

## Porting of RTLinuxPro on ARM9 Platform

Rahul Desai <sup>1</sup>, Geeta Patil <sup>2</sup>, Vaishali Ingale <sup>3</sup>

1. Asst. Prof, Army Institute of Technology, Pune
2. Asst. Prof, Army Institute of Technology, Pune,
3. Asst Prof, Army Institute of Technology, Pune,

### Abstract

*This paper describes the porting of RTLinuxPro on ARM9 platform. RTLinuxPro is FSMLabs RTCore POSIX PS51 robust "hard" real-time kernel plus a full-embedded Linux development system. EP9302 is a high-performance ARM9 System-On-Chip Embedded Processor on the Glomation GESBC-9302 Embedded Single Board Computer. This paper also discusses designing real-time applications and executing them on this platform*

### 1. Introduction

RTLinuxPro is a hard realtime operating system designed to support applications that have real, serious, non-negotiable deadlines

The RTOS is called "RTCore" which is a highly efficient, hard real-time implementation of POSIX Standard threads [1]. The general purpose operating system is "Linux" [2] – sophisticated free software supported by the world's largest technology companies. Figure 1 shows the design of RTLinuxPro.

RTLinuxPro is a full POSIX 1003.13 [3] complaint hard realtime operating system (RTCore) that runs a general purpose operating system (Linux) as fully preemptible application. This provides the modularity. Applications run either in RTCore as real-time task or in Linux as user space applications. All the features of Linux are available as well as required hard real-time performance. RTCore is designed to make real-time programming more convenient and less mysterious. One way to think of RTCore is as a small operating system that runs a second operating system as its lowest priority task. All the non-time-critical applications can be put in the second operating system. 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.

RTLinuxPro is used for everything from satellite controllers, telescopes, and jet engine test stands to routers and computer graphics. RTLinuxPro runs on a wide range of platforms from high-end clusters of multiprocessor P4s/Athlons to low power devices like the MPC860 and ARM7.

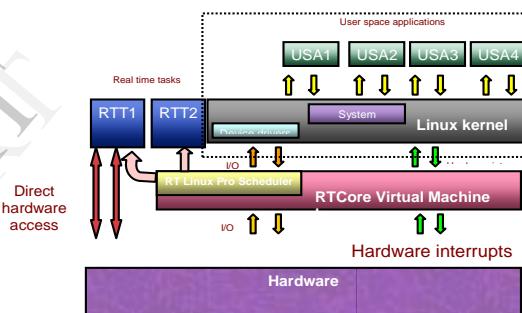


Figure 1: RTLinuxPro Design

The EP93xx is a high-performance system-on-chip design that includes 200 MHz ARM9 processor and is ideal for a range of industrial and consumer electronic applications. The EP9302 features an advanced ARM920T processor [4] design with MMU that supports Linux, Windows CE and many other embedded operating systems. The ARM920T's 32-bit microcontroller architecture, with a five-stage pipeline [5], delivers impressive performance at very low power. The basic block diagram of EP-9302 [6] is as shown in Figure 2.

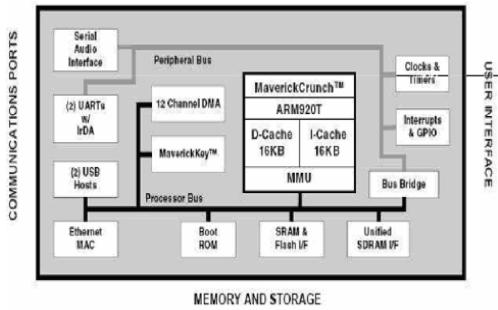


Figure 2: EP9302 System-On-Chip Layout

The GESBC-9302 a low cost compact sized single board computer based on Cirrus Logic EP9302 processor. With a large peripheral set targeted to a variety of applications, the GESBC-9302 is well suited for industrial controls, 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 list below summarizes the features of the GESBC- 9302.

- 200MHz Processor Core-ARM920T
- 32M SDRAM, 4~16M FLASH
- Ethernet Media Access Controller (EMAC)
- 5 channel 12-bit Analog-to-Digital Converter (ADC)
- Universal Asynchronous Receiver / Transmitters (UARTs) with RS-485 Support
- 2 USB Host Port and Real-Time Clock
- Hardware Debug Interface

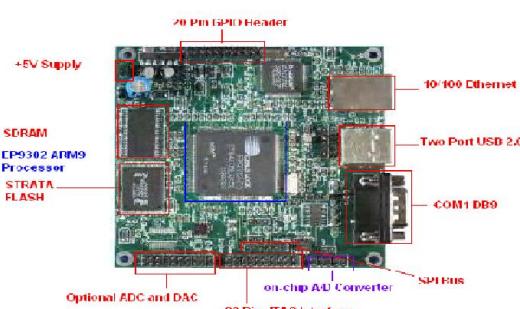


Figure 3: Glomation GESBC-9302 Embedded Single Board Computer

The general flow of compiling and porting RTLinuxPro is as shown in Figure 4.

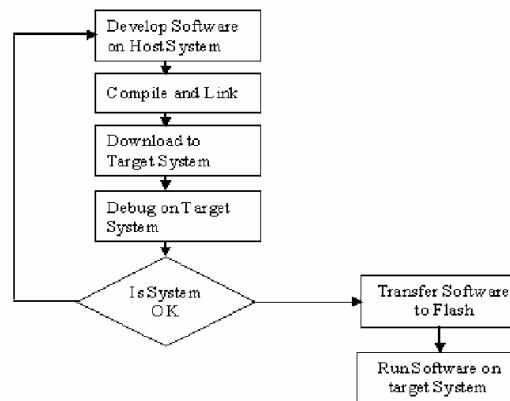


Figure 4: Deployment Flow chart

## 2. GESBC-9302 Board Software

Linux with few test applications and network utilities is shipped with the GESBC-9302 Board. This software is programmed into the system FLASH located on the board prior to shipment [7].

Download Utility provides the user with a tool for programming the flash memory on the GESBC-9302 with a binary image.

RedBoot provides a simple interface for loading operating systems and applications onto the GESBC-9302 board. The board is shipped with Redboot pre-installed.

### 2.1 Starting tftp server

Here basically two important terms are host machine and target machine. 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 GESBC-9302. File transfer service (tftp) should be executed on host machine to transfer the file to target board.

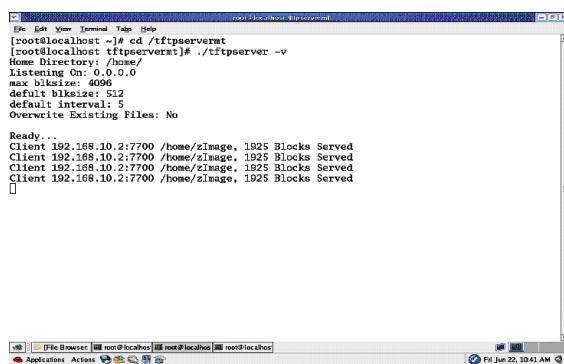


Figure 5: tftp server

## 2.2 About download utility

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 RedBoot can use the Ethernet interface.

## 2.3 Boot Loader

RedBoot is the standard embedded system Boot Loader from Red Hat. RedBoot provides a wide set of tools for downloading and executing programs on embedded target systems.

RedBoot uses a serial console for its input and output. The default serial port setting is 57600,8,N,1.

It also supports the built-in Ethernet port and a flash file system and general flash programming.

## 2.4 Designing root file system

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. 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`,

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, by entering the following commands at the terminal console.

```
load -v -r -b 0x800000 -h 192.168.10.1  
ramdisk.gz
```

Where -v: verbose, -r: binary format, -b: base address in memory, -h: IP address of Host

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.

### 3. 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 type `make menuconfig`, you'll get a nice text-based color menu system that you can use to configure the kernel. `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 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.

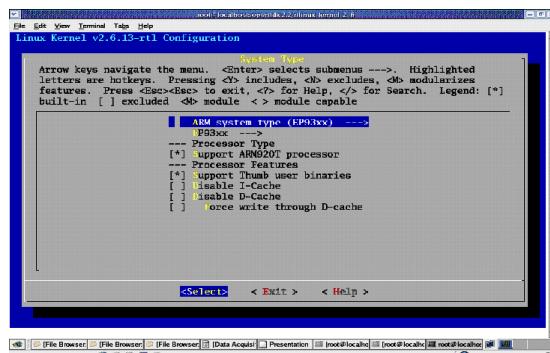


Figure 6: Compiling RTLinuxPro kernel

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.

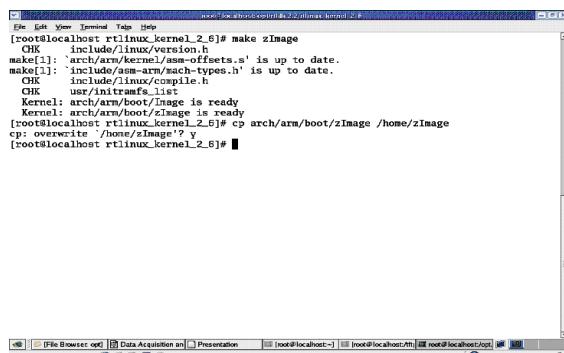


Figure 7: Making RRTLINUXPro Kernel Image

## 4. Installing RRTLINUXpro

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 located in modules directory of RRTLINUXPro on the Board.

The next step is to load RRTLINUX kernel image onto the onboard SDRAM, issue the following command at the terminal console connected to the GESBC-9302 board,

**RedBoot> load -v -r -b 0x80000 zImage**

-v = verbose, -r for remote, -b for base address.

Next, load Root File System on GESBC-9302 board

**RedBoot> fis load ramdisk**

Finally Execute

**RedBoot> exec**

This will load the minimal ramdisk image, the RRTLINUX kernel, and then start it running. After a few seconds, it will come to # prompt by executing /sbin/init specified in init/main.. The following figure 8 shows a snapshot of GESBC-9302 board Debug Information on Terminal Program.

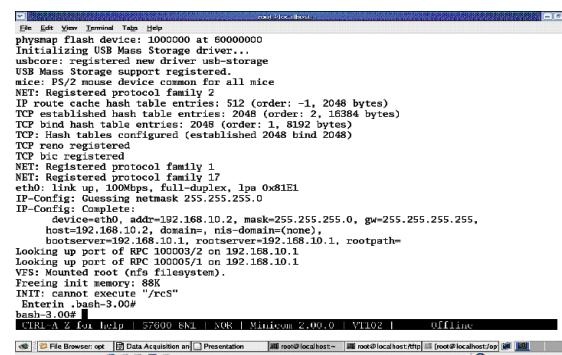


Figure 8: Once the board is ready....

## 5. Designing 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. This is most simple example to create a real time thread. A real-time application is

usually composed of several “threads” of execution.

```
#include<stdio.h>
#include<pthread.h>

void *thread_fun(void *arg)
{
rtl_printf("Inside \'thread_fun\' Thread\n");
}

int main()
{
    void *ptr;
    pthread_t thread;
    rtl_printf("Inside main Thread\n");

    pthread_create(&thread,NULL,thread_fun,
NULL);
    pthread_cancel(thread);
    pthread_join(thread,&ptr);
    return 0;
}
```

The entire process to run the RTLinux application on the target board is depicted as below:

1) Compile desired program on host machine using provided arm-linux-gcc toolchain.

```
[root@localhost examples]#      arm-linux-
gcc -o sample sample.c
```

2) Transfer the compiled file without the c extension to the /home directory on your host machine. On the console of the development board, create a directory in your ramdisk “sample”

```
bash-3.00#      mkdir /usr/sample
```

3) Once again, on the console of your development board, invoke a TFTP session to transfer your newly compiled program from the HOST to your TARGET over Ethernet:

```
bash-3.00#      tftp      -g      -r      sample
192.168.10.1
```

4) As “Execute” permission is not enabled on this file. We need to add this attribute. And execute the program. +x stands for grant execute permission. **chmod** stands for change file permission mode.

```
bash-3.00#      chmod +x sample
bash-3.00#      ./sample.o
```

Running the example (*./sample.o*) 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* that 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. **rtl\_printf()** is fully capable and can handle any format that a normal **printf ()** can handle.

**pthread\_create()** is for creating a new thread. The new thread created is of type **pthread\_t**, defined in the header *pthread.h*. This thread executes the function **thread\_fun()**. The *attr* argument specifies thread attributes to be applied to the new thread. If *attr* is NULL, default attributes are used. So here, **thread\_fun ()** is invoked with no argument. It has three components – initialization, run-time and termination. In RTLinux, all threads share the Linux kernel address space. The advantage of using threads is that switching between threads is quite inexpensive when compared with context switch. We can have complete control over the execution of a thread by using different functions present in RTLinux.

Once the message has been printed, **pthread\_cancel()** is invoked to cancel the thread.



Figure 9: Running RTLinux application on GESBC-9302

With RTLinuxPro 2.0, real-time applications are very portable, and even recompilable as normal Linux applications.

## 6. Conclusion

This paper presents guidelines showing how to port RTLinuxPro 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 describes the steps for running any real-time applications on ARM platform.

## 7. Acknowledgements

We would like to thank Sakithvel, Madhava Rao SS and Viswanathan S from FSM labs for their continuous support and valuable guidance as and when required.

## 8. References

- [1] Where you can learn more about thread programming: Newsgroup, Newsgroup, Sun: Workshop Developer Products--Threads , IEEE Parallel & Distributed Technology
- [2] Using Linux for real time applications- Marchesin. A; Volume 21 IEEE- Sep-Oct 2004.
- [3] IEEE Std 1003.1b-1993 IEEE Standard for Information Technology. Portable operating system interface (POSIX) part 1: System application programming interface, amendment 1: Realtime extensions. Technical report, IEEE, New York, 1994.
- [4] ARM System-on-chip Architecture 2<sup>nd</sup> Edition – Steve Furber
- [5] ARM System Developer's Guide by Andrew Sloss, Dominic Symes, Chris Wright.
- [6] GESBC-EP9302 User Manual  
Website: [www.cirrus.com](http://www.cirrus.com)
- [7] FSM Labs Inc. RTCore/BSD Released, 2002.  
<http://www.fsmlabs.com>.
- [8] FSM Labs Inc.  
FSMLabs RTLinux Development, 2002.  
[www.fsmlabs.com/developers](http://www.fsmlabs.com/developers).  
[www.fsmlabs.com/products/software.htm](http://www.fsmlabs.com/products/software.htm).

[9] Linux Journal. The Monthly Magazine of the Linux Community, 2002.  
<http://www.linuxjournal.com>.