International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 4, No. 3, November 2015, pp. 161~172
ISSN: 2089-4864

Development of BSP for ARM9 Evaluation Board

Vinayak Pandit K., Sanket Dessai, Shilpa Chaudhari
Department of Computer Engineering, M.S. Ramaiah School of Advanced Studies, Bangalore-560058, India

Article Info

Article history:

Received Mar 2, 2015
Revised Jun 19, 2015
Accepted Jul 12, 2015

Keyword:

ARM9
Board
BSP
Evaluation
GNU

ABSTRACT

With an increasing usage of ARM9 core for different kinds of applications ranging from data acquisition to Mobile application, there arises the need for developing ARM9 based board. To bring up this board, board supporting package (BSP) is must. Board supporting package virtualizes the platform hardware so that the different drivers can be ported easily on any hardware. The boot loader is the initial stage of firmware, which initializes the hardware components presents on the board. A universal Bootloader is chosen and is to be customized with respect to target board. In the later section bootloader is interfaced to the kernel which is obtained form an authorized distributor under general purpose license. The customized board specific routines as well drivers are ported onto the hardware. Then the compiled kernel image is ported onto the target board using a debugger and SAM-BA utility. Linux kernel has seen major releases; the basic architecture of the Linux kernel has remained more or less unchanged. The latest 2.6 version of Linux kernel is ported onto target hardware. Kernel support for many architectures and high-end I/O devices gives the independence to choose appropriate hardware for developing system. The bootloader customization is the critical step, which involves a lot of modifications in the header files. BSP components such as bootloader, kernel is compiled using GNU tool chain; obtained image is ported on target using debugger. BSP porting is a very complex task, which required knowledge of hardware and software control sequence and boot strategy of the controller.

Corresponding Author:

Sanket Dessai,
Departement of Computer Engineering,
M.S.Ramaiah School of Advanced Studies,
#470-P, Peenya Industrial Area, Bangalore, Karnataka, India. 560058.
Email: sanketdessai@msrsas.org

1. INTRODUCTION

In the wake of increase in the usage of ARM9 processors for different kinds of applications ranging from simple automation to the complex mobile applications, there arises a need for developing applications based on ARM9 processors. Therefore, in recent years, embedded systems play a key role in the era of cutting edge technology. With this thought I have taken up the project to development ARM9 based development board. Initially Development board is considering being a raw board which consist many hardware components. The Development board needs appropriate firmware or software to make the hardware work properly. Nonetheless, the requirements of current embedded applications are too heavy and complicated to be handled by most of firmware and single process software systems. Nevertheless, well-designed operating systems can handle those heavy and complicated requirements. Therefore, more and more operating systems are running on embedded systems. One killer application of Linux is to be used as the operating system of an embedded system. ARM is an architecture preferred by many embedded Linux developers. So that in this project Linux Operating System plays vital role in development environment. AT91SAM9260 controller is the heart of the development board. USB, Ethernet, SDRAM, Flash Memory, and MMC card interfaced with the controller for future application. Through out document I will discuss how

to deploy the basic system onto an ARM9 development board. Basic system is nothing but a Board Supporting Package.

A BSP or "Board Support Package" is the set of firmware software used to initialize the hardware components or devices on the board and implement the board-specific routines that can be used by the kernel and device drivers alike. BSP is thus a hardware abstraction layer gluing hardware to the OS by hiding the details of the processor and the board. The BSP hides the board- and CPU-specific details from the rest of the OS, so portability of drivers across multiple boards and CPUs becomes extremely easy. Another term that is often used instead of BSP is the Hardware Abstraction Layer (HAL).

The BSP has two components:

1. The microprocessor support*:* Linux has wide support for all the leading processors in the embedded market such as ARM, MIPS, and soon the PowerPC.
2. The board-specific routines*:* A typical HAL for the board hardware will include:
   a. Bootloader support
   b. Memory map support
   c. System timers
   d. Interrupt controller support
   e. Real-Time Clock (RTC)
   f. Serial support
   g. Bus support (PCI/ISA)
   h. DMA support
   i. Power management

2. BOARD HARDWARE AND DEVELOPMENT SUITE

The One killer application of Linux is to be used as the operating system of an embedded system. ARM is an architecture prefered by many embedded Linux developers. Development of Evaluation kit is a very challenging role, we need to consider both hardware and software specifications. BSP is a set of software packages which mainly initializes the board specific routines. BSP is thus a hardware abstraction layer gluing hardware to the OS by hiding the details of the processor and the board. The BSP hides the board- and CPU-specific details from the rest of the OS. Project deals with Linux BSP and porting issues on AT91SAM9260-EK board. Free Linux kernel version is used to implement BSP. Kernel 2.6 series version supports following devices:

- A 32-bit ARM processor
- 64 MB of SDRAM
- 8 MB of flash
- 256 Mbytes of NAND Flash memory
- 1 Serial Data-Flash
- 1 serial EEPROM
- USB(2.0) Device and Host Interface
- MMC Card Interface

2.1. Development Suite

Figure 2 illustrates the overall design of the project. Toolchain setup is initial step of the embedded linux development. The toolchain that is used on embedded systems is known as the cross-platform toolchain. The tool chain need to put together to cross-develop applications for any target includes the binary utilities, such as ld, gas, and ar, the C compiler, gcc, and the C library, glibc. Bootstrap program does the minimum hardware configuration such as power management control, PIO and clock configuration. Bootstrap is compiled by using toolchain which is already installed on host computer. In the next step is to customize the bootloader, the boot loader is expected to initialize various devices, and eventually call the Linux kernel, passing information to the kernel. BSP components can be selected using latest Kernel version. Kernel source consist the board supporting routines, required components can be selected, by using make menuconfig, which displays the console window; select required components for target board. The list of what software needs to go into the kernel and what can be compiled as modules can be specified using this step. At the end of this step, kbuild records this information in a set of known files so that the rest of kbuild is aware of the selected components. Obtained bootloader and kernel image is ported on target board using debugger. SAM-BA In system programmer is used to port the images.

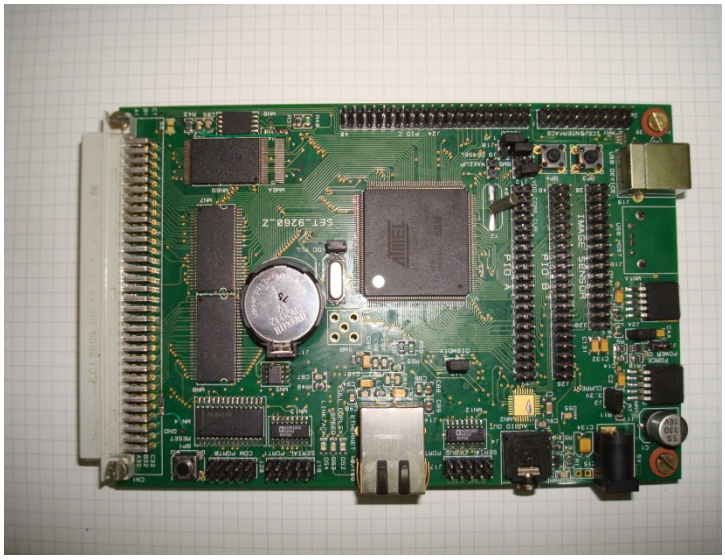

Figure 1. AT91SAM92600 Evaluation Kit

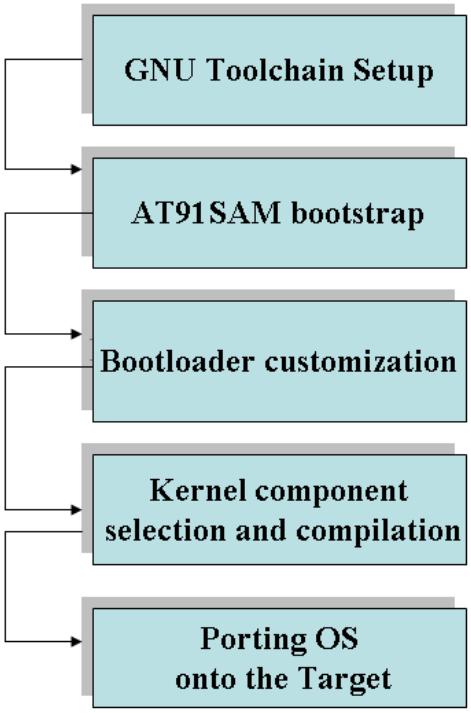

Figure 2. Development Steps

2.2. GNU Tool Setup

One of the initial steps in the embedded Linux movement is setting up the toolchains for building the kernel and the applications. The toolchain that is used on embedded systems is known as the cross-platform toolchain. The tool chain we need to put together to cross-develop applications for any target includes the binary utilities, such as ld, gas, and ar, the C compiler, gcc, and the C library, glibc.

The first step in building the tool chain is selecting the component versions we will use. This involves selecting a binutils version, a gcc version, and a glibc version. Because these packages are maintained and released independently from one another, not all versions of one package will build properly when combined with different versions of the other packages. Here we have used binutils-2.16, gcc compiler

(gcc-3.4.4), library (glibc-2.3.5) and kernel version (linux-2.6.22.6). Download all required tool chain package from ftp.gnu.org . Most tool chain build steps involve carrying out the following actions:-

1. Unpack the package.
2. Configure the package for cross platform development
3. Build the package.
4. Install the package.

Cross platform environment will ready after toolchain installation. Figure 3 and Figure 4 shows the binary utilities and kernel images.

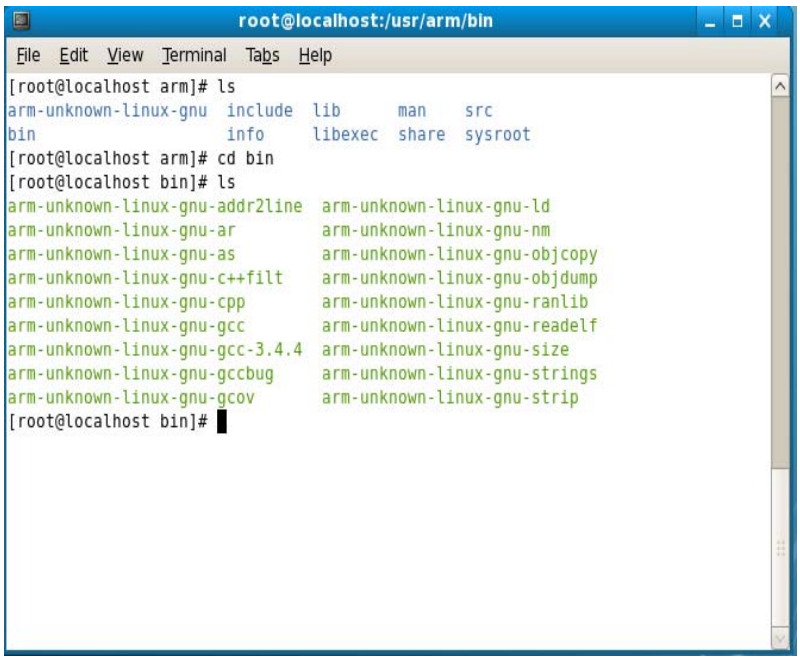

Figure 3. Binary Utility Window

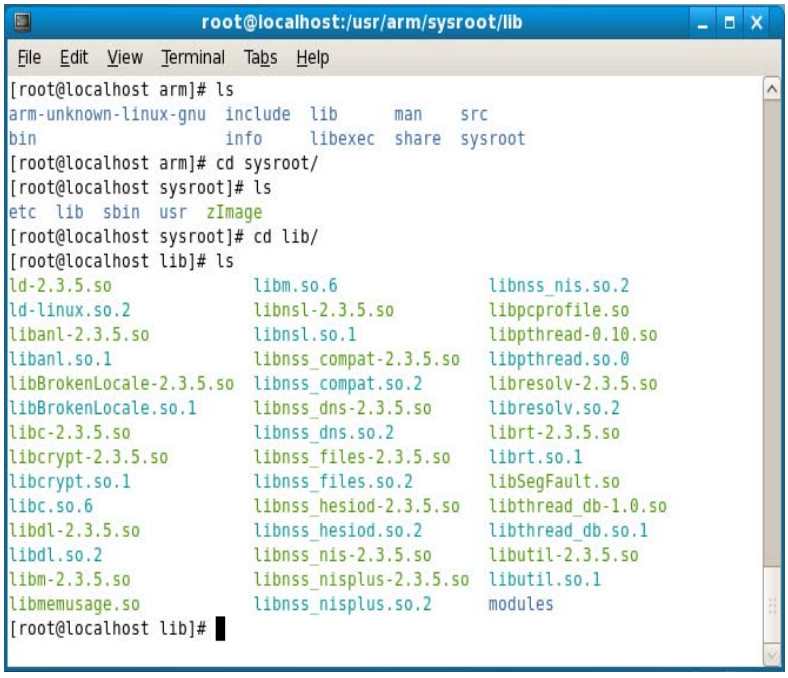

Figure 4. Kernel Image and Lib Utility Window

3. AT91SAM BOOTSTRAP FRAMEWORK

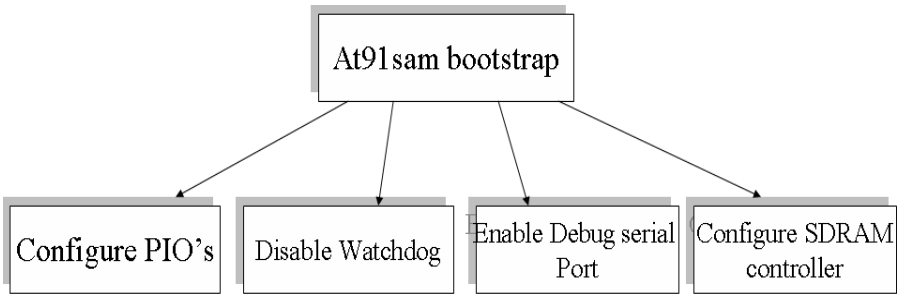

Figure 5. Bootstrap Framework

AT91Bootstrap application is considered as a first level bootloader. AT91Bootstrap is a modular application so it becomes easy to specialize the framework for a particular deployment strategy. AT91Bootstrap also provides clear examples, for a particular device, on how to perform some basic static configurations such as PMC or PIOs. AT91Bootstrap can be easily configured using a higher level protocol as shown in Figure 5. AT91Bootstrap integrates different sets of algorithms:

- Device initialization such as Clock Speed configuration, PIO settings.
- Peripheral drivers such as PIO, PMC or SDRAMC.
- Physical media algorithm such as DataFlash, NandFlash, Parallel Flash...
- File System drivers such as JFFS2 or FAT.
- Compression and Cipher algorithms
- Application Launcher for ELF, Linux.

Using this set of algorithms, it becomes easy to get a basic bootloader which, for example, is located in DataFlash and will be copied to internal SRAM by SAM-BA Boot. That bootloader could, for example, perform the processor initialization (PLLs, PIOs, SDRAMC, and SPI) then could load U-Boot from DataFlash sectors to SDRAM and finally jump to it.

3.1. Compiling an AT91 Bootstrap Project

To compile AT91Bootstrap GNU tool chain is must install. Once your toolchain is installed, instal AT91Bootstrap in a directory and cd into it

Make Command

To compile an AT91Bootstrap project:

- Go into the board directory
- Select your board by going into corresponding board directory
- Select your project by going into corresponding project directory
- Configure your project (Makefile and header file)
- Type make

> cd board/at91sam9260ek/nandflash
> make

3.2. ARM Bootloader

When system is powered on, the first piece of code called bootstrapping initializes basic hardware and then brings up OS kernel. It can be included in a bootloader or in OS directly. First of all, to make an OS, like Linux, boot, we must copy the kernel in memory and put the machine in a state that allow it to work properly in the first critical steps. The little piece of software that is in charge on this job is the boot loader. Classical boot loaders like LILO and GRUB are designed for more complex machines. For ARM based platform, few boot loaders are available, but our choice was to take a boot loader named das U-Boot, for universal boot. U-Boot supports the ARM Integrator/CP. In our project U-boot-1.1.5 version is chosen for firmware design.

There are many Bootloader is available for arm target. ARMboot, Redboot, U-boot etc. following points to be considered for bootloader selection:

- The Processor architectures and platforms that it supports
- The OS that it supports Debugging abilities, Stability and Portability Functionalities:

- Simple bootup, or with monitoring capability and lots of drivers support.
- The medium they are installed in usually ROM, EEPROM or Flash
- The storage device from which the kernel image is downloaded -- host PC, local Flash, disk, CF, etc.
- The device and protocol by which the kernel image is transmitted Serial port, USB, Ethernet, IDE, etc.
- The 'host-side' software used to transmit the kernel image

Based on these have selected das U-Boot for Universal boot. U-boot stands “Universal Bootloader”, it’s an open source bootloader available under GPL license. Source code available at http://sourceforge.net/projects/u-boot. It supports many CPU architectures and platform types like ARM, PPC, and MIPS etc.

ARM Linux cannot be started on a machine without a small amount of machine specific code to initialize the system. Most boot loaders start from the flash. They do the initial processor initialization such as configuring the cache, setting up some basic registers, and verifying the onboard RAM. Also they run the POST routines to do validation of the hardware required for the boot procedure such as validating memory, flash, buses, and so on. In the first stage control sequence will set the exception vector and handlers, next reset the control signal and set the CPU to following modes

- Set SVC Mode
- Turnoff Watchdog timers
- Disable IRQs
- Set the system clock

Once it is done control is given to cpu_init_crit, in this it has to flush the I and D cache as well as flush the TLB and MMU stuff. Further we need to set the memory according to the CPU memory mapping. After setting the memory control signal is return to start.s for further execution. Setup C environment and run on RAM and copy U-Boot code/ initialized data to RAM.

3.3. Software Configuration

Configuration is usually done using C preprocessor defines; the rationale behind that is to avoid dead code whenever possible. There are two classes of configuration variables:

❖ Configuration _OPTIONS_:

These are selectable by the user and have names beginning with "CONFIG_".

❖ Configuration _SETTINGS_:

These depend on the hardware etc. should not be meddled with if you don't know what you're doing; they have names beginning with “CFG_". Configuration depends on the combination of board and CPU type; all such information is kept in a configuration file "include/configs/<board_name>.h".

3.4. Inserting BSP in Kernel Module

The Linux BSP source code resides under arch/ and include/<asm-arm directories. Thus arch/arm will contain the source files for the arm-based board and include/asm-arm will contain the header files. Under processor directory, boards based on that CPU are categorized again into subdirectories. The important directories under each subdirectory are:

Kernel*:* This directory contains the CPU-specific routines for initializing,
IRQ set-up, interrupts, and traps routines.

mm*:* Contains the hardware-specific TLB set-up and exception-handling code

For example, ARM BSP has the two subdirectories arch/arm/kernel and arch/arm/mm that hold the above code. Along with these two directories there is a host of other subdirectories; these are the BSP directories that hold the board-specific code only. The user needs to create a subdirectory tree under the appropriate processor directory that contains the files necessary for the BSP. The next step is to integrate the BSP with the build process so that the board-specific files can be chosen when the kernel image is built. This may require that the kernel component selection process done using the “make menuconfig” command while the kernel is built is aware of the board. Linux kernel components are selected using the make config (or the make menuconfig/make xconfig) command. The heart of the configuration process is the configuration file placed under the processor directory. You need to edit the file arch/arm/kconfig.in (for the 2.6 kernel) for including AT91SAM based board components in the kernel build process.

3.5. Kernel Component Selection and Compilation

Once all above configuration is done, we have to select the kernel components by using make menuconfig, which displays the console window; you can select required components for your target board. The list of what software needs to go into the kernel and what can be compiled as modules can be specified

using this step. At the end of this step, kbuild records this information in a set of known files so that the rest of kbuild is aware of the selected components. Component selection objects are normally:

a. Processor selection
b. Board selection
c. Driver selection
d. Some generic kernel options

There are many front ends to the configuration procedure; the following figure shows the kernel configuration file. The top-level Makefile in the kernel sources is responsible for building both the kernel image and the modules. It does so by recursively descending into the subdirectories of the kernel source tree. Every architecture (the processor port) needs to export a list of components for selection during the configuration process; this includes: Any processor flavor. For example, if your architecture is defined as ARM, then you will be prompted as to which ARM flavor needs to be chosen.

- The hardware board
- Any board-specific hardware configuration
- The kernel subsystem components, which more or less remain uniform

Once the components selection is done configuration is saved to configuration file. Then zImage can be generating by using make zImage.

3.6. Memory Mapping

The following Figure 6 shows the controller memory map a first level of address decoding is performed by the Bus Matrix, i.e., the implementation of the Advanced High Performance Bus (AHB) for its Master and Slave interfaces with additional features. Decoding breaks up the 4G bytes of address space into 16 banks of 256 Mbytes. The banks 1 to 7 are directed to the EBI that associates these banks to the external chip selects EBI_NCS0 to EBI_NCS7. Bank 0 is reserved for the addressing of the internal memories, and a second level of decoding provides 1 Mbyte of internal memory area. Bank 15 is reserved for the peripherals and provides access to the Advanced Peripheral Bus (APB). Other areas are unused and performing an access within them provides an abort to the master requesting such an access.

Each Master has its own bus and its own decoder, thus allowing a different memory mapping per Master. However, in order to simplify the mappings, all the masters have a similar address decoding. Regarding Master 0 and Master 1 (ARM926 Instruction and Data), three different Slaves are assigned to the memory space decoded at address 0x0: one for internal boot, one for external boot, one after remap. The system always boots at address 0x0. To ensure a maximum number of possibilities for boot, the memory layout can be configured with two parameters. REMAP allows the user to lay out the first internal SRAM bank to 0x0 to ease development. This is done by software once the system has booted. Refer to the Bus Matrix Section for more details. When REMAP = 0, BMS allows the user to lay out to 0x0, at his convenience, the ROM or an external memory. This is done via hardware at reset. The following flowchart depicts the internal AT91SAM memory map. If BMS = 1, Boot on Embedded ROM The system boots using the Boot Program.If BMS = 0, Boot on External Memory Boot on slow clock (On-chip RC or 32,768 Hz). Boot with the default configuration for the Static Memory Controller, byte select mode, 16-bit data bus, Read/Write controlled by Chip Select, allows boot on 16-bit non-volatile memory.

The internal memory is mapped from 0x0000 0000 to 0x0FFF FFFF. Boot memory is start from 0x0000 0000 address to 0x100000. 32k Bytes ROM is mapped from 0x10 0000 to 0x10 2000 is shown. Two 4 KB Fast SRAM0 and SRAM1 are also mapped. For more detail refer datasheet of this controller. The system always boots at address 0x0. To ensure a maximum number of possibilities for boot, the memory layout can be configured with two parameters. REMAP allows the user to lay out the first internal SRAM bank to 0x0 to ease development. This is done by software once the system has booted. Refer to the Bus Matrix Section for more details. When REMAP = 0, BMS allows the user to lay out to 0x0, at his convenience, the ROM or an external memory. This is done via hardware at reset. The AT91SAM9260 matrix manages a boot memory that depends on the level on the BMS pin at reset. The internal memory area mapped between address 0x0 and 0x000F FFFF is reserved for this purpose. If BMS is detected at 1, the boot memory is the embedded ROM. If BMS is detected at 0, the boot memory is the memory connected on the Chip Select 0 of the External Bus Interface.

The 64 MB of SDRAM is made available for running both the boot loader and Linux kernel. Normally the boot loader is made to run from the bottom of the available memory so that once it transfers control to the Linux kernel, the Linux kernel can easily reclaim its memory. The Linux memory map setup is divided into four stages.

- The Linux kernel layout — the linker script
- The boot memory allocator
- Creating memory and IO mappings in the virtual address space

- Creation of various memory allocator zones by the kernel

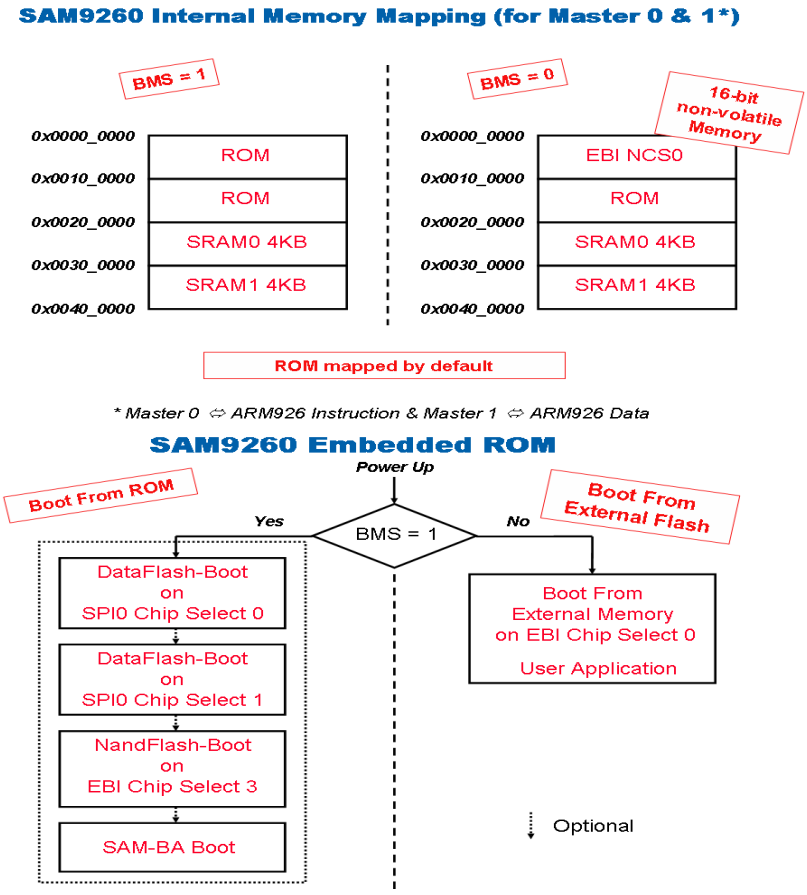

Figure 6. Internal Memory Mapping

3.7. Root File Systems

A journaling flash filesystem (JFFS2) can be chosen to store our target's root filesystem on flash memory. We must have to create the JFFS2 image by using the mkfs.jffs2 utility and a directory tree containing the filesystem that we want to load into flash. The Journaling Flash Filesystem, Version 2, (JFFS2) is a filesystem specially designed for use on embedded devices, though it can also be used in other environments. JFFS2 features automatic compression and decompression as files are written to and read from the filesystem. Journaling means that updates to files and the filesystem itself (known as filesystem metadata) are first stored in a special portion of the filesystem, called a journal or log, before actually being written to the file system. Journaling file systems generally provide higher performance and require less time at restart than non-journaled file systems because they minimize the need for the operating system to verify the integrity of its file systems while the system is rebooting.

In ordinary, non-journaled file systems, changes are made directly to files and the file system. Applications must typically wait until updates are successfully completed before they can continue execution. This can reduce overall system performance and can also result in incomplete updates if a system crashes while an application is in the middle of updating files. The default kernel configuration file includes support for the JFFS2 filesystem. If you have modified or recreated this file and plan to use a JFFS2 file system with your distribution, you should verify that JFFS2 support is still active in your kernel configuration. This support is activated by selecting **filesystem > Journaling Flash filesystem version 2 (EXPERIMENTAL)** from the configuration menu. To use a JFFS2 root filesystem, you first need to create a directory and populate it with the applications you want to include in your JFFS2 root filesystem. You must then copy the contents into a JFFS2 file system image. Creating a JFFS2 filesystem is done by using the mkfs.jffs2 utility Because of ARM9 architecture, JFFS2 implements garbage collection on MTD blocks. This scheme works fine in most cases. When the file system approaches its limits, however, JFFS2 spends an increasing amount

of time garbage collecting. Furthermore, as the file system reaches its limits, the system is unable to truncate or move files and the access to files is slowed down. If you are using JFFS2, make sure your application's data does not grow to fill the entire file system. In other words, make sure your applications check for available file system space before writing to it in order to avoid severe slow down and system crashes.
The creation of a JFFS2 image is fairly simple:-

$ cd ${PRJROOT}
$ mkfs.jffs2 -r rootfs/ -o images/rootfs-jffs2.img

We use the -r option to specify the location of the directory containing the root filesystem, and the -o option to specify the name of the output file where the file system image should be stored. The JFFS2 compression ratio is much smaller than CRAMFS. For a root file system containing 7840 KB, for example, the resulting JFFS2 image is 6850 KB in size. The compression ratio is a little above 10%.

4. CONFIGURING THE TARGET

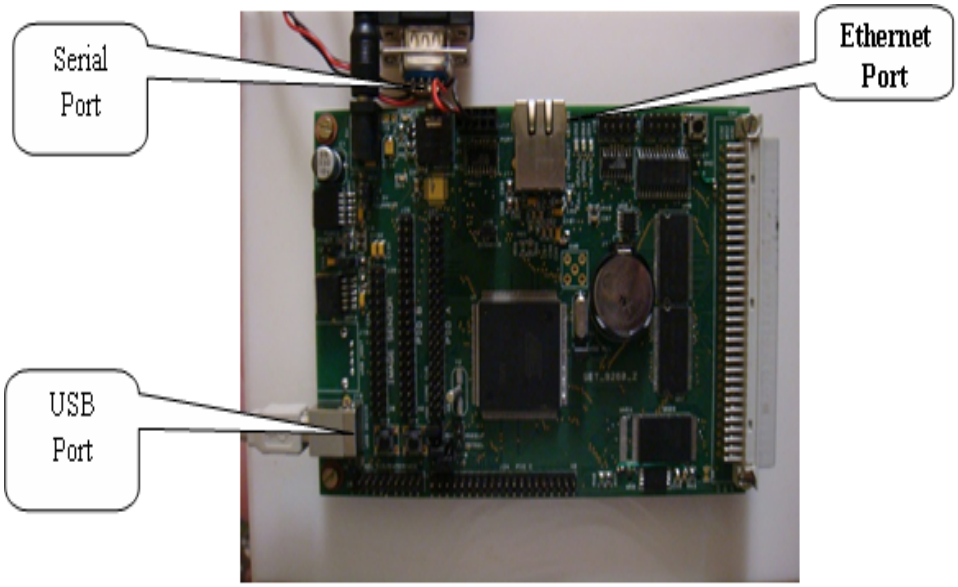

Figure 7. Configuring the Board and its Setup

Figure 7 shows the board configuration and setup, Connect the power supply included with the AT91SAM9260 connect the AT91SAM9260 to the host computer by plugging the serial cable into the board. Before porting bootloader, kernel image into DataFlash SAMBA-ISP (In System Programmer) is required, this software is freely available, download this from atmel web site. Here we have used Windows operating system to port the images.

4.1. Booting Strategy of AT91SAM Product

Several pieces of software are involved to boot a Linux kernel on SAM9 products. First is the ROM code which is in charge to check if a valid application is present on supported media (FLASH, DATAFLASH, NANDFLASH, and SDCARD). The boot sequence of Linux for SAM9260 is done in several steps:

1. Boot Program - Check if a valid application is present in FLASH and if it is the case download it into internal SRAM
2. AT91Bootstrap - In charge of hardware configuration, download U-Boot binary from FLASH to SDRAM, start the bootloader
3. U-Boot - The bootloader, in charge of download kernel binaries from FLASH, network, USB key, etc. Start the kernel.
4. Linux kernel - The operating system kernel.
5. Root Filesystem - Contains applications which are executed on the target, using the OS kernel services.

4.2. Load AT91Bootstrap on SAM9 board

AT91Bootstrap is a first step bootloader providing a set of algorithms to manage hardware initialization (GPIO, Clock, SDRAM, etc), to download your main application from specified FLASH media to main memory and to start it.

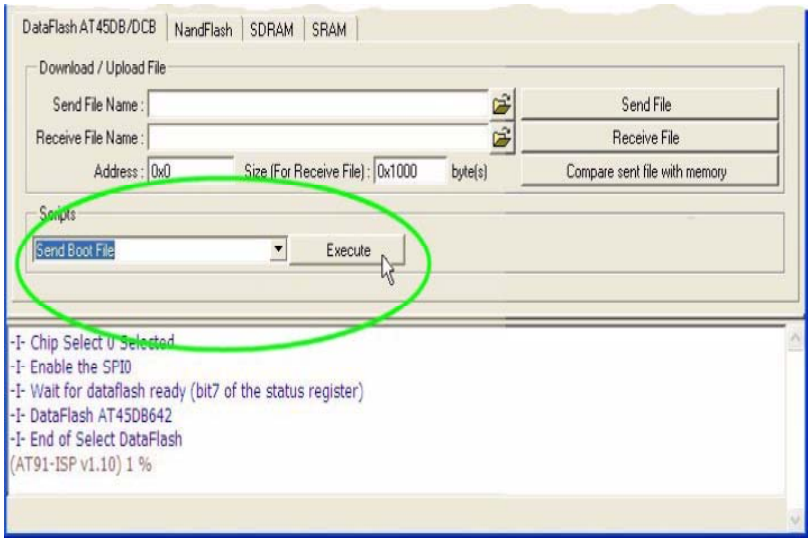

Figure 8. Bootstrap Execution Window

4.3. Load AT91Bootstrap on SAM9 board

This section describes How to load u-boot into the boot media with SAM-BA. Copy the U-Boot file into DataFlash as follows:

1. In the Send File Name field, click the Browse button.
2. Select u-boot-9260-env-dataflash.bin and click Open.
3. In the Address text field, enter 0x8000, and then click the Send File button is shown in Figure 9.

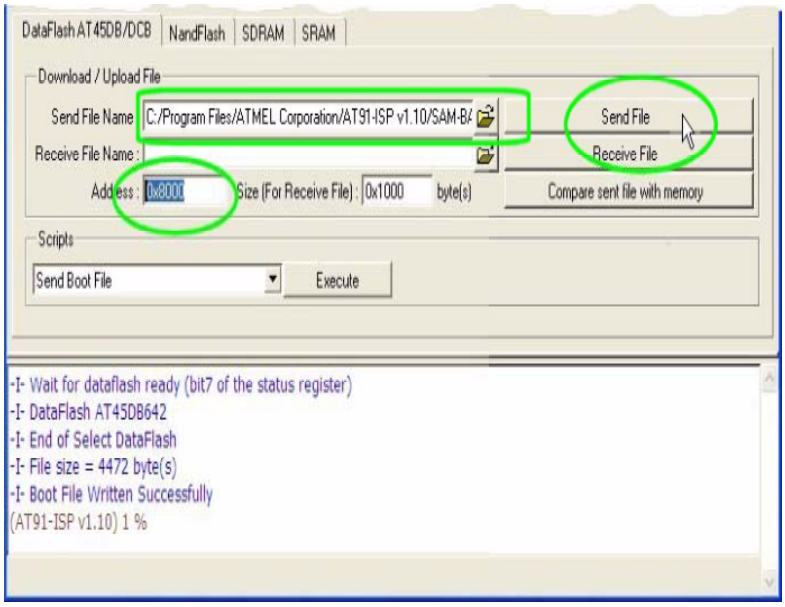

Figure 9. Bootloader Installation

After bootloader installation, when you start or reset the board, U-Boot autoboots with a display in the console like the following one. Press any key to stop the autoboot process, so that you can set the target's environment variables. Set the kernel loading address, IP address, boot argument values etc is shown below figure 9. If you do not change this envi-ronment variable, the kernel might not boot successfully, reporting a GZIP error or restarting while uncompressing the kernel image. This is particularly likely if you are trying to boot a large kernel over the network.

```
U-Boot> saveenv
Saving Environment to dataflash...
U-Boot>

U-Boot> printenv
baudrate=115200

ipaddr=10.0.0.10
serverip=10.0.0.2
bootfile=uImage.at91sam9260
ethaddr=00:03:04:05:06:07
bootargs=console=ttyS0,115200 ip=dhcp root=/dev/nfs
loadaddr=0x21400000

Environment size: 230/16380 bytes
U-Boot>
```

Figure 10. U-boot Environment Setup

4.4. Downloading and Booting the Kernel

Connect your Ethernet cable to the target board and to your host computer. Then, return to the U-Boot> prompt in U-Boot and use the tftp command to download the kernel image. After a few seconds, the bootloader transfers the kernel image to the board and the prompt returns. When the download is complete, use the bootm command to boot the kernel.

```
U-Boot> bootm

## Booting image at 0x21400000 ...
   Image Name:   Linux-2.6.22
   Image Type:   ARM Linux Kernel Image(uncompressed)
   Data Size:    1365008 Bytes =  1.3 MB
   Load Address: 20008000
   Entry Point:  20008000
   Verifying Checksum ... OK
OK
```

```
U-Boot> tftp uImage.at91sam9260
TFTP from server 10.0.0.2; our IP address is 10.0.0.10
Filename 'uImage.at91sam9260'.
Load address: 0x21400000
Loading:#################################################################
#################################################################
        ##########
done
Bytes transferred = 1365072 (14d450 hex)
U-Boot>
```

Figure 11. Kernel Image Compilation

5. CONCLUSION

Board supporting package was customized and successfully implemented on ARM9 evaluation kit. The Cadenux development environment consists of the GUI-based BSP configuration tool, memconfig. This is a one stop for building all the components of BSP such as the Linux kernel, bootloader, and file systems. The BSP porting is a complicated task. It involves good understanding of the hardware, Linux Kernel, bootloader, memory mapping and specialized knowledge of debugger operation. It was realized that the understanding of the Hardware and Software flow maps was essential. Note that both arm-elf- and arm-linux-ARM GCC cross-compiler types are suitable for u-boot building Bootloader. Choosing the right Linux kernel version is very important for the successful completion of this project. Because old kernel version will not provide the quality of feature, By using a Linux-based host development environment, most of the applications that are to be run on the target hardware can be tested on a Linux host, reducing time to port applications. A journaling file system built specifically for storage on flash, JFFS2 was added only for the 2.6 kernel. The 2.6 kernel can be considered a good port for using NAND flash. So we have used JFFS2 as a file system.

REFERENCES

[1] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. “Porting Choices to ARM Architecture Based Platforms”, Campbell, University of Illinois at Urbana-Champaign, http://choices.cs.uiuc.edu/, Draft of 2007/01/12 15:12.
[2] Inder M. Singh, "Embedded Linux the 2.6 kernel is ideal for specialized devices of all sizes", white paper, Aug-2004, permission of Quarter Power Media, publishers of Linux Magazine, www.linuxmagazine.com
[3] Steven L. Rodgers and James B. Calvin, Jr. "The Complleing Case for Open Source Embedded Tools” White Paper, 8 December 2002.
[4] William von Hagen, “Migrating custom Linux installations to 2.6”, White Paper, 5 March 2004.
[5] William von Hagen and David, “Overview of Device Drivers and Loadable Kernel Modules”, white paper, 13 Feb 2004.
[6] David Woodhouse, “JFFS: The Journalling Flash File System”, White Paper, Red Hat, Inc. 13 Jan 2005
[7] Katayama, T. Saisho, and K.Fukuda, “Prototype of the device driver generation system for UNIX-like operating systems”, International Symposium on Linux System, pp. 302-310, March 2000.
[8] AT91SAM9260 Datasheet.
[9] Karim Yaghmour, "Building Embedded Linux System", First Edition, O'Reilly and Associates Inc., 2003
[10] P. Raghavan, Amol Lad and Sriram Neelakandan “Embedded Linux System Design and Development”, First Edition, Auerbach Publications, 2006.