

Developing an ARM based GNU/Linux Operating System for Single Board Computer – Cubietruck

S.Pravin Kumar^{#1}, G.Pradeep^{#2}, G.Nantha Kumar^{#3}, C.Dhivya Devi^{#4}

^{1,2}Student

^{3,4}Assistant Professor

[#]Department of CSE, Anjalai Ammal Mahalingam Engineering College,
Kovilvenni, Tamil Nadu, India

Abstract—The design and implementation of a Monolithic-Kernel Single Board Computer (SBC) - Cubietruck GNU/Linux-like operating system on ARM platform in technical details, including boot loader design - UBOOT, building the Kernel - uImage, design of root file system and init process. The Single Board Computer Operating System (SBC OS) is developed on Linux platform with GNU tool chain. The SBC OS can be used for both SBC system application development and related curriculum teaching. Single Board Computer like embedded system related curriculums are already become necessary components for undergraduate computer majors. The system is mainly designed for the purpose of technical research and curriculum based teaching and students to learn, study and more readable, of which the source codes can be provided to students, guiding them to design tiny operating system on ARM platform from scratch.

Keyword-Single Board Computer, UBOOT, Monolithic-Kernel, Init Process, ARM, UImage

I. INTRODUCTION

With the rapid developments of electronic and computer technologies, Single Board Computers have already become more and more popular in the wide variety of fields. As the core component of computer system as well as embedded system, operating system has been playing a very important role [1].

For the purpose of technical research and curriculum based teaching, the author's of this paper develops a Monolithic-Kernel Single Board Computer GNU/Linux like operating system on ARM platform. The advantage of the system is described in the following.

A. Monolithic-Kernel Architecture

Unlike micro-kernel in MINIX operating system which slower processing system due to additional message passing, the SBC OS is designed as a Monolithic-Kernel analogous to the famous GNU/Linux. With such kind of architecture, the faster processing, the modularity and structure can be improved significantly therefore is suitable for Single Board Computers.

B. For both Single Board Computer Development and Curriculum Teaching

On one hand, the essential techniques related to operating systems and ARM machines are involved, e.g., boot loader design - UBOOT, building the Kernel - uImage, design of root file system and init process. All of these are obviously helpful for development on ARM based Single Board Computer as well as for students to learn and study [2]. On the other hand, the SBC OS is designed more readable, of which the source codes can be provided to students, guiding them to design tiny Single Board Computer operating system on ARM platform from scratch.

C. Modularity and Structure

Each separate functionality should be found in a separate module, and the file layout of the paper should reflect this. Depending on their function, many capabilities can also be built into optional, runtime-loadable, modular components. These can be loaded later when the particular capability is required. Within each module, complex functionality is subdivided in an adequate number of independent functions. These (simpler) functions are used in combination to achieve the same complex end-result.

II. ARCHITECTURE OF THE SBC OS

At the top of our SBC OS contains the user or application space where the user applications are executed. Below the user space is the Kernel space where the SBC OS Kernel exists. Fig. 1 represents the architecture of the SBC OS.

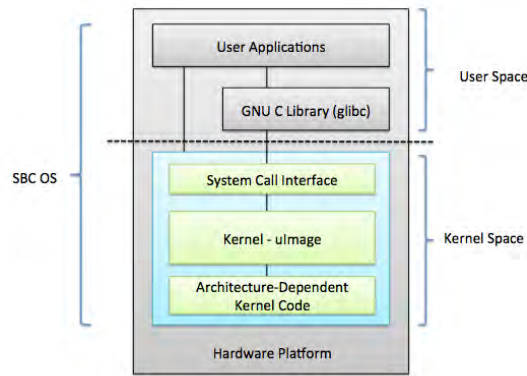


Fig. 1. Architecture of SBC OS

Our SBC OS also contains a GNU C Library (glibc) which provides the system call interface that connects to the SBC OS Kernel and provides the mechanism to transition between the user or application space and the SBC OS Kernel. This is important because the Kernel and user application occupy different protected address spaces while each user or application space process occupies its own virtual address space, SBC OS Kernel occupies a single address space.

The SBC OS Kernel can be further divided into three gross levels. At the top is the system call interface, which implements the basic functions such as read and write. Below the system call interface is the SBC OS Kernel code, which can be more accurately defined as the architecture-independent Kernel code. This code is common to all of the processor architectures supported by SBC OS. Below this is the architecture-dependent code, which forms what is more commonly, called a BSP (Board Support Package). This code serves as the processor and platform-specific code for the given architecture.

III. DESIGN AND IMPLEMENTATION

A. Boot Loader Design – UBOOT

U-Boot is an open-source, cross-platform boot loader that provides out-of-box support for hundreds of Single Board Computers and many CPUs, including PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze, and x86.

Our SBC OS normally reside in large-capacity devices such as hard disks, CD-ROMs, USB disks, network servers, and other permanent storage media [8]. When the processor is powered on, the memory does not hold an operating system, so special software is needed to bring the SBC OS into memory from the media on which it resides. This software is normally a small piece of code called the *boot loader*. On a desktop PC, the boot loader resides on the master boot record (MBR) of the hard drive and is executed after the PC's *basic input output system* (BIOS) performs system initialization tasks. In a Single Board Computer, the boot loader's role is more complicated because these systems rarely have BIOS to perform initial system configuration. Although the low-level initialization of the microprocessor, memory controllers, and other board-specific hardware varies from board to board and CPU to CPU, it must be performed before an OS can execute. U-Boot is highly customizable to provide both a rich feature set and a small binary footprint. U-Boot has a command shell (also called a monitor) for working with U-Boot commands to create a customized boot process.

Although a boot loader runs for a very short time during the system's startup and is mainly responsible for loading the Kernel, it is nevertheless a very important system component.

At a minimum, a boot loader for our SBC OS performs these functions:

- Initializing the hardware, especially the memory controller
- Providing boot parameters for the SBC OS
- Starting the SBC OS

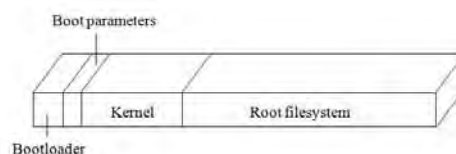


Fig. 2. Storage of Boot Loader, Kernel and Root File System in the ROM Memory (Flash)

Compiling the Boot Loader – UBOOT

Boot Loader compilation for the SBC OS can be done using the following command:

```
$ make clean && make cubietruck CROSS_COMPILE=arm-linux-gnueabihf-
```

B. Building the Kernel – uImage

The Kernel is the most fundamental software component of SBC OS. It is responsible for managing the bare hardware within our chosen target system and bringing order to what would otherwise be a chaotic struggle between each of the many various software components on a typical system.

In essence, this means the SBC OS Kernel is a resource broker [5]. It takes care of scheduling use of (and mediating access to) the available hardware resources within a particular SBC OS. Resources managed by the Kernel include system processor time given to programs, use of available RAM, and indirect access to a multitude of hardware devices—including those custom to our chosen target. The Kernel provides a variety of software abstractions through which application programs can request access to system resources, without communicating with the hardware directly.

The precise capabilities provided by any particular build of the SBC OS Kernel are configurable when that Kernel is built. Kernel configuration allows us to add and remove the peripheral devices. Depending on their function, many capabilities can also be built into optional, runtime-loadable, modular components. These can be loaded later when the particular capability is required. Fig. 3 represents the architecture of the SBC OS Kernel.

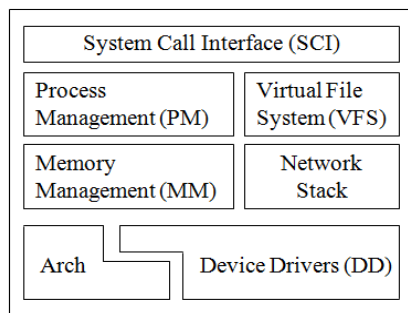


Fig. 3. Architecture of the SBC OS Kernel

1) *Configuring the Kernel:* We need to configure the options that are needed to have it in our Kernel before building it. The target is to have an appropriate .config file in our Kernel source distribution. Depending on our target, the option menus available will change, as will their content [6]. Some options, however, will be available no matter which embedded architecture we choose. After the environmental setup, make menuconfig runs a text-based menu interface is shown in Fig. 4.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

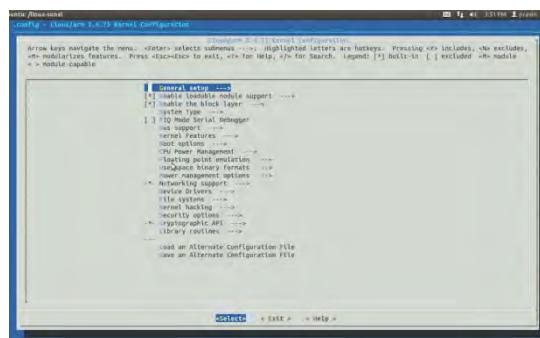


Fig. 4. Snapshot of Menuconfig

- <*> indicates that the feature is on
- <M> indicates that the feature is configured as mobile
- <> indicates that the feature is off

2) *Compiling the Kernel and Modules:* After saving the Kernel configuration in the kernel root directory, our main goals are to compile the uImage compressed Kernel and uImage Kernel modules for our SBC OS using the following commands:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- uImage -j4
```

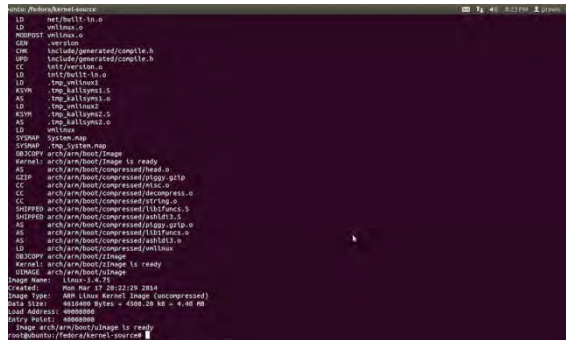


Fig. 5. Snapshot of SBC OS Kernel Compilation - uImage

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules -j4
```

3) *Installing Modules:* Before we install and boot from our new SBC OS Kernel, we should put the new Kernel modules in /lib/modules with the following command:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules_install
```

Our new modules should appear in /lib/modules/version, where version is new Kernel version of the SBC OS

C. *Design of Root File System*

One of the last operations conducted by the Linux Kernel during system startup is mounting the root file system. The Linux Kernel itself doesn't dictate any file system structure, but user space applications do expect to find files with specific names in specific directory structures. Therefore, it is useful to follow the de facto standards that have emerged in Linux systems.

Each of the top-level directories in the root file system has a specific purpose. Many of these, however, are meaningful only in multiuser systems in which a system administrator is in charge of many servers or workstations employed by different users. In most embedded Linux systems, where there are no users and no administrators, the rules for building a root file system can be loosely interpreted. This doesn't mean that all rules can be violated, but it does mean that breaking some of them will have little to no effect on the system's proper operation. Interestingly, even mainstream commercial distributions for workstations and servers sometimes deviate from the de facto rules for root file systems.

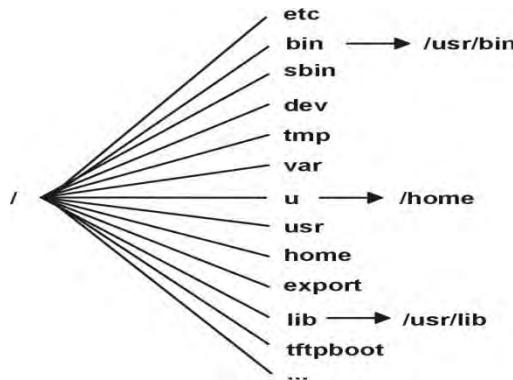


Fig. 6. Structure of Root File System

Most UNIX file system types have a similar general structure [3], although the exact details vary quite a bit. The central concepts are *superblock*, *inode*, *data block*, *directory block*, and *indirection block*. The superblock contains information about the file system as a whole, such as its size (the exact information here depends on the file system). An inode contains all information about a file, except its name. The name is stored in the directory, together with the number of the inode. The basic file system can be installed using the following command.

```
$ debotstrap --no-check-gpg --arch=armhf --foreign wheezy
```

TABLE 1
Root File System Directories and Contents

Directory	Content
<i>bin</i>	Essential user command binaries
<i>boot</i>	Static files used by the boot loader
<i>Dev</i>	Devices and other special files
<i>etc</i>	System configuration files, including startup files
<i>home</i>	User home directories
<i>lib</i>	Essential libraries, such as the C library, and kernel modules
<i>Media</i>	Mount points for removable media
<i>mnt</i>	Mount points for temporarily mounted file systems
<i>opt</i>	Add-on software packages
<i>proc</i>	Virtual file system for kernel and process information
<i>root</i>	Root user's home directory
<i>sbin</i>	Essential system administration binaries
<i>Sys</i>	Virtual file system for system information and control (buses, devices, and drivers)
<i>Tmp</i>	Temporary files
<i>Usr</i>	Secondary hierarchy containing most applications and documents useful to most users, including the X server
<i>Var</i>	Variable data stored by daemons and utilities

D. Init Process and Runlevels

In conventional Linux systems, *init* is the first process started when a Linux Kernel boots, and it is the ancestor of all processes [7]. Its primary role is to start appropriate service processes for the “state” the system is to run in at boot and to shutdown/start appropriate services if the system state changes (such as changing to the halt/shutdown state). It can also create consoles and respond to certain types of events.

Init's behavior is determined by its configuration file */etc/inittab*. Lines in */etc/inittab* have the following syntax:

id:runlevels:action:process

where:

id — 1–4 (usually 2) character name for the line, totally arbitrary;
runlevels — a list of runlevels the line applies to;
action — what *init* is to do and/or under what conditions;
process — program/command to be run.

The ID for a line is completely arbitrary though there are some conventions for standard lines. For example, the IDs for lines that start services are “ln” where n is the runlevel. UNIX had runlevels 0–6, and these are the most commonly used runlevels, though Linux can use 0–9. The runlevel 1 can also be denoted by “s” (on some systems s may cause password checking that 1 does not). The runlevels component of an *inittab* line can be a list of runlevels. E.g., 123 would mean the line applies to runlevels 1, 2, and 3.

Actions are a bit hard to explain, as some action specs are really what *init* should do while others refer to a condition under which the process spec should be run. Some key actions are:

initdefault — runlevel to enter at boot (process ignored);
sysinit — process run first at boot (runlevels ignored)
wait — run process when runlevels entered then wait for termination;
respawn — restart process when terminated;
ctrlaltdel — run process when *init* receives the SIGINT signal from CTRL-ALT-DEL key combination.

Many Linux distros are set up so that the “state” of the running system is determined by its runlevel. By state we mean a set of services/processes that are to be running. “Redhat-family” distros tend to make significant use of runlevels, while “Debian-family” distros make less use of them. Here are the typical runlevels and their meaning for Redhat-family distros:

0 — halt system
1 — single user mode (no GUI)

- 2 — multiuser mode, no networking (no GUI)
- 3 — multiuser mode, networking (no GUI)
- 4 — unused
- 5 — multiuser mode (GUI/X11) //FUTURE WORK
- 6 — reboot system

The meaning of runlevels 0, 1, and 6 are quite standard among distros, but the use of other runlevels can vary even among Redhat-family distros. Debian-family systems generally do not distinguish among runlevels 2–5 (all the same). The default runlevel—i.e., the runlevel the system will boot into—is specified in the file `/etc/inittab` via a line that looks like:

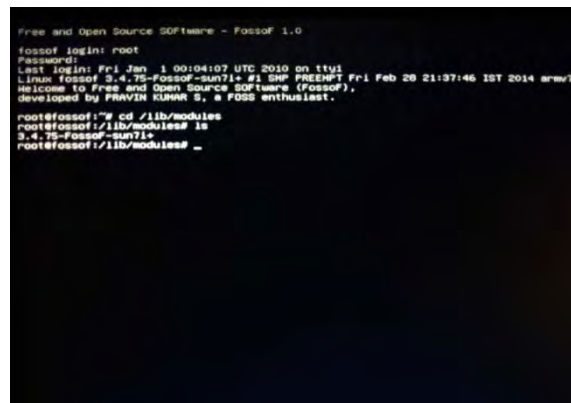
```
id:5:initdefault:
```

The current runlevel of a system can be changed by using the `init` (or `telinit`) command, e.g., `init 3` (this would mean to change to runlevel 3, which might terminate X and put you into a non-GUI console).

IV. DEMO OF THE SBC OS AND SINGLE BOARD COMPUTER – CUBIETRUCK

FOSSOF 1.0 – Free and Open Source SOFtware is a SBC OS, which can be obtained from <https://github.com/gselvapraavin/FossoF> developed by the author's of this paper. It can be cloned by using the following command:

```
$ sudo apt-get -y install git
$ cd ~
$ git clone https://github.com/gselvapraavin/FossoF
$ chmod +x ./FossoF/fossof.sh
$ cd ./FossoF
$ ./fossof.sh
```



```
Free and Open Source Software - FossoF 1.0
fossof login: root
Password:
Last login: Fri Jan 1 00:04:07 UTC 2010 on tty1
Linux fosssof 3.4-TS-FossoF-sun71 #1 SMP PREEMPT Fri Feb 28 21:37:46 IST 2014 armv7l
Welcome to Free and Open Source Software (FossoF),
developed by PRAVIN KUMAR S, a FOSS enthusiast.

root@fossof:~# cd /lib/modules
root@fossof:/lib/modules# ls
3.4-TS-FossoF-sun71
root@fossof:/lib/modules# _
```

Fig. 7. Demo of the SBC OS

The compiled image will be located in `/tmp/FossoF/output/debian_rootfs.raw.gz`. For writing it into SD card decompress it and use Image Writer (Windows) or DD-it in Linux by using the following command:

```
$ dd bs=1M if=FossoF_x.x_vga.raw of=/dev/sdx
```



Fig. 8. Single Board Computer – Cubietruck

Cubietruck is Single Board Computer, is the 3rd board of Cubieteam, also name it Cubieboard3 [4]. It's a new PCB model adopted with Allwinner A20 main chip, just like Cubieboard2. But it is enhanced with some features, such as 2GB memory, VGA display interface on-board, 1000M nic, WIFI+BT on-board, support Li-battery and RTC, SPDIF audio interface.

V. CONCLUSION AND FUTURE WORK

The purpose of the SBC OS described in this paper is to design and implement a Monolithic-Kernel SBC operating system. Some further improvements will be developed on next stage, including following main components:

A. X Window System

A X Window System will be added to this SBC OS, The graphical user interface on Linux systems is based on the X Window System. Today, X Window System is currently at version 11 revision 6 and is properly known as X11R6, X11, or just X. The following init process can be achieved in future.

init 5 — multiuser mode (GUI/X11)

B. Multimedia Support

A Multimedia Support will be added to this SBC OS, which offers people to listen to music. Granted, a computer is an expensive radio or CD player if that were all it was used for. Many people listen to the music while they work, like yours truly. This is a far cry from the muted sounds that emanated from the internal speakers of older computers. The computer's capability to process sound has grown drawn automatically. Today, sound cards not only play back music, they can help to create music as well – through the Musical Instrument Digital Interface (MIDI) port. This is just one of the capabilities of the modern sound card. The quality of a recording depends on the number of digital bits that are used when converting from sound to digital data – generally 8 or 16 bits. Another factor affecting quality is the rate at which the sound is sampled. The sample rate range is 5kHz to 44.1 kHz, or 5,000 to 44,100 samples per second. The faster the sample rate, the better the quality of the recording, which also means the larger the size of the resulting data file.

REFERENCES

- [1] Bo Qu and Zhaozhi Wu, "Design of Mini Multi-Process Micro-Kernel Embedded OS on ARM", Proceedings of the 2nd International Symposium on Computer, Communication, Control and Automation, 2013, pp. 0295.
- [2] Bo Qu and Zhaozhi Wu, "Design of ARM Based Embedded Operating System Micro Kernel", Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering, 2013, pp. 2040.
- [3] A.S. Tanenbaum and A.S. Wookhull, Operating Systems: Design and Implementation, 3E, Prentice Hall, Inc ., 2008.
- [4] Cubieboard / Cubietruck Debian Wheezy SD Card image, 2013, (<http://www.igorpecovnik.com/2013/12/24/cubietruck-debian-wheezy-sd-card-image>)
- [5] William Stallings, Operating Systems: Internals and Design Principles, 6E, Prentice Hall, Inc., 2009.
- [6] How to make a Cubieboard System, 2013, (http://www.dl.cubieboard.org/docs/How_to_make_a_cubieboard_system.pdf)
- [7] init: Booting, Runlevels, Services, GUI, Replacements, 2012, (<http://www2.cs.siu.edu/~cs406/handouts/init.pdf>)
- [8] U-Boot Reference Manual, 2007, (http://ftp1.digi.com/support/documentation/90000852_c.pdf)

AUTHOR PROFILE



S.Pravin Kumar, a FOSS enthusiast, and Student - Bachelor of Engineering in Computer Science at Anjalai Ammal Mahalingam Engineering College. His research interests are in Operating System Internals, Open Source Tools and Wireless Network Systems.



G.Pradeep, Student – Bachelor of Engineering in Computer Science at Anjalai Ammal Mahalingam Engineering College. His research interests are in Operating Systems and Graphics and Multimedia.



G.Nantha Kumar received the B.E degree in Computer Science and Engineering from Madurai Kamaraj University in 1999 and received his M.E degree in Computer Science and Engineering from Anna University Chennai in 2004. He is a research scholar of Anna University Chennai. Currently; he is an Assistant Professor at Anjalai Ammal Mahalingam Engineering College Kovilvenni. His research interests are in Mobile computing and Wireless Network Systems.



C.Dhivya Devi received the B.E degree in Computer Science and Engineering from Anna University in 2011 and received her M.E degree in Computer Science and Engineering from Sastra University, Thanjavur in 2013. She is an Assistant Professor at Anjalai Ammal Mahalingam Engineering College Kovilvenni. Her research interests are in Wireless Sensor Networks and Network security.