Therefore you have to name the package before your package.

```
dummy: uclibc $(TARGET_DIR)/$(DUMMY_TARGET_BINARY)
```

Also consider to register any dependencies with the configuration system. If you do that, all needed software is selected for the build upon the selection of your package. This results in a build without broken dependencies. Also take a look at the kbuild (and kconfig) documentation in the kernel sources on how to add dependencies to your package. A short introduction is available in chapter 1.1.

Since it should be possible to work with Buildroot offline, it is mandatory to implement a rule which downloads your package without building it. In addition is it useful to check if all package-sources are available. Because of these reasons following line is mandatory.

```
dummy-source: $(DL_DIR)/$(DUMMY_SOURCE)
```

The last two rules serve as cleanup. The purpose of the first rule (dummy-clean) is to clean the build directory by calling the source Makefile. This will force a new build of the package upon a new Buildroot build. The second clean rule (dummy-dirclean) removes the whole build directory and thus forcing a new extraction, patching and rebuild of the package the next time the Buildroot build process is initiated.

```
dummy-clean:
        -$(MAKE) -C $(DUMMY_DIR) clean
dummy-dirclean:
        rm -rf $(DUMMY_DIR)
```

The last lines of the file add the target dummy to the list of targets to be compiled by Buildroot by first checking if the configuration option for this package has been enabled with the configuration tool. If that is the case, Buildroot adds this package to its TARGETS global variable and it will be built upon the next build.

```
ifeq ($(BR2_PACKAGE_DUMMY),y)
TARGETS+=dummy
endif
```

After adapting Config.in and dummy.mk for your package, Buildroot should be able to download, patch, configure, compile and copy it to the target file system. This is not an “absolute” example which you have to follow in detail. There are many ways to do things but the above example gives a basic overview. Take a look at the other packages and compare their implementations. A similar example is also available from http://buildroot.uclibc.org/buildroot.html.

2 Creating a package

Unless you are not using automake and its buddies which take care of most of the configuration, compilation and clean processes you have to take care of this by yourself. The integral part in your software is the Makefile and this file must be adapted for Buildroot. An example Makefile for the dummy package could look like this:

```
GCC := $(CROSS_COMPILE)gcc
CFLAGS := -Wall -Os -g

.PHONY: all install clean
```

```
all: dummy

dummy: dummy.o
        $(GCC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)
%.o: %.c
        $(GCC) -c $< -o $@ $(CFLAGS)
install:
        install -m 755 dummy $(DESTDIR)/usr/bin
clean:
        rm -f dummy dummy.o
```

Especially important is the install rule. If you do not want to take care of it in the Buildroot package Makefile (dummy.mk) you have to add the DESTDIR path variable to the copy command to ensure that it is copied to the right place. This variable points to the root of the target root filesystem.

Pack the Makefile its associated sources in a tar archive, compress it either with the gz or bz2 algorithm and name it according to your package name and version. Place this archive in the download folder or on the internet site which you have specified. Alternatively copy your sources to the build directory and create the files ".unpacked" and ".configured". This omits the download, decompression and configuring but you have to do this again if you make a clean. At last run *make menuconfig*, select your package in the configuration and run *make* to build it for the target.

3 References

Description of the kernel configuration system (Kconfig):
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux2.6.git;a=tree;f=Documentation/kbuild

AVR®32 Buildroot application note:
http://www.atmel.com/dyn/resources/prod_documents/doc32062.pdf

An example of extending Buildroot with more software:
http://buildroot.uclibc.org/buildroot.html

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature