Hi3516A/Hi3516D Linux Development Environment

User Guide

Issue 02

Date 2015-06-16

HiSilicon Technologies Co., Ltd.

Address:	Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China
Website:	http://www.hisilicon.com
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the Linux development environment of the Hi3516A. This document also explains how to set up the Linux and network development environments, burn the Linux kernel, and root file system, and start Linux-based applications.

After reading this document, customers will clearly understand the Linux development environment.

NOTE

This document uses the Hi3516A as an example. Unless otherwise stated, Hi3516D and Hi3516A contents are consistent.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3516A	V100
Hi3516D	V100

Intended Audience

This document is intended for:

- Technical support personnel
- Software development engineers

Change History

Changes between document issues are cumulative. Therefore, the latest document issue contains all changes made in previous issues.

Issue 02 (2015-06-16)

This issue is the second official release, which incorporates the following changes:

Chapter 4 Root File System

In section 4.3.5, the directory that stores mksquashfs is changed to **SDK/package/osdrv/tools/pc**.

Issue 01 (2014-12-20)

This issue is the first official release, which incorporates the following changes:

The contents related to the Hi3516D are added.

Issue 00B02 (2014-09-14)

This issue is the second draft release.

Issue 00B01 (2014-07-25)

This issue is the first draft release.

Contents

About This Document ... i

1 Development Environment ... 1

1.1 Embedded Development Environment ... 1

1.2 Overview of the Hi3516A Linux Development Environment ... 2

1.3 Setting Up the Linux Development Environment ... 3

1.3.1 Installing an OS on the Linux Server ... 3

1.3.2 Installing the Cross Compiler ... 3

1.3.3 Installing the Hi3516A SDK ... 4

2 U-boot ... 5

3 Linux Kernel ... 6

3.1 Kernel Source Codes ... 6

3.2 Configuring the Kernel ... 6

3.3 Compiling the Kernel and Generating the Kernel Image uImage ... 7

4 Root File System ... 8

4.1 Introduction to the Root File System ... 8

4.2 Creating a Root File System by Using the BusyBox ... 9

4.2.1 Obtaining the Source Code of the BusyBox ... 9

4.2.2 Configuring the BusyBox ... 9

4.2.3 Compiling and Installing the BusyBox ... 10

4.2.4 Creating a Root File System ... 10

4.3 Introduction to File Systems ... 11

4.3.1 CRAMFS ... 11

4.3.2 JFFS2 ... 12

4.3.3 YAFFS2 ... 13

4.3.4 initrd ... 13

4.3.5 Squashfs ... 14

5 Application Development ... 15

5.1 Compiling Code ... 15

5.2 Running Applications ... 15

A Acronyms and Abbreviations ... 16

Figures

Figure 1-1 Development in embedded mode... 1

Figure 1-2 Setting up the Hi3516A Linux development environment 2

Figure 4-1 Structure of the root file system's top directory 8

Tables

Table 1-1 Software running in the Hi3516A Linux development environment 2

Table 4-1 Some directories that can be ignored 9

Table 4-2 JFFS2 parameters 13

1 Development Environment

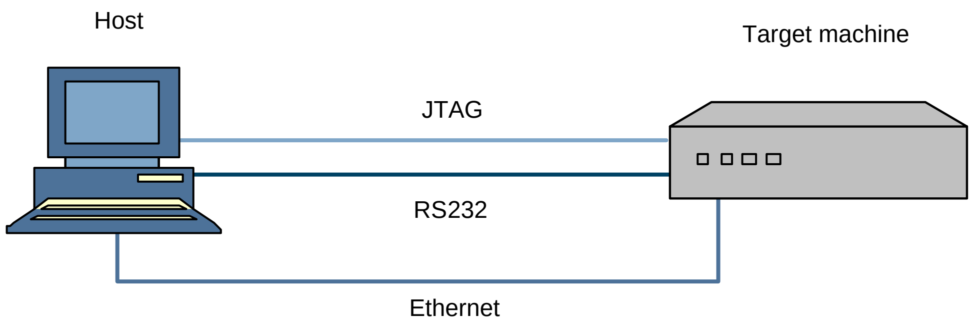
1.1 Embedded Development Environment

The development and debugging tool cannot be run on the embedded board, because the resources of the embedded board are limited. The embedded board is typically developed in cross compilation mode, that is, in host+target machine (evaluation board) mode. The host and target machine are typically connected through the serial port. However, they can also be connected through the network port or Joint Test Action Group (JTAG) interface, as shown in Figure 1-1.

The processors of the host and the target machine are different. A cross compilation environment must be built on the host for the target machine. After a program is processed through compilation, connection, and location, an executable file is created. When the executable file is burnt to the target machine by some means, the program can then run on it.

After the target machine's bootloader is started, operational information about the target machine is transmitted to the host and displayed through the serial port or the network port and displayed. You can control the target machine by entering commands on the host's console.

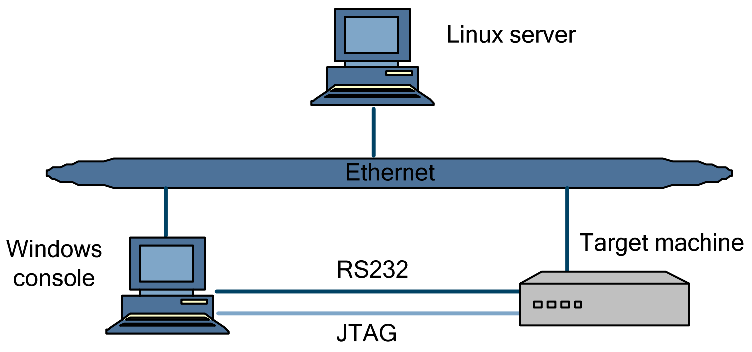
Figure 1-1 Development in embedded mode



1.2 Overview of the Hi3516A Linux Development Environment

The Hi3516A Linux development environment consists of a Linux server, a Windows console, and a Hi3516A DMEB (target machine) on the same network, as shown in Figure 1-2.

Figure 1-2 Setting up the Hi3516A Linux development environment



After a cross compilation environment is set up on the Linux server, and the Windows console is connected to the Hi3516A reference board (REFB) through the serial port or network interface, the developer can develop programs on the Windows console or on the Linux server through remote login. Table 1-1 describes the software running in the Hi3516A Linux development environment.

NOTE

Although a Windows console exists in the development environment, many operations can be completed on the Linux server, such as replacing the HyperTerminal with Minicom. Therefore, you can adjust the development environment to your personal preferences.

Table 1-1 Software running in the Hi3516A Linux development environment

Software		Description
Windows console	Operating system (OS)	Windows 98, Windows me, Windows 2000, Windows XP, or Windows 7
	Application software	Putty, HyperTerminal, Trivial File Transfer Protocol (TFTP) server, and DS-5.
Linux server	OS	Ubuntu, Redhat, and Debian are supported, and there are no special requirements. The kernel 2.6.18 or later is supported. Additionally, full installation is recommended.
	Application software	Network file system (NFS), telnetd, Samba, and VIM, and ARM cross compilation environment (Gcc 4.8.3). Other application software varies according to the actual development requirements. The required software is pre-installed by default. You need only configure the software before using it.

Software		Description
Hi3516A	Boot program	Fastboot
	OS	HiSilicon Linux (HiLinux for short). The HiLinux kernel is developed based on the standard Linux kernel V3.4.y, and the root file system is developed based on the BusyBox V1.20.2.
	Application software	Supports common Linux commands, such as **telnetd** and **gdb server**.
	Program development library	uClibc-0.9.33.2 and glibc-2.16-2012.09

1.3 Setting Up the Linux Development Environment

1.3.1 Installing an OS on the Linux Server

It is recommended that you install the latest stable release of an OS on the Linux server (such as the RedHat Fedora Core series, SUCE10, or Ubuntu10) to ensure easily available technical support. The following lists some recommended OS versions:

- Later versions of RedHat, such as the RedHat Fedora Core series, Redhat Enterprise Linux, and Red Hat 3.4.4-2.
- Earlier versions of RedHat, such as RedHat 9.0.

The stable releases of Debian are also commonly used. The advantage of Debian is that several types of installation packages are available and can be easily updated online.

1.3.2 Installing the Cross Compiler

CAUTION

A cross compiler obtained from other sources (such as Internet) may not be compatible with the existing kernel, and may result in unexpected issues during development.

The release package provides two compilation tool chains: arm-hisiv300-linux (based on Uclibc) and arm-hisiv400-linux (based on Glibc). In this document, **arm-hisiXXX-linux** is used to represent the two tool chains.

To install the cross compiler, perform the following steps:

Step 1 Go to **osdrv/toolchain/arm-hisiXXX-linux**, and decompress the tool chain by running the following commands:

```
cd  toolchain/arm-hisiXXX-linux/
```

```
tar  -xvf arm-hisiXXX-linux.tar.bz2
```

Step 2 Install the tool chain by running **sudo ./cross.install**.

----End

1.3.3 Installing the Hi3516A SDK

The Hi3516A SDK is a software development package based on the Hi3516A REFB. It contains all the tools and source code used in Linux-related development. Therefore, the Hi3516A SDK acts as the basic software platform for chip development.

To install the Hi3516A SDK on a Linux server, perform the following steps:

Step 1 Copy the **Hi3516A_V100R001XX.tgz** package (XX indicates the version number) to the Linux server.

Step 2 Decompress the preceding package by running **tar –zxf   Hi3516A_V100R001XX.tgz**.

If no message is displayed during decompression, wait until the command is executed.

Step 3 Open the **Hi3516A_V100R001XX** folder, and run **./ sdk.unpack**.

If you do not have the **root** permission, the system asks you to enter the password of the **root** user or **sudo** user. If the system displays a message indicating that you have no execution permission, run **chmod 777 ./sdk.unpack**.

----End

2 U-boot

For details, see *Hi3516A U-boot Porting Development Guide*.

3 Linux Kernel

3.1 Kernel Source Codes

After the Hi3516A SDK is installed, the kernel source code is saved in **osdrv** folder of the SDK. You can open the folder to perform related operations.

3.2 Configuring the Kernel

CAUTION

If you are not familiar with the kernel and Hi3516A platform, do not change the default configuration. However, you can add modules as required.

To configure the kernel, run the following commands:

Step 1 Copy the **.config** file manually:

```
hisilicon$cd kernel/linux-3.4.y
hisilicon$cp arch/arm/configs/hi3516A_full_defconfig .config
```

Step 2 Configure the kernel by running the **make menuconfig** command

```
hisilicon$make ARCH=arm CROSS_COMPILE= arm-hisiXXX-linux- menuconfig
```

Step 3 Select modules as required.

Step 4 Save the settings and exit.

----End

NOTE

The two parameters **ARCH = arm** and **CROSS_COMPILE = arm-hisiXXX-linux-** must be added after the **make** command during kernel compilation. The **CROSS_COMPILE** parameter indicates the tool chain. In this document, the **CROSS_COMPILE = arm-hisiXXX-linux-** parameter indicates the following two situations:

- Hi3516A_V100R001C01SPCxxx corresponds to uclibc. If the uclibc tool chain is used, the **CROSS_COMPILE** parameter is set to **arm-hisiv300-linux-**.
- Hi3516A_V100R001C02SPCxxx corresponds to glibc. If the glibc tool chain is used, the **CROSS_COMPILE** parameter is set to **arm-hisiv400-linux-**.

3.3 Compiling the Kernel and Generating the Kernel Image uImage

After settings are saved, run **make ARCH=arm CROSS_COMPILE=arm-hisiXXX-linux-uImage** to compile the kernel and generate the kernel image. This may take several minutes.

NOTE

If errors occur during kernel compilation, execute commands in the following sequence:

- **make ARCH=arm CROSS_COMPILE=arm-hisiXXX-linux- clean**
- **make ARCH=arm CROSS_COMPILE=arm-hisiXXX-linux- menuconfig,**
- **make ARCH=arm CROSS_COMPILE=arm-hisiv100nptl-linux- uImage.**

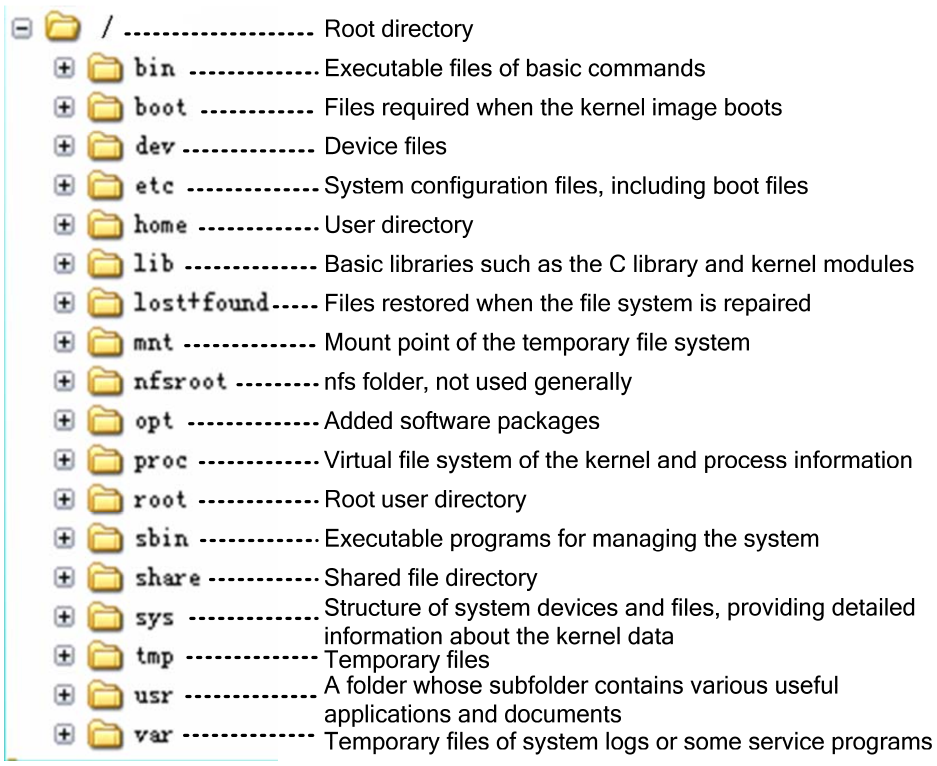
4 Root File System

4.1 Introduction to the Root File System

The top layer of the Linux directory structure is the root directory called "/". After loading the Linux kernel, the system mounts a device to the root directory. The file system of the device is called the root file system. All the mount points of system commands, system configuration, and other file systems are all located in the root file system.

The root file system is typically stored in the common memory, flash memory, or network-based file system. All the applications, libraries, and other services required by the embedded system are stored in the root file system. Figure 4-1 shows the structure of the top directory of the root file system.

Figure 4-1 Structure of the root file system's top directory



/	Root directory
bin	Executable files of basic commands
boot	Files required when the kernel image boots
dev	Device files
etc	System configuration files, including boot files
home	User directory
lib	Basic libraries such as the C library and kernel modules
lost+found	Files restored when the file system is repaired
mnt	Mount point of the temporary file system
nfsroot	nfs folder, not used generally
opt	Added software packages
proc	Virtual file system of the kernel and process information
root	Root user directory
sbin	Executable programs for managing the system
share	Shared file directory
sys	Structure of system devices and files, providing detailed information about the kernel data
tmp	Temporary files
usr	A folder whose subfolder contains various useful applications and documents
var	Temporary files of system logs or some service programs

A common Linux root file system consists of all the directories in the root file system's top directory structure. In an embedded system, the root file system needs to be simplified. Table 4-1 describes some directories that can be ignored.

Table 4-1 Some directories that can be ignored

Directory	Description
/home, /mnt, /opt, and /root	Directories that can be expanded by multiple users.
/var and /tmp	The **/var** directory stores system logs or temporary service program files. The **/tmp** directory stores temporary user files.
/boot	**The /boot** directory stores kernel images. During startup, the PC loads the kernel from the **/boot** directory. For an embedded system, the kernel images are stored in the flash memory or on the network server instead of the root file system to conserve space. Therefore, this directory can be ignored.

NOTE
Empty directories do not increase the size of a file system. If there is no specific reason, it is recommended to retain these directories.

4.2 Creating a Root File System by Using the BusyBox

Before creating a root file system by using the BusyBox, you need to obtain the BusyBox source code, and then configure, compile, and install the BusyBox.

4.2.1 Obtaining the Source Code of the BusyBox

After the SDK is installed successfully, the complete source code of the BusyBox is saved in **osdrv** folder. You can also download the source code of the BusyBox from http://www.busybox.net.

4.2.2 Configuring the BusyBox

To configure the BusyBox, you need to go to the directory of the BusyBox, and then run the following command:

1. hisilicon$ cp osdrv/busybox/busybox-1.20.2/busybox_cfg_hi3516a_XXX osdrv/busybox/busybox-1.20.2/.config //Specify a configuration file.

   The tool chain varies according to busybox_cfg_hi3516a_XXX.

   - busybox_cfg_hi3516a_v300 corresponds to the tool chain arm-hisiv300-linux.
   - busybox_cfg_hi3516a_v400 corresponds to the tool chain arm-hisiv400-linux.
2. hisilicon$ make menuconfig

The configuration GUI of the BusyBox is the same as that of the kernel. By using the simple and intuitive configuration options, you can configure the BusyBox as required. Pay attention to the following two options that are displayed after you choose **BusyBox Settings** > **Build Options**:

```
[*]Build BusyBox as a static binary (no shared libs)
[*] Build with Large File Support (for accessing files > 2 GB)
(arm-hisiv300-linux-) Cross Compiler prefix
() Path to sysroot
(-mcpu=cortex-a7 -mfloat-abi=hard -mfpu=neon-vfpv4) Additional CFLAGS
(-mcpu=cortex-a7 -mfloat-abi=hard -mfpu=neon-vfpv4) Additional LDFLAGS
() Additional LDLIBS
```

Note the following points:

- The first option is used to determine whether to compile the BusyBox as an executable file with a static link. If the option is selected, the compiled BusyBox has a static link. In this case, the BusyBox does not depend on the dynamic library and has a large size. If the option is deselected, the compiled BusyBox has a dynamic link. In this case, the BusyBox has a small size but requires the support of the dynamic library.
- The second option is used to select a cross compiler recommended in the SDK. After configuration, you need to save the settings and close the BusyBox.

For details about the options of the BusyBox, see the *BusyBox Configuration Help*.

4.2.3 Compiling and Installing the BusyBox

To compile and install the BusyBox, run the following commands:

```
hisilicon$ make
hisilicon$ make install
```

If the BusyBox is compiled and installed successfully, the following directories and files are generated in the **_install** directory of the BusyBox:

```
drwxr-xr-x 2 lnan lnan 4096 2014-05-23 14:37 bin
lrwxrwxrwx 1 lnan lnan   11 2014-05-23 14:37 linuxrc -> bin/busybox
drwxr-xr-x 2 lnan lnan 4096 2014-05-23 14:37 sbin
drwxr-xr-x 4 lnan lnan 4096 2014-05-23 14:37 usr
```

4.2.4 Creating a Root File System

After the SDK is installed successfully, the created root file system is saved in the **osdrv/pub** directory.

If necessary, you can create a root file system based on the BusyBox.

To create a root file system, perform the following steps:

Step 1 Run the following commands:

```
hisilicon$mkdir rootbox
hisilicon$cd rootbox
hisilicon$cp -R packet/os /busybox-1.20.2/_intsall/* .
hisilicon$mkdir etc dev lib tmp var mnt home proc
```

Step 2 Provide required files in the **etc**, **lib**, and **dev** directories.

1. For the files in the **etc** directory, see the files in **/etc** of the system. The main files include **inittab**, **fstab**, and **init.d/rcS**. You are recommended to copy these files from the **examples** directory of the BusyBox, and then modify them as required.
2. You can copy device files from the system by running the **cp –R file** command or generate required device files by running the **mknod** command in the **dev** directory.
3. The **lib** directory is used to store the library files required by applications. You need to copy related library files based on applications.

----End

After the preceding steps are completed, a root file system is generated.

NOTE

If you have no special requirements, the configured root file system in the SDK can be used directly. If you want to add applications developed by yourself, you only need to copy the applications and related library files to the corresponding directories of the root file system.

4.3 Introduction to File Systems

In the embedded system, the common file systems include the compressed RAM file system (CRAMFS), journaling flash file system v2 (JFFS2), NFS, initrd, YAFFS2, and Squashfs. These file systems have the following features:

- CRAMFS and JFFS2 have good spatial features; therefore, they are applicable to embedded applications.
- CRAMFS and Squashfs are read-only file systems.
- Squashfs provides the highest compression rate.
- JFFS2 is a readable/writable file system.
- NFS is suitable for the commissioning phase at the initial stage of development.
- YAFFS2 is applicable only to the NAND flash.
- initrd uses the read-only CRAMFS.

4.3.1 CRAMFS

CRAMFS is a new file system that was developed based on Linux kernel V2.4 and later. CRAMFS is easy to use, easy to load, and has a high running speed.

The advantages and disadvantages of CRAMFS are as follows:

- Advantages: CRAMFS stores file data in compression mode. When CRAMFS runs, the data is decompressed. This mode can save the storage space in flash memory.
- Disadvantages: CRAMFS cannot directly run on flash memory, because it stores compressed files. When CRAMFS runs, the data needs to be decompressed and then copied to the memory, which reduces read efficiency. Also, CRAMFS is read-only.

For Linux running on the board to support CRAMFS, you must add the **cramfs** option when compiling the kernel. The process is as follows: After running **make menuconfig**, choose **File > systems**, select **miscellaneous filesystems,** and then select **Compressed ROM file system support** (the option is selected in the SDK kernel by default).

The mkfs.cramfs tool is used to create the CRAMFS image. To be specific, the CRAMFS image is generated after you process a created file system by using mkfs.cramfs. This procedure is similar to the procedure for creating an ISO file image using a CD-ROM. The related command is as follows:

```
hisilicon$mkfs.cramfs  ./ rootbox ./cramfs-root.img
```

Where, **rootbox** is a created root file system, and **cramfs-root.img** is the generated CRAMFS image.

4.3.2 JFFS2

JFFS2 is on the successor of the JFFS file system created by David Woodhouse of RedHat. JFFS2 is the actual file system used in original flash chips of embedded mini-devices. As a readable/writable file system with structured logs, JFFS2 has the following advantages and disadvantages:

- Advantages: The stored files are compressed. The most important feature is that the system is readable and writable.
- Disadvantages: When being mounted, the entire JFFS2 needs to be scanned. Therefore, when the JFFS2 partition is expanded, the mounting time also increases. Flash memory space may be wasted if JFFS2 format is used. The main causes of this are excessive use of log files and reclamation of useless storage units of the system. The size of wasted space is equal to the size of several data segments. Another disadvantage is that the running speed of JFFS2 decreases rapidly when the memory is full or nearly full due to trash collection.

To load JFFS2, perform the following steps:

Step 1 Scan the entire chip, check log nodes, and load all the log nodes to the buffer.

Step 2 Collate all the log nodes to collect effective nodes and generate a file directory.

Step 3 Search the file system for invalid nodes and then delete them.

Step 4 Collate the information in the memory and release the invalid nodes that are loaded to the buffer.

---End

The preceding features show that system reliability is improved at the expense of system speed. Additionally, flash chips with large capacity, the loading process is slower.

To enable kernel support for JFFS2, you must select the **JFFS2** option when compiling the kernel (the released kernel of HiSilicon supports JFFS2 by default). The process is as follows: After running the **make menuconfig** command, choose **File** > **systems**, select **miscellaneous filesystems**, and then select **Journaling Flash File System v2 (JFFS2) support** (the option is selected in the SDK kernel by default).

To create a JFFS2 file system, run the following command:

```
hisilicon$mkfs.jffs2 -d ./ rootbox -l -e 0x20000 -o  jffs2-root.img
```

You can download the mkfs.jffs2 tool from the Internet or obtain it from the SDK. **rootbox** is a created root file system. Table 4-2 describes the JFFS2 parameters.

Table 4-2 JFFS2 parameters

Parameter	Description
d	Specifies the root file system.
l	Indicates the little-endian mode.
e	Specifies the buffer size of the flash memory.
o	Exports images.

4.3.3 YAFFS2

YAFFS2 is an embedded file system designed for the NAND flash. As a file system with structured logs, YAFFS2 provides the loss balance and power failure protection, which ensures the consistency and integrity of the file system in case of power failure.

The advantages and disadvantages of YAFFS2 are as follows:

- Advantages
  - Designed for NAND flash and provides optimized software structure and fast running speed.
  - Stores the file organization information by using the spare area of the hardware. Only the organization information is scanned in the case of system startup. In this way, the system starts fast.
  - Adopts the multi-policy trash recycle algorithm. Therefore, YAFFS2 improves the efficiency and fairness of trash recycle for the loss balance.
- Disadvantages

  The stored files are not compressed. Even when the contents are the same, a YAFFS2 image is greater than a JFFS2 image.

In the SDK, YAFFS2 is provided as a module. To generate the YAFFS2 module, you need to add the path of the related kernel code to the **Makefile** in the YAFFS2 code package, and then perform compilation.

Both the YAFFS2 image and CRAMFS image can be generated by using tools. To generate a YAFFS2 image, run the following command:

```
hisilicon$ mkyaffs2image  ./ rootbox yaffs2-root.img pagesize ecctype
```

Where, **rootbox** is a created root file system, **yaffs2-root.img** is a generated YAFFS2 image, **pagesize** is the page size of the NAND flash welded on the board, and ecctype is the error checking and correcting (ECC) type of the NAND flash.

4.3.4 initrd

The initrd is equivalent to the store media and supports the formats such as ext2 and cramfs. Therefore, the kernel must support both initrd and CRAMFS. The following configurations must be complete to enable the initrd to work normally. In fact, the following configurations are complete in the SDK by default. If you change the configurations by mistake, do as follows:

- Choose **Device Drivers** > **Block devices**, and select **RAM disk support** and **Initial RAM disk (initrd) support**.
- Choose **File** > **systems**, and select **Miscellaneous filesystems** and **Compressed ROM file system support**.

To create an initrd image, perform the following steps:

Step 1 Create a CRAMFS image. For details, see section 4.3.1 "CRAMFS."

Step 2 Create an initrd image based on the created CRAMFS image by running the **mkimage -A arm -T ramdisk -C none -a 0 -e 0 -n cramfs-initrd -d ./cramfs-image cramfs-initrd** command.

----End

4.3.5 Squashfs

Squashfs is a read-only file system based on the Linux kernel and features high compression rate.

Squashfs has the following features:

- Compresses data, inode, and directories.
- Retains 32-bit UID/GIDS and file creation time.
- Supports a maximum of 4 GB file system.
- Detects and deletes duplicate files.

To use squashfs, perform the following steps:

Step 1 Create a kernel image supporting squashfs, go to the **linux-3.4.y** directory, and run the following commands:

```
cp arch/arm/configs/hi3516a_mini_defconfig .config
make ARCH=arm CROSS_COMPILE=arm-hisiXXX-linux- menuconfig (Save and then
exit)
make ARCH=arm CROSS_COMPILE=arm-hisiXXX-linux- uImage
```

Step 2 Create the image of the squashfs file system by using mksquashfs stored in **SDK/package/osdrv/tools/pc**. To use mksquashfs, run the following command:

```
./mksquashfs rootfs ./ rootfs.squashfs.img -b 64K -comp xz
```

where **rootfs** is the created root file system; **rootfs.squashfs.img** is the generated image of the squashfs file system; **-b 64K** indicates that the block size of the squashfs file system is 64 KB (which determines the actual block size of the SPI flash); **–comp xz** indicates that the algorithm for compressing the file system is XZ. You need to modify the parameters as required.

----End

5 Application Development

5.1 Compiling Code

You can choose a code compilation tool as desired. In general, the Source Insight is used in Windows, and Vim+ctags+cscope is used in Linux.

5.2 Running Applications

Before running compiled applications, you need to add them to the target machine and perform the following operations:

- Add the applications and required library files (if any) to the directories of the root file system of the target machine. Generally, applications are placed in the **/bin** directory, library files are placed in the **/lib** directory, and configuration files are placed in the **/etc** directory.
- Creates a root file system containing new applications.

NOTE

Before running applications, you need to write and read the file system. You are recommended to use the YAFFS2 or JFFS2 file system.

To create CRAMFS, YAFFS2 or JFFS2, you need to create corresponding file system (see section 4.3 "Introduction to File Systems"), burn the root file system to the specified address in the flash memory, and set the related boot parameters. In this case, new applications can run properly after Linux starts.

NOTE

To enable new applications to run automatically when the system starts, edit the **/etc/init.d/rcS** file, and then add the paths of the applications to be started.

A Acronyms and Abbreviations

A	
ARM	advanced RISC machine
C	
CRAMFS	compressed RAM file system
D	
DMS	digital media solution
E	
ELF	executable and linkable format
G	
GCC	GNU compiler collection
GNU	GNU's not Unix
I	
IP	Internet Protocol
J	
JFFS2	journaling flash file system V2
JTAG	joint test action group

P

PC	personal computer

S

SDRAM	synchronous dynamic random access memory
SDK	software development kit

Y

YAFFS2	yet another flash file system v2