RTS3901/RTS3902 SDK User Manual

Release 1.1

Realtek

April 07, 2017

COPYRIGHT

DISCLAIMER

Realtek provides this document ``as is'', without warranty of any kind, neither expressed nor implied, including, but not limited to, the particular purpose. Realtek may make improvements and/or changes in this document or in the product described in this document at any time. This document could include technical inaccuracies or typographical errors.

TRADEMARKS

Realtek is a trademark, of Realtek Semiconductor Corporation. Other names mentioned in this document are trademarks/registered trademarks of their respective owners.

USING THIS DOCUMENT

This document is intended for the software and firmware engineer's general information on the Realtek IP Camera IC. Though every effort has been made to ensure that this document is current and accurate, more information may have become available subsequent to the production of this guide. In that event, please contact your Realtek representative for additional information that may help in the development process.

REVISION HISTORY

Revision	Release Date	Author	Summary
1.0	2015/10/16	Wei Wang	Initial Version
1.1	2016/6/6	Hurray Niu	update add user applications and libs

Contents

List of Figures

List of Tables

1 SDK Introduction

1.1 Directory Architecture

SDK's directory consists of the following parts:

```
[DIR] boards                => Board Support Package ( BSP )
[DIR] bootloader            => Uboot
[DIR] config                => Configuration scripts and files
[DIR] users                 => User program and library
[DIR] toolchain             => toolchain
[DIR] linux-3.10            => Linux Kernel
[DIR] utils                 => Host PC utilities
```

After configuration, the following directories are created:

```
[DIR] target                => A symbolic link to the target board BSP
[DIR] images                => Store the binary image file
[DIR] romfs                 => Binary executable file
[DIR] extfs                 => Binary executable file (not included in the final image file)
[DIR] tmpfs                 => Temporary executable file and library
```

2 Linux development environment

2.1 Prepare the Linux development environment

2.1.1 OS

Currently the SDK has only done a lot of testing on the Ubuntu 14.04 LTS, so it is recommended to use this version as a development system.

Build Ubuntu system

Before using the SDK, you need to follow the steps listed below.

1. Configure /bin/sh for bash
   - sudo dpkg-reconfigure dash ===> Select ``No''
2. Set the environment variables TERM and TERMINFO
   - export TERM=xterm
   - export TERMINFO=/lib/terminfo

Note: You can save the above two commands to the ~/.profile file to avoid re-configured at each boot

3. Install the necessary packages
   - sudo apt-get install libncurses5-dev autoconf automake cmake libtool gettext texinfo gawk flex bison mtd-utils
4. If you are using a 64-bit system, you will need to install additional 32-bit packages
   - sudo dpkg --add-architecture i386
   - sudo apt-get update
   - sudo apt-get install gcc-multilib lib32z1-dev lib32ncurses5-dev lib32ncursesw5-dev

2.2 Build the SDK

Building the SDK requires the following steps:

- config: Configure the SDK

- make：Compile SDK
- pack：Pack the different binary images into a bin file
- clean：Clear the intermediate configuration and object files

2.2.1 config

Before compiling the SDK, you can select a different configuration by using the command **make config** or **make menuconfig**.

The top configuration menu is Figure 2.1 :

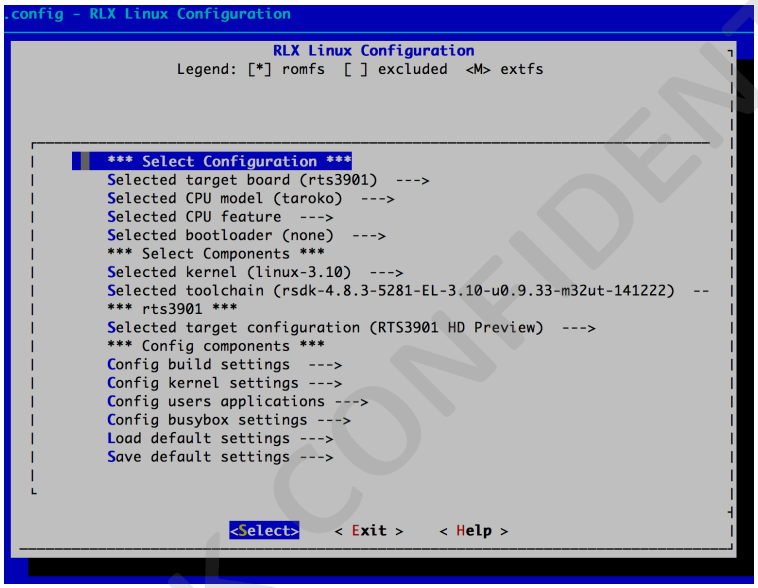

Figure 2.1: Top configuration menu

If you need to configure the kernel, user mode program or busybox, you can select ``Config components'' under the corresponding options To make a separate set. The ``Load default settings'' option is based on the selection in ``Selected target configuration'' To load the specified default system configuration. Likewise, ``Save default settings'' will also store your current configuration according to the selection To the corresponding default configuration file.

``Config users applications'' Menu is Figure 2.2 :

When the configuration is complete, the symbolic link ``target'' is created and the corresponding ``target'' is selected according to the ``Selected target board'' **boards** directory。

2.2.2 make

If everything is fine, execute the **make** command to produce a binary image that is available to the target board. During the compilation process, the intermediate files will be stored in **tmpfs**, **romfs** and **extfs** and other directories. The binary images will be placed in **images** directory.

2.2.3 pack

Execute the **make pack** command to package one or more binary images into a separate bin file.

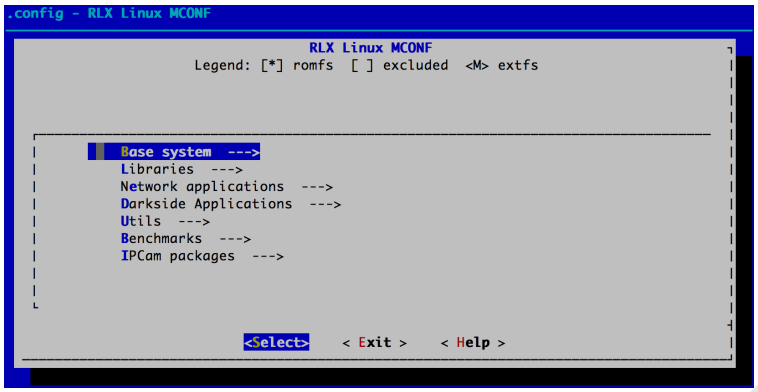

Figure 2.2: Configure the user mode program

2.2.4 clean

make clean clear the compile-time generated temporary object files, **make mrproper** used to clear the temporary configuration file.

3 U-boot

3.1 Introduction

Bootloader is a small program that runs before the Linux kernel, which is used to initialize the hardware device and boot the operating system kernel. The SDK uses U-boot as the bootloader.

3.1.1 Start U-boot

Power on the development board, the default standard input and standard output is UART1. The UART setting is:

- Buadrate: 57600
- Data: 8
- Parity: none
- Stop: 1
- Flow control: none

3.1.2 U-boot Common commands

Note: When you enter some of the letters of the command, press the Tab key. The system will automatically fill or list the list of possible commands.

Commands	Descriptions
?	Get all the command list or list the help of a command.
help	Same as ?
printenv	print environment parameter
setenv	Set or delete variables
setethaddr	Set the Ethernet MAC address
setipaddr	Set the Ethernet IP address
ping	Determine the destination host network status
tftpsrv	Start tftp server
bootm	Set the runtime environment and start executing binaries.
update	Start tftp server and the received image file written to a specific address in the flash

3.2 Programing Image

Programing Image to flash via tftp.

3.2.1 Preparation

If there is only U-boot on development board, and did not write the correct Linux kernel, boot directly into the U-boot command line. If the Linux kernel have written correctly, U-boot will have a 3 seconds countdown, before the end of the countdown by pressing any key on the keyboard can also enter the U-boot command line.

Configure the network

First need to confirm whether the U-boot network is properly configured, you can execute the command **printenv** to confirm, such as Figure 3.1 show, Check whether the **ethaddr**, **ipaddr**, **netmask** variables are set correctly.

If the network is not configured correctly, execute the following command settings:

```
# setethaddr 00:e0:4c:02:00:40
# setenv ipaddr 10.0.2.24
# setenv netmask 255.255.240.0
```

Note: Assumes that the IP address of the development board is 10.0.2.24, netmask is 255.255.240.0

Install tftp client on PC side

The following is a recommended version of Windows and Linux client and use examples.

```
rlxboot# printenv
=5
addmisc=setenv bootargs ${bootargs}console=ttyS0,${baudrate}panic=1
baudrate=57600
bootaddr=(0xBC000000 + 0x10000 * (4 + 2 + 4 + 8))
bootcmd=bootm 0xbc120000
bootfile=/vmlinux.img
ethact=r8168#0
ethaddr=00:e0:4c:02:00:40
fileaddr=82000000
filesize=6b2369
gatewayip=10.0.0.1
ipaddr=10.0.2.24
load=tftp 80500000 ${u-boot}
loadaddr=0x82000000
netmask=255.255.240.0
stderr=serial
stdin=serial
stdout=serial

Environment size: 443/131068 bytes
```

Figure 3.1: printenv output result

- Windows：tftpd32, offical website: http://tftpd32.jounin.net/

  Use as follows Figure 3.2 , **Host** fill in the development board's IP address, **Port** Fill in tftp's default port number 69 or just keep empty,**Local File** Select the local image file for upload. Click the **Put** button.

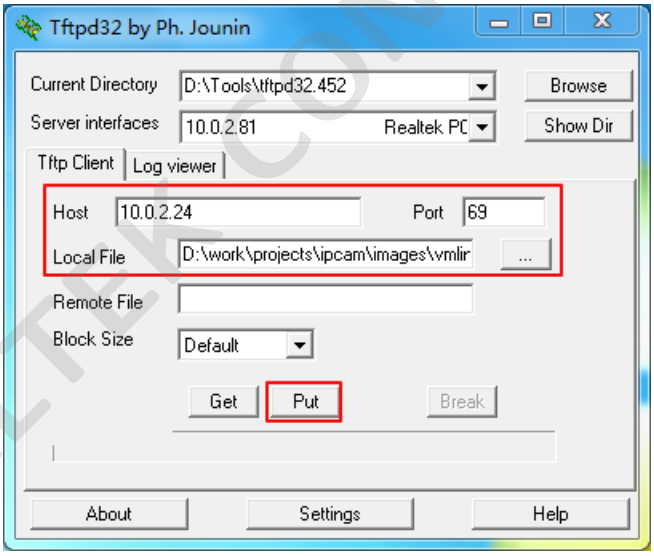

Figure 3.2: tftpd32 interface

- Linux：tftp-hpa

  Use as follows

```
$ tftp 10.0.2.24 -m binary -c put vmlinuz.img
```

3.2.2 Burn U-boot

1. Connect the development board and the PC via Ethernet

2. Execute the command at the U-boot

```
# update uboot
```

3. PC side

```
$ tftp 10.0.2.24 -m binary -c put u-boot.bin
```

4. Receives the U-boot image and burning the image

3.2.3 Burn Linux Kernel

1. Connect the development board and the PC via Ethernet
2. Execute the command at the U-boot

```
# update kernel
```

3. PC side

```
$ tftp 10.0.2.24 -m binary -c put vmlinuz.img
```

4. Receives the Linux kernel image and burning the image

3.2.4 Burn MCU F/W

1. Connect the development board and the PC via Ethernet
2. Execute the command at the U-boot

```
# update fw
```

3. PC side

```
$ tftp 10.0.2.24 -m binary -c put mcu_fw.bin
```

4. Receives the MCU FW image and burning the image

3.2.5 Burn RootFS

1. Connect the development board and the PC via Ethernet
2. Execute the command at the U-boot

```
# update rootfs
```

3. PC side

```
$ tftp 10.0.2.24 -m binary -c put rootfs.bin
```

4. Receives the RootFS image and burning the image

3.2.6 Burn linux.bin

linux.bin is generated by the ``**make pack**'', linux.bin can include one or more partitions.

For more information about linux.bin file structure and flash partition, please refer to 6

The programming steps are as follows:

1. Connect the development board and the PC via Ethernet
2. Execute the command at the U-boot

```
# update all
```

3. PC side

```
$ tftp 10.0.2.24 -m binary -c put linux.bin
```

4. Receives the linux.bin image and burning the image

3.2.7 Test Linux Kernel

If you just want to test the Linux kernel and not burn flash, you can refer to the following steps:

1. Execute the command at the U-boot

```
# tftpsrv
```

2. PC side

```
$ tftp 10.0.2.24 -m binary -c put vmlinuz.img
```

3. When transfer is completed, execure ``**bootm**'' at command line

```
# bootm
```

4 Linux Kernel

4.1 Kernel Source Code

Linux kernel source code in the root directory of the SDK linux-3.10 directory, the other BSP-related code is under target/bsp.

4.2 Configure Kernel

If you do not have sufficient understanding of the RTS3901/RTS3902 platform, do not modify the default configuration. But can add the required modules.

The steps to configure the kernel are as follows:

1. In the root directory of the SDK run **make config**, select **Load default settings** (Figure 4.1)**:**

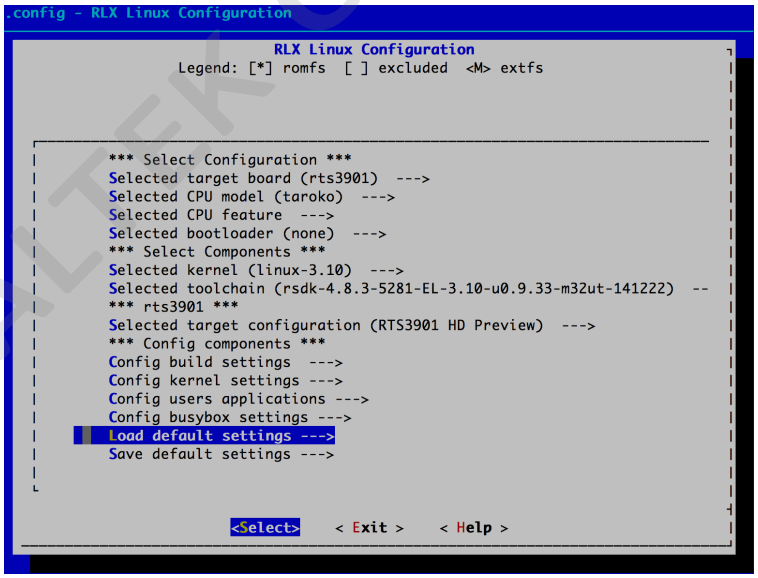

Figure 4.1: Load default settings

2. If you need to modify the kernel configuration items, select **Config kernel settings** to enter the Linux kernel configuration interface (Figure 4.2)

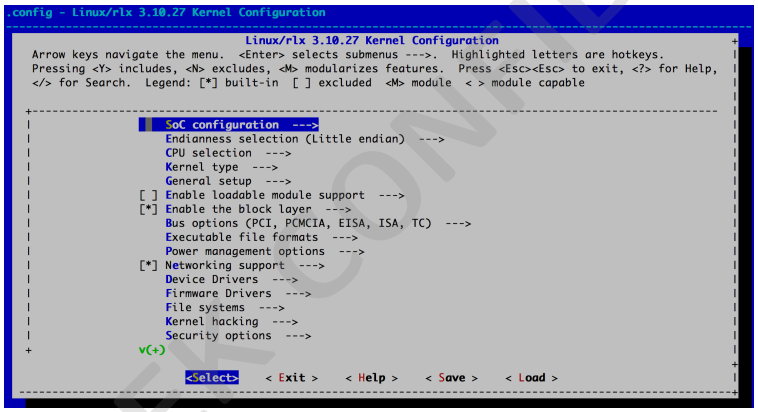

Figure 4.2: Config kernel settings

5 File System

5.1 RootFS Introduction

The root directory of Linux is `/', and the system mounts the device as the root directory after the kernel is started. All system mount points, system configuration, etc. are located under the root file system.

Root file system to save the conventional memory, flash memory or network file system. All applications, libraries, and other services need to be located under the root file system. Form *RootFS Directory List* shows the directories under the root directory.

Table 5.1: RootFS Directory List

Directory	Description
bin	The basic command of the executable file
dev	Device file node
etc	System configuration files, including startup files
lib	Basic libraries such as C libraries, kernel modules
media	Temporary file system mount point (usually used to mount mobile devices, such as SD card)
mnt	Temporary file system mount point
proc	The virtual file system used to export kernel and process information
sys	System Device and files
tmp	Temporary file
usr	User applications and files
var	System logs and temporary files for services

5.2 SquashFS File System

SquashFS file system is designed for embedded systems read-only compressed file system, currently supports zlib, lzma, lzo compression algorithm.

5.2.1 Create and use the SquashFS file system

1. Configure the kernel to add SquashFS support

```
=> File systems
   => Miscellaneous filesystems
      => SquashFS 4.0 - Squashed file system support
         => Include support for XZ compressed file systems
```

2. Make SquashFS image

   The Makefile automatically calls the mksquashfs tool to make the SquashFS image. If you need to manually generate SquashFS image file (using lzma compression format), you can execute the following command:

```
$ ./config/linux/mksquashfs ./romfs/ ./image/rootfs.bin -comp xz
```

5.3 JFFS2 File System

JFFS2 is based on the JFFS file system. As an embedded small file system, JFFS2 has read and write functions, and has the following characteristics:

- Storage compression file, the biggest feature is the ability to read and write
- The disadvantage is that the mount process scans the entire file system, so the mount time is positively related to the partition size.

5.3.1 Create and use the JFFS2 file system

1. Configure the kernel to add MTD_BLOCK support

```
=> Device Drivers
   => Memory Technology Device (MTD) support (MTD [=y])
      => Caching block device access to MTD devices
```

2. Configure the kernel and add JFFS2 support

```
=> File systems
   => Miscellaneous filesystems
      => Journalling Flash File System v2 (JFFS2) support
```

3. Make JFFS2 image

   The SDK merges all partition images into a single file with the command **make pack**. Before you package, you need to configure the path of each image, Specifically you need to edit the file **utils/scripts/merge.ini**. If you want to specify other images, you can modify the path directly.

   We assume that the default path is used, and the JFFS2 image is generated into the directory **image**. Note that the JFFS2 image generate based on the directory `jffs2', If the directory does not exist, the JFFS2 image will not be packaged.

   To modify the JFFS2 image is also very simple, Only need to modify the `jffs2' under the file can be.

4. Burn merge image

   Please refer to 6.3 - 6.4.

5. Use JFFS2 Partitions

   By default, the JFFS2 partition will be mounted to the /**usr**/**conf** directory, If the directory of the file is modified, then the corresponding data will be saved to Flash, So after the system restarts, the new data will not be lost.

6 Flash Image

6.1 Flash Partitions

By default, the SDK will configure seven partitions:

1. U-boot partition (boot)
2. MCU F/W partition (mcu)
3. Parameter configuration partition (hconf)
4. User data partition (userdata)
5. Linux kernel partition (kernel)
6. Rootfs partition (rootfs)
7. LDC Mapping table partition (ldc)

Assume that the flash size is 16M and its partition is as Figure 6.1. The U-boot partition must be placed at the start address.

Flash partition can be divided according to demand, by editing the file **target/bsp/partition.ini** To configure the partition table. The configuration file can be divided into multiple segments. It is important to note that the global segment must be placed at the beginning. The global segment is mainly used to specify the total size of the flash and the sector size, The size must be expressed in hexadecimal. As for the other segments, only the size must be specified, the unit is the number of erased units, expressed in decimal.

```
[global]
flash_size = 0x1000000
sector_size = 0x10000
[boot]
offset = 0
sectors = 4
[mcu]
sectors = 4
[hconf]
sectors = 4
[userdata]
sectors = 8
[kernel]
sectors = 64
[rootfs]
sectors = 128
[ldc]
sectors = 8
```

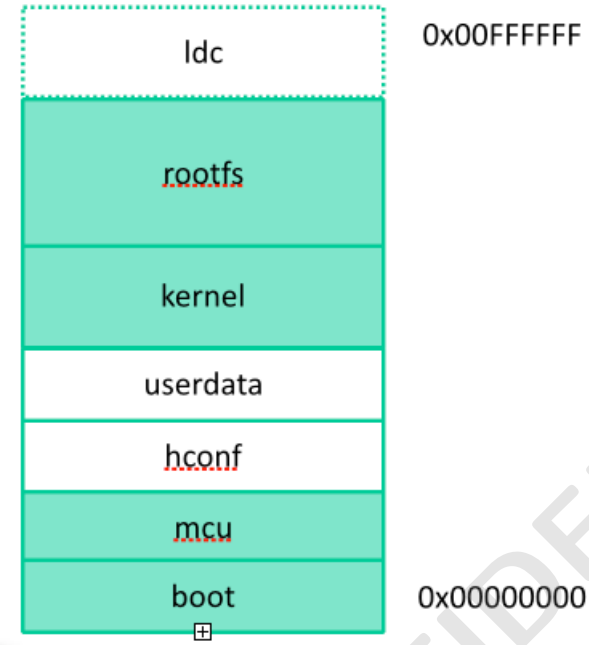

Figure 6.1: Flash Partitions

As in the example above, offset can also be ignored and it is recommended to ignore it so that the offset is automatically calculated by the software. Where the offset and sectors are the number of sectors.

Boot partition must be placed at the beginning of the flash, the order of other partitions can be adjusted. Note that the names of these predefined partitions can not be modified. If you want to add a new partition, you can place any place after the boot partition. Modify the partition.ini configuration file, re-compile the SDK (including U-boot), the new partition can take effect

6.2 Image file structure

According to the flash partition, which U-boot partition, MCU F/W partition and Linux kernel partition need to use before programming. The programming of these partitions can be done separately in the U-boot, or it can be packaged as a merged image file **linux.bin**.

This section introduces the linux.bin image file structure.

Linux.bin file structure can refer to Figure 6.2 。

Header section is fixed to 256 bytes, used to save the basic information of the image, this part will not be programmed into the flash.

Each programming section includes: magic number 4 bytes, address 8 bytes, length 8 bytes, data area, check code 4 bytes. （ Refer to Figure 6.2 ）

Write the partition format

6.3 Preparation for burning image

Notice: If you want to use JFFS2 partition, please refer to 5.3.1 See how to create a JFFS2 partition.

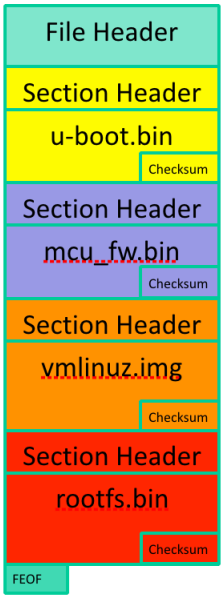

Figure 6.2: linux.bin file structure

magic	resv	resv	
burnaddr		burnlen	
data		data	
data		data	
data		...	checksum

1. Configure the flash partition

   Edit the file **target/bsp/partition.ini**

2. Compile kernel

   Use make to build a image. At the end of the compilation, you can find the kernel image in the directory **image**.

3. Configure merge image

   You can configure merge rules by editing **utils/scripts/merge.ini**. Specifies the path to the partition image, and you can add `#' in front of the line to ignore this image.

4. Merge image

   Run **make pack** to generate a merged **linux.bin** file. If you want to include a JFFS2 partition, refer to 5.3.1.

6.4 Upgrade linux.bin on WebUI

In the PC through the browser to access the development board, in the ``Admin->Image Upgrade'' page, the linux.bin directly burn to the development board flash. (Refer to Figure 6.3)

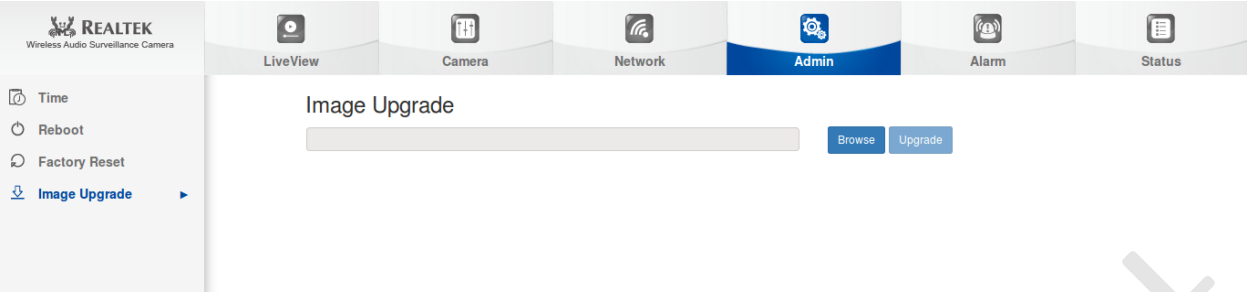

Figure 6.3: Upgrade image through the Web UI

6.5 Upgrade linux.bin via NFS

6.5.1 Set up an NFS server on Ubuntu

Assume that the Ubuntu username is `ipcam' and set the NFS share path to `/home/ipcam/share/nfs' IP Address:10.0.1.10。

- Install NFS Server

```
$ sudo apt-get install nfs-kernel-server
```

- Configure the shared directory, edit /etc/exports

```
/home/ipcam/share/nfs  *(ro,sync,no_root_squash)
```

- Restart NFS Service

```
$ sudo service nfs-kernel-server restart
```

- Copy Image `linux.bin' to shared directory `/home/ipcam/share/nfs'

6.5.2 Configure the NFS client on the device

- Configure the local network

```
# ifconfig eth0 10.0.1.10
```

- Mount remote NFS shared directory

```
# mount -t nfs -o nolock 10.0.1.10:/home/ipcam/share/nfs /mnt/
```

- Start the programming program

```
# neuralyzer /mnt/linux.bin
```

6.6 Through the UBOOT programming raw.bin

Sometimes need to burn the original raw.bin, you can edit `utils/scripts/raw.ini' Configure which partitions raw.bin need to include and specify the file path. If there is no need to burn the partition, you can add # at the first place of the partition Note that in order to avoid can not start without UBOOT, UBOOT partition must be configured.

- Generate raw.bin

```
Run 'make pack' to generate a merged 'raw.bin' file.
```

- Get raw.bin via tftp

```
# tftpboot 0x81800000 192.168.1.10:raw.bin;
```

- flash command to burn raw.bin

```
# sf probe 0;sf erase 0x0 0x1000000;sf write 0x81800000 0x00000 0x1000000;
```

7 Develop applications and libraries

The applications and libraries are stored under the **users** directory and are grouped into different subdirectories by type, such as **users**/**system**, **users**/**network** and so on. The library is stored in the **users**/**libs** subdirectory.

In the SDK, the application and library configuration is managed by a system called **formosa**. The following sections describe how to add new library and application to the SDK.

7.1 Add library

Suppose we add a library named **libbar** in the SDK, proceed as follows:

- Create the **libbar** directory under **users**/**libs** and add the source code
- Create subdirectory **.formosa** under **libbar**
- In **.formosa** directory, create **rlxbuild.in** and **rlxbuild.mk** two files (see the file below)
- Modify the upper directory of the **Kconfig** file, add **libbar** configuration

7.1.1 rlxbuild.in

Example:

```
pkg_$(CONFIG_PACKAGE_libbar) += libs/libbar
```

7.1.2 rlxbuild.mk

Example:

```
all: libbar tmpfs

libbar: libbar.c
    $(CC) -shared -fpic $^ -o libbar.so
    $(CC) -static $^ -c
    $(AR) -r libbar.a libbar.o

tmpfs:
    $(TMPFSINST) libbar.h /include
    $(TMPFSINST) libbar.a /lib

romfs:
    $(ROMFSINST) libbar.so /lib
```

```
clean:
    rm -f *.o *.a *.so
```

7.1.3 Kconfig

Example:

```
config PACKAGE_libbar
    bool "libbar"
    help
      libbar package
```

7.2 Add application

Suppose we add a application named **foo** under **users/utils**. And the application relies on the libbar library that was added above. Proceed as follows:

- Create the **foo** directory under **users/utils** and add the source code
- Create subdirectory **.formosa** under **foo**
- In **.formosa** directory, create **rlxbuild.in** and **rlxbuild.mk** two files (see the file below)
- Modify the upper directory of the **Kconfig** file, add **foo** configuration

7.2.1 rlxbuild.in

Example:

```
pkg_$(CONFIG_PACKAGE_foo) += utils/foo
foo: libbar
```

When **CONFIG_PACKAGE_foo** (defined in **users/utils/Kconfig**) is selected, **utils/foo** will be compiled. In addition, **foo** depends on **libbar**, so **libbar** will be compiled before **foo**.

Note: The name of the package under the /**users** must be unique. In this case, the name of **foo** must be unique.

7.2.2 rlxbuild.mk

rlxbuild.mk is the package real Makefile, in this Makefile can define four target:

- **all**: make all package
- **clean**: make clean
- **romfs**: install the compiled target files into the directory **romfs**
- **tmpfs (optional)**: install the compiled target temporial files into the directory **tmpfs**

SDK provides two auxiliary variables related to **romfs**:

- **$(ROMFSINST)**: The script used to install the target file into the directory **romfs**
- **$(DIR_ROMFS)**: The absolute path to the directory **romfs**

If the package contains some other programs will use the file, we need to implement **tmpfs** target。

Note: Applications rarely use **tmpfs** target, but the library will often use it.

Similiar to **romfs**, SDK also provides two auxiliary variables related to **tmpfs**:

- **$(TMPFSINST)**: The script used to install the target file into the directory **tmpfs**
- **$(DIR_TMPFS)**: The absolute path to the directory **tmpfs**

Example:

```
CFLAGS += -I$(DIR_ROMFS)/include
DFLAGS += -L$(DIR_TMPFS)/lib

all: foo

foo: foo.c
    $(CC) $(CFLAGS) $^ -o $@ $(DFLAGS) -lbar

romfs:
    $(ROMFSINST) foo /bin/foo

clean:
    rm -f foo *.o
```

7.2.3 Kconfig

Adding the package to the SDK's configuration menu needs to modify **Kconfig**, continue the previous example, we need to modify **users**/**utils**/**Kconfig** to add **foo** configuration:

```
config PACKAGE_foo
    bool "foo"
    help
      foo package
```

7.3 Compile

After completing the code, you need to run it in the root directory of the SDK **make menuconfig**, select new libbar and foo, and then in the root directory **make** to compile, if you just want to compile the single package, you can execute the command:

```
$ make foo_build
```

Clear the compiled result:

```
$ make foo_clean
```

By default, the detailed compilation message is hidden, if you want to view the details of the compiler, you can **make** followed by ``**V=1**''

```
$ make V=1
```

Or:

```
$ make foo_build V=1
```

Realtek Semiconductor Corp.
Headquarters
No. 2, Innovation Road II
Hsinchu Science Park, Hsinchu 300, Taiwan
Tel.: +886-3-578-0211. Fax: +886-3-577-6047
http://www.realtek.com.tw