AVR32737: AVR32 AP7 Linux Getting Started

32-bit AVR®
Microcontrollers

Application Note

Features

- **Linux development tools overview**
- **Introduction to the Linux boot process**
- **Compiling, running and debugging applications**

1 Introduction

This application note is aimed at helping the reader become familiar with Linux® development with the Atmel AVR®32 AP7 Application Processor.

Rev. 32091A-AVR32-02/08

2 Development Tools

The software provided with this application note requires the following components to be available.

2.1 Your personal computer (PC)

Developing Linux applications for embedded systems is most conveniently done on a Linux development host.

It is recommended that developers install, use and become familiar with Linux on the Desktop PC.

This can typically be done through the use of a pre-made Virtual Machine image running inside the existing operating system or by installing a Linux distribution (Ubuntu, Fedora, Debian or other) in a dual-boot configuration on the PC.

2.2 AVR32 Linux BSP

The AVR32 Linux BSP is a collection of everything you need to start Linux development on the AVR32 AP7 platform. It includes the AVR32 GNU Toolchain, the Linux kernel, the U-Boot boot loader as well as an assortment of useful applications. It also comes with a set of scripts to rebuild the whole environment from scratch.

2.2.1 Buildroot

Buildroot is an open-source project, used by Atmel to build its Linux board support packages for development kits and reference designs.

Buildroot is a configurable and fully automated build system for embedded systems. The main idea is that the user selects what he wants installed on the system, and buildroot takes care of compiling everything from sources, creating a custom file system image that can be programmed into flash, put on an MMC/SD card, or unpacked on an NFS server.

If you are familiar with buildroot, you may easily customize a board support package for your target board, recompile the Linux kernel, build your own toolchain or make changes to existing applications. Atmel provides binary releases to help you get started.

For more information about Buildroot, see application note AVR32003 AVR32 AP7 buildroot.

2.3 AVR32 GNU Toolchain

The AVR32 GNU Toolchain is a set of command-line utilities for AVR32 development, including a compiler, assembler, linker, debugger and several other utilities. Although the AVR32 GNU Toolchain distribution includes tools for both standalone development and Linux development, only the avr32-linux part of the toolchain is required.

A ready-to-use toolchain is available for several platforms, as well as in source code form, on http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118. It is also distributed as a part of the AVR32 Linux BSP and Buildroot.

Finally, on Debian and Ubuntu systems, the AVR32 GNU toolchain can be installed using apt-get by adding the following line to **/etc/apt/sources.list**:

```
deb http://www.atmel.no/avr32/ubuntu/dapper binary/
```

2.3.1 Using the toolchain provided by Buildroot

Buildroot includes the source code to the whole avr32-linux toolchain, and it uses this to build its own toolchain from scratch instead of relying on what's installed on the host. This usually includes several libraries not distributed with the official AVR32 GNU Toolchain packages, which may come in handy when building custom applications.

Buildroot installs its toolchain under "staging_dir" inside the Buildroot source tree[1]. To use it, simply add the "bin" directory to PATH. For example, if Buildroot was unpacked under "src" in your home directory, the following command will make the Buildroot-provided toolchain available to other projects.

```
export PATH=$HOME/src/buildroot/staging_dir/bin:$PATH
```

Note that merely running this command will only affect the shell you're currently in, so it's usually a good idea to add this line to $HOME/.profile to make it available in future sessions.

Consult the AVR32003 AVR32 AP7 Buildroot application note for information on how to build and configure buildroot.

2.4 Development board

Any development board with an AVR32 AP7 processor and at least 8MiB RAM is sufficient for Linux application development. For GUI development using Qtopia or similar toolkits, at least 16MiB RAM is recommended. ATNGW100 and ATSTK1000 both satisfy these requirements. ATSTK1000 can be upgraded to have more than the default 8MiB of RAM, but that is outside the scope of this application note.

2.5 Debugging tools

The JTAGICEmkII emulator is only necessary when debugging the kernel or the bootloader. It is not required for application-level debugging, but it can be used to reprogram on-board flash memory, typically used when changes has been done to the U-Boot bootloader.

Application debugging on Linux utilizes the debugging functionality integrated in the Linux kernel itself through a small helper program, *gdbserver*.

2.6 AVR32 Studio

AVR32 Studio is an IDE provided by Atmel which can be used to develop and debug AVR32 Linux applications, as well as standalone AVR32 applications. For more information, please consult the application note AVR32015 Getting started with AVR32 Studio.

[1] This is configurable, however.

3 Booting the system

Booting the Linux kernel is usually a two-step process. First, the boot loader, which is stored in parallel flash, runs. After doing some basic hardware initialization, like setting up external SDRAM and configuring the main system clock, it loads the Linux kernel into memory and executes it, passing some important pieces of information along.

The Linux kernel can be loaded from a variety of sources, depending on the development board:

- Directly from external flash, without going through any filesystem

```
    bootm <address where image is stored>
```

- External flash using the JFFS2 filesystem

```
    fsload
    bootm
```

- External SD- or MMC card formatted with a FAT or ext2 filesystem

```
    mmcinit
    fatload mmc 0:1 0x10400000 <filename>
    bootm
          -or-
    mmcinit
    ext2load mmc 0:1
    bootm
```

- TFTP from a server

```
    tftpboot
    bootm
          -or-
    dhcp
    bootm
```

Most of these commands take optional parameters where the load address and/or filename can be specified. By default, they use the “bootfile” and “loadaddr” environment variables, respectively. One notable exception is the “fatload” command, which requires these parameters to be specified on the command line.

3.1 The root file system

After the Linux kernel has been successfully loaded by u-boot and bootstrapped itself, it needs a root filesystem to continue. The root filesystem contains a handful of essential system files necessary to initialize other filesystems and services. At a bare minimum, it must contain the files “/sbin/init” and “/dev/console”; the former is the program that the kernel will execute right after bootstrapping itself, while the latter is where /sbin/init will send its output messages.

The root filesystem is specified through u-boot’s “bootargs” parameter, for example like this:

```
setenv bootargs root=mtd1 rootfstype=jffs2
```

3.1.1 Root file system in flash

Since the system needs a flash chip in which to store the boot loader anyway, storing the root file system in flash as well can be a simple, cheap and robust solution.

Although almost any kind of file system can be used to store files in flash, using a file system with built-in wear leveling is highly recommended. One of the most common file systems for use with flash is JFFS2: The Journaling Flash File System version 2. Note that when using JFFS2, the file system type must be specified explicitly on the command line, as shown above.

3.1.2 Root file system on a MMC or SD card

MMC and SD memory cards may offer cheap and convenient storage, and is therefore often preferable in systems where the on-board parallel flash provides too little room. Such cards are usually removable, but in some cases it may be preferable to solder them to the board.

To use a root filesystem stored on an MMC or SD card, set up the command line as follows:

```
setenv bootargs root=/dev/mmcblk0p1 rootwait
```

The last parameter, “rootwait”, is necessary because memory cards are initialized asynchronously and may not have been fully initialized when the init thread starts looking for a root file system to use. This parameter will tell the init thread to wait until the specified root device shows up. Note that with some kernels distributed by Atmel, this parameter must be specified as “rootwait=1”, but this is incompatible with the official kernels from kernel.org. The newest 2.6.22- and 2.6.23-based kernels from Atmel accept both forms.

When using a root file system stored on an MMC or SD card, make sure that the following features are linked statically into the kernel:

- MMC/SD card support
- MMC block device driver
- Atmel Multimedia Card Interface support
- The file system that the card uses (ext3 is recommended.)

Note that even though the Linux kernel supports the FAT file system and its derivatives (VFAT, FAT32, etc.), it is not recommended to use this as a root file system because it lacks support for several features commonly used by Linux systems, including hard and soft (symbolic) file links, which the “busybox” application depends on.

3.1.3 Root file system on NFS

During development, using NFS as a root file system makes it very easy to update the target as the application is being developed, especially if the development host is being used as the NFS server.

Assuming the NFS server has been correctly set up, the kernel can be told to use root file system on NFS by adding the following to the kernel command line:

```
root=/dev/nfs ip=dhcp nfsroot=192.168.1.1:/srv/stk1000
```

where “192.168.1.1” should be replaced with the IP address of the NFS server, and “/srv/stk1000” with the directory on the server to be used as a root filesystem on the target.

Now, the target filesystem can be updated on the fly by adding/deleting/modifying files under “/srv/stk1000” on the server. But beware that replacing essential system files like /lib/libuClibc-0.9.29.so on the server may crash the target system.

3.1.3.1 Setting up an NFS server

Setting up an NFS server in a Debian or Ubuntu environment may be accomplished by following the steps below. For further information, please consult the documentation of your Linux distribution.

1. Install the NFS server package.

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

2. Create the directory to be used as a root file system on the target.

   ```
   sudo mkdir /srv/stk1000
   ```

3. Populate the root file system with a tar image created by e.g. buildroot.

   ```
   cd /srv/stk1000
   sudo tar xjvf .../binaries/atstk1000/rootfs.avr32_nofpu.tar.bz2
   ```

4. Add the following line to the file **/etc/exports** using your favourite text editor.

   ```
   /srv/stk1000        10.0.0.0/255.0.0.0(rw,async,no_root_squash)
   ```

5. Refresh the NFS server configuration.

   ```
   /etc/init.d/nfs-kernel-server restart
   ```

Replace ”10.0.0.0/255.0.0.0” above with your actual network address and mask.

4 Application example: Hello World

Since Linux is a full-featured Unix-like operating system, all the difficult code dealing with low-level initialization, USARTs and other hardware devices are hidden away inside the kernel or standard libraries. This means that a simple “Hello World” application can be written in only a few lines of code:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf(“Hello World!\n”);
    return 0;
}
```

This program will also run and produce the same result on any Linux-based development host. Although the result may not come as a surprise to anyone familiar with the C language in this particular case, this can be very valuable when dealing with more complicated programs – even if a real development board isn’t available, it is still possible to do initial testing on a regular PC running Linux (or, in most cases, other operating systems like Windows®.)

4.1 Building the application

Even for simple applications like the Hello World example above, creating a Makefile to control the build process is highly recommended. A simple Makefile is shown below.

```
CROSS_COMPILE     := avr32-linux-
CC                := $(CROSS_COMPILE)gcc
CFLAGS            := -O1 –g –Wall
```

```
TARGETS          := hello
OBJ              := hello.o

all: $(TARGETS)

hello: $(OBJ)
    $(CC) $(CFLAGS) –o $@ $^
```

Add the contents shown above to a file called **Makefile**, and then run the **make** command on the development host to build the application.

4.2 Running and debugging the application

Before the application can be run or debugged, it must be transferred to the target. The default configuration of Buildroot includes the dropbear ssh server, so one way to do achieve is to use scp:

```
$ scp hello root@192.168.1.2:/tmp
```

Replace “192.168.1.2” with the IP address of the target board. The default root password is “roota”.

After the application has been transferred to the target, verify that it has the correct permissions and correct them if necessary. If the ‘x’ (execute) bit isn’t set, Linux will refuse to execute the application.

```
# ls –l hello
-rw-r--r-- 1 root root 5120 2007-12-10 16:54 hello
# ./hello
-bash: ./hello: Permission denied
# chmod a+x hello
# ls –l hello
-rwxr-xr-x 1 root root 5120 2007-12-10 16:54 hello
# ./hello
Hello World!
#
```

Now, the application does seem to work as expected, but if it didn’t, some debugging might be required. Two additional pieces of software are required in order to do this: gdbserver (on the target) and avr32-linux-gdb (on the host.)

On the target, start gdbserver like this:

```
# gdbserver :4242 ./hello
```

where “4242” is the TCP port number that gdbserver will listen to. Now, start avr32-linux-gdb on the host and connect to the target.

```
$ avr32-linux-gdb hello
GNU gdb 6.4.atmel.1.0.0
[…]
(gdb) set solib-absolute-prefix $HOME/src/buildroot/staging_dir
(gdb) target remote 192.168.1.2:4242
```

Notice the second-to-last line setting the solib-absolute-prefix variable. This tells gdb where to find any shared libraries the application may have linked against, which is necessary because the libraries may contain debugging information.

At this point, all the normal gdb debugging commands, e.g. “step”, “next” and “break”, may be used just as if the program was running on the development host.

5 References

- Embedded Linux Primer, ISBN: 978-0131679849

Headquarters

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

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
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
Avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature