

**Quentin Cartier**

# **Work Log – M2M Linux Distribution – Part 1 2021**

## **1. Preface by Pr. Olivier Gruber**

This document is a guide through building our own small Linux Distribution.

You have been given a fully functional script to build a bootable disk image with a Linux kernel mounting an

absolutely minimal root file system. The goal is therefore to understand how it is done.

This document is meant to help you navigate through the different required steps and tools. It has a structure,

it points to important tricks or skills or tools, it also ask questions. If you are discovering this domain, we encourage you to follow the overall guiding contents of this document. If you are more of an expert, feel free to re-organize this document into something that suits your skills and expertise. The starting point is to look at the overall structure in directories and files. Then, immediately look at the shell script "mkdist.sh", it is a completely functional script:

1. Build the "hello" directory
  1. Compile and link the "init process"
  2. Generate the initial ramdisk
2. Creates a bootable disk image with GNU MBR stage1
3. Partition the disk and create a file system
4. Populate the file system with enough to boot the Linux kernel from the directory "MiniDist".
5. Installing GRUB (manually or automatically)
6. Boot on QEMU

Try to get a general feel of these steps. Overtime, the goal is about making sense of these steps and learning

the required skills and necessary tools to achieve them.

### **A word about safety:**

Make sure that you do regular backups of your Linux install.

Be mindful and rigorous... shell scripting is potentially harmful.

Below is an example of a simple typo mistake that is fatal:

```
DIR=~john/M2M/tmp"
```

```
rm -rf ${DOR}/
```

The script above will result in wiping out your entire disk!!!! Bye-bye your Linux and your Data!

## **2. QEMU**

```
#Pre-requisite in order to launch the script
# We need qemu-system-x86
sudo apt install qemu-system-x86
sudo apt-get install libncurses5 libncurses5:i386

#Notes
-The script does not seem to work with Ubuntu 20.04
-A `sudo` may be needed in order to launch `parted` or/and `losetup`
```

The emulated platform is a regular Intel-based Personal Computer (PC). You will notice that we use "qemu-system-i386". You will notice that a new window pops up. This new window is the "screen and keyboard" of the PC. Indeed, as you know, a PC has a screen and a keyboard, always. The terminal window in which you launched QEMU remains the terminal connected via a serial line, like before when we worked with the Versatile-PB board.

FYI: Keyboard Emulation with QEMU:

You can ask QEMU to emulate a specific keyboard. Indeed, often, French users have an AZERTY keyboard and GRUB behaves as it is QWERTY.

```
(AWERTY keyboard):
$ qemu-system-i386 -k fr -m 256 -serial stdio -drive
format=raw,file=disk.img
```

```
(QWERTY keyboard):
# qemu-system-i386 -k en-us -m 256 -serial stdio -drive
format=raw,file=disk.img
```

## **3. Minimal File-System Layout**

Look at the script "mkdist.sh" and figure out the rôle of the directory "MiniDist".

MiniDist is the directory where we copy the distribution

<b>dd</b>	A command-line utility whose primary purpose is to convert and copy files.
<i>bs=BYTES</i>	read and write up to BYTES bytes at a time (default: 512); overrides ibs and obs
<i>if=FILE</i>	<i>Input File</i> Read from FILE instead of stdin
<i>of=FILE</i>	<i>Output File</i> Write to FILE instead if stdout
<i>count=N</i>	Copy only N input blocks
<i>conv=CONVS</i>	Convert the file as per the comma separated symbol list <b>Notrunc</b> is important to prevent truncation when writing into a file, it has no effect on a block device such as sda or sdb.

```
#Create the raw disk image
dd if=/dev/zero of=$DISK bs=512 count=256000 seek=256
```

```
(mkdist.sh : Steps)
# Install GRUB MBR ( stage1)
## Copy grub stage 1
dd conv=notrunc if=$GRUB_DIR/stage1 of=$DISK bs=512 count=1
[...]

#Partitioning the disk image
## Create msdos label, always the first step in creating a dist
parted -s $DISK mklabel msdos

## Create primary partition, starting at sector 256, taking all the
remaining sectors
parted -s $DISK mkpart primary ext2 256s 256000s
```

```
# Loop mount the disk image
## Grab available loop device (-f = find, determine the first non
used loop peripheral.)
DEVLOOP=`losetup -f`
sudo losetup -o131072 $DEVLOOP $DISK

# Creating the file system
sudo mkfs -t ext2 $DEVLOOP
sudo mkdir -p $MOUNT_DIR
sudo mount $DEVLOOP $MOUNT_DIR

# Copying the mini-dist contents

[...]
```

## 4. GRUB Bootloader

When launching the script "**mkdist.sh**", you get a GRUB menu on the screen of the emulated PC. Look at the script "mkdist.sh" and figure out how GRUB is installed on the created virtual disk for the emulated PC. You have been given the GRUB manual in the directory "*grub-0.97*", there are a few sections on installing GRUB. The GRUB menu is in MiniDist/boot/grub/menu.lst



Look in that menu at the two boot options, with or without the loading of an initial ramdisk:

1. *Linux 4.4 - Initial RamDisk Boot*
2. *Linux 4.4. - HardDisk Boot*

Explain the related options given to the Linux kernel:

```
root=/dev/ram rdinit=/hello
root=/dev/sda1 init=/hello
```

Look at this init process called hello and discuss the way we compiled and linked this executable “hello”.

Also discuss what is the makefile target “**initrd.hello**” about. Look up in Google “Linux initrd” and “Linux initial ramdisk”.

```
(grub-menu-1st)
```

```
title Linux 4.4 - root=/dev/ram - rdinit=/hello
kernel /boot/vmlinuz-4.4.113-disk root=/dev/ram rdinit=/hello
console=ttyS0,115200n8 console=tty0
initrd /boot/initrd.hello
boot
```

```
title Linux 4.4 - root=/dev/sda1 - init=/hello
kernel /boot/vmlinuz-4.4.113-disk root=/dev/sda1 init=/hello
console=ttyS0,115200n8 console=tty0
boot
```

<b>rdinit</b> =full_path_name	Run the init process from the ramdisk. This file must be on the kernel ramdisk instead of on the root filesystem.
<b>init</b> =filename	Program to run at init time. Run the specified binary as the init process instead of the default /sbin/init program

### **What it *initrd* ( Initial RamDisk) ?**

Initrd is a method for loading a temporary root file system into memory. The image is a file system image, which is made available in a special block device /dev/ram. That special device is then mounted as the initial file system.

## 5. Minimal File-System Layout

Under the directory "MiniDist", we have very little, essentially a directory "boot" containing the GRUB bootloader and two Linux kernels, with two distinct build configurations. You can compare the two configurations, there are text files, using a tool like "meld" for instance. Look at **CONFIG\_PRINTK**, see it is turned on in config-4.4.113-disk and not in config-4.4.113-ram. Now, reboot on QEMU the two kernels and notice how one is more verbose than the other?

(Comparing config with meld)



CONFIG\_PRINTK is set for disk, and not for ram.

`CONFIG\_PRINTK` :

- Enables normal printk support. Removing it eliminates most of the message strings from the kernel image and makes the kernel more or less silent. (source:<https://cateee.net/lkddb/web-lkddb/PRINTK.html>)

(Boot from ram)

```
Booting 'Linux 4.4 - Initial RamDisk Boot'
kernel /boot/vmlinuz-4.4.113-ram root=/dev/ram rdinit=/hello
[Linux-bzImage, setup=0x3c00, size=0xf5190]
initrd /boot/initrd.hello
[Linux-initrd @ 0xff86000, 0x49969 bytes]
boot
Hello! What is your name:
```

(Boot from hardisk)

Enabling CONFIG\_PRINTK makes the boot more verbose. It is useful to diagnose system issues at boot time.

```

Machine View
BIOS EDD facility v0.16 2004-Jun-25, 1 devices found
ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
ata2.00: configured for MWDMA2
ata1.00: ATA-7: QEMU HARDDISK, 2.5+, max UDMA/100
ata1.00: 256256 sectors, multi 16: LBA48
ata1.00: configured for MWDMA2
scsi 0:0:0:0: Direct-Access      ATA          QEMU HARDDISK    2.5+ PQ: 0 ANSI: 5
sd 0:0:0:0: [sdal] 256256 512-byte logical blocks: (131 MB/125 MiB)
sd 0:0:0:0: [sdal] Write Protect is off
sd 0:0:0:0: [sdal] Write cache: enabled, read cache: enabled, doesn't support DPC
or FUA
sda: sda1
scsi 1:0:0:0: CD-ROM           QEMU        QEMU DVD-ROM     2.5+ PQ: 0 ANSI: 5
sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
cdrom: Uniform CD-ROM driver Revision: 3.20
sd 0:0:0:0: [sdal] Attached SCSI disk
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/i
input4
JFS: Mounted root (ext2 filesystem) readonly on device 8:1.
Freeing unused kernel memory: 280K
tsc: Refined TSC clocksource calibration: 2711.970 MHz
clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x271769d4601, max_idle_r
s: 440795256941 ns
clocksource: Switched to clocksource tsc
Hello! What is your name:

```

Also look at CONFIG\_BLK\_DEV\_RAM, see the support for a ramdisk is only included in the kernel vmlinuz-4.4.113-ram which is consistent with the GRUB menu configuration. Now look at CONFIG\_MSDOS\_PARTITION, CONFIG\_IDE, CONFIG\_BLK\_DEV\_SD, and CONFIG\_BLK\_DEV\_RD...

It is clear that one kernel has support for IDE/SCSI mass storage peripherals (hard disks) and not the other.

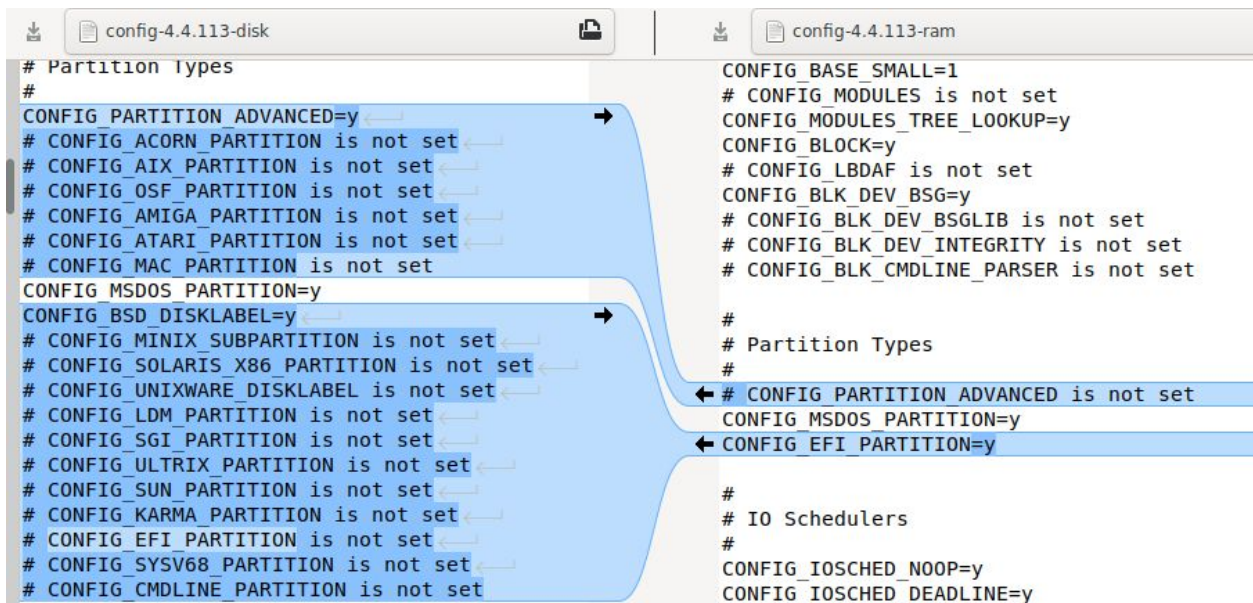
`CONFIG\_BLK\_DEV\_RAM`:

- RAM block device support. Saying yes (Y) allows to use a portion of the RAM memory as a block device. That means that it allows file systems to be on RAM, with possibility to read and write as a normal block device ( eg. : Hard drive) This is useful for example when using a live CD.
- Defined in : drivers/blocks/Kconfig



`CONFIG\_MSDOS\_PARTITION`:

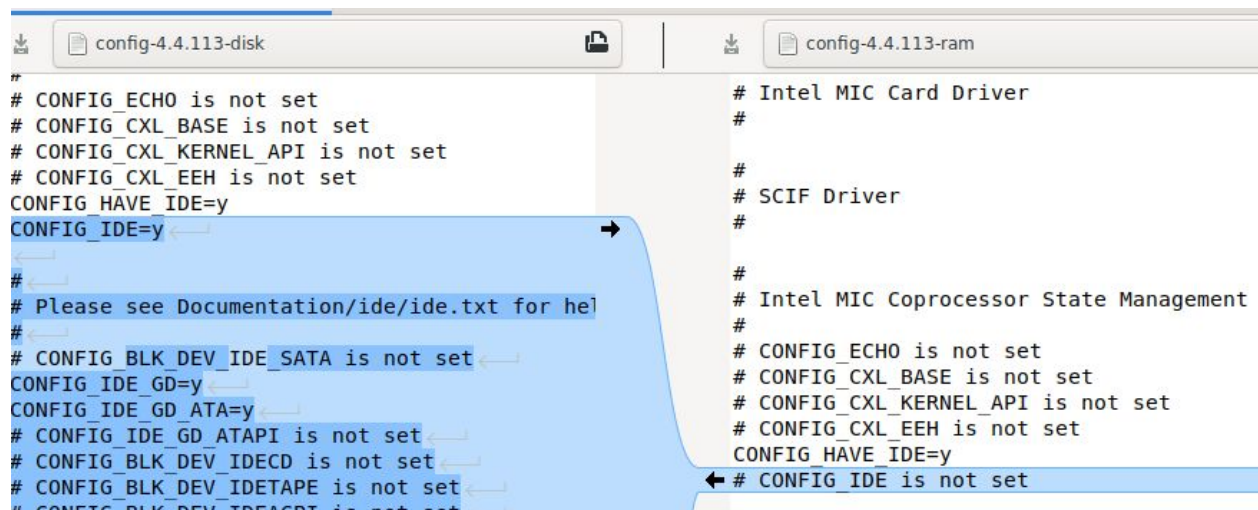
- PC BIOS (MSDOS partition tables) support
- defined in : block/partitions/Kconfig



`CONFIG\_IDE`:

- Y: Enable kernel to manage low cost mass storage units such as ATA/(E)IDE and ATAPI units
- Defined in : drivers/ide/Kconfig

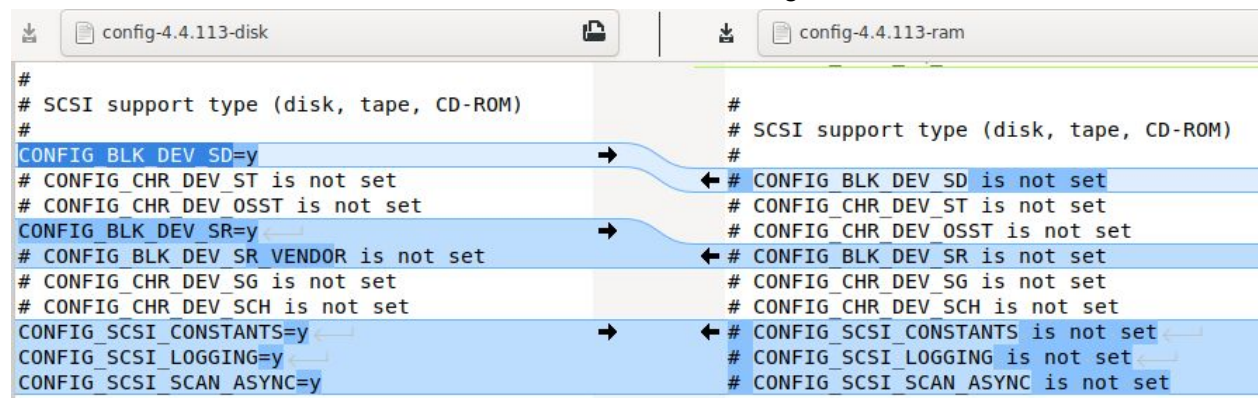




The configuration for the kernel on disk enables the support for IDE, whereas the configuration for the kernel on ram does not.

`CONFIG\_BLK\_DEV\_SD`:

- SCSI disk support. (Not SCSI CD-ROMs)
- Defined in : drivers/scsi/Kconfig



`SCSI`:

- Accr. :Small Computer Systems Interface.
- A standard interface and command set for transferring data between devices on a bus.

**Note:** you do not have to build your own kernel here, but it helps understand that the two kernels have been built with a different configuration that explains why one is suited to boot only an initial ramdisk while the other is suited to boot directly from the hard disk and does not support the use of an initial ramdisk.

## 6. The "init" Process

Look into the folder "hello" to see how we build our own executable for the "init" process.

- a. *Look at the makefile and explain however how the "hello" executable is linked to become a standalone executable?*

### **(hello/Makefile)**

```
# Under 64bit linux, you need the -m32 flag
# to force the gnu compiler to produce a 32bit executable.
# If your host environment is a 32bit linux, then you may omit that
flag.
# However, you must always retain the -static flag.
CFLAGS= -m32 -static
GCC=gcc

all: hello initrd.hello

#   echo Please copy initrd.hello under MiniDist/boot/
#   echo Also copy hello under /MiniDist/
#   echo Finally, make sure that you have an entry in
MiniDist/boot/grub/menu.lst for it!

clean:
    rm -f hello initrd.hello *~

# Produce the most simple root file system,
# using cpio, and solely containing the hello program.
initrd.hello: hello
    echo hello | cpio -o --format=newc | gzip -9 > initrd.hello
    file initrd.hello

# Compile the hello program in order to be able to
# execute it as the init program launch by a Linux kernel.
hello: hello.c
    $(GCC) $(CFLAGS) -o hello hello.c
```

We first build the executable hello with the flags -m32 and -static. The **static** option forces all dependencies to be linked statically, so that the binary can run on a machine without that the runtime installed.

**b. Explain why it is so big? Around 700KB...**

```
(ls hello) -rwxr-xr-x 1 1000 1000 670992 Feb 15 00:24 hello
```

The reason for that is that as the executable is linked statically, that means that the dependencies must exist inside the executable. When we look at the file hello.c, we see the following include directives:

```
#include<stdio.h>
#include<stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
```

In a statically linked executable, these includes must be contained in the executable, that is why the file is bigger than we would think.

**c. Explain why it is necessary to do so? Knowing that we will boot a Linux kernel on an almost-empty file system, including the hello executable and the “/boot” directory.**

In the case of an almost-empty file system, the strict minimum is installed, so the necessary shared libraries to use with hello.c are not installed . Because of that, we need to include the dependencies inside the executable, using static linking.

**d. Look at the GRUB menu and at the corresponding "init" options passed to the kernel:**

```
root=/dev/ram rdinit=/hello
root=/dev/sda1 init=/hello
```

**(grub-menu-lst)**

```
title Linux 4.4 - root=/dev/ram - rdinit=/hello
kernel /boot/vmlinuz-4.4.113-disk root=/dev/ram rdinit=/hello
console=ttyS0,115200n8 console=tty0
initrd /boot/initrd.hello
boot

title Linux 4.4 - root=/dev/sda1 - init=/hello
kernel /boot/vmlinuz-4.4.113-disk root=/dev/sda1 init=/hello
console=ttyS0,115200n8 console=tty0
boot
```

Look rapidly at the code, it is an easy read. The hello program is a simple parrot-like console that repeats whatever you type. Notice that you are back to regular programming here, unlike our previous experiments with bare-metal programming. Why? The reason is that you have the regular libC and a regular Linux kernel available.

To make sure you understood the compilation/linking/shared library, let's do a small modification here.

1. *Compile the hello program normally, no more static linking of libraries*
2. *Check what libraries it now depends on (use ldd)*

```
@debian:~/m2m_workload_code_m2gi_2021/Step1-Linux/hello$
ldd hello
    linux-gate.so.1 (0xf7fa2000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d9d000)
    /lib/ld-linux.so.2 (0xf7fa4000)
```

3. *Copy the necessary libraries in the right place*

Where can you find the needed libraries?

Could you use the ones from your development distribution, the one that you have installed on your personal machine? Well... we hope that you know you cannot: 64-bit vs 32-bit as a starter... and maybe not the same version of the kernel... and maybe not the same version of the toolchain, not configured the same way... We gave you some libraries and binaries that are compatible in the directory "debian-initrd". Where did we find them? Well, the kernel is from a Debian distribution, so we opened up the initrd from that distribution, and it obviously contains compatible binaries and libraries. Try out your work now, your recompiled hello program. Does it work with both boot options? It may, if you were thorough. Most probably, it runs only when booting directly from the disk and not when booting from the initial ramdisk. Explain why?

I created a `lib` directory at the root of MiniDist, where I put the shared libs from debian-initrd. From the Makefile, I removed the -static option, and rebuilt the executable. We can see that the hello file is no about 15kb only:

```
-rwxr-xr-x 1 1000 1000 15732 Feb 15 19:03 hello
```

### ***Why it doesn't work when booting from the initial ramdisk?***

The reason is that initrd.hello only contains a hello executable. However, it does not contain the content of MiniDist in which we installed the shared libraries, so we need to include the libs to the initrd.hello.

```
# Produce the most simple root file system,
# Old: using cpio, and solely containing the hello program.
# New: We include the MiniDist content in initrd.hello
initrd.hello: hello
    #echo hello | cpio -o --format=newc | gzip -9 > initrd.hello
    (cd ../MiniDist; find . | cpio -o --format=newc | gzip -9 >
../hello/initrd.hello)
    file initrd.hello
```

## **7. The Shell Setup**

Let's configure our minimal distribution to have a regular shell on the console, that is, the screen of the PC.

### ***7.1 The script "init"***

So we will use the following script, as our "init" process:

```
#!/bin/sh
exec /bin/sh +m
```

This is possibly the smallest init shell script ever !  
Put this script in the file MiniDist/init

***Try it out...Well?***

***Did you remember to update the GRUB menu, creating a new entry?***

To execute the script, we need to create a new entry in the GRUB menu

```
(New entry in menu.lst)
[...]
title Linux 4.4 - HardDisk Boot (Init)
kernel /boot/vmlinuz-4.4.113-disk root=/dev/sda1 init=/init
Boot
[...]
```

Did you make sure the "init" script is executable (rwxr-xr-x)

```
(ls)
-rwxr-xr-x 1 1000 1000 27 Feb 15 20:30 init
```

Try it again, it should work now.

- No: Panic

Still does not work? Well, look at the first line of the script: `#!/bin/sh`

Well, do you have a shell at `/bin/sh` ? Nope. In fact, do you have any commands available, right?

## ***7.2 The shell***

So we need a shell program... but wait, modern Linux distributions use something called busybox... we gave

You have a compatible version of it in the directory "debian-initrd".

`Busybox`:

- "The Swiss Army knife of Embedded Linux"
- Several unix utilities in a single exec. This single executable replaces basic functions of more than 300 common commands
- Runs in a variety of POSIX env. ( Linux, Android, FreeBSD).

So go ahead, research what busybox is about and install the core commands in your minimal distribution:

MiniDist/bin/busybox

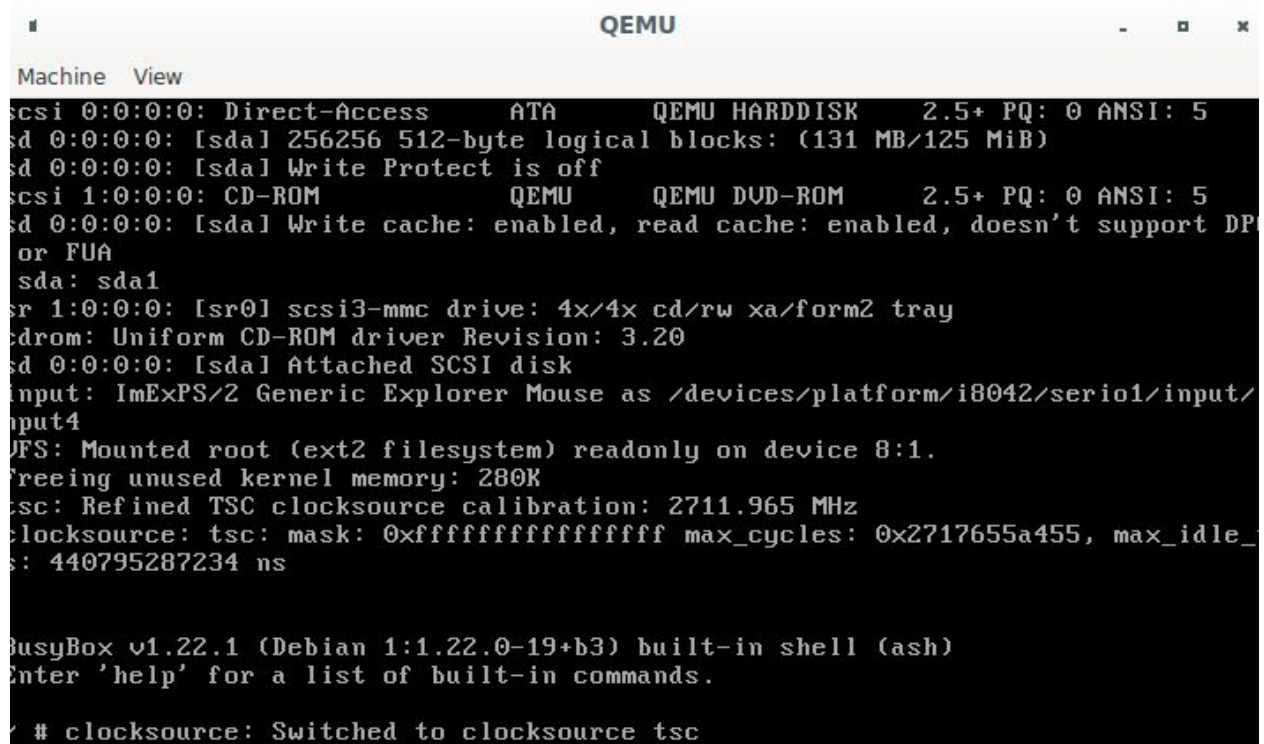
MiniDist/bin/sh // as a symbolic link to busybox

MiniDist/bin/lm // as a symbolic link to busybox

```
gigi@debian:~/m2m_workload_code_m2gi_2021/Step1-Linux/MiniDist/bin$  
ls -ln  
total 316  
-rwxr-xr-x 3 1000 1000 322316 Jan 16 2020 busybox  
lrwxrwxrwx 1 1000 1000 7 Feb 15 20:58 lm -> busybox  
lrwxrwxrwx 1 1000 1000 7 Feb 15 20:57 sh -> busybox
```

Try again to boot... It works, great!

(Working)



```
QEMU  
Machine View  
scsi 0:0:0:0: Direct-Access ATA QEMU HARDDISK 2.5+ PQ: 0 ANSI: 5  
sd 0:0:0:0: [sdal] 256256 512-byte logical blocks: (131 MB/125 MiB)  
sd 0:0:0:0: [sdal] Write Protect is off  
scsi 1:0:0:0: CD-ROM QEMU QEMU DVD-ROM 2.5+ PQ: 0 ANSI: 5  
sd 0:0:0:0: [sdal] Write cache: enabled, read cache: enabled, doesn't support DP  
or FUA  
sda: sda1  
sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray  
cdrom: Uniform CD-ROM driver Revision: 3.20  
sd 0:0:0:0: [sdal] Attached SCSI disk  
input: ImEXPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/  
input4  
VFS: Mounted root (ext2 filesystem) readonly on device 8:1.  
Freeing unused kernel memory: 280K  
tsc: Refined TSC clocksource calibration: 2711.965 MHz  
clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x2717655a455, max_idle_  
s: 440795287234 ns  
  
BusyBox v1.22.1 (Debian 1:1.22.0-19+b3) built-in shell (ash)  
Enter 'help' for a list of built-in commands.  
  
# clocksource: Switched to clocksource tsc
```

Well, not so great since you cannot do much, can you? There are no commands available. Why? Is it a path problem, well try it, once booted, print the environment variable PATH:

```
/ # echo $PATH
/sbin:/usr/sbin:/bin:/usr/bin
```

```
/ # echo $PATH
/sbin:/usr/sbin:/bin:/usr/bin
/ #
```

So it is not a PATH problem, the path is quite fine. It is just we do not have much commands under "/bin".

By the way, before we solved that problem, what is this weird prompt?

```
/ # cd bin
/bin # echo $PS1
\w \ $
/ #
```

Well, you can change it by changing the environment variable:

```
/bin # export PS1="$ "
$
```

Or you can leave it alone, at least, you learned about the existence of the PS1 environment variable that controls the prompts of your shell, even in your regular Linux.

## 7.3 The basic commands

So let's get back at not having commands. Well? Do you know how to fix the problem?

1. Copy more executable under MiniDist/bin?
2. Create symbolic links?

If you picked (1), you missed the busybox thing entirely.

So now, you can create the symbolic links under MiniDist/bin... manually, or we can do it from the init

Script:

```
(init)
```



```
#!/bin/sh
LINKS="echo sleep ls mkdir rmdir rm mv cp cat more dmesg mknod"
for link in $LINKS ; do[ ! -e /bin/$link ] && ln -s /bin/busybox
/bin/$link
done
sleep 2
echo "\n\nWelcome!\n"
exec /bin/sh +m
```

So? Oh well, did we forgot to say that the root file system is mounted **read-only** by the Linux kernel? Yep, it is...

## ***7.4 Remounting the root file system***

So we need to remount it read-write, like this:

```
echo "Remounting /dev/root in rw mode"
mount -o rw,remount /dev/root
```

So go ahead, add these two lines to your script "init".  
Works?

Not really, right? Unless you remembered to check that the command "mount" is provided by busybox and that you created the right symbolic link. You are lucky, our busybox provides the command "mount".

```
Remounting /dev/root in rw mode
mount: can't read '/proc/mounts': No such file or directory
ln: /bin/echo: Read-only file system
ln: /bin/sleep: Read-only file system
ln: /bin/ls: Read-only file system
ln: /bin/mkdir: Read-only file system
ln: /bin/rmdir: Read-only file system
ln: /bin/rm: Read-only file system
ln: /bin/mv: Read-only file system
ln: /bin/cp: Read-only file system
clocksource: Switched to clocksource tsc
ln: /bin/cat: Read-only file system
ln: /bin/more: Read-only file system
ln: /bin/dmesg: Read-only file system
ln: /bin/mknod: Read-only file system
/init: line 9: sleep: not found
\n\nWelcome!\n

BusyBox v1.22.1 (Debian 1:1.22.0-19+b3) built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

Still not working? What is the error message telling you? The message is about a missing `/proc/mounts`. We discussed the special file system `/proc`. As a special file system, it is mounted with a special arguments to the command "mount":

```
mount -t proc none /proc
```

Now it should all work fine.

**Note:** The directory ``proc`` should be created before, otherwise it will fail to mount none on `proc/mounts`.

## 7.5 Device Files

Now that we have a read/write root file system, we can add some code to the script "init" to create the

necessary device files:

```
mknod -m 666 /dev/ttyS0 c 4 64
```

```
mknod -m 666 /dev/tty0 c 4 0
```

```
mknod -m 666 /dev/tty1 c 4 1
```

```
mknod -m 666 /dev/null c 1 3
```

```
mknod -m 666 /dev/zero c 1 5
mknod -m 444 /dev/random c 1 8
mknod -m 444 /dev/urandom c 1 9
mknod -m 666 /dev/tty c 5 0
mknod -m 622 /dev/console c 5 1
mknod /dev/sda b 8 0
mknod /dev/sda1 b 8 1
```

Don't forget to do what is necessary for these lines to succeed and not fail.

## 7.6 The nano editor

Let's add a small editor, called "nano", to our small distribution. The binary is given in the directory "debian-initrd", it is an executable of its own right and not a symbolic link to busybox.

The nano executable is put in the bin directory.

**Note:** *since we did bare metal programming and you understood better the concept of a terminal and its relationship with the standard input/output of a C program, now you understand how the nano editor works by sending escape sequences to the terminal to move the cursor or change the colors.*

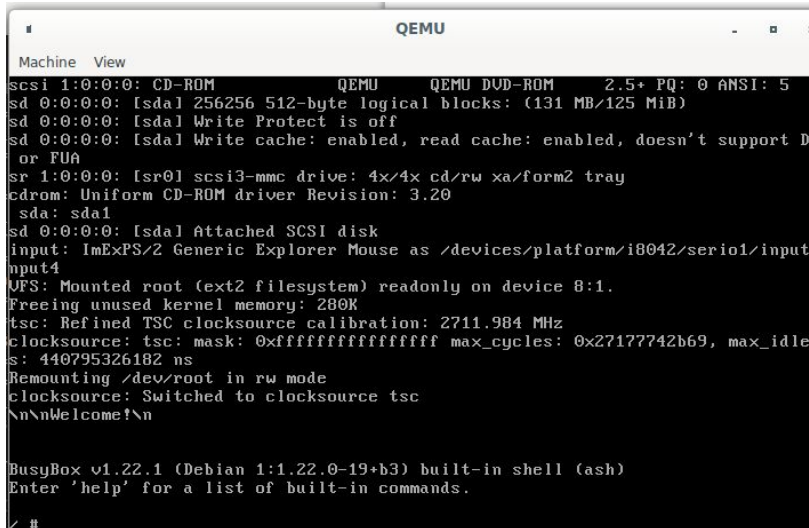
**Note:** *The same is true for programs like "top" or "htop" that show you the activity on your machine. Even better, you know now that these programs rely on /proc to obtain the information that they display.*

## 7.7 The Console

Let's discuss the console, as the terminal used to interact with our Linux kernel. We use QEMU to emulate a Intel-based personal computer, which means **QEMU emulates a screen and a keyboard, by default**. This is why QEMU opens a new window named "QEMU" when launched. So let's not confuse the two windows. One emulates the screen and keyboard devices of the emulated computer. The other, where you launch QEMU could be used as a terminal if QEMU was launched asking for a serial line on its standard input and output (-serial stdio). That being clear, you notice that when booting the Linux kernel configured to boot directly from disk, it prints a lot of stuff but only the last 24 lines or so remain visible. Also notice that the kernel prints stuff later one, once our shell has started, garbling the output.

So let's try to work with the console option passed to the kernel by the boot loader. If you pass this option, nothing changes:

```
console=tty0
```



```
Machine View
scsi 1:0:0:0: CD-ROM          QEMU   QEMU DVD-ROM    2.5+ PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 256256 512-byte logical blocks: (131 MB/125 MiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support D
or FUA
sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
cdrom: Uniform CD-ROM driver Revision: 3.20
sda: sda1
sd 0:0:0:0: [sda] Attached SCSI disk
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input
input4
VFS: Mounted root (ext2 filesystem) readonly on device 8:1.
Freeing unused kernel memory: 280K
tsc: Refined TSC clocksource calibration: 2711.984 MHz
clocksource: tsc: mask: 0xfffffffffffff max_cycles: 0x27177742b69, max_idle
s: 440795326182 ns
Remounting /dev/root in rw mode
clocksource: Switched to clocksource tsc
\n\nWelcome!\n

BusyBox v1.22.1 (Debian 1:1.22.0-19+b3) built-in shell (ash)
Enter 'help' for a list of built-in commands.

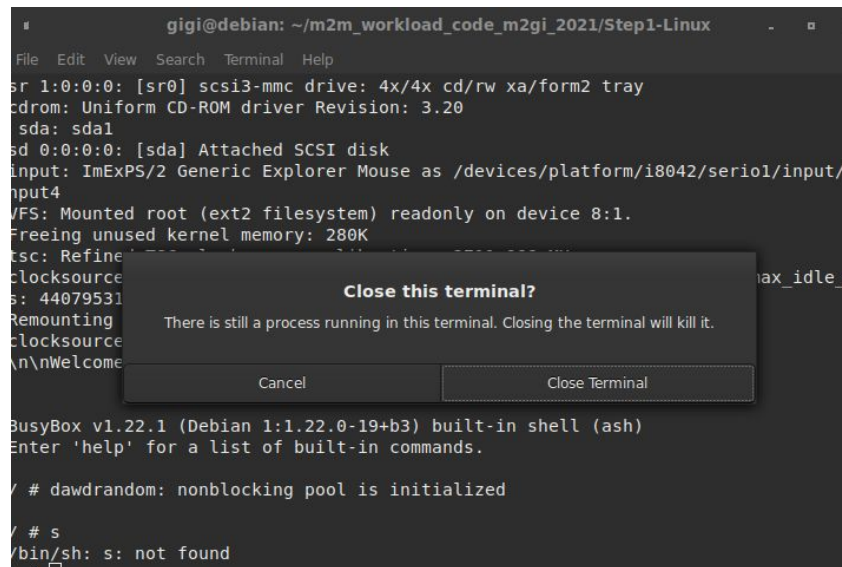
#
```

(A window in QEmu)

What happens if you pass this option instead?

`console=ttyS0,115200n8`

Ans: In that case, the terminal window is not a separate QEmu window, but is running in my linux terminal



```
gigi@debian: ~/m2m_workload_code_m2gi_2021/Step1-Linux
File Edit View Search Terminal Help
sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
cdrom: Uniform CD-ROM driver Revision: 3.20
sda: sda1
sd 0:0:0:0: [sda] Attached SCSI disk
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/
input4
VFS: Mounted root (ext2 filesystem) readonly on device 8:1.
Freeing unused kernel memory: 280K
tsc: Refined TSC clocksource calibration: 2711.984 MHz
clocksource: tsc: mask: 0xfffffffffffff max_cycles: 0x27177742b69, max_idle
s: 440795326182 ns
Remounting /dev/root in rw mode
clocksource: Switched to clocksource tsc
\n\nWelcome!\n

BusyBox v1.22.1 (Debian 1:1.22.0-19+b3) built-in shell (ash)
Enter 'help' for a list of built-in commands.

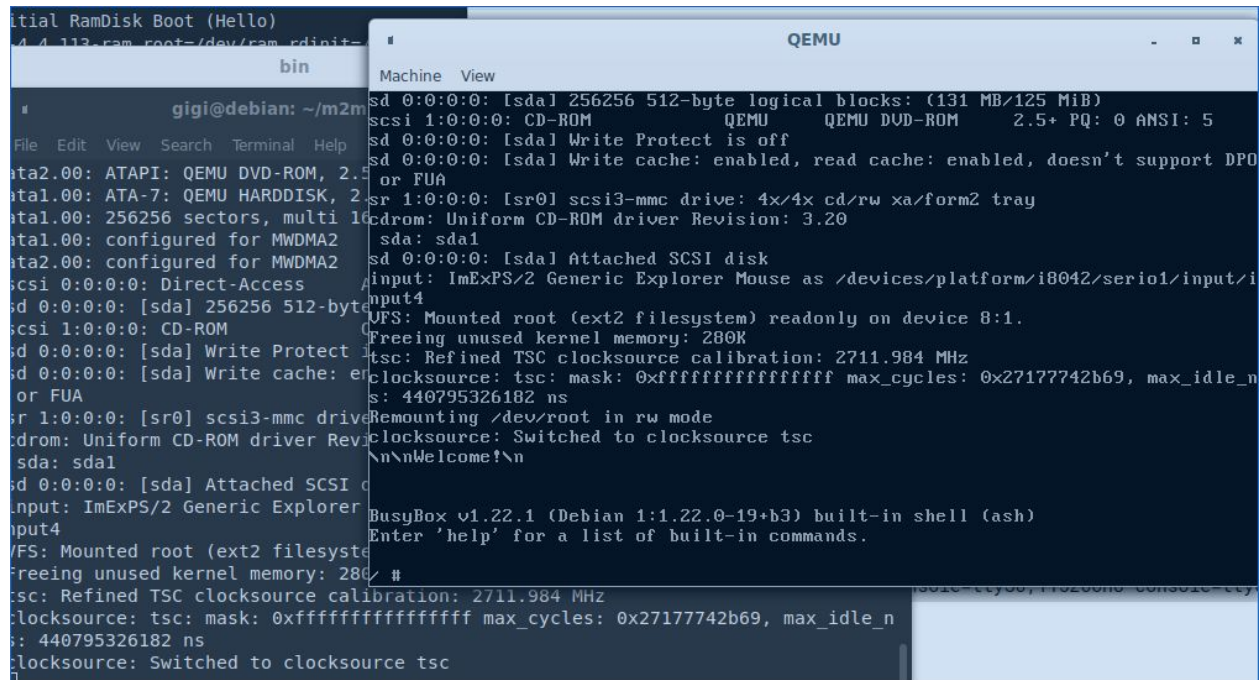
# dawdrandom: nonblocking pool is initialized

# s
/bin/sh: s: not found

#
```

What happens if you pass this option instead?

`console=ttyS0,115200n8 console=tty0`



The image shows a QEMU window titled "QEMU" with a "Machine View" tab. The main area displays boot logs for a virtual machine. The logs include information about the SCSI disk (sda), the CD-ROM (sr0), and the clocksource (tsc). The boot process completes, and the system switches to the clocksource tsc. A terminal window is overlaid on the QEMU window, showing the prompt "gigi@debian: ~/m2m" and the command "exec /bin/sh +m </dev/tty1 >/dev/tty1 2>&1". The terminal output shows the boot logs and the prompt "Welcome!\n".

```
Initial RamDisk Boot (Hello)
4.4.113-ram-root=/dev/ram rdinit=

bin

#
gigi@debian: ~/m2m
File Edit View Search Terminal Help

ata2.00: ATAPI: QEMU DVD-ROM, 2.5
ata1.00: ATA-7: QEMU HARDDISK, 2
ata1.00: 256256 sectors, multi 16
ata1.00: configured for MWDMA2
ata2.00: configured for MWDMA2
scsi 0:0:0:0: Direct-Access
sd 0:0:0:0: [sda] 256256 512-byte
scsi 1:0:0:0: CD-ROM
sd 0:0:0:0: [sda] Write Protect
sd 0:0:0:0: [sda] Write cache: en
or FUA
sr 1:0:0:0: [sr0] scsi3-mmc drive
cdrom: Uniform CD-ROM driver Revision: 3.20
sda: sda1
sd 0:0:0:0: [sda] Attached SCSI d
input: ImExPS/2 Generic Explorer
input4
VFS: Mounted root (ext2 filesystem)
Freeing unused kernel memory: 280K
tsc: Refined TSC clocksource calibration: 2711.984 MHz
clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x27177742b69, max_idle_n
s: 440795326182 ns
clocksource: Switched to clocksource tsc
Welcome!\n

BusyBox v1.22.1 (Debian 1:1.22.0-19+b3) built-in shell (ash)
Enter 'help' for a list of built-in commands.

gigi@debian: ~/m2m
#
tsc: Refined TSC clocksource calibration: 2711.984 MHz
clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x27177742b69, max_idle_n
s: 440795326182 ns
clocksource: Switched to clocksource tsc
```

The output is in both windows

Let's try to separate the output from the kernel and the one from our init shell. The kernel would print to the serial line, while the shell would be on the screen. For this, we need to combine a change in the GRUB menu with a change to the script "init".

For the GRUB menu, we want to pass the following option to the kernel:

```
console=ttyS0,115200n8
```

So the console will remain on the serial line. If we do nothing, the script "init" will also use the console as input/output. In fact, we cannot change that for the script "init" itself, but at the end, when it executes another shell, we can set a different tty for execution, like this:

```
exec /bin/sh +m </dev/tty1 >/dev/tty1 2>&1
```