Performance Comparison of RTOS

Shahmil Merchant, Kalpen Dedhia
{Kmd83 and srm96}@columbia.edu
Dept Of Electrical Engineering
Columbia University

***Abstract**: Embedded systems are becoming an integral part of commercial products today. Mobile phones, watches, flight controllers etc are to name a few. There is a strong and compatible relationship between the system hardware and the software, primarily the operating system to ensure hard real time deadlines. The real time operating system has to interface/communicate well with the hardware below it to prevent casualty.*

We look into two such, freely available, real-time operating systems ecos and rtlinux. We analyze the real time attributes, like timing latency, context switch latency and interrupt latency, of these operating systems by means of simple applications. We intend to give a better understanding of the finer intricacies of ecos and rttinux and study the pros and cons of them.

1. INTRODUCTION

Real time applications have become a very common phenomenon these days. Developers are given the enviable task of making software with real time constraints. A large number of RTOS are available in the market and one does get confused as to which one to use such that it provides the best over all benefits in terms of cost and operability. There are set of certain benchmarks, which one could examine in an RTOS, such as latency, susceptibility to different loads.

The main goal of our paper would be one of providing a larger picture of the job at hand in the form of an evaluation rather than providing a detailed study of measurement. We chose Ecos and Rtlinux due to the fact that they are free ware and are stable Linux ports with POSIX compliance. Obenland's [5] paper looks at POSIX in real-time systems and POSIX thread extensions and compares the performance of two general-purpose operating systems and two real-time operating systems.

Stewart's paper [4] illustrates different methods for estimating execution time of both user level and operating system overhead. Coarse grain timing measurements is calculated in software with time granularity in milliseconds. Hardware must be used to accurately predict the execution time. We are going to follow the software approach.

The main goal of this project is to study and analyze real-time operating systems and what semantics go into developing one. This project provides an initial approach about how to compare real-time operating systems.

The organization of the paper is as follows: in section 2 we survey the related work in comparing real time operating systems, different measurement approaches. Section 3, we explain some characteristics of the two real time operating systems and test metrics for comparing RTOSes. Section 4 discusses the experimental results and compares ecos and rtlinux based on these results. Finally, in section 5 we provide a summery and directions for future work.

2. RELATED WORK

Liu and Layland [3] came up with Rate Monotonic scheduling (priority driven scheduling) of real-time operating systems which is premier scheduling algorithm implemented in almost all the RTOSes. Better scheduling algorithm - Earliest Deadline First (deadline driven scheduling) is too complex to be implemented in real-time operating system. Keelings [2] article about the effects of priority inversion and its solution gives an insight to the hazards of priority inversion.

Stewarts [4] explains different approaches to calculate timing and performance, writing user code to measure performance and identifying timing errors. Manas [6] discusses Linux as real time operating system and different approaches for real-time Linux kernel. Timmerman [7] describes the framework for evaluation of real-time operating systems. This article makes a really good point of comparing RTOS under different load conditions.

3. RTOS AND TEST METRICS

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. Some other important features of every RTOS are as follows:

1. It should have some support for multi-tasking (threads) and it should be pre-emptive priority driven system.
2. RTOS should support thread synchronization using semaphores or mutexes.
3. RTOS must have sufficient number of priority levels.
4. RTOS must avoid *priority inversion.*

Edwards [1] book explains priority inversion in brief.

Scheduling a task in RTOS is critical. RTOS scheduler follows one of the following mentioned scheduling polices: FIFO (First In-First Out), STF (Shortest Time First), EDF (Earliest Deadline First), Priority scheduling defined by Liu and Layland [3], etc.

3.1 Test Metrics

RTOS has strict timing constraints and it should respond, under all possible load condition, in a very predictable way. This means different latencies of the systems should be predictable. Best operating system should have minimum latency and least amount of standard deviation. Following are the test metrics to evaluate eCos[1] and Rtlinux[2].

1. eCos is distributed by RedHat, Inc.
2. Rtlinux is distributed by Finite State Machine Labs, Inc.

3.1.1 Thread Switch Latency

A thread is defined by different states such as waiting, running, runnable etc. The time taken for the thread to move between different states is a parameter for testing RTOS performance. The context switch overhead in switching from one thread to another is lesser than a process as a thread is a lightweight process.

3.1.2 Interrupt Latency

Probably the most important feature for evaluating the performance of a RTOS is its ability to respond to interrupts. The time taken by the interrupt handler to deal with an interrupt and get back to regular program execution is extremely important in systems governed by hard real time constraints.

3.1.3 Thread creation and destruction

The RTOS under study have POSIX 1.0 compatibility. One could use the time to create the thread and destroy the thread as a good metric due to the simple reason it would show how well memory management would work in the system under consideration.

3.1.4 File System management

File systems provide an abstraction to the higher levels of software code the way as to which programs are stored in disk. Files on disk could be stored in the following manner:

Contiguous Allocation: Where processes are stored one after the other in the form of a heap and termination of a process could result in holes being created on the File system. Defragmentation is a solution.

Allocation table: A directory like structure wherein the directory has a pointer to each process on disk. FAT used in Window is an example.

Indexed Allocation: Where indexes to a linked list of processes exist. Best method allocation.

Hence by testing the time needed to create and open close a file could test the feasibility of different allocation schemes.

3.1.5 Synchronization

Shared resources form an integral part of an Operating system. Great care has to be taken when dealing with resources that can be used by different objects. Semaphore and Monitor implementation takes care of the Critical Section Problem.

The way an RTOS behaves to different system loads is an integral part of testing of an operating system. We test the real-time operating system under different load conditions.

3.2 RTOS under consideration

Both eCos and RTLinx are supplied as open source.

3.2.1 eCos

eCos, developed by Cygnus (now owned by Redhat), is distributed by Redhat, Inc. eCos supports broad range of targets like ARM, Hitachi SH3, Intel x86, MIPS, Matsushita AM3x, PowerPC and SPARC including 16,32, 64 bit architectures and Digital Signal Processors.

eCos meets the requirements of the real-time systems that Linux cannot yet reach. eCos has highly configurable kernel. You can build application specific real-time operating system with really small footprint. eCos has prioritized FIFO and Bitmap scheduling policies with 32 priority levels. It also supports priority inheritance and priority ceiling to tackle priority inversion problem. eCos has only soft real-time support for interrupt latency. All other latencies are CPU dependent.

More about eCos real-time operating system, development tools and device drivers can be found on Redhat website [9].

3.2.2 RTLinux

RTLinux is developed and distributed by Finite State Machine Labs, Inc. RTLinux support a limited number of architectures, like x86, PowerPC, MIPS, Alpha, unlike eCos. RTLinux is a hard real-time kernel that co-exists with linux kernel. Linux kernel is a lowest priority task or RTLinux kernel and it can be fully pre-empted. RTLinux kernel communicates with Linux kernel using shared memory. This approach allows real-time applications to take advantage of non real-time features of Linux.

RTLinux supports prioritized FIFO scheduler and extensible scheduler with 1024 priority levels. Lock-free data structure and priority ceiling are two approaches to avoid priority inversion. RTLinux claims to have hard real-time interrupt latency. All the other latencies are CPU dependent.

Yodaiken's [8] paper explains hard real-time approach of RTLinux and its one of the first papers written on RTLinux. More about RTLinux can be found on Finite State Machine Labs official website [10].

4. EXPERIMENTAL RESULTS

We have successfully installed both the operating systems that are eCos and Rtlinux. eCos was installed using a Linux synthetic target. Rtlinux version 2.2a (pre-patched) was used .The installation techniques are described in the appendix as given later.

One of the most challenging aspects was just to get these two real-time operating systems working on the x86 architecture. We used AMD K6 266 MHz processor and Redhat's Linux distribution 6.2 and kernel version 2.2.14.

For Rtlinux our approach to calculate the thread creation and deletion latency is simple. We create a thread and just before doing so we measure the time and immediately after creation of the thread we measure the time again. The values that we got are given in the table below.

We also measure the context-switching overhead between threads. We define a shared resource and create two threads. We then create a lock on it. Any thread that comes in acquires the lock and using the resource. We calculate the time required to access the lock and also the time when the thread relinquishes the lock. Rtlinux uses generic C libraries but provides a few API's for measuring real time issues. Programs require one to build modules, which are added at run time to the Linux kernel. Hence compiling Rtlinux adds new modules.

eCos does provide an abstraction of a real time operating system and its because of its wide range of configuration options. Once the eCos kernel has been configured and built it provides a Hardware abstraction layer (HAL), which is a set of binaries built above the Linux kernel, which ensure the real time environment.eCos requires, to build this binary Hardware Abstraction Layer and run your application on either of the 2 targets – A Linux synthetic target which is the same machine and where the actually scheduling is done by the Linux scheduler but with real time support provided by the binaries or An external target where in the you connect to it by means of a serial port or Ethernet. The target machine was booted using an image file that was created during the eCos "stub" creation process. For eCos we used a couple of files that we generated during the installation process that enabled us to calculate the thread creation switching etc time. The values obtained have been tabulated below.

Test Metrics	eCos	RTLinux
Thread Creation Latency	16130-34300 ns	137216 ns
Thread Deletion Latency	5350-6450 ns	8448 ns
Mutex latency	5043-25340 ns.	504346912 ns

Table 1

We can see from table, eCos has a range of values and standard deviation is large. Thus, if system requirement are hard real-time, you should choose RTLinux as its latency values are fairly predictable. We are yet at a nascent stage of analysis and any assumptions would require further probing.

5. CONCLUSION

Studying the differences between the two RTOS, we came to the conclusion that scheduling policies strictly dependent on priorities and with built in priority inversion avoidance. Hard real-time support for interrupt latency in RTLinux is vital. We need to consider the size of the kernel, its ability to support hard real-time timing constraints and performance under different load conditions before judging which RTOS better than the other RTOS.

We have done the groundwork. We have tested basic applications to measure semantics regarding time. Other semantics such as support for additional devices, drivers, portability to different hardware and virtual memory issues need to be discussed. Time and resource constraints have really not given us much time to study these issues. Our analysis does give the reader a overview of what actually goes into developing an operating system with real time constraints and how similar operations on different RTOS require different time to execute. This does open room to think of different ways of optimizing a Kernel for Real time applications by taking the best features of each.

6. ACKNOWLEDGEMENTS

We would like thanking Prof. Stephen Edwards for giving us the opportunity to implement such an interesting project and giving us valuable guidance as and when required. We would also like to thank Jonathan Larmour and Gary Thomas from Redhat and Michael Berbanov from FSM labs for their continuous support.

7. BIBLIOGRAPHY

[1] S Edwards. Languages for Digital Embedded Systems. Kluwer, 2000.

[2] N.J. Keeling. How Priority Inversion messes up real-time performance and how the Priority Ceiling Protocol puts it right. Real-Time Magazine 99(4): 46-50. April, 1999.

[3] C. Liu and James Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. Journal of the Association for Computing Machinery, 20(1): 46-61, January 1973.

[4] David Stewart. Measuring Execution Time and Real-Time Performance. In Embedded Systems conference, San Francisco, April 2001.

[5] K. Obenland. Real-Time Performance of Standards based Commercial Operating

Systems. In Embedded Systems conference, San Francisco, April 2001.

[6] Manas Saksens. Linux as Real-Time Operating System. In Embedded Systems conference, San Francisco, April 2001.

[7] Martin Timmerman. RTOS Evaluations Kick Off. Real-Time Magazine 98(3): 6–10. March 1998.

[8] Victor Yodaiken. RTLinuX approach to hard real-time. http://luz.nmt.edu/~rtlinux

[9] http://www.redhat.com/embedded

[10] http://www.fsmlabs.com

APPENDIX

Installing RTlinux

Download and tar

Download a clean prepatched kernel from Rtlinux.org. http://RTlinux.org/

rtlinux-2.0 is intended to be used with Linux 2.2.13
rtlinux-2.2 is intended to be used with Linux 2.2.14
rtlinux -2.2a is also intended to be used with Linux 2.2.14

I placed this file in the home directory of root.

I want to untar it into /usr/src

```
tar xzvf /root/rtlinux-2.2a-prepatched.tar.gz /usr/src
```

Note 2.2/2.2a - Both use the same rtlinux-2.2 directory when taken out of the tar ball.

A whole bunch of stuff goes across the screen indicating all of the files that tar is placing in /usr/src. This creates a new directory /usr/src/rtlinux-2.2 and sets up the new kernel structure below it.

The linux link

```
[root@localhost /root]# cd /usr/src
```

I see from the light red line that - linux ->linux-2.2.14. So the linux file in this directory is a link. Since it is only a link I can delete it and then check to make sure it is gone.

```
[root@localhost src]# rm linux
```

Now if it is a real directory with a bunch of sub directories, I'd need to move it to some other location. I'd do that by first seeing that it was a directory rather than a link to a directory.

I see from the light red line that linux is a common directory -- its line begins with (d) rather than (l) and it just has a (/) after its name. I need to move this linux directory out of the way before I compile a new kernel. I'll called it linuxold . The **mv** command with the linux name and linuxold as the new name does the deed for me.

Now you need to make a new link in place of the old /usr/src/linux

```
[root@localhost src]# ln -s  rtlinux-2.2/linux   linux
```

Make certain that the link goes from /usr/src/linux to the realtime/linux. Not the other way round.

Configuring the kernel

But at least I have a start at knowing what needs to be included when I run **make oldconfig, make config, make menuconfig, or make xconfig**.   I'm going to make sure that I am in the correct directory before I enter the config command.

```
[root@localhost linux]# pwd
/usr/src/rtlinux-2.0/linux
```

Yes, I am in the correct location for the configuration process. So I need to select the method that I will use. The possibilities again are:

- make oldconfig
- make config
- make menuconfig
- make xconfig

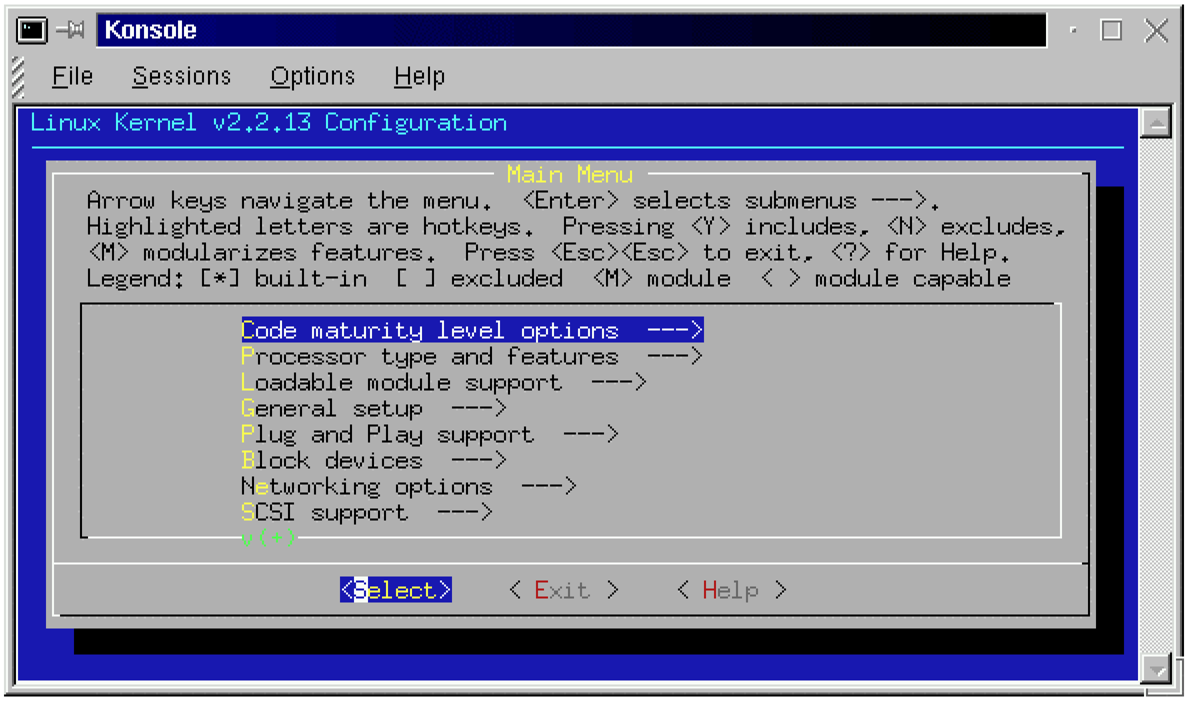
This is what each looks like. I use make menuconfig but it was just a matter of choice.

make menuconfig

To start this method you can just enter make menuconfig at the prompt.

```
[root@localhost linux]# make menuconfig
```

After a bit of checking and compiling, the following window shows up.

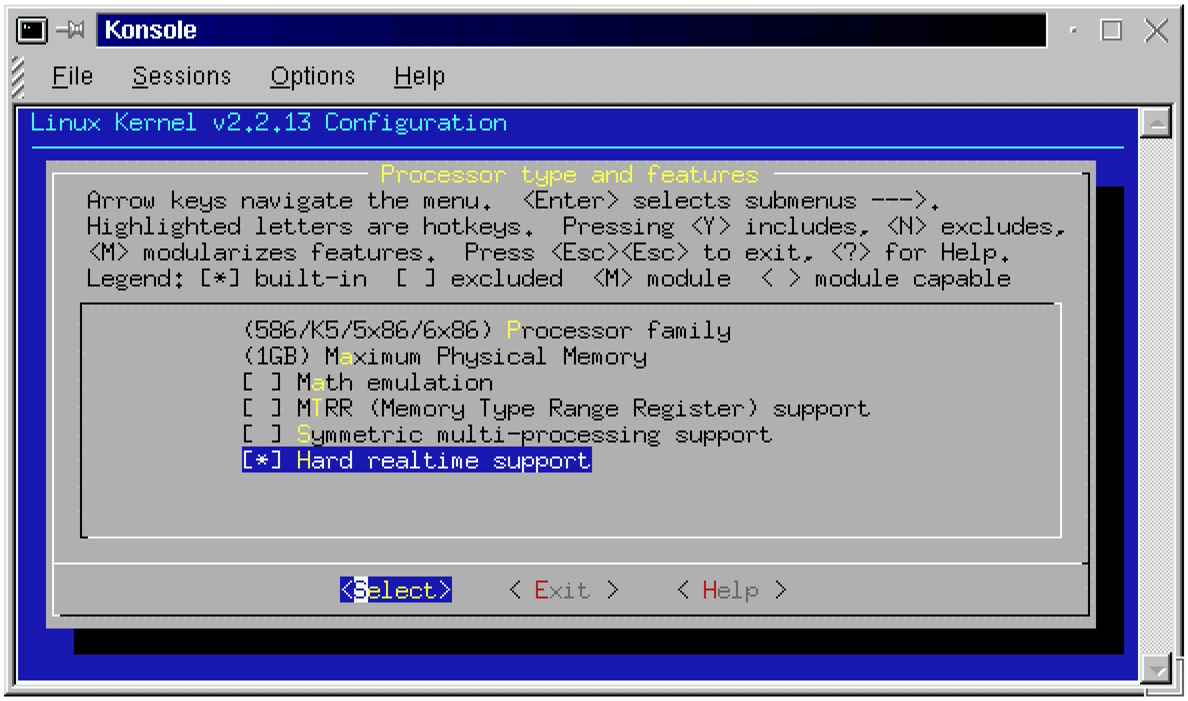


```
Linux Kernel v2.2.13 Configuration
                          Main Menu
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

            Code maturity level options  --->
            Processor type and features  --->
            Loadable module support  --->
            General setup  --->
            Plug and Play support  --->
            Block devices  --->
            Networking options  --->
            SCSI support  --->
            v(+)

                 <Select>    < Exit >    < Help >
```

This seems to be an easier option than the make config. It also allows you to skip around among the many sections of the config file and repeat your work in each section. The following screen shows the processor type and features menu.



```
Linux Kernel v2.2.13 Configuration
                   Processor type and features
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

            (586/K5/5x86/6x86) Processor family
            (1GB) Maximum Physical Memory
            [ ] Math emulation
            [ ] MTRR (Memory Type Range Register) support
            [ ] Symmetric multi-processing support
            [*] Hard realtime support

                 <Select>    < Exit >    < Help >
```

You can navigate around the menu choices with the up and down arrow keys. You can also navigate across the select, exit, and help items with the left and right arrow keys.

Whenever a parameter is highlighted, you can select it with (y), deselect it with (n), or make it a module with (m). The help files are very helpful here.

Making the kernel

There are several things that you will need to do as a part of making the kernel. They include:

- make dep
- make clean
- make bzImage
- make modules
- make modules_install
- move bzImage-RTL2.2 and System.map-RTL2.2
- make a new link from System.map

make dep

Now I am ready to begin to build the new kernel. At this point make is going to set up all of the dependencies between the many files that will have to be compiled into the new bzImage file.

```
[root@localhost linux]# make dep
```

make clean

This command cleans up the file system by removing any unnecessary files that were created during the previous step.

```
[root@localhost linux]# make clean
[root@localhost linux]#
```

make bzImage

```
 [root@localhost linux]# make bzImage
```

This is the place where you really test that /usr/src/linux link to /usr/src/rtlinux-2.2/linux. If that link is not correct. you may get the following error message.

```
[root@localhost linux]# make bzImage
In file included from /usr/include/errno.h:36,
     from scripts/split-include.c:26:
/usr/include/bits/errno.h:25: linux/errno.h: No such file
```

```
or directory
make: *** [scripts/split-include] Error 1
```

If you get a message like this, fix that link.

During this operation, the interpreter issues several warnings but none of them are fatal. As it approaches the conclusion of this process I get the following lines.

```
 ...
Root device is (3, 5)
Boot sector 512 bytes.
Setup is 1308 bytes.
System is 584 kB
[root@localhost linux]
```

These messages are significant because they say that a kernel has been made and tell what its size is. The size of your system will be different from that shown above. You can see the result from the following link.

```
[root@localhost linux]# ls -l arch/i386/boot/bz*
-rw-r--r--   1 root     root       669362 Nov  5 14:22
arch/i386/boot/bzImage
```

And there it is. There should also be a new system map in the linux directory.

```
[root@localhost linux]# ls -l Sys*
-rw-r--r--   1 root     root       218189 Nov  5 14:22
System.map
```

These are the two files that we will need to copy into the /boot directory. This will be done right after I make and install the kernel modules.

make modules

```
 [root@localhost linux]# make modules
make -C  kernel CFLAGS="-Wall -Wstrict-prototypes - (much
more here)
...
```

There were a number of warning messages issued by the interpreter during this process. There were no errors that caused the procedure to abort.

make modules_install

Now I need to move those modules in the right place so that the new kernel can find them.

```
[root@localhost linux]# make modules_install
Installing modules under /lib/modules/2.2.14-rtl2.2/net
```

The returned lines indicate that modules were found and that they were installed in the libraries.

Move bzImage and System.map

In the directory /boot, the file System.map-* is the currently running system map because it is pointed to by the link from System.map. Also when lilo asks for vmlinuz it will get vmlinux-*. I want to preserve some of the same ability to link simple file names to the files that I just created. To do this, I am going to call them by their name and add the -RTL2.2 suffix to each. Since I am still in the /usr/src/rtlinux-2.2/linux directory, I can reference the new files from there.

```
[root@localhost linux]# pwd
/usr/src/rtlinux-2.2/linux
[root@localhost linux]# cp System.map /boot/System.map-
RTL2.2
[root@localhost linux]# cp arch/i386/boot/bzImage
/boot/bzImage-RTL2.2
```

and the file copy deed is done.

I can boot directly from bzImage-RTL2.0 as we will see in a minute when we edit LILO.config. But I need to delete the old System.map link and make a link from System.map to the new System.map-RTL22.

```
[root@localhost linux]# rm /boot/System.map
rm: remove `/boot/System.map'? y
[root@localhost linux]# ln -s /boot/System.map-RTL2.0
/boot/System.map
```

Look again at the directory of /boot to see that everything is okay.

Editing LILO

My system uses LILO to boot up. The file /etc/lilo.conf sets up LILO so I need to edit it and include the new image

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
vga=normal
default=linux
keytable=/boot/us.klt
prompt
timeout=150
message=/boot/message

image =/boot/bzImageRTL22
        label=rtl22
        root=/dev/hda5
        append="mem=63M" (this is the size of my ram memory
-1M for shared)
        read-only
image=/boot/vmlinuz
        label=mandrake
        root=/dev/hda5
        append=""
        read-only
```

Now with lilo.config set up like this, I need to run LILO to modify the boot sector.

```
[root@localhost linux]# /sbin/lilo
Added linux *
Added rtl22
```

LILO will report what it has done. The * shows that linux will be the default kernel. If it doesn't report that it has added your new kernel you should edit /etc/lilo.conf and try again.

Running the kernel

When LILO starts up it will wait a while before it loads the default kernel. During that time you can press the tab key and get a list of possible entries. If your new kernel does not come up properly, you can enter one of the other kernel names at that time and it will be booted.

Making the rt modules

make modules

```
 [root@localhost /root]# cd /usr/src/rtlinux-2.2
[root@localhost rtlinux-2.2]# make
Kernel version 2.2.14-rtl2.2
...
make[1]: Leaving directory `/usr/src/rtlinux-
2.2/drivers/rt_printk'
Now you need to become root and do "make install"
```

make install

Make install is run from the same directory that the make command was run in.

Note 2.0 - The 2.0 report will show the rtl directory and other kernel names.

```
[root@localhost rtlinux-2.2]# make install

Testing for the mbuff device... mknod /dev/mbuff c 10 254
Testing for FIFOs... created /dev/rtf0 - 7 (major 150)
```

```
Instaling modules in /lib/modules/2.2.14-rtl2.2/misc
/sbin/depmod -a
Installing man pages to /usr/local/man
Installing header /usr/include/rtlinux
install -c -m 644 rtl.mk /usr/include/rtlinux;
```

After following all directions on installing the rtlinux-2.2 kernel, I had two files that needed permissions changed before it would let me see the examples operate. Those files are, rmrtl and insrtl and they are both in the rtlinux-2.2 directory.

```
[root@localhost rtlinux-2.2]# chmod -v a+rx insrtl rmrtl
mode of insrtl changed to 0755 (rwxr-xr-x)
mode of rmrtl changed to 0755 (rwxr-xr-x)
[root@localhost rtlinux-2.2]#
```

Installing ECOS for linux synthetic target.

Downloading eCos

The latest development versions of eCos are now provided via the eCos anonymous CVS repository, and it is strongly recommended that wherever possible this mechanism is used in preference to downloading eCos 1.3.1. eCos 1.3.1 does not have the functionality, platform coverage, nor bug fixes that are available in the latest versions of eCos.

Installing the eCos RPM

To install the RPM format eCos distribution, you should first ensure that you have the RPM tool. This tool is supplied by default on all Red Hat Linux distributions. The RPM must be installed by the root user, so you may need to ask your systems administrator to install it for you. To install it simply run the following command as root:

```
    rpm -i ecos-1.3.1-1.i386.rpm
```

On completion, the eCos repository will be found in the directory `/opt/ecos/ecos-1.3.1`.

Users without root access may still be able to extract the files from the RPM by using the following command:

```
    rpm2cpio ecos-1.3.1-1.i386.rpm | cpio -i -d
```

GCC 2.95.2 sources can be downloaded from the GCC page. Both the core distribution and the C++ distribution files are required for eCos. Install either of the following distributions.

- gcc-core-2.95.2.tar.gz
- gcc-core-2.95.2.tar.bz2
- gcc-g++-2.95.2.tar.gz
- gcc-g++-2.95.2.tar.bz2

Note that the instructions for building GCC here are only intended for use with eCos. In any other environment, the tools may not function correctly

GNU Debugger

Instructions for downloading the GNU Debugger (GDB) are provided on the GDB home page. However, Red Hat has also released an open source graphical front-end to GDB based on Tcl/Tk called *Insight*, which has a separate home page

The Insight sources are a superset of the standard GDB sources. It is also still possible to run GDB in command-line mode by using the `-nw` command-line option when invoking GDB, so there is nothing to lose by using the Insight sources.

The latest release (version 5.0) is recommended and may be downloaded via the GDB home page. Download the files directly:

- Insight-5.0.tar.bz2
- gdb-5.0.tar.bz2

Preparing the sources for building

Once the tools sources have been downloaded, they must be prepared before building. These instructions assume that the tool sources will be extracted in the `/src` directory hierarchy. Other locations may be substituted throughout. Similarly placeholders of the form ***`YYYYMMDD`*** and ***`YYMMDD`*** should be replaced with the actual date of the downloaded files. Ensure that the file system used has sufficient free space available. The contents of each archive will expand to occupy approximately 6 times the space required by the compressed archive itself.

The following steps should be followed at a *sh*, *ksh* or *bash* prompt. Users of the *csh* and *tcsh* shells should replace `2>&1` with `|&` throughout:

1. Create a directory for each set of tool sources, avoiding directory names containing spaces as these can confuse the build system:
2. `mkdir -p /src/binutils /src/gcc /src/gdb`
3. Extract the sources for each tool directory in turn. For *bzip2* archives:

   ```
   cd /src/binutils
   bunzip2 < binutils-2.10.1.tar.bz2 | tar xvf -
   cd /src/gcc
   bunzip2 < gcc-core-2.95.2.tar.bz2 | tar xvf -
   bunzip2 < gcc-g++-2.95.2.tar.bz2 | tar xvf -
   cd /src/gdb
   bunzip2 < insight-5.0.tar.bz2 | tar xvf -
   ```

   For *gzip* archives:

   ```
   cd /src/binutils
   gunzip < binutils-2.10.1.tar.gz | tar xvf -
   cd /src/gcc
   gunzip < gcc-core-2.95.2.tar.gz | tar xvf -
   gunzip < gcc-g++-2.95.2.tar.gz | tar xvf -
   cd /src/gdb
   gunzip < insight-5.0.tar.gz | tar xvf -
   ```

   The following directories should be generated and populated during the extraction process:

```
    /src/binutils/binutils-2.10.1
    /src/gcc/gcc-2.95.2
    /src/gdb/insight-5.0
```

If the standard GDB source distribution was downloaded rather than Insight, then the GDB tools source directory will be `/src/gdb/gdb-5.0` rather than `/src/gdb/insight-5.0`.

4. You may now need to apply a small number of source patches that are required to fix outstanding problems and add eCos support to the tools Download the ecos-gcc-2952.pat patch to a file and apply it:

   ```
       cd /src/gcc/gcc-2.95.2
       patch -p0 < ecos-gcc-2952.pat
   ```

   You must then reset the source file timestamps to ensure that makefile dependencies are handled correctly:

   ```
       contrib/egcs_update --touch
   ```

   If the *patch* utility reports the following message:

   ```
       Reversed (or previously applied) patch detected! Assume -R?
   [n]
   ```

   then type *n* because this indicates the patch has already been applied in the master sources.

Building the tools

1. Configure the GNU Binary Utilities:

   ```
       mkdir -p /tmp/build/binutils
       cd /tmp/build/binutils
       /src/binutils/binutils-2.10.1/configure --target=i686-pc-
   linux-gnu \
         --prefix=/tools \
         --exec-prefix=/tools/H-i686-pc-linux-gnu \
         -v 2>&1 | tee configure.out
   ```

   If there are any problems configuring the tools, you can refer to the file `configure.out` as a permanent record of what happened.

2. Build and install the GNU Binary Utilities:

   ```
       make -w all install 2>&1 | tee make.out
   ```

   If there are any problems building the tools, you can use the file `make.out` as a permanent record of what happened.

3. Configure GCC, ensuring that the GNU Binary Utilities are at the head of the PATH:

```
    PATH=/tools/H-i686-pc-linux-gnu/bin:$PATH ; export
PATH
```
(for *sh*, *ksh* and *bash* users)
```
    set path = ( /tools/H-i686-pc-linux-gnu/bin $path
)
```
(for *csh* and *tcsh* users)
```
    mkdir -p /tmp/build/gcc
    cd /tmp/build/gcc
    /src/gcc/gcc-2.95.2/configure --target=i686-pc-linux-gnu \
      --prefix=/tools \
      --exec-prefix=/tools/H-i686-pc-linux-gnu \
      --with-gnu-as --with-gnu-ld --with-newlib \
      -v 2>&1 | tee configure.out
```

4. Build and install GCC:

```
    make -w all-gcc install-gcc \
      LANGUAGES="c c++" 2>&1 | tee make.out
```

5. Configure Insight:

```
    mkdir -p /tmp/build/gdb
    cd /tmp/build/gdb
    /src/gdb/insight-5.0/configure --target=i686-pc-linux-gnu \
      --prefix=/tools \
      --exec-prefix=/tools/H-i686-pc-linux-gnu \
      -v 2>&1 | tee configure.out
```

6. Build and install Insight:

```
    make -w all install 2>&1 | tee make.out
```

   On completion, the Intel x86 Linux development tool executable files will be located at `/tools/H-i686-pc-linux-gnu/bin`. This directory should be added to the head of your PATH.

Setting up the eCos environment

Having installed eCos and the development tools, the environment must be setup prior to use.

1. Indicate the location of the eCos repository to the eCos host tools. For *sh*, *ksh* and *bash* users:

```
    ECOS_REPOSITORY=/opt/ecos/ecos-1.3.1/packages
    export ECOS_REPOSITORY
```

For *csh* and *tcsh* users:

```
setenv ECOS_REPOSITORY /opt/ecos/ecos-1.3.1/packages
```

2. Add the location of the eCos host tools and the build tools to the PATH. For *sh*, *ksh* and *bash* users:

```
PATH=/opt/ecos/ecos-1.3.1/tools/bin:$PATH
PATH=/tools/H-i686-pc-linux-gnu/bin:$PATH
export PATH
```

For *csh* and *tcsh* users:

```
set path = ( /opt/ecos/ecos-1.3.1/tools/bin $path )
set path = ( /tools/H-i686-pc-linux-gnu/bin $path )
```

The Build Process

Build Tree Generation

An eCos build actually involves three separate trees. The component repository acts as the source tree, and for application developers this should be considered a read-only resource. The build tree is where all intermediate files, especially object files, are created. The install tree is where the main library `libtarget.a`, the exported header files, and similar files end up. Following a successful build it is possible to take just the install tree and use it for developing an application: none of the files in the component repository or the build tree are needed for that. The build tree will be needed again only if the user changes the configuration. However the install tree does not contain copies of all of the documentation for the various packages, instead the documentation is kept only in the component repository.

By default the build tree, the install tree, and the `ecos.ecc` savefile all reside in the same directory tree.

I build 2 directories called ecos_app and ecos_src

In the ecos_src I do the following

ecosconfig new linux //since the target is linux synthetic

ecosconfig tree

ecosconfig check (this basically checks on any conflicts and modifies the configuration file)

Once the build tree is made then we can just run “make” which will build the corresponding directories.

The application directory.

Copy the makefile from the /examples directory into the application directory .Modify the PKG_INSTALL_DIR to point to the build tree PATH. Also modify the makefile to include the files to be executed.

Remote target procedure

In order to run the above for a remote target build another directory called ecos_stub and do the following

ecosconfig new pc stubs

ecosconfig tree

make

copy the gdb_module.bin file using dd in linux into floppy.

The PATH and configuration would then accordingly change depending on the target you have(whether i386 or arm etc)

Then boot your target machine from the floppy and connect your machine to the target machine using a null modem serial line .

Run gdb on your machine and do the following:

i386-elf-gdb executable

At the gdb prompt run the following

Set remotebaud 38400

Set debug remote 1

Target remote /dev/yourserialport

Then once you get through this stage type

Load

Then continue

This is ensure the running of ecos applications.