



gridARM-EVBPEAK Linux BSP

Full operational Linux environment

User's Guide v0.9.0

Product names mentioned in this manual may be the trademarks or registered trademarks of their respective companies. They are not explicitly marked by "™" and "®".

© 2013 PEAK-System Technik GmbH

PEAK-System Technik GmbH
Otto-Roehm-Strasse 69
64293 Darmstadt
Germany

Phone: +49 (0)6151 8173-20
Fax: +49 (0)6151 8173-29

www.peak-system.com
info@peak-system.com

Document version 0.9.0 (2013-11-14)

Table of Content

Overview	4
Writing Conventions.....	5
Linux Host Commands.....	5
Board Commands	5
Board Support Package Usage	6
Requirements	6
Installation of the development environment.....	7
Building the U-Boot binary image	8
Building the Linux binary images.....	8
Using the binary images	10
Requirements	10
Downloading images using the network.....	10
Flashing from U-Boot	11
Flashing U-Boot	11
Flashing the Linux kernel.....	12
Flashing the root file system	13
Flashing from Linux	14
Flashing the Linux kernel.....	14
Flashing the root file system	15
NFS File System Access.....	16
Installing the NFS file system on the Host.....	16
Configuring the network interface	17
Mounting a NFS access from the Board	17
Booting from NFS	17
Linux Environment.....	19
Serial TTY	19
Status LED.....	20
I2C EEPROMs	21
I2C Real Time Clock	22
I2C PCA9555PW.....	23
I2C PCA954PW.....	25
NOR Flash	27
Analog to Digital Converter	30

MicroSD Card.....	31
SPI Flash.....	33
GEMAC/PHY.....	34
CAN Controller.....	35
USB Device Port (UDP)	37
Adding User Applications	42

Overview

This manual describes how to use the Board Support Package (BSP) provided by PEAK-System to install a full Linux environment, able to create binaries ready to run on the PEAK-System gridARM-EVBPEAK board.

The BSP is intended to work with the **uClinux-dist** development environment v20121024, which can be downloaded from:

<http://www.uclinux.org/pub/uClinux/dist/uClinux-dist-20121024.tar.bz2>

The provided package contains the following files:

File	Description
uImage	The Linux kernel image based on a 2.6.36 kernel modified to run on the gridARM-EVBPEAK board.
u-boot.bin	The U-Boot binary image, based on v2009.09 and modified to boot on the gridARM-EVBPEAK board.

If you plan to flash the board with these original image files, go on with paragraph "Using the binary images" on page 10.

If you plan to rebuild one of these files since you want to modify things, go on with paragraph "Board Support Package Usage" on page 6. In that case, it is strongly recommended to have some knowledge of the uClinux-dist development environment. For more information on uClinux-dist, please, see <http://www.uclinux.org/>.

Writing Conventions

The following writing and typographical conventions are used in this document.

Linux Host Commands

If not explicitly written, commands to type on the host command line interface or texts displayed on its screen are prefixed with a "\$" prompt and the police used is a "Courier" style. For example:

```
$ mkdir -p /mnt/gridarm_evb-rootfs
$
```

Board Commands

If not explicitly written, commands to type on the board command line interface, or texts displayed on its console are prefixed with a "/ #" prompt and the police used is a "Courier" style. For example:

```
/ # mount -t nfs -o nolock,rw 192.168.0.1:/mnt/gridarm_evb-rootfs /mnt/nfs
/ #
```

Board Support Package Usage

That BSP is provided to be used in the uClinux-dist environment v20121024. This environment enables to (re)build the Linux kernel and the root file system images, from the source files it contains. This paragraph describes the steps to follow to rebuild these binary images from scratch.

Requirements

In order for uClinux-dist (v20121024) to run, the Linux host MUST be installed with – at least - the several following packages:

```
mtd-utils  
  
genromfs  
  
zlib1g-dev
```

Moreover, the CodeSourcery ARM Linux tool chain must be installed on the Linux host. Here are the steps to do that:

1. Download the package:

```
$ wget https://sourcery.mentor.com/GNUToolchain/package8746/public/arm-  
uclinuxeabi/arm-2011.03-46-arm-uclinuxeabi.bin
```



Note that the below command MUST be typed without any carriage return except the one used to enter the whole command

2. Install the tool chain:

```
$ /bin/sh arm-2011.03-46-arm-uclinuxeabi.bin -i console  
->1- Typical  
  
Where would you like to install?  
Default Install Folder: /home/xxxx/CodeSourcery/Sourcery_G++_Lite  
  
ENTER AN ABSOLUTE PATH, OR PRESS <ENTER> TO ACCEPT THE DEFAULT  
: /where/is/installed/2011.03-46-arm-uclinuxeabi  
  
4- Don't create links  
ENTER THE NUMBER OF AN OPTION ABOVE, OR PRESS <ENTER> TO ACCEPT THE DEFAULT  
: 4
```



Warning: some recent versions of Ubuntu uses **Dash** as the default Shell, which is incompatible with some scripts in **GCC**. Operator MUST first move the symbolic link `/bin/sh` to another one, saying Bash, for example:

```
% sudo dpkg-reconfigure -plow dash
```

Then choose 'No' in the 'Configuring dash' popup dialog and press enter. You can run following command and check that `/bin/sh` points to 'bash':

```
$ ls -l /bin/sh
..... /bin/sh -> bash
```

3. Modify the system PATH according to the tool chain installation directory. For example:

```
$ export PATH="/where/is/installed/2011.03-46-arm-uclinuxeabi/bin:$PATH"
```

Installation of the development environment

1. Download and install the uClinux-dist environment:

```
$ wget http://www.uclinux.org/pub/uClinux/dist/uClinux-dist-20121024.tar.bz2
$ tar -xjf uClinux-dist-20121024.tar.bz2
$ cd uClinux-dist
$
```



Warning: the uClinux-dist environment CAN'T be moved easily, so the installation directory SHOULD be chosen carefully.

2. Uncompress the PEAK-System BSP for the gridARM-EVBPEAK board:

```
$ cd uClinux-dist
$ tar -xzf /somewhere/uClinux-vendors-PEAK-System-gridARM-EVBPEAK-2.11.0.tar.gz
$
```

3. Install the BSP:

```
$ make -C vendors/PEAK-System/gridARM-EVBPEAK install
$
```



Note: the first installation process could take time because it downloads the right Linux Kernel and Busybox packages from the Internet and applies the patches contained in the BSP.

4. Configure the BSP:

```
$ make menuconfig

Vendor/Product Selection --->
  --- Select the Vendor you wish to target
  Vendor (PEAK-System) --->
  --- Select the Product you wish to target
  PEAK-System Products (gridARM-EVBPEAK) --->

Kernel/Library/Defaults Selection --->
  --- Kernel Version (linux-2.6.36) --->
  Libc Version (uClibc) --->
```

Exit then save the new configuration.

Building the U-Boot binary image

The U-Boot binary image is made with the below special "make uboot" command, in the uClinux-dist environment root directory:

```
$ cd uClinux-dist
$ make -C vendors/PEAK-System/gridARM-EVBPEAK uboot
...
```

At the end of this command, the binary image of U-Boot is stored into the images directory:

```
$ ls images/u-boot.bin
u-boot.bin
$
```

Building the Linux binary images

Building the binaries is done with the "make" command, in the uClinux-dist environment root directory:

```
$ cd uClinux-dist
$ make
...
```

The first "make" command takes a long time, mainly because uClinux-dist has to configure the sources packages before building them. All further "make" commands should take less time.

At the end of the "make" command, all of the Linux binary images have been built.

The root file system image(s) are stored into the images directory:

```
$ ls images
romfs.img  rootfs.cpio  rootfs.jffs2  rootfs.tar.gz
$
```

The Linux Kernel image to be booted from U-Boot is located in the Kernel tree:


```
$ ls linux-2.6.36/arch/arm/boot/uImage  
linux-2.6.36/arch/arm/boot/uImage  
$
```



Note that the original configuration of the BSP generates an *initramfs* image of the Linux Kernel, so that it can be downloaded into the board as a standalone Linux system, without the need of flashing any root file system too.

Using the binary images

This chapter deals with how to use the binary images of Linux and the root file system. The gridARM-EVBPEAK board should already be flashed with these images, especially with a U-Boot image.

Requirements

A serial 232 cable has to be connected to the DBGU TTY port of the board. The other end should be connected to a serial port of the host, where a terminal emulator software is able to run and to handle a 38400 Baud , 8 bits with no parity and 1 stop bit serial connection. Under Linux O.S., operator may use “minicom” software on the correct *tty* port. For example, if the cable is connected to *ttyS1*:

```
$ minicom -w -D /dev/ttyS1 -b 38400 -8
```

After power-up, and in order to stop the auto boot process of the board, operator has got a few seconds to first hit any key.

Downloading images using the network

In order to download anything from the network from U-Boot, U-Boot network environment has to be correctly setup. In particular, operator has to refer to his network administrator and to the U-Boot documentation to set valid IP addresses.

The rest of that document supposes that the board IP address is the fixed value 192.168.0.111 and the network mask is the corresponding 255.255.255.0 value:

```
gridARM-EVBPEAK U-Boot> printenv
bootdelay=3
baudrate=115200
loadaddr=0x80800000
ethaddr=xx:xx:xx:xx:xx:xx
ipaddr=192.168.0.111
serverip=192.168.0.1
...
Environment size: XXX/XXXXXX bytes
gridARM-EVBPEAK U-Boot>
```

Operator has also to set the “*serverip*” environment variable to the IP address of the host used to download the image files (see the *setenv* command in U-Boot documentation). That host must also run a TFTP server (192.168.0.1 in the above example). On a Linux host, operator may refer to the *inetd/xinetd* and *tftpd* related documentation.

In order to test the network connection with the server host, operator may use the U-Boot “*ping*” command:

```
gridARM-EVBPEAK U-Boot> ping 192.168.0.1
host 192.168.0.1 is alive
gridARM-EVBPEAK U-Boot>
```

Downloading any file on the board is done using the U-Boot “tftp” command. For example, downloading the “uImage” file from the TFTP server repository:

```
gridARM-EVBPEAK U-Boot> tftp 20000000 uImage
TFTP from server 192.168.0.1; our IP address is 192.168.0.111
Filename 'uImage'.
Load address: 0x20000000
Loading: #####
#####
#####
#####
#####
done
Bytes transferred = NNNNN (XXXXX hex)
gridARM-EVBPEAK U-Boot>
```

Second parameter of the “tftp” command is the memory address where the download data are stored. Here, data are stored at memory location 0x20000000 which corresponds to a memory area left to download usage.

NNNNN (XXXXX) is the size in decimal (hexadecimal) of the downloaded data. This value will be used later.

Flashing from U-Boot

This paragraph describes how to flash an image into the flash memory of the board, from the U-Boot environment, when JP200 is closed and JP201 is on position 2-3.

First step before flashing anything from U-Boot is to download the file into the board memory space. Please, refer to paragraph “Downloading images using the network” on page 10.

When JP200 is closed and JP201 is on position 2-3, the following static partitions table is defined for the gridARM-EVBPEAK board:

Address	Size	Description
0x00000000	0x00400000 (4 MB)	U-Boot image in first 10 sectors
0x30000000	0x00400000 (4 MB)	Linux kernel partition
0x30400000	0x00400000 (4 MB)	JFFS2 root file system partition

Flashing U-Boot

Flashing a U-Boot image needs first that the right image being downloaded into the memory of the board. See for example “Downloading images using the network” on page 10, to know how downloading a file using TFTP and a network connection.



Warning: flashing a U-Boot image SHOULD BE done with extreme attention! Flashing any wrong image instead of a valid one COULD definitively damage the board!

```
gridARM-EVBPEAK U-Boot> tftp 20000000 u-boot.bin
Using GridARM_GMAC device
TFTP from server 192.168.0.1; our IP address is 192.168.0.111
Filename 'u-boot.bin'.
Load address: 0x20000000
Loading: #####
done
Bytes transferred = 206032 (324d0 hex)
gridARM-EVBPEAK U-Boot>
```

Flashing U-Boot needs first to unprotect the sectors before copying SDRAM content into Flash memory:

```
gridARM-EVBPEAK U-Boot> protect off 0 4ffff
Un-Protected 12 sectors
gridARM-EVBPEAK U-Boot> erase 0 4ffff
..... done
Erased 12 sectors

gridARM-EVBPEAK U-Boot> cp.b 20000000 0 324d0
Copy to Flash... 9....8....7....6....5....4....3....2....1....done
gridARM-EVBPEAK U-Boot>
```

Note that the above `cp.b` command copies **206032** bytes (*324d0H*) from SDRAM to the NOR flash. This value corresponds to the size of the **u-boot.bin** image downloaded first on the board in the example, using the `tftp` command. This size could be different.

Flashing the Linux kernel

Flashing a kernel image needs that the image being first downloaded into the memory of the board. See for example “Downloading images using the network” on page 10, to know how downloading a file, using TFTP and a network connection.

Supposing that the Kernel image is saved in SDRAM from address 0x20000000, the commands to erase then program the Flash memory from U-Boot are:

```
gridARM-EVBPEAK U-Boot> erase 30000000 +400000
..... done
Erased 39 sectors

gridARM-EVBPEAK U-Boot> cp.b 20000000 30000000 400000
gridARM-EVBPEAK U-Boot>
```

Note that **400000** tells the `cp.b` command to copy up to 4 MB of data from SDRAM to the NOR flash. This value corresponds to the whole size of the flash partition. To speed-up the copy, this value could be replaced by the value of the real count of bytes of the flashed image of the Kernel (see `tftp` command output).

Flashing the root file system

Any Linux Kernel needs a root file system to boot. The original configuration of the BSP creates an *initramfs* version of the Kernel, but it can be useful to boot on a root file system flashed in NOR instead. The BSP also creates a JFFS2 image of the root file system. To boot on, the JFFS2 image must be downloaded first (see also “Downloading images using the network” on page 10).


For example, here are the commands to download then flash a root file system JFFS2 image into the right partition of the flash memory:

```
gridARM-EVBPEAK U-Boot> tftp 20000000 rootfs.jffs2
  TFTP from server 192.168.0.1; our IP address is 192.168.0.111
  Filename 'rootfs.jffs2'.
  Load address: 0x20000000
  Loading: #####
           #####
           #####
           #####
           #####
done
  Bytes transferred = NNNNN (XXXXXX hex)
gridARM-EVBPEAK U-Boot>
```

The corresponding Flash partition must be erased before being flashed:

```
gridARM-EVBPEAK U-Boot> erase 30400000 +400000
..... done
Erased 39 sectors

gridARM-EVBPEAK U-Boot> cp.b 20000000 30400000 400000
gridARM-EVBPEAK U-Boot>
```



Note that 0x400000 tells the `cp.b` command to copy up to 4 MB of data from SDRAM to the NOR flash. This value corresponds to the whole size of the flash partition. To speed-up the copy, this value could be replaced by a rounded value of the real count of bytes of the flashed image of the root file system (see XXXXX in the `tftp` command output).

In order to use a flashed root file system, the Kernel MUST first be configured WITHOUT the `initramfs` option:

```
$ cd uClinux-dist
$ make menuconfig

Kernel/Library/Defaults Selection --->
  [*] Customize Kernel Settings
```

Then Exit and Save to enter the Kernel configuration menu. Be sure to remove the "*" in the "Initial RAM file system and RAM disk (`initramfs/initrd`) support" option:

```
General setup --->
[ ] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```

Then Exit then Save everything. Rebuild the Kernel and transfer then flash the new *ulmage* to the board:

```
$ make linux
```

When a root file system is to be used from Flash, the boot arguments of U-Boot MUST be changed accordingly (note that the below command MUST be typed without any carriage return except the one used to enter the whole command):

```
gridARM-EVBPEAK U-Boot> setenv bootargs console=ttyS0,38400 root=/dev/mtdblock1
rootfstype=jffs2 rw
gridARM-EVBPEAK U-Boot>
```

Flashing from Linux

This paragraph describes how to write an image in the NOR flash of the board, from the Linux environment.

In configuration JP200 closed and JP201 on position 2-3, the Kernel statically defines the following partitions table for the 32-bits wide 8MB NOR flash:

```
/ # cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00400000 00020000 "kernel"
mtd1: 00400000 00020000 "rootfs"
mtd2: 00400000 00001000 "user"
/ #
```

Both first 4MB partitions are defined for storing a Kernel image and a root file system (JFFS2) image.

Flashing the Linux kernel

Flashing a kernel image needs that the image being first downloaded onto the board.

Example below shows the steps to download a Kernel image using TFTP¹ and to flash it in the dedicated flash partition and how to reboot on it:

¹ A network interface MUST be configured before (see page 11).

```
/ # cd /tmp
/var/tmp # tftp -g -r uImage 192.168.0.1
/var/tmp # ls -l
total 1896
drwxr-xr-x    2 root    root          80 Jan  1 23:53 dbus
-rw-r--r--    1 root    root           6 Jan  1 23:53 ifstate
srw-rw-rw-    1 root    root           0 Jan  1 23:53 log
-rw-r--r--    1 root    root           4 Jan  1 23:53 messagebus.pid
-rw-r--r--    1 root    root        7854 Jan  2 00:04 messages
drwxr-xr-x    2 root    root          60 Jan  1 23:53 subsystems
-rw-r--r--    1 root    root       1915364 Jan  2 00:05 uImage
-rw-r--r--    1 root    root           0 Jan  1 23:53 utmp
```

The downloaded image is then flashed into *mtd0* partition using the *flashcp* tool:

```
/var/tmp # flashcp uImage /dev/mtd0
/var/tmp #
```

Once the Kernel image flashed, reboot the board, copy the image from the NOR flash and boot from U-Boot:

```
/var/tmp # reboot
...
gridARM-EVBPEAK U-Boot> cp.b 0x30000000 0x20000000 0x300000
gridARM-EVBPEAK U-Boot> bootm
```

Flashing the root file system

Any Linux Kernel needs a root file system to boot. To flash a new root file system in the Flash memory, its JFFS2 image must be downloaded first onto the board. Next, the image must be written into the right partition.



Warning: a new file system image SHOULD be carefully flashed from Linux, especially when the running Kernel's root file system is mounted on the Flash (in this condition, the root file system overwrites itself!)

The following example works when Linux booted on a root file system NOT in Flash (NFS or *initramfs* configuration of the Kernel, for example):

```

/ # cd /tmp
/var/tmp # tftp -g -r rootfs.jffs2 192.168.0.1
/var/tmp # ls -l
total 1896
...
-rw-r--r--    1 root    root      8126464 Jan  2 00:20 rootfs.jffs2
...
/var/tmp # cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00400000 00020000 "kernel"
mtd1: 00400000 00020000 "rootfs"
mtd2: 00400000 00001000 "user"

/var/tmp # flashcp -v rootfs.jffs2 /dev/mtd1
Erasing blocks: 21/21 (100%)
Writing data: 2688k/2688k (100%)
Verifying data: 2688k/2688k (100%)

/var/tmp #

```

This new JFFS2 file system can be tested by mounting the corresponding block device onto a newly created mount point:

```

/var/tmp # mkdir -p /mnt/nor
/var/tmp # mount -t jffs2 /dev/mtdblock1 /mnt/nor
/var/tmp # ls /mnt/nor
bin  dev  etc  home  init  lib  mnt  proc  sbin  sys  tmp  usr  var
/var/tmp #

```

The new root file system is now ready to boot on (see "Flashing the root file system" on page 13 to see how the `bootargs` environment variable in U-Boot should be changed before).

NFS File System Access

This paragraph deals with the Network File System access from the board.

Installing the NFS file system on the Host

First of all, a valid file system for the gridARM-EVBPEAK board has to be installed on a host, in the local network. The next steps describe how to install such a file system on a Linux host, as well as how to configure that host to be a valid NFS server for the board. Root privileges for some of the below commands are requested.

1. Uncompress the provided root file system compressed archive to a safe place. For example:

```

$ cd uClinux-dist
$ mkdir -p /mnt/gridarm_evb-rootfs
$ tar -C /mnt/gridarm_evb-rootfs -xzf images/rootfs.tar.gz
$

```

2. Export the new directory so it will be visible by any NFS client, by adding to the NFS server `/etc/exports` file, the following line

```

/mnt/gridarm_evb-rootfs *(rw,sync,no_root_squash,no_subtree_check)

```


3. Finally, restart the NFS server. For example:

```
$ /etc/init.d/nfs-kernel-server restart
```

Configuring the network interface

On gridARM-EVBPEAK, the Ethernet interface name is "eth0". Before using any network access, this connection must be configured. Here are the steps that show how to setup a static IP address (192.168.0.11 for example) to interface "eth0":

```
/tmp # ifconfig eth0 192.168.0.111
/tmp # ifconfig eth0 up
/tmp # ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 data bytes
64 bytes from 192.168.0.1: seq=0 ttl=64 time=3.125 ms
64 bytes from 192.168.0.1: seq=1 ttl=64 time=1.035 ms
^C
--- 192.168.0.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 1.035/2.080/3.125 ms
```

Mounting a NFS access from the Board

When Linux boots on the gridARM-EVBPEAK board, it might be useful to use the network file system access to get or put some files to/from a Linux host on the network. This BSP is setup to enable mounting a NFS file system access from the gridARM-EVBPEAK board. This can be done with the next commands (supposing that the NFS server IP address is 192.168.0.1 and that it exports the /mnt/gridarm_evb-rootfs directory):

```
/ # mkdir -p /mnt/nfs-rootfs
/ # mount -t nfs -o nolock,rw 192.168.0.1:/mnt/gridarm_evb-rootfs /mnt/nfs-rootfs/
/ # ls -l /mnt/nfs-rootfs/
total 1
drwxr-xr-x  2 root  root           0 Jul 12  2011 bin
drwxr-xr-x  6 root  root           0 Jul 22  2011 dev
drwxr-xr-x  5 root  root           0 Jun 15  2011 etc
drwxr-xr-x  5 root  root           0 Jul 12  2011 home
lrwxrwxrwx  1 root  root           9 Jun 15  2011 init -> sbin/init
drwxr-xr-x  3 root  root           0 Dec 31 17:00 lib
lrwxrwxrwx  1 root  root          11 Jul 12  2011 linuxrc -> bin/busybox
drwxr-xr-x  3 root  root           0 Jul 12  2011 mnt
drwxr-xr-x  2 root  root           0 Jul 12  2011 opt
drwxr-xr-x  2 root  root           0 Jul 12  2011 proc
drwxr-xr-x  2 root  root           0 Jun 15  2011 root
drwxr-xr-x  2 root  root           0 Jul 12  2011 sbin
drwxr-xr-x  2 root  root           0 Jul 12  2011 sys
drwxrwxrwt  3 root  root           0 Dec 31 17:00 tmp
drwxr-xr-x  6 root  root           0 Jul 12  2011 usr
drwxr-xr-x  3 root  root           0 Jun 15  2011 var
```

Booting from NFS

This BSP is setup to enable the Linux Kernel to boot on a Network File System, from the gridARM-EVBPEAK board. This can be done with the next commands (supposing that the NFS server IP address is 192.168.0.1 and that it exports the /mnt/gridarm_evb-rootfs directory):

1. First step is to change the `bootargs` environment variable from U-Boot, to tell Linux to get its IP address with DHCP and its root file system on the remote host at address 192.168.0.1:

```
gridARM-EVBPEAK U-Boot> setenv bootargs 'console=ttyS0,38400 ip=dhcp root=/dev/nfs
nfsroot=192.168.0.1:/mnt/gridarm_evb-rootfs rw'
gridARM-EVBPEAK U-Boot>
```

2. Next, download (or copy) any Linux image into SDRAM and boot it:

```
gridARM-EVBPEAK U-Boot> bootm
## Booting kernel from Legacy Image at 20000000 ...
   Image Name:   Linux-2.6.36
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2170176 Bytes = 2.1 MiB
   Load Address: 20008000
   Entry Point:  20008000
   Loading Kernel Image ... OK
OK

Starting kernel ...

Linux version 2.6.36 (peak@Ubuntu-12) (gcc version 4.5.2 (Sourcery G++ Lite 2011
.03-46) ) #549 Thu Jan 3 16:23:00 CET 2013
CPU: Gridconnect-gridARM [83770946] revision 6 (ARMv4T), cr=00000000
CPU: VIVT data cache, VIVT instruction cache
Machine: PEAK-System gridARM EVB
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: console=ttyS0,38400 ip=dhcp root=/dev/nfs nfsroot=192.168.
0.1:/home/peak/linux/rootfs/gridarm_evb rw
...
eth0: Link down.
eth0: gridARM ethernet at 0x90000000 irq=26 (00:23:24:25:26:22)
eth0: Micrel KSZ902x PHY (Copper)
...
eth0: Link down.
Sending DHCP requests .
eth0: Link now 100-FD
eth0: Link down.
eth0: Link now 100-FD
.. OK
IP-Config: Got DHCP answer from 192.168.0.1, my address is 192.168.0.116
IP-Config: Complete:
   device=eth0, addr=192.168.0.116, mask=255.255.255.0, gw=255.255.255.255,
   host=192.168.0.116, domain=, nis-domain=(none),
   bootserver=192.168.0.1, rootserver=192.168.0.1, rootpath=
Looking up port of RPC 100003/3 on 192.168.0.1
Looking up port of RPC 100005/3 on 192.168.0.1
VFS: Mounted root (nfs filesystem) on device 0:9.
Freeing init memory: 92K
init started: BusyBox v1.10.2-uc0 (2013-01-03 16:24:43 CET)
starting pid 24, tty '': '/etc/rc'
starting pid 30, tty '/dev/ttyS0': '/bin/sh'
udhcpd (v1.10.2-uc0) started
/ # Sending discover...
Sending select for 192.168.0.116...
Lease of 192.168.0.116 obtained, lease time 43200
udhcpd: cannot background in uclinux (yet)

/ #
```

Linux Environment

The Linux package is made of two parts: the Kernel space and the user space.

The embedded Linux Kernel is based on v2.6.36. It is configured for the gridARM-EVBPEAK platform from PEAK-System Technik, and includes all the necessary drivers to access the numerous components present on the board:

- The serial TTYS, handled by the specific driver `drivers/serial/gridarm_serial.c`.
- The "status" LED present on the front of the gridARM-EVBPEAK is controlled by the Linux "heartbeat" trigger.
- The SPI controller handled by the specific driver `drivers/spi/gridarm_spi.c`.
- The I2C controller supported by the specific driver `drivers/i2c/busses/i2c-gridarm-irq.c`.
- I2C EEPROMs supported by the standard driver `drivers/misc/eeprom/at24.c`.
- I2C Real-Time Clock handled by `drivers rtc/rtc-m41t80.c`.
- I2C SPI Flash, handled by the specific driver `drivers/mtd/devices/m25p80.c`.
- I2C PCA9555PW and PCA9554PW GPIO Extenders controlling User LEDs, buttons and LCD display, handled by the driver `drivers/gpio/pca953x.c`.
- GPIO driven LCD display handled by the specific driver `drivers/staging/panel-gpio/panel-gpio.c`.
- The NOR Flash chips, supported by `drivers/mtd/chips/jedec_probe.c`.
- The Analog-to-Digital Converter, controlled by the ADC specific driver `drivers/staging/iio/adc/gridarm-adc.c`.
- The micro-SD card reader.
- The Gigabit Ethernet Controller as well as the PHY are driven by `drivers/net/arm/gridarm_eth.c`.
- The CAN controller is driven by the *linux-can* stack, through the specific driver `drivers/net/can/gridarm_can.c`.
- The USB Device "gadget" port, through the specific driver `drivers/usb/gadget/gridarm_udc.c`.

Moreover, the BSP is setup to also include all the necessary tools and/or configuration files to enable access to these components from the user space.

Serial TTY

Two serial ports are available on the gridARM-EVBPEAK platform:

- The gridARM SoC DBGU unit, which is connected to `ttys0`. The Linux system is setup to start a Shell on this TTY so that operator gets a command line interface as soon as the Kernel boot sequence ended. The default baud rate of this connection is 38400. This serial connection is also the default console used by the Kernel for early debug messages.

```
static struct resource dbgu_resources[] =
{
    [0] = {
        .start = GA_VA_BASE_SYS + GA_DBGU,
        .end   = GA_VA_BASE_SYS + GA_DBGU + SZ_512 - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = GA_AIC_ID_SYS,
        .end   = GA_AIC_ID_SYS,
        .flags = IORESOURCE_IRQ,
    },
};
```

- The gridARM SoC USART0 unit, which is connected to *ttyS1*. The Linux system is setup to start a Shell on this TTY at 115200 bauds.

```
static struct resource uart0_resources[] =
{
    [0] = {
        .start = GA_BASE_US0,
        .end   = GA_BASE_US0 + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = GA_AIC_ID_USART0,
        .end   = GA_AIC_ID_USART0,
        .flags = IORESOURCE_IRQ,
    },
};
```

Both TTY connections need a crossover cable.

Connecting to the board using USART0 needs to log in. The default configuration of the system defines a unique user "root" with no password. It is strongly recommended to define a new password to root as soon as possible, using embedded `passwd` command. For example:

```
/ # passwd root
Changing password for root
New password: alakazam
Retype password: alakazam
Password for root changed by root
/ #
```

Note that the password is encrypted and saved into the current root file system. Booting the same Kernel on another root file system will have different behavior!

Status LED

The status LED is controlled with PIN15 of the gridARM PIOA. The Kernel automatically attaches the "heartbeat" system trigger to this LED, if the option is configured in the Kernel (default):

```
static struct gpio_led evb_leds[] = {
#ifdef CONFIG_LEDS_GPIO_PLATFORM
    {
        .name           = "cpu",
        .default_trigger = "heartbeat",
        .active_low     = 1,
        .gpio           = GA_PIN_PA(15),
    },
#endif
};
```

The Kernel is also configured to enable access to this *cpu* LED, from the user space, through the *sysfs* interface:

```
/ # ls /sys/class/leds/cpu
brightness      max_brightness  subsystem      uevent
device          power            trigger

/ # cat /sys/class/leds/cpu/trigger
none timer [heartbeat] gpio
```

I2C EEPROMs

The gridARM-EVBPEAK board includes a mandatory I2C 256B EEPROM at address 0x50:

```
...
i2c /dev entries driver
at24 0-0050: 256 byte 24c02 EEPROM (writable)
...
```

```
static struct at24_platform_data mc24c02 = {
    .byte_len   = SZ_2K / 8,          /* 256B */
    .page_size  = 4,                  /* 4-Byte Page Write Buffer */
};

static struct i2c_board_info __initdata evb_i2c_devices[] = {
    {
        .addr = 0x50,
        .type = "24c02",
        .platform_data = &mc24c02,
    },
};

...
};
```

The Kernel is configured to enable read access to the EEPROM through the *sysfs* interface so that dumping their content can simply be done with using the included *hexdump* tool:

```
/ # hexdump /sys/bus/i2c/drivers/at24/0-0050/eeprom
00000000 ffff ffff ffff ffff ffff ffff ffff
*
0000100
/ #
```

User space file system also include `eeeprog` tool which is able to program the EEPROM chips. For example, writing the characters string "Hello World!" in EEPROM at address 0x50 is done with the following command:

```
/ # echo "Hello World!" | eeeprog -f -16 /dev/i2c-0 0x50 -w 0:13
eeeprog 0.7.5, a 24Cxx EEPROM reader/writer
Copyright (c) 2003 by Stefano Barbato - All rights reserved.
  Bus: /dev/i2c-0, Address: 0x50, Mode: 8bit
  Writing stdin starting at address 0x0
.....
/ #
```

Checking correct writing can be done with the same tool:

```
/ # eeeprog -f /dev/i2c-0 0x50 -r 0:13
eeeprog 0.7.5, a 24Cxx EEPROM reader/writer
Copyright (c) 2003 by Stefano Barbato - All rights reserved.
  Bus: /dev/i2c-0, Address: 0x50, Mode: 8bit
  Reading 13 bytes from 0x0
Hello World!
/ #
```

... or using `hexdump` through `sysfs`:

```
/ # hexdump -C /sys/bus/i2c/drivers/at24/0-0050/eeprom
00000000 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0a ff ff ff |Hello World!....|
00000010 ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....|
*
00000100
/ #
```

I2C Real Time Clock

The gridARM-EVBPEAK board includes an I2C RTC chip at address 0x68:

```
...
rtc-m41t80 0-0068: chip found, driver version 0.05
rtc-m41t80 0-0068: rtc core: registered m41t81s as rtc0
...
rtc-m41t80 0-0068: setting system clock to 2012-12-19 12:24:01 UTC (1355919841)
...
```

The RTC clock is defined as a device connected to the I2C bus, at address 0x68:

```
static struct i2c_board_info __initdata evb_i2c_devices[] = {
...
    /* RTC device */
    {
        I2C_BOARD_INFO("m41t80", 0x68),
        .type = "m41t81s",
    },
...
};
```

Moreover, the Kernel is configured to propose all three user interfaces to the RTC:

1. *procfs* interface

```
/ # cat /proc/driver/rtc
rtc time      : 12:25:49
rtc_date     : 2012-12-19
alarm_time   : 00:00:00
alarm_date   : 2012-**-23
alarm_IRQ    : yes
alarm_pending : no
24hr        : yes
battery      : ok
```

2. *sysfs* RTC interface

```
/ # cat /sys/class/rtc/rtc0/date
2012-12-19
/ # cat /sys/class/rtc/rtc0/time
12:26:03
/ # cat /sys/class/rtc/rtc0/time
12:26:05
/ #
```

3. Character device interface (*ioctl*). The *ioctl* interface is used by the `hwclock` Busybox applet included in the root file system. Example below shows how to set the system date AND the RTC clock, using `hwclock --systohc` command:

```
/ # date
Wed Jun  5 14:13:42 UTC 2013
/ # date 15:19
Wed Jun  5 15:19:00 UTC 2013
/ # hwclock -r
Wed Jun  5 14:15:03 2013  0.000000 seconds
/ # hwclock --systohc
/ # hwclock --show
Wed Jun  5 15:19:14 2013  0.000000 seconds
/ #
```

I2C PCA9555PW

The gridARM-EVBPEAK board includes an I2C-bus I/O port chip (GPIO extender) at address 0x27, offering up to 2x8 I/O lines:

- I/O lines from 0 to 6 are attached to 7 "user" LEDs.
- I/O line number 7 is attached to a speaker
- I/O lines from 8 to 13 are attached to 6 "user" push buttons

The GPIO extender is defined as a device connected to the I2C bus. The architecture of the gridARM-EVBPEAK board defines the base of the GPIO pin number next to the range of the PIOA pins (`PIN_BASE+32`):

```
#define EVBPEAK_EXT0_GPIO_BASE (PIN_BASE+32)
...
static struct pca953x_platform_data pca955xpw_data[] = {
    [0] = {
        .gpio_base      = EVBPEAK_EXT0_GPIO_BASE,
        .setup           = evbpeak_pca9555_setup,
    },
};

static struct i2c_board_info __initdata evb_i2c_devices[] = {
    /* PCA9555PW */
    {
        I2C_BOARD_INFO("pca9535", 0x27),
        .platform_data = &pca955xpw_data[0],
        .type = "pca9555",
    },
};
```

On the gridARM-EVBPEAK, 7 of these new GPIO pins are connected to 7 "user" LEDs. The systems is configured to handle them as `gpio-leds`, in the same way than the `cpu` LED (see "Status LED" on page 20):

```
static struct gpio_led evb_leds[] = {
#ifdef CONFIG_LEDS_GPIO_PLATFORM
    /* One CPU LED + 7 from PCA9555PW */
    {
        .name                = "cpu",
        .default_trigger     = "heartbeat",
        .active_low          = 1,
        .gpio                 = GA_PIN_PA(15),
    },

    /* PCA9555PW */
    { .name = "yellow:1", .active_low = 1, .gpio = EXT0_PIN(0), },
    { .name = "yellow:2", .active_low = 1, .gpio = EXT0_PIN(1), },
    { .name = "yellow:3", .active_low = 1, .gpio = EXT0_PIN(2), },
    { .name = "yellow:4", .active_low = 1, .gpio = EXT0_PIN(3), },
    { .name = "green:5", .active_low = 1, .gpio = EXT0_PIN(4), },
    { .name = "green:6", .active_low = 1, .gpio = EXT0_PIN(5), },
    { .name = "red:7", .active_low = 1, .gpio = EXT0_PIN(6), },
#endif
};
```

Thus, their access from user space is the same than for the `cpu` LED:

```
/ # ls /sys/class/leds
cpu      green:6  yellow:1  yellow:3
green:5  red:7    yellow:2  yellow:4
/ #
```

Lighting on, for example, the red LED (number 7 on the gridARM-EVBPEAK board) is done with:

```
/ # echo 1 > /sys/class/leds/red:7/brightness
/ #
```

Setting the heartbeat trigger to, for example, one of the yellow LEDs (number 3 on the gridARM-EVBPEAK board) is done with:


```
/ # echo heartbeat > /sys/class/leds/yellow:3/trigger
/ #
```

Last unused GPIO pin is connected to the buzzer of the gridARM-EVBPEAK board. It is also exported to user space by the standard GPIO library *sysfs* interface of the Kernel:

```
static int evbpeak_pca9555_setup(struct i2c_client *client,
                                unsigned gpio_base, unsigned ngpio,
                                void *context)
{
    gpio_request(EXT0_PIN(7), "buzzer");
    gpio_direction_output(EXT0_PIN(7), 0);
    gpio_export(EXT0_PIN(7), 0);

    return 0;
}
```

Exporting pin 7 of the PCA9555 allows user to switch on/off the buzzer of the gridARM-EVBPEAK board:

```
/ # echo 1 > /sys/class/gpio/gpio71/value
/ # echo 0 > /sys/class/gpio/gpio71/value
```

I2C PCA954PW

The gridARM-EVBPEAK board also includes an I2C-bus 8 x I/O port chip at address 0x27. The architecture of the gridARM-EVBPEAK board defines the base of the GPIO pin number next to the range of the 16 x PCA9555 GPIO pins (EVBPEAK_EXT0_GPIO_BASE+16):

```
#define EVBPEAK_EXT1_GPIO_BASE (EVBPEAK_EXT0_GPIO_BASE+16)
...
static struct pca953x_platform_data pca955xpw_data[] = {
...
    [1] = {
        .gpio_base      = EVBPEAK_EXT1_GPIO_BASE,
        .setup           = evbpeak_pca9554_setup,
    },
};

static struct i2c_board_info __initdata evb_i2c_devices[] = {
...
    /* PCA9554PW */
    {
        I2C_BOARD_INFO("pca9535", 0x21),
        .platform_data = &pca955xpw_data[1],
        .type = "pca9554",
    },
...
}
```

This GPIO extender allows to control an optional Hitachi HD44780 compatible LCD panel. The GPIO pins are thus configured for a 4-bits access to such a device, thanks to the *setup* callback:

```

static int evbpeak_pca9554_setup(struct i2c_client *client,
                                unsigned gpio_base, unsigned ngpio,
                                void *context)
{
    /* power contrast: on */
    gpio_request(EXT1_PIN(0), "lcd_pwr");
    gpio_direction_output(EXT1_PIN(0), 1);

    gpio_request(EXT1_PIN(1), "lcd_rs"); /* cmd/data */
    gpio_direction_output(EXT1_PIN(1), 0);

    gpio_request(EXT1_PIN(2), "lcd_rw"); /* read/write */
    gpio_direction_output(EXT1_PIN(2), 0);

    gpio_request(EXT1_PIN(3), "lcd_en"); /* 0->1 : 4,5 + 7-14 */
    gpio_direction_output(EXT1_PIN(3), 0);

    gpio_request(EXT1_PIN(4), "lcd_d4"); /* data */
    gpio_direction_output(EXT1_PIN(4), 0);

    gpio_request(EXT1_PIN(5), "lcd_d5");
    gpio_direction_output(EXT1_PIN(5), 0);

    gpio_request(EXT1_PIN(6), "lcd_d6");
    gpio_direction_output(EXT1_PIN(6), 0);

    gpio_request(EXT1_PIN(7), "lcd_d7");
    gpio_direction_output(EXT1_PIN(7), 0);

    return 0;
}

```

The LCD panel is also defined as a platform device in the Kernel:

```

extern void panel_puts(char *s);

/*
 * LCD Display
 */

static int evb_lcd_gpio_pins[] = {
    -1, /* BL - not controlled from driver EXT1_PIN(0) */
    EXT1_PIN(1), /* RS */
    EXT1_PIN(2), /* RW */
    EXT1_PIN(3), /* EN */

    EXT1_PIN(4), /* D4 */
    EXT1_PIN(5), /* D5 */
    EXT1_PIN(6), /* D6 */
    EXT1_PIN(7), /* D7 */

    -1, -1, -1, -1 /* 4-bit mode only */
};

static struct platform_device evb_lcd = {
    .name = "lcd-gpio",
    .id = -1,
    .dev = {
        .platform_data = &evb_lcd_gpio_pins,
    },
};

```

This platform device is handled by the `lcd-gpio` device driver which instantiates the device node `/dev/lcd`. This device node enables user space to write some simple text strings on the optional LCD panel:

```
/ # ls -l /dev/lcd
crw----- 1 root    root      10, 156 Jan  1 1970 /dev/lcd
```

This device node proposes a simple interface which enables to write character strings onto lines of the LCD panel, from user space. For example:

```
/ # echo -n "Hello World!" > /dev/lcd
```

The `lcd-gpio` driver handles the following control characters:

Control Character	Function
<code>\f</code>	Clear the entire screen of the LCD display and move cursor at the top left corner of the screen
<code>\n</code>	Set cursor on most left position of next line, scroll up display if cursor was on last one.
<code>\b</code>	Move cursor one position to the left, ignored if cursor already at the left-most position of the line.
<code>\r</code>	Move cursor to the most left position of the cursor line.
<code>\t</code>	This character is changed into a space character.
<code>\v</code>	Resets the LCD panel

NOR Flash

The gridARM-EVBPEAK board embeds three identical NOR Flash chips. Their usage is defined by JP200 and JP201 combination:

JP200	JP201	NOR Usage
Closed	2-3	<ol style="list-style-type: none"> 1 x chip giving a 16-bit wide 4MB NOR Flash (at NCS0) 2 x parallel chips implementing a 32-bit wide 8MB NOR Flash (at NCS2)
Open	1-2	1 x 32-bit wide NOR Flash (at NCS0)



Linux ARM architecture needs to setup the exception vector at address 0 so that early code of the Kernel setup NCS0 bank to SDRAM. For Linux to access any NOR Flash memory, JP200="closed" and JP201="2-3" is mandatory. In the other case, NOR Flash access is not possible.

When JP200="closed" and JP201="2-3", the NOR Flash address space is defined with the NCS2 bit so that the NOR memory addresses start from 0x30000000.

```

...
physmap platform flash device: 00800000 at 30000000
physmap-flash: Found 2 x16 devices at 0x0 in 32-bit bank. Manufacturer ID 0x0000
7f Chip ID 0x0022f6
Amd/Fujitsu Extended Query Table at 0x0040
  Amd/Fujitsu Extended Query version 1.1.
physmap-flash: Swapping erase regions for top-boot CFI table.
number of CFI chips: 1
cmdlinepart partition parsing not available
RedBoot partition parsing not available
Using physmap partition information
Creating 2 MTD partitions on "physmap-flash":
0x0000000000000-0x0000000400000 : "kernel"
0x0000000400000-0x0000000800000 : "rootfs"
...

```

The Kernel architecture defines two partitions in this 8 MB space:

Start	Size	Name	Usage
0x00000000	4 MB	kernel	This partition is defined to save a Kernel image that can be read from U-Boot, to boot on.
0x00400000	4 MB	rootfs	This partition is defined to save an image of a root file system used by Linux to boot on.

See also "Flashing the root file system" on page 13 to see how to use these partitions.

The NOR flash and its partition scheme are statically defined in the architecture of the Kernel:

```

#define NOR_BASE          GA_CHIPSELECT_2
#define NOR_SIZE          SZ_8M

static struct resource evb_flash_resource[] = {
    {
        .start  = NOR_BASE,
        .end    = NOR_BASE + NOR_SIZE - 1,
        .flags  = IORESOURCE_MEM,
    },
};

static struct mtd_partition evb_flash_parts[] = {
    {
        .name    = "kernel",
        .offset  = 0x000000,
        .size    = SZ_4M,
    },
    {
        .name    = "rootfs",
        .offset  = MTDPART_OFS_NXTBLK,
        .size    = MTDPART_SIZ_FULL,
    },
};

static struct physmap_flash_data evb_flash_data = {
    .width      = 4, /* 32-bits width */
    .parts      = evb_flash_parts,
    .nr_parts   = ARRAY_SIZE(evb_flash_parts),
};

static struct platform_device evb_flash = {
    .name       = "physmap-flash",
    .id        = -1,
    .resource   = evb_flash_resource,
    .num_resources = 1,
    .dev = {
        .platform_data = &evb_flash_data,
    },
};

```

The MTD driver offers a *procfs* unified interface to user space for Flash devices. Both NOR partitions on the gridARM-EVBPEAK are `mtd0` and `mtd1`.

```

/ # cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00400000 00020000 "kernel"
mtd1: 00400000 00020000 "rootfs"
mtd2: 00400000 00001000 "user"
/ #

```

The root file system also includes the `mtd-utils` package which offers tools to write into any Flash chips. For example, flashing a new Kernel image from the Linux system is done like this (the new image is first downloaded using the *tftp* program):

```

/ # cd /tmp
/var/tmp # tftp -g -r uImage 192.168.0.1
/var/tmp # ls -l
-rw-r--r--    1 root    root    3011936 Dec 21 16:23 uImage

/var/tmp # flashcp -v uImage /dev/mtd0
Erasing blocks: XX/XX (100%)
Writing data: YYYYk/YYYYk (100%)
Verifying data: YYYYk/YYYYk (100%)

/var/tmp #

```

Flashing a new file system is done in the same way. Mounting an access point allows to check if the file system has been correctly flashed:

```

/ # cd /tmp
/var/tmp # tftp -g -r rootfs.jffs2 192.168.0.1

/var/tmp # ls -l rootfs.jffs2
-rw-r--r--    1 root    root    2752512 Jan  1 00:32 rootfs.jffs2

/var/tmp # flashcp -v rootfs.jffs2 /dev/mtd1
Erasing blocks: 21/21 (100%)
Writing data: 2688k/2688k (100%)
Verifying data: 2688k/2688k (100%)

/var/tmp # mkdir -p /mnt/nor
/var/tmp # mount -t jffs2 /dev/mtdblock1 /mnt/nor
/var/tmp # ls /mnt/nor
bin  dev  etc  home  init  lib  mnt  proc  sbin  sys  tmp  usr  var

/var/tmp #

```



Warning: Of course, flashing a new file system in `mtd1` SHOULD *ONLY* be done when the running Kernel has booted on an other root file system!

Analog to Digital Converter

The gridARM SoC includes an Analog to Digital Converter Controller, connected to IRQ11.

```

static struct resource adc_resources[] = {
    [0] = {
        .start = GA_BASE_ADC,
        .end   = GA_BASE_ADC + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = GA_AIC_ID_ADC,
        .end   = GA_AIC_ID_ADC,
        .flags = IORESOURCE_IRQ,
    },
};

```

```

...
gridarm_adc gridarm_adc: device registered irq=11 clk=5000000
...
/ # cat /proc/interrupts
          CPU0
  1:      3825          AIC   tick, ttyS0
  4:         98          AIC   ttyS1
  6:        151          AIC   gridarm spi.0
 11:         0          AIC   gridarm_adc
 18:     11478          AIC   gridarm_i2c
 25:         1          AIC   eth0_phy
 26:         4          AIC   eth0
 34:         0          GPIO   mmc-spi-detect
Err:         0

```

The gridARM-EVBPEAK board is equipped with 4 potentiometers connected to channel AD0 to AD3 of the ADC controller of the gridARM SoC.

```

static struct gridarm_adc_data evb_adc_data = {
    .channels_used = BIT(0) | BIT(1) | BIT(2) | BIT(3) | BIT(4),
    .use_external_triggers = false,
    .vref = 3300, /* 3V3 */
};

```

The SoC doesn't connect any external trigger so that the Linux kernel ADC driver "only" reads value from the corresponding channel on external request. Thus, the driver adds to the Kernel sysfs one new entry per potentiometer, to enable their values to be read from user space:

```

/ # ls /sys/devices/platform/gridarm_adc/device0/
in0_raw  in2_raw  in4_raw  in6_raw  name      subsystem
in1_raw  in3_raw  in5_raw  in7_raw  power     uevent
/ # cat /sys/devices/platform/gridarm_adc/device0/name
gridarm_adc
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
0
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
0
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
1022
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
1023
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
1022
/ # cat /sys/devices/platform/gridarm_adc/device0/in0_raw
1022

```

In the above example, potentiometer 0 went from left to right, from 0 to 1024 (by default, the ADC is setup in 10-bits resolution mode).

MicroSD Card

The gridARM-EVBPEAK board embeds a microSD card reader connected to the SPI bus, on CS#1. Thus, the Kernel MUST be configured to support MMC/SD/SDIO over SPI:

```
Device Drivers --->
[*] MMC/SD/SDIO card support --->
[*]   MMC block device driver
[*]     Use bounce buffer for simple hosts
[*]   MMC/SD/SDIO over SPI
```

CS#1 is defined in the architecture of the gridARM-EVBPEAK board:

```
static struct spi_board_info evbpeak_spi_devices[] = {
    {
        /* SPI micro-SD */
        .modalias = "mmc_spi",
        .chip_select = 1,      /* CS1 */
        .max_speed_hz = 15 * 1000 * 1000,
    },
};
```

```
...
mmc_spi spi0.1: SD/MMC host mmc0, no WP, no poweroff, cd polling
...
```

The Kernel polls on card detection so that inserting an SD-card is detected. For example, inserting a 2GB SD-card in the slot produces such a display:

```
mmc0: host does not support reading read-only switch. assuming write-enable.
mmc0: new SD card on SPI
mmcblk0: mmc0:0000 SU02G 1.84 GiB
mmcblk0: p1
```

If the SD-card is formatted with a known file system on one of its partition, it can be mounted to get access on it. For example, above SD-card defines partition *p1*. The root file system is configured to provide access to such a partition, by defining the below device node:

```
/ # ls -l /dev/mmcblk0p1
brw----- 1 root root 179, 1 Feb 1 2013 /dev/mmcblk0p1
```

So, getting access to the content of this partition is done like this:

```
/ # mkdir /mnt/sdcard
/ # mount /dev/mmcblk0p1 /mnt/sdcard
EXT3-fs: barriers not enabled
kjournald starting. Commit interval 5 seconds
EXT3-fs (mmcblk0p1): using internal journal
EXT3-fs (mmcblk0p1): recovery complete
EXT3-fs (mmcblk0p1): mounted filesystem with writeback data mode
/ # ls /mnt/sdcard
bin      home      lost+found  proc      sys      var
dev      lib       mnt        root      tmp
etc      linuxrc  opt        sbin     usr
/ #
```


SPI Flash

The gridARM-EVBPEAK board embeds a Flash chip connected on the SPI bus at CS#0.

```
...
m25p80 spi0.0: sst25vf032b (4096 Kbytes)
Creating 1 MTD partitions on "spi_flash":
0x0000000000000-0x0000004000000 : "user"
...
```

The Kernel architecture is configured to create one *user* MTD partition to provide access to the whole SPI Flash memory:

```
static struct mtd_partition evb_spi_partitions[] = {
    {
        .name      = "user",
        .offset    = 0,
        .size      = MTDPART_SIZ_FULL,      /* 4xMB, read from chip */
    },
};

static struct flash_platform_data evb_spi_flash_platform_data = {
    .name          = "spi_flash",
    .type          = "sst25vf032b",
    .parts         = evb_spi_partitions,
    .nr_parts      = ARRAY_SIZE(evb_spi_partitions)
};

static struct spi_board_info evbpeak_spi_devices[] = {
    {
        /* SPI Flash */
        .modalias = "m25p80",
        .bus_num = 0,
        .chip_select = 0,      /* CS0 */
        .max_speed_hz = 6 * 1000 * 1000,
        .platform_data = &evb_spi_flash_platform_data,
    },
};
```

This "user" partition is visible from the user space like any other Flash memories. The gridARM-EVBPEAK board defines 3x MTD partitions: the first two map the NOR Flash partitions while third one maps the entire SPI Flash:

```
/ # cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00400000 00020000 "kernel"
mtd1: 00400000 00020000 "rootfs"
mtd2: 00400000 00001000 "user"
```

This SPI Flash may also be used for file system usage. For example, writing a JFFS2 image is done like this:

```
/var/tmp # mkdir /mnt/flash
/var/tmp # flash_erase -q /dev/mtd2 0 0
/var/tmp # nandwrite -p -q /dev/mtd2 user.jffs2
/var/tmp # mount -t jffs2 /dev/mtdblock2 /mnt/flash
/var/tmp # ls /mnt/flash/
README
/var/tmp # cat /mnt/flash/README
(c) 2013 PEAK-System
/var/tmp #
```

GEMAC/PHY

The gridARM-EVBPEAK board connects the GEMAC to a Gigabit Ethernet PHY and an RJ45 connector, to enable full network access from the embedded applications.

```
...
gridarm_eth: using slot #1 MAC address
eth0: Link down.
eth0: gridARM ethernet at 0x90000000 irq=26 (AA:BB:CC:DD:EE:FF)
eth0: Micrel KSZ902x PHY (Copper)
...
```

The Kernel is configured with the entire TCP/IP Ipv4 stack, enabling access to the entire Internet from the board. Moreover, the Kernel is also configured with NFS so that accessing to a remote file system through the network is allowed as well as booting on such a Network File system (see "Booting from NFS" on page 17). The Kernel is also configured to request a dynamic IP address from the Network (according to the `ip=` option of the Kernel command line).

```
...
eth0: Link down.
Sending DHCP requests .
eth0: Link now 100-FD
., OK
IP-Config: Got DHCP answer from 192.168.0.1, my address is 192.168.0.116
IP-Config: Complete:
    device=eth0, addr=192.168.0.116, mask=255.255.255.0, gw=255.255.255.255,
    host=192.168.0.116, domain=, nis-domain=(none),
    bootserver=192.168.0.1, rootserver=192.168.0.1, rootpath=
Looking up port of RPC 100003/3 on 192.168.0.1
Looking up port of RPC 100005/3 on 192.168.0.1
VFS: Mounted root (nfs filesystem) on device 0:9.
Freeing init memory: 92K
init started: BusyBox v1.10.2-uc0 (2013-01-03 16:24:43 CET)
...
```

The root file system is also configured to request a DHCP address when starting.

```

...
eth0: Link now 100-FD
...
/ # udhcpc (v1.10.2-uc0) started
Sending discover...
Sending select for 192.168.0.116...
Lease of 192.168.0.116 obtained, lease time 43200
udhcpc: cannot background in uclinux (yet)

/ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:23:24:25:26:22
          inet addr:192.168.0.116  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:692 (692.0 B)  TX bytes:1188 (1.1 KiB)
          Interrupt:26

/ # ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 data bytes
64 bytes from 192.168.0.1: seq=0 ttl=64 time=5.939 ms
64 bytes from 192.168.0.1: seq=1 ttl=64 time=1.536 ms
64 bytes from 192.168.0.1: seq=2 ttl=64 time=1.080 ms
64 bytes from 192.168.0.1: seq=3 ttl=64 time=1.496 ms
...
/ #

```

CAN Controller

The gridARM-EVBPEAK board embeds a CAN controller.

```

...
CAN device driver interface
gridarm_can gridarm_can: device registered at 0xffffd8000, irq=20
...

```

The CAN controller is defined in the Kernel architecture:

```

...
static struct resource can_resources[] = {
    [0] = {
        .start = GA_BASE_CAN,
        .end   = GA_BASE_CAN + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = GA_AIC_ID_CAN,
        .end   = GA_AIC_ID_CAN,
        .flags = IORESOURCE_IRQ,
    },
};

static struct platform_device gridarm_can_device = {
    .name       = "gridarm_can",
    .id         = -1,
    .resource   = can_resources,
    .num_resources = ARRAY_SIZE(can_resources),
};
...

```

The Kernel includes a driver for the CAN controller which is compatible with the linux-can architecture so that `ifconfig` and/or `ip` tools are able to configure it:

```
/ # ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
-00

        NOARP  MTU:16  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:10
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
        Interrupt:20
```

```
/ # ip -s -d link show can0
2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN qlen 10
  link/can
  can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
  bitrate 0 sample-point 0.000
  tq 0 prop-seg 0 phase-seg1 0 phase-seg2 0 sjw 0
  : tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1
  clock 80000000
  re-started bus-errors arbit-lost error-warn error-pass bus-off
  0          0          0          0          0          0
  RX: bytes  packets  errors  dropped overrun mcast
  0          0          0          0          0          0
  TX: bytes  packets  errors  dropped carrier collsns
  0          0          0          0          0          0
```

For example, setting the bit rate to 500 kbps is done using `ip link` tool:

```
/ # ip link set can0 type can bitrate 500000
/ #
```

Finally, the CAN network interface must be set to UP:

```
/ # ifconfig can0 up
gridarm_can gridarm_can: BRP=10 SJW=1 PROP_SEG=6 PHASE_SEG1=7 PHASE_SEG2=2
gridarm_can gridarm_can: writing GA_BR: 0x00090561
/ #

/ # ip -s -d link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UNKNOWN qlen 10
  link/can
  can state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
  bitrate 500000 sample-point 0.875
  tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
  : tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1
  clock 80000000
  re-started bus-errors arbit-lost error-warn error-pass bus-off
  0          0          0          0          0          0
  RX: bytes  packets  errors  dropped overrun mcast
  0          0          0          0          0          0
  TX: bytes  packets  errors  dropped carrier collsns
  0          0          0          0          0          0
/ #
```

Once configured, the CAN interface can be used through the Socket-CAN API. The root file system is setup to include some of the `can-utils` sample programs. For example, dumping any CAN frames received on the `can0` interface with absolute timestamp is done like this:

```
/ # candump -ta can0
(1370617149.530720) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.540613) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.550634) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.560586) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.570631) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.580606) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.590581) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.600626) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.610587) can0 101 [8] 55 55 55 55 55 55 55 55
(1370617149.620646) can0 101 [8] 55 55 55 55 55 55 55 55
...
```

Sending a CAN frame with ID=0x123 and data bytes = 0x01 0x02 0x03 is done like this:

```
/ # cansend can0 123#010203
/ #
```

USB Device Port (UDP)

The gridARM-EVBPEAK board is equipped with one USB "device" connector connected to the USB Device Port (UDP) of the gridARM SoC. This UDP is compliant with the USB V2.0 full-speed device specification.

If the Kernel USB Gadget option is set (default), the architecture of the Kernel is configured with the gridARM UDC resources:

```
static struct resource udc_resources[] = {
    [0] = {
        .start = GA_BASE_UDP,
        .end   = GA_BASE_UDP + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = GA_AIC_ID_UDP,
        .end   = GA_AIC_ID_UDP,
        .flags = IORESOURCE_IRQ,
    },
};

static struct platform_device grid_udc_device = {
    .name = "gridarm_udc",
    .id   = -1,
    .dev  = {
        .platform_data = &udc_data,
    },
    .resource = udc_resources,
    .num_resources = ARRAY_SIZE(udc_resources),
};
```

The UDC driver is statically linked to the default Kernel and V-Bus signal is not connected to any PIO on the gridARM-EVBPEAK board:

```
...
no VBUS detection, assuming always-on
gridarm_udc version 1.0.0
...
```

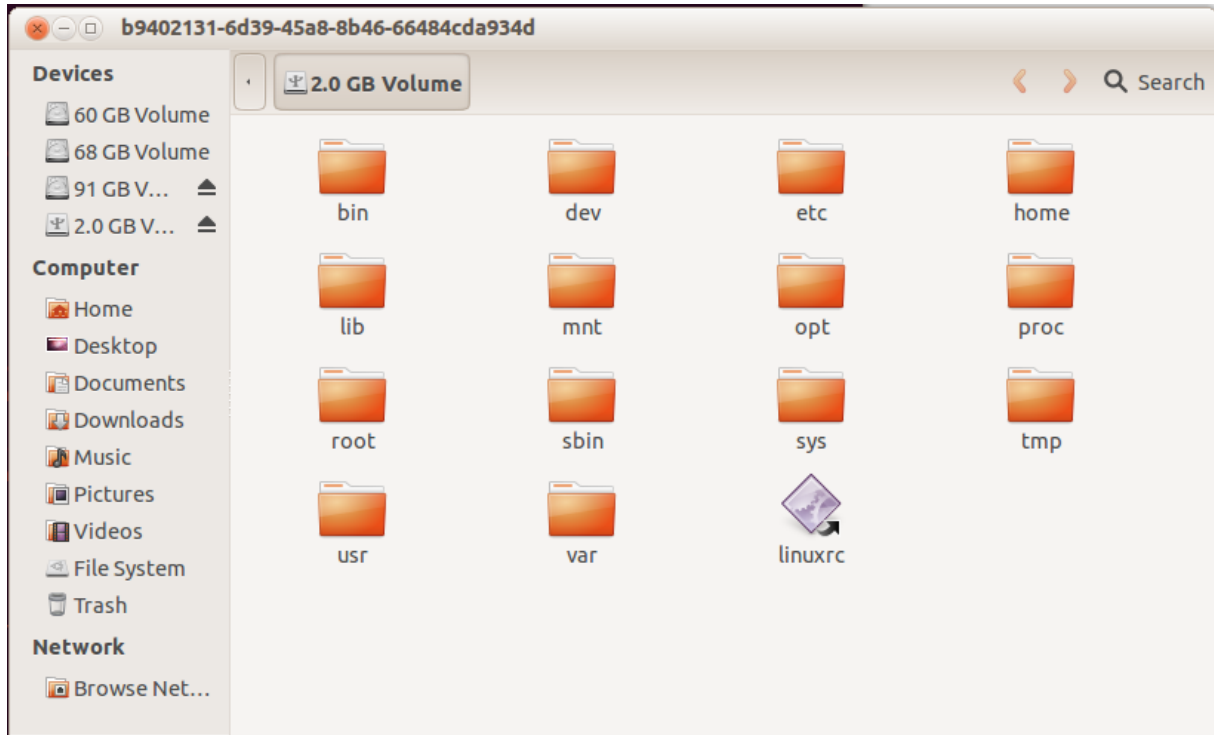
The USB Gadget driver module is configured by default to act as an USB Mass Storage disk driver. This module enables access to a regular file or a block device.

For example, here are the steps to do to enable access to the SD-Card reader of the gridARM-EVBPEAK board, from a Linux PC (host):

1. Load the file USB File Storage gadget driver module with partition #1 of the SD-Card acting as backing file:

```
...
mmc_spi spi0.1: SD/MMC host mmc0, no WP, no poweroff, cd polling
mmc0: host does not support reading read-only switch. assuming write-enable.
mmc0: new SD card on SPI
mmcblk0: mmc0:0000 SU02G 1.84 GiB
  mmcblk0: p1
...
/ # modprobe g_file_storage file=/dev/mmcblk0p1
g_file_storage gadget: Userspace failed to provide serial number; Failing back to
default
g_file_storage gadget: Default serial number provided: 3230204E6F76
g_file_storage gadget: File-backed Storage Gadget, version: 20 November 2008
g_file_storage gadget: Number of LUNs=1
g_file_storage gadget-lun0: ro=0, nofua=0, file: /dev/mmcblk0p1
gridarm_udc bound to g_file_storage
/ # g_file_storage gadget: full speed config #1
```

2. Next, plug the other end of the USB cable to the Linux host. Since SD-Card partition is an EXT3-fs, the GUI should automatically opens a folder window which gives access to the SD-Card content:



Having a look to the Linux host system kernel messages gives more details about the entire process:

```

$ dmesg | tail -n 20
[91178.528033] usb 6-2: >new full-speed USB device number 28 using uhci_hcd
[91178.722296] usb 6-2: >New USB device found, idVendor=0525, idProduct=a4a5
[91178.722302] usb 6-2: >New USB device strings: Mfr=1, Product=2, SerialNumber=3
[91178.722306] usb 6-2: >Product: File-backed Storage Gadget
[91178.722310] usb 6-2: >Manufacturer: Linux 2.6.36 with gridarm_udc
[91178.722313] usb 6-2: >SerialNumber: 3230204E6F76
[91178.742387] usb-storage 6-2:1.0: >Quirks match for vid 0525 pid a4a5: 10000
[91178.742414] scsi7 : usb-storage 6-2:1.0
[91179.744331] scsi 7:0:0:0: >Direct-Access      Linux      File-Stor Gadget 0333 PQ:
0 ANSI: 2
[91179.745367] sd 7:0:0:0: >Attached scsi generic sg11 type 0
[91179.755313] sd 7:0:0:0: >[sdk] 3846144 512-byte logical blocks: (1.96 GB/1.83
GiB)
[91179.758308] sd 7:0:0:0: >[sdk] Write Protect is off
[91179.758313] sd 7:0:0:0: >[sdk] Mode Sense: 0f 00 00 00
[91179.761308] sd 7:0:0:0: >[sdk] Write cache: enabled, read cache: enabled,
doesn't support DPO or FUA
[91179.839341]  sd: unknown partition table
[91179.851307] sd 7:0:0:0: >[sdk] Attached SCSI disk
[91184.262368] kjournald starting. Commit interval 5 seconds
[91184.268311] EXT3-fs (sdk): using internal journal
[91184.306312] EXT3-fs (sdk): recovery complete
[91184.306315] EXT3-fs (sdk): mounted filesystem with ordered data mode

$ cat /proc/partitions
major minor #blocks name

   8          0 244198584 sda
...
   8          8  4096      sdk

$ mount
...
/dev/sdk on /media/peak/b9402131-6d39-45a8-8b46-66484cda934d type ext3
(rw,nosuid,nodev,uhelper=udisks2)

```

The USB Gadget file storage driver has been setup as a driver module that can be loaded and unloaded dynamically. This enables to export any other backing file or block device to the Linux host.

For example, operator could next export the content of the root file system partition to the Linux host:

1. First, unload the file storage driver module (if it was loaded first). This should also close any opened window on the remote host:

```

/ # rmmod g_file_storage
gridarm_udc unbound from g_file_storage

```

2. Load the file storage USB Gadget (again) but with a backing file which corresponds to the *rootfs* block device MTD partition (*mtdblock1*):


```

/ # modprobe g_file_storage file=/dev/mtdblock1
g_file_storage gadget: Userspace failed to provide serial number; Failing back t
o default
g_file_storage gadget: Default serial number provided: 3230204E6F76
g_file_storage gadget: File-backed Storage Gadget, version: 20 November 2008
g_file_storage gadget: Number of LUNs=1
g_file_storage gadget-lun0: ro=0, nofua=0, file: /dev/mtdblock1
gridarm_udc bound to g_file_storage
/ # g_file_storage gadget: full speed config #1

```

3. On the remote host, the system always detects an USB file storage device and attaches a block device on it (`sdk` in this example):

```

$ cat /proc/partitions
major minor #blocks name
 8          0 244198584 sda
...
 8         160      4096 sdk

```

4. Since the block device is created by the system, mounting a JFFS2 file system on a Linux system is done by loading the `block2mtd` driver module which maps a device block (here: `/dev/sdk`) to a (128 KB erase size) MTD partition. In Linux PC systems, the `mtdblock` driver module should be loaded first:

```

$ sudo modprobe mtdblock
$ ll /dev/mtd*
crw----- 1 root root 90, 0 Jun 11 10:58 /dev/mtd0
crw----- 1 root root 90, 1 Jun 11 10:58 /dev/mtd0ro
brw-rw---- 1 root disk 31, 0 Jun 11 10:58 /dev/mtdblock0
$ sudo modprobe block2mtd block2mtd=/dev/sdk,128KiB

```

5. Finally, the JFFS2 file system can be mounted on `/dev/sdk` through `/dev/mtdblock0`, which enables read/write access to the gridARM-EVBPEAK root file system saved in NOR flash:

```

$ sudo mkdir -p /mnt/gridARM-EVBPEAK-rootfs
$ sudo mount -t jffs2 /dev/mtdblock0 /mnt/gridARM-EVBPEAK-rootfs

$ ls /mnt/gridARM-EVBPEAK-rootfs
bin dev etc home init lib mnt proc sbin sys tmp usr var

$ ls /mnt/gridARM-EVBPEAK-rootfs/home/peak/
demo rc
$ sudo cp README /mnt/gridARM-EVBPEAK-rootfs/home/peak/
$ ls /mnt/gridARM-EVBPEAK-rootfs/home/peak/
demo rc README

```

Adding User Applications

The following procedure describes how to add a user written application to the uClinux memory image(s).

1. First, a new directory must be created in `.../uClinux-dist/user/` to store the source files for the new application. In this example the program and directory will be called *hello*. Next move the source files for the application into this directory.

```
$ cd uClinux-dist
$ mkdir user/hello
$
```

2. Next the file `./uClinux-dist/user/Makefile` must be edited. Add as many lines similar to the following as there will be executables to generate. In this example, only one executable is to be built:

```
dir_${CONFIG_USER_HELLO} += hello
```

3. Next the file `./uClinux-dist/user/Kconfig` must be edited. Add an entry similar to the following :

```
config USER_HELLO
    bool "hello"
    help
        Sample Hello World! application
```

This file contains the text of the menus displayed by `make menuconfig`, when customizing Application settings:

```
$ make menuconfig

Kernel/Library/Defaults Selection --->
  [*] Customize Application/Library Settings
```

Exit then Save configuration. User configuration menu automatically opens next and the added items are displayed:

```
My own Applications --->
  [*] hello (NEW)
```

Ensure to select the user application before exiting and saving the configuration.

- Next ensure that there is an appropriate Makefile inside the *hello* program's directory, in this example: `./uClinux-dist/user/hello/Makefile`. This Makefile should have a form similar to the following:

```
#
# Makefile
#
#       Sample Hello World program Makefile
#
#       (c) PEAK-System GmbH
#
EXEC = hello
OBJS = hello.o

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)

romfs:
        $(ROMFSINST)    /bin/$(EXEC)

clean:
        rm -f $(EXEC) *.elf *.gdb *.o
~
```

In this example, the file `hello.c` looks like:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, World!\n");
    return 0;
}
```

- Finally, do the global `make` and if the user application compiled successfully it should now appear in the `romfs/bin` directory on the host system:

```
$ make
...

$ ls -l romfs/bin/hello
-rwxr--r-- 1 peak peak 30548 Jun 12 11:33 romfs/bin/hello
```

It should also appear in `/bin` directory on the target system:

```
/ # ls -l /bin/hello
-rwxr-xr-x  1 root root    30548   Jan 1 1970 /bin/hello
/ # hello
Hello, World!
/ #
```