ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 10, October 2013

Configuration of Real Time Linux on EP93XX Platform

Geeta Patil[1], Vaishali Ingale[2], Ashwini Sapkal[3], Rahul Desai[4]

Asst Professor, Information Technology Department, Army Institute of Technology, Pune, Maharashtra, India

Abstract: This paper describes the porting of RTLinuxPro on EP93xx platform which contains ARM920T based processor. This paper also discusses designing real-time application and executing them on this platform. RTLinuxPro is licensed software from FSM Lab specifically designed for real time embedded applications.

Keywords: RTLinux, EP93XX, FSMLab, TFTP etc.

I. INTRODUCTION

FSMLab (Finite State Machine Labs Inc.) RTLinux (Real Time Linux) is a small real time operating system that is used for systems where precise timing, down to a few microseconds or less, is needed [1]. For example, RTLinux is used to run telescopes (at Kitts Peak), instruments (by NASA and many others), machine tools (NIST), high speed network switches (Huawei and Alcatel), and all sorts of other things where "usually fast enough" is not good enough.

RTLinux follows the POSIX 1003.13 standard and so its API is pretty close to ordinary POSIX (Portable Operating System Interface) threads/signals [2]. RTLinux kernel applications look like threads and signal handlers running on a tiny operating system close to the bare machine. Real-time applications on RTLinux are almost always made up of two parts: a real time kernel and the parts that do data logging, non-real time networking, GUIs, data analysis or display, and anything else that does not need precise timing. This non-real time part runs in either Linux or BSD UNIX and uses the ordinary programming interface of these systems. One of the big advantages of RTLinux is that real time programmers can use a simple, very efficient, threads/signals environment for hard real time software, use a regular operating system with many features for everything else, and glue the parts together. A typical application might use a standard database, a Perl-script, and a data analysis package - all driven by a real time thread

Real time operating systems are systems, which respond to any external unpredictable event in a predictable way and with strict timing constraints. Real time operating systems are required to be very deterministic.RTLinux uses a patented process to run a general purpose operating system like Linux or BSD Unix as its lowest priority thread and to make sure that this general purpose operating system can always be preempted (interrupted) whenever a real time operation needs to run.

The RTCore OS is a small, hard real-time operating system that can run Linux or BSD UNIX as an application server [5]. This allows a standard operating system to be used as a component of a real-time application. A typical RTCore application consists of one or more real-time components that run under the direct control of the real-time kernel and a set of non real-time components that run as user-space programs.

RTLinux is an extension of Linux to a Real time OS originally developed by V. Yodaiken at the New Mexico Institute of Mining and Technology. Now, RTLinux is available as a community supported free version, called RTLinux Free, as well as a commercial version from FSMLabs, called RTLinux Pro. RTLinux almost runs on any platform including x86, Power PC, Alpha, ARM etc. RTLinux also works on i486. The MiniRTL release fits RTLinux, Linux and some applications on a single floppy disk and runs in 4M of memory. The RTLinux software for x86 is available on internet for free download.

RTLinux supports hard real-time (deterministic) operation through interrupt control between the hardware and the operating system. Interrupts needed for deterministic processing are processed by the real-time core, while other interrupts are forwarded to the non-real time operating system. The operating system (Linux) runs as a low priority thread. First-In-First-Out pipes or shared memory can be used to share data between the operating system and the real-time core.

Three major attributes make RTCore work: It disables all hardware interrupts in the GPOS. It provides interrupts via interrupt emulation. It runs full featured non real-time Linux (or BSD) as the lowest priority task. It is the idle task of the RTOS, meaning that it is run whenever the real-time system has nothing else to execute.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 10, October 2013

The main problem in adding hard real-time capabilities to GPOD is that the disabling of interrupts is widely used in the kernel for synchronization purposes. The strategy of disabling interrupts in critical code sequences (as opposed to using synchronization mechanisms like semaphore or Mutex), is quite efficient. It also makes code simpler, since it does not need to be designed to be re-entrant. But disabling the interrupts for long period results in lost events.

To maintain the structure of the GPOS kernel while providing real-time capabilities, one must provide an ”interrupt interface” that gives full control over interrupts, but at the same time appears to the rest of the system like regular hardware interrupts. This interrupt interface is essentially an interrupt emulation layer, and is one of the core concepts in RTCore. Interrupt emulation is achieved by replacing all occurrences of STI and CLI with emulation code. This introduces a software layer between the hardware interrupt controller and the GPOS kernel, allowing the real-time kernel to handle interrupts as needed by real-time code, but still allowing the general purpose OS to handle them if there is a need.

Interrupts that are not destined for a real-time task must be passed on to the GPOS kernel for proper handling when there is time to deal with them. In other words, RTCore has full control over the hardware and non real-time GPOS sees soft interrupts, not the real interrupts. Hardware interrupt interaction is simply emulated in the GPOS. This means that there is no need to recode GPOS drivers, provided there are no hard-coded instructions in binary-only drivers that bypass the emulation.

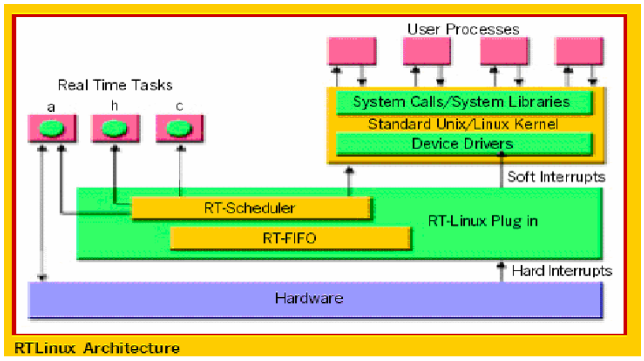

Fig 1: RTLinux Kernel

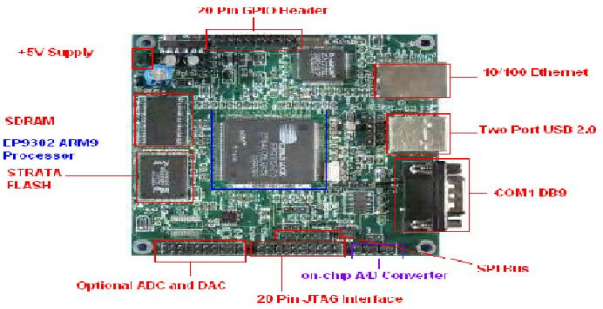

Fig 2: EP93XX SBC

The EP93xx is a high-performance system-on-chip design that includes a 200 MHz ARM9 processor and is ideal for a range of industrial and consumer electronic applications [3]. The EP9302 features an advanced ARM920T processor design with a memory management unit (MMU) that supports Linux, Windows CE and many other embedded operating systems [4]. The ARM920T's 32-bit microcontroller architecture, with a five-stage pipeline, delivers impressive performance at very low power. Designers of industrial controls, internet radios, digital media servers, audio jukeboxes, thin clients, set-top boxes, point-of-sale terminals, biometric security systems and GPS devices will benefit from the EP9302's integrated architecture and advanced features. The basic block diagram of EP9302 SBC [6] is as shown in Figure 3.

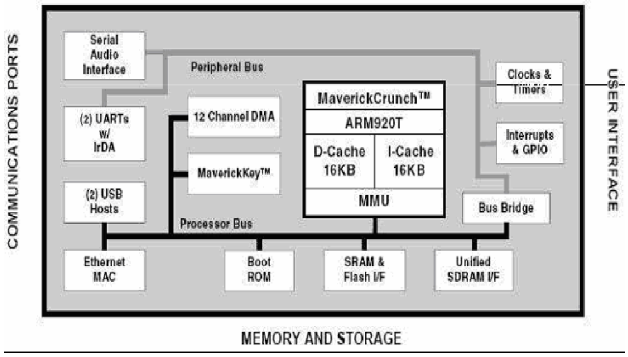

Fig 3: EP9302 SBC Block Diagram

The general flow of compiling and porting RTLinuxPro is as shown in Figure 3.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 10, October 2013

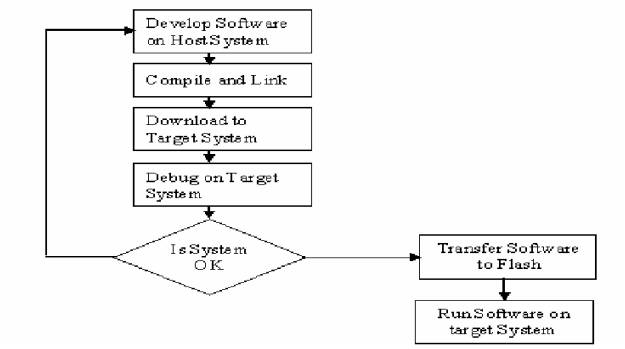

Fig 4: Deployment Flow chart

II. STARTING TFTP SERVER

Host Machine is the development workstation on which all binaries are compiled. Binaries and file systems are built so that they will run on the target machine. Target Machine is the computer which runs the binaries and file system compiled by the host machines. It could also be the host machine itself. Host machine is our personal desktop while our target is EP9302 SBC. File transfer service (tftp) should be executed on host machine to transfer the file to target board.

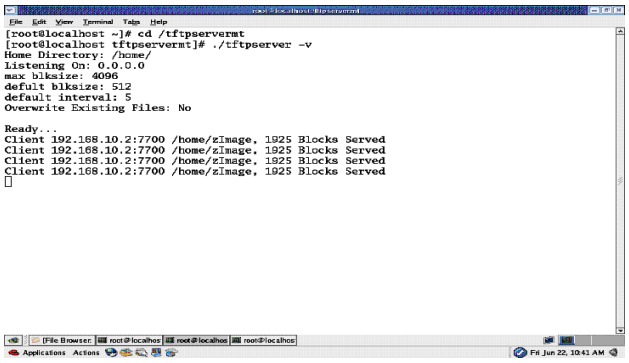

Fig 5: stating tftp server

III. DESIGNING ROOT FILE SYSTEM

Download utility is used to program the image "redboot.bin" into the flash of the board to be used. It is also used to program the Ethernet MAC address so that the Ethernet interface can be used by RedBoot.

The Linux kernel expects several important files to exist in a root file system when it boots. In embedded systems, these files are stored in ramdisk. There are two limitations on the size of a ramdisk. If the compressed ramdisk image is stored in partition in on board Flash memory and the compressed ramdisk.gz is bigger than this partition, the boor loader will not program it to on board flash. The compressed ramdisk image is decompressed into RAM. The bigger your uncompressed ramdisk is, the less RAM you have remaining for the kernel and user programs. This limitation depends on the amount of RAM installed and the amount needed by the kernel and your software to run. Enter the command "df" to see how much free space you have in your ramdisk.

The basic file system structure contains minimum set of directories /dev, /bin, /etc, /lib, /sbin, Basic set of utilities sh, ls, cp, mv, etc, Minimum set of config files: rc, inittab, fstab, etc., Devices /dev/hd*, /dev/tty*, /dev/fd0, etc and Runtime library to provide basic functions used by utilities.

Once RedBoot (with networking) is running on the board, Linux can be loaded and run. The images to be loaded by RedBoot must be placed into the area used by the tftp server on your host machine. The boot-loader loads the zImage and the root file system from on-board Flash. The default configuration of EP9302 is using part of SDRAM as RAM disk for Linux root file system. The ramdisk image must be stored in the on-board FLASH memory and loaded by Redboot for the Linux kernel. The image must be loaded into dynamic memory before it can be stored in the on board FLASH memory. Since the file system is in RAM, it is fast and can be mounted rw (read/write) but the changes are not preserved after a reboot. The compressed image will remain unchanged and provide the same environment each time the system starts

IV. COMPILING RTLINUXPRO

Before you compile your kernel, you need to configure it. Configuration choice is kept in **/linux/.config file**. Configuration is your opportunity to control exactly what kernel features are enabled (and disabled) in your new kernel. You'll also be in control of what parts get compiled into the kernel binary image (which gets loaded at boot-time), and what parts get compiled into load-on-demand kernel module files. The old-fashioned way of configuring a kernel is **make config**. New Way to configure is to use **make menuconfig or make xconfig**. If you'd like to configure your kernel, type one of these options. If you type **make menuconfig**, you'll get a nice text-based color menu system that you can use to configure the kernel. If you type **make xconfig**, you'll get a very nice X-based GUI that can be used to configure various kernel options. **make config** runs the Bash script Configure, which reads in the arch/arm/config.in file, which is located in the architecture directory and holds the definitions of the kernel configuration options and default assignments and interrogates it to see which components are to be included in the kernel. **arch/arm/config.in** resorts to the config.in files contained in the directories of the individual subsystems of the kernel. During this process, the two files **linux/autoconfig.h** and **.config** are created. The .config file

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 10, October 2013

controls the sequencing of the compilation run which **linux/autoconfig.h** takes care of conditional compiling within the kernel sources. The .config file is used if configure is called again to determine the default responses to individual questions. A fresh configuration will thus return the last values as the defaults. The command **make oldconfig** ensures that the default values are accepted without further interrogation. This enables .config file to be included in a new version of Linux so that the kernel is compiled with the same configuration.

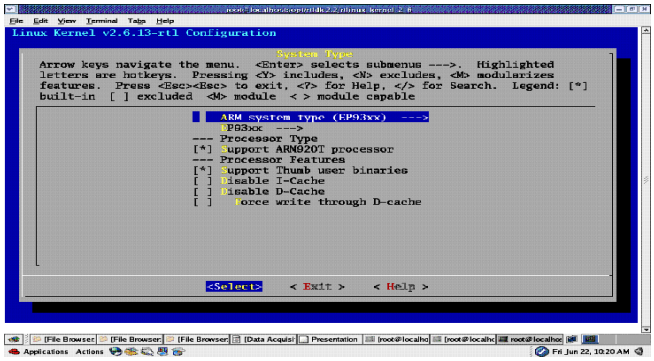

Fig 6: Compiing RTLinux kernel

Once your kernel is configured, it's time to get it compiled. Change the directories to the RTLinux kernel directory on the host system. Use "**make zImage**" to build the Linux boot image. After several minutes, compilation will complete and you'll find the zImage file in **/arch/arm/boot**.

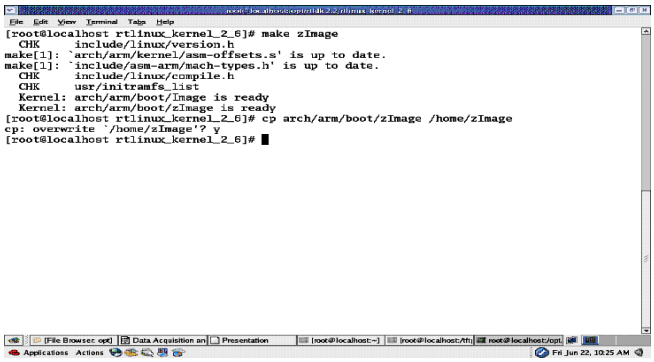

Fig 7: Creating zImage

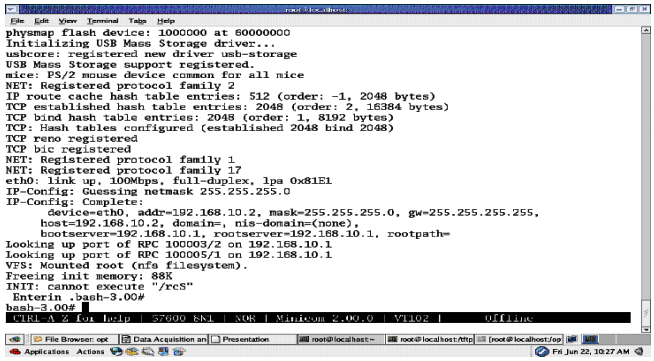

Fig 8: Once the board is ready….

V. INSTALLING RTCORE

A typical RTCore application consists of one or more real time components that run under the direct control of the real time kernel and a set of non real time components that run as under space programs. RTCore must be loaded in order for any real time services to be available. Hence in order to run any RTCore application we need to load RTCore modules on the Board. The process is simple, once the patched kernel has been compiled and installed. Once the system is running the correct kernel, RTCore module can be loaded located in modules directory of RTLinuxPro.

For this example, the RTCore OS needs to be loaded on edb93xx SBC. This is most simple example to print Hello message.

```
#include <stdio.h>
int main(void)
{
printf ("Hello from the RTL base system\n");
return 0;
}
```

Running the example (./hello.rtl) forces the RTCore OS to load the application and enter the main () context. Here it prints a message out through standard I/O for the user to see and exits. A simple makefile is needed to build this, which includes rtl.mk which will set up the build environment - compilers, CFLAGS, and so on. Including Rules.make will provide the build rules needed to transform C source to an RTCore application. In old RTLinux versions (2.0 and prior) printf () messages are appearing in the kernel's ring buffer, but now they print through stdout just like any other application. Also, there is a standard printf (), rather than the rtl_printf (). This printf () is fully capable and can handle any format that a normal printf () can handle. Once the message has been printed, the program exits, and RTCore unloads the application.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 10, October 2013

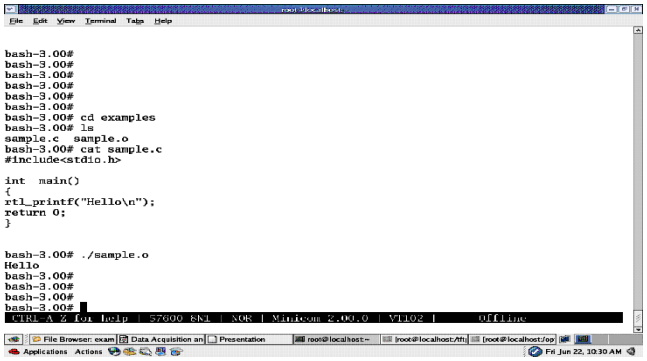

Fig 9: Running RTLinux application on EP9302

CONCLUSION

This paper presents guidelines showing how to port RTLinux on ARM platforms. While configuring the real time kernel for ARM platform, we need to select proper processor type. i.e. EP9302 processor. And we also need to enable "Network File System" and "Root File System on NFS" options as a root file system for target GESBC-9302 board. This paper also described the steps for running any real-time applications on ARM platform.

REFERENCES

1. Using Linux for real time applications- Marchesin. A; Volume 21 IEEE- Sep-Oct 2004.
2. Victor Yodaiken. RTLinux approach to hard real-time.
3. ARM System-on-chip Architecture 2nd Edition – Steve Furber
4. ARM System Developer's Guide by Andrew Sloss, Dominic Symes, Chris Wright FSM Labs Inc. FSMLabs RTLinux Development, 2002. www.fsmlabs.com/developers. www.fsmlabs.com/products/software.htm.
5. GESBC-EP9302 User Manual
6. Website: www.cirrus.com