LITE5200BUBPG
Rev. 0, 5/2006

U-Boot Quick Reference

for the Lite5200B Development Platform

by: Jonathan Wang
Infotainment, Multimedia, and Telematics Division

1 Overview

This manual is a Quick Reference for U-Boot commands on the Lite5200B Development Platform. Each command overview gives a description, directions for usage, and an execution example.

A list of U-Boot commands can be accessed while in the U-Boot prompt. Type **help** for a complete listing of available commands for the development platform.

Contents

freescale™ semiconductor

2 Attribution

This manual is excerpted from the Denx UBOOT manual as described below. Primarily it has been made specific to the Lite5200B board and has been reformatted. It is subject to GPL copyright restrictions as described below and at the URL given below. You have the freedom to distribute copies of this document in any format or to create a derivative work of it and distribute it provided that you:

- Distribute this document or the derivative work at no charge at all. It is not permitted to sell this document or the derivative work or to include it into any package or distribution that is not freely available to everybody.
- Send your derivative work (in the most suitable format such as sgml) to the author.
- License the derivative work with this same license or use GPL.
- Include a copyright notice and at least a pointer to the license used.
- Give due credit to previous authors and major contributors.

This document is derived from the DENX™ U-Boot User Manuals. (Copyright © 2001-2006 by Wolfgang Denk, DENX Software Engineering.)

The manual can be found at:

http://www.denx.de/wiki/DULG/Manual

3 Disclaimer

Use the information in this document at your own risk. Freescale disavows any potential liability for the contents of this document. Use of the concepts, examples, and/or other content of this document is entirely at your own risk. All copyrights are owned by their owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

4 List of Commands

4.1 AUTOSCR

Run script from memory:

```
autoscr [addr] - run script starting at addr - A valid autoscr header must be present
```

The **autoscr** command allows “shell” scripts to run under U-Boot. To create a U-Boot script image, commands are written to a text file. Then the **mkimage** tool [of a suitable compiler] is used to convert this text file into a U-Boot image using the image type script. This image can be loaded like any other image file. **Autoscr** runs the commands in this image.

For example, consider the following text file:

```
echo
echo Network Configuration:
echo ----------------------
echo Target:
printenv ipaddr hostname
echo
echo Server:
printenv serverip rootpath
echo
```

Convert the text file into a U-Boot script image using the **mkimage** command as follows:

```
bash$ mkimage -A ppc -O linux -T script -C none -a 0 -e 0 \
> -n "autoscr example script" \
> -d /tftpboot/TQM860L/example.script /tftpboot/TQM860L/example.img
Image Name:   autoscr example script
Created:      Mon Apr  8 01:15:02 2002
Image Type:   PowerPC Linux Script (uncompressed)
Data Size:    157 Bytes = 0.15 kB = 0.00 MB
Load Address: 0x00000000
Entry Point:  0x00000000
Contents:
   Image 0:      149 Bytes =    0 kB = 0 MB
```

Load and execute this script image in U-Boot:

```
=> tftp 100000 /tftpboot/TQM860L/example.img

ARP broadcast 1
TFTP from server 10.0.0.2; our IP address is 10.0.0.99
Filename '/tftpboot/TQM860L/example.img'.
Load address: 0x100000
Loading: #
done
Bytes transferred = 221 (dd hex)

=> autoscr 100000

## Executing script at 00100000

Network Configuration:
----------------------
Target:
```

```
ipaddr=10.0.0.99
hostname=tqm

Server:
serverip=10.0.0.2
rootpath=/opt/hardhat/devkit/ppc/8xx/target
```

4.2 BASE

Print or set address offset:

```
base - print address offset for memory commands
base off - set address offset for memory commands to 'off'
```

Use the **base** command (short: **ba**) to print or set a "base address" used as an address offset for all memory commands; the default value of the base address is 0, so all addresses you enter are used unmodified. However, when you repeatedly have to access a certain memory region (like the internal memory of some embedded PowerPC processors) it can be very convenient to set the base address to the start of this area and then use only the offsets:

```
=> base
Base Address: 0x00000000
=> md 0 c
00000000: feffffff 00000000 7cbd2b78 7cdc3378    ........|.+x|.3x
00000010: 3cfb3b78 3b000000 7c0002e4 39000000    <.;x;...|...9...
00000020: 7d1043a6 3d000400 7918c3a6 3d00c000    }.C.=...y...=...
=> base 40000000
Base Address: 0x40000000
=> md 0 c
40000000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
40000010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
40000020: 2031393a 35353a30 34290000 00000000     19:55:04)......
=>
```

4.3 BDINFO

Print board info structure:

The **bdinfo** command (short: **bdi**) prints the information that U-Boot passes about the board such as memory addresses and sizes, clock frequencies, MAC address, etc. This type of information is generally passed to the Linux kernel.

```
=> bdinfo
memstart    = 0x00000000
memsize     = 0x04000000
flashstart  = 0x40000000
flashsize   = 0x00800000
flashoffset = 0x00030000
sramstart   = 0x00000000
sramsize    = 0x00000000
immr_base   = 0xFFF00000
bootflags   = 0x00000001
intfreq     =     50 MHz
busfreq     =     50 MHz
ethaddr     = 00:D0:93:00:28:81
IP addr     = 0.0.0.0
baudrate    = 115200 bps
=>
```

4.4 BMP

Manipulate bmp image data:

```
bmp info <imageAddr>          - display image info
bmp display <imageAddr> [x y] - display image at x,y
```

4.5 BOOTD

Boot default, i.e., run 'bootcmd'.

The **bootd** (short: **boot**) executes the default boot command, i.e. what happens when you don't interrupt the initial countdown. This is a synonym for the **run bootcmd** command.

4.6 BOOTM

Boot application image from memory:

```
bootm [addr [arg ...]] - boot application image stored in memory passing arguments 'arg
...'; when booting a Linux kernel,'arg' can be the address of an initrd image
```

The **bootm** command is used to start operating system images. From the image header it gets information about the type of the operating system, the file compression method used (if any), the load and entry point addresses, etc. The command will then load the image to the required memory address, uncompressing it on the fly if necessary. Depending on the OS it will pass the required boot arguments and start the OS at it's entry point. The first argument to **bootm** is the memory address (in RAM, ROM or flash memory) where the image is stored, followed by optional arguments that depend on the OS.

For Linux, exactly one optional argument can be passed. If it is present, it is interpreted as the start address of a **initrd** ramdisk image (in RAM, ROM or flash memory). In this case the **bootm** command consists of three steps: first the Linux kernel image is uncompressed and copied into RAM, then the ramdisk image is loaded to RAM, and finally control is passed to the Linux kernel, passing information about the location and size of the ramdisk image.

To boot a Linux kernel image without a initrd ramdisk image, the following command can be used:

```
=> bootm $(kernel_addr)
```

If a ramdisk image is used, type:

```
=> bootm $(kernel_addr) $(ramdisk_addr)
```

Both examples imply that the variables used are set to correct addresses for a kernel and a **initrd** ramdisk image.

When booting images that have been loaded to RAM (for instance using TFTP download) you have to be careful that the locations where the (compressed) images were stored do not overlap with the memory needed to load the uncompressed kernel. For instance, if you load a ramdisk image at a location in low memory, it may be overwritten when the Linux kernel gets loaded. This will cause undefined system crashes.

4.7 BOOTP

Boot image via network using bootp/tftp protocol:

```
bootp [loadAddress] [bootfilename]
```

4.8 CMP

Memory compare:

```
cmp [.b, .w, .l] addr1 addr2 (count)
```

The **cmp** command tests of the contents of two memory areas and determines whether or not the contents of the two memory areas are identical or not. The command will either test the whole area as specified by the 3rd (count) argument or stop at the first difference if the count argument is not specified.

The following example demonstrates comparing the memory ranges 0x100000 - 0x10002F to 0x400000 - 0x40002F. The contents of the two memory ranges are shown below.

```
00100000: 27051956 50ff4342 6f6f7420 312e312e    '..VP.CBoot 1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......

40000000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
40000010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
40000020: 2031393a 35353a30 34290000 00000000     19:55:04)......


=> cmp 100000 40000000 400

word at 0x00100004 (0x50ff4342) != word at 0x40000004 (0x50504342)
Total of 1 word were the same

=>
```

Like most memory commands the **cmp** command accesses the memory in different sizes: 32 bit (long word), 16 bit (word) or 8 bit (byte) data. If invoked just as **cmp** the default size (32 bit or long words) is used; the same can be selected explicitly by typing **cmp.l** instead. To access memory as 16 bit (word data), use the variant **cmp.w**; to access memory as 8 bit (byte data) use **cmp.b**. Please note that the count argument specifies the number of data items to process, i.e. the number of long words or words or bytes to compare.

4.9 CONINFO

Print console devices and information.

The **coninfo** command (short: **conin**) displays information about the available console I/O devices.

```
=> conin
List of available devices:
serial   80000003 SIO stdin stdout stderr
=>
```

The output contains the device name, flags, and the current usage. For example, the output “serial 80000003 SIO **stdin stdout stderr**” means that the serial device is a system device (flag 'S') which

provides input (flag 'I') and output (flag 'O') functionality and is currently assigned to the 3 standard I/O streams **stdin**, **stdout**, and **stderr**.

4.10 CP

Memory copy:

```
cp [.b, .w, .l] source target count
    - copy memory
```

The memory copy command copies data in memory, starting at the “source” address to the “target” address. The “count” field specifies then number of bytes, words or long words to be copied depending upon the extension field of the **cp** command. If a “**.b**” extension is used, the count field specifies the number of bytes. Likewise, if a “**.w**” or “**.l**” extension is used, the count field respectively specifies the number of words or long words.

The **cp** command is used as a FLASH programming command.

The **cp** command can copy data from one memory element to another memory element. The source can be RAM/ROM/FLASH/EPROM or any other type of memory. The destination or target memory is usually RAM; however the target memory can also be FLASH or other type of programmable, non-volatile memory. If the destination for the data is FLASH or other type of programmable, non-volatile memory, the U-Boot monitor program will determine the type of memory used as the destination and choose the appropriate programming algorithm.

The following is a typical sequence to program FLASH memory on the Media5200 Board using U-Boot.

```
setenv ldlx tftp 1000000 /tftpboot/uImage
setenv ldfs tftp 1000000 / tftpboot/fsimg
setenv uplx run ldlx \; erase ffe00000 ffefffff \; cp.b 1000000 ffe00000 \$\(filesize\)
setenv upfs run ldfs \; erase ff05000 ffdfffff \; cp.b 1000000 ffe00000 \$\(filesize\)
setenv bootdelay 2
saveenv
```

4.11 CRC32

Checksum calculation:

```
5crc32 address count [addr]
    - compute CRC32 checksum [save at addr]
=> crc 100004 3FC
CRC32 for 00100004 ... 001003ff ==> d433b05b
=>
```

The **crc32** command (short: **crc**) can be used to calculate a CRC32 checksum over a range of memory:

```
=> crc 100004 3FC
CRC32 for 00100004 ... 001003ff ==> d433b05b
=>
```

When used with 3 arguments, the command stores the calculated checksum at the given address:

```
=> crc 100004 3FC 100000
CRC32 for 00100004 ... 001003ff ==> d433b05b
=> md 100000 4
00100000: d433b05b ec3827e4 3cb0bacf 00093cf5    .3.[.8'.<.....<.
=>
```

As you can see, the CRC32 checksum was not only printed, but also stored at address 0x100000.

4.12 DCACHE

Enable or disable data cache:

```
dcache [on, off]
    - enable or disable data (writethrough) cache
```

4.13 DHCP

Invoke dhcp client to obtain ip/boot params.

4.14 DISKBOOT

Boot from ide device.

```
diskboot loadAddr dev:part
```

4.15 ECHO

Echo args to console:

```
echo [args..] - echo args to console;
                \c suppresses newline
```

The **echo** command echoes the arguments to the console:

```
=> echo The quick brown fox jumped over the lazy dog.

The quick brown fox jumped over the lazy dog.
=>
```

4.16 EEPROM

Eeprom sub-system:

```
eeprom read  addr off cnt
eeprom write addr off cnt
       - read/write `cnt' bytes at EEPROM offset `off
```

4.17 ERASE

Erase flash memory:

```
reset - No help available.

erase start end
    - erase FLASH from addr 'start' to addr 'end'
erase N:SF[-SL]
    - erase sectors SF-SL in FLASH bank # N
erase bank N
    - erase FLASH bank # N
erase all
    - erase all FLASH banks
```

4.18 FATINFO

Print information about filesystem:

```
fatinfo <interface> <dev[:part]>  - print information about filesystem from 'dev' on
'interface'

=>
```

4.19 FATLOAD

Load binary file from a dos filesystem:

```
fatload <interface> <dev[:part]>  <addr> <filename> [bytes] - load binary file 'filename'
from 'dev' on 'interface'to address 'addr' from dos filesystem

=>
```

4.20 FATLS

List files in a directory (default /):

```
fatls <interface> <dev[:part]> [directory]
```

4.21 FLINFO

Print flash memory information:

```
flinfo
    - print information for all FLASH memory banks
flinfo N
    - print information for FLASH memory bank # N

=>
```

The command **flinfo** (short: **fli**) can be used to get information about the available flash memory (see Flash Memory Commands below).

```
=> fli

Bank # 1: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40000000 (RO) 40008000 (RO) 4000C000 (RO) 40010000 (RO) 40020000 (RO)
    40040000      40060000      40080000      400A0000      400C0000
    400E0000      40100000      40120000      40140000      40160000
    40180000      401A0000      401C0000      401E0000      40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
```

```
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=>
```

4.22 GO

Start application at address 'addr':

```
go addr [arg ...]
    - start application at address 'addr'
      passing 'arg' as arguments
```

U-Boot has support for so-called standalone applications. These are programs that do not require the complex environment of an operating system to run. Instead they can be loaded and executed by U-Boot directly, utilizing U-Boot's service functions like **console I/O** or **malloc()** and **free()**.

This can be used to dynamically load and run special extensions to U-Boot like special hardware test routines or bootstrap code to load an OS image from some filesystem. The go command is used to start such standalone applications. The optional arguments are passed to the application without modification.

4.23 HELP

Print online help:

```
help [command ...]
    - show help information (for 'command')
'help' prints online help for the monitor commands.

Without arguments, it prints a short usage message for all commands.

To get detailed help information for specific commands you can type
'help' with one or more command names as arguments.
```

The **help** command (short: **h** or **?**) prints online help. Without any arguments, the **help** command prints a list of all U-Boot commands that are available in your configuration of U-Boot. You can get detailed information for a specific command by typing its name as argument to the **help** command:

4.24 ICACHE

Enable or disable instruction cache:

```
icache [on, off]
    - enable or disable instruction cache
```

4.25 ICRC32

Checksum calculation:

```
icrc32 chip address [.0, .1, .2] count -- compute CRC32 checksum
```

4.26 IDE

IDE sub-system:

```
ide reset - reset IDE controller

ide info  - show available IDE devices

ide device [dev] - show or set current device

ide part [dev] - print partition table of one or all IDE devices

ide read  addr blk# cnt

ide write addr blk# cnt - read/write `cnt' blocks starting at block `blk#'    to/from
memory address `addr'
```

4.27 ILOOP

Infinite loop on address range:

```
iloop chip address[.0, .1, .2] [# of objects]
    - loop, reading a set of addresses
```

4.28 IMD

I2C memory display:

```
imd chip address[.0, .1, .2] [# of objects]
    - i2c memory display
```

4.29 IMINFO

Print header information for application image:

```
iminfo addr [addr ...]
    - print header information for application image starting at
      address 'addr' in memory; this includes verification of the
      image contents (magic number, header and payload checksums)
```

Iminfo (short: **imi**) is used to print the header information for images like Linux kernels or ramdisks. It prints (among other information) the image name, type and size and verifies that the CRC32 checksums stored within the image are OK.

```
=> imi 100000

## Checking Image at 00100000 ...
   Image Name:   Linux-2.4.4
   Created:      2002-04-07  21:31:59 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    605429 Bytes = 591 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
```

NOTE

The exact operation of this command can be controlled by the settings of some U-Boot environment variables.

4.30 IMLS

List all images found in flash:

```
imls
    - Prints information about all images found at sector
      boundaries in flash.
```

4.31 IMM

I2C memory modify (auto-incrementing):

```
imm chip address[.0, .1, .2]
    - memory modify, auto increment address
```

4.32 IMW

Memory write (fill):

```
imw chip address[.0, .1, .2] value [count]
    - memory write (fill)
```

4.33 INM

Memory modify (constant address):

```
inm chip address[.0, .1, .2]
    - memory modify, read and keep address
```

4.34 IPROBE

Probe to discover valid I2C chip addresses.

4.35 ITEST

Return true/false on integer compare

```
itest [.b, .w, .l, .s] [*]value1 <op> [*]value2
```

4.36 LOADB

Load binary file over serial line (kermit mode)

```
loadb [ off ] [ baud ]
    - load binary file over serial line with offset 'off' and baudrate 'baud'
```

With kermit you can download binary data via the serial line. Here we show how to download uImage, the Linux kernel image. Configuring the "kermit" command and then type:

```
=> loadb 100000
## Ready for binary (kermit) download ...
Ctrl-\c
(Back at denx.denx.de)
--------------------------------------------------
C-Kermit 7.0.197, 8 Feb 2000, for Linux
 Copyright (C) 1985, 2000,
  Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
Kermit> send /bin /tftpboot/pImage
...
Kermit> connect
Connecting to /dev/ttyS0, speed 115200.
The escape character is Ctrl-\ (ASCII 28, FS)
Type the escape character followed by C to get back,
or followed by ? to see other options.
--------------------------------------------------
= 550260 Bytes
## Start Addr      = 0x00100000
=> iminfo 100000

## Checking Image at 00100000 ...
   Image Name:   Linux-2.4.4
   Created:      2002-07-02  22:10:11 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    550196 Bytes = 537 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
```

4.37 LOADS

Load S-Record file over serial line:

```
loads [ off ]
    - load S-Record file over serial line with offset 'off'
```

4.38 LOOP

Infinite loop on address range:

```
loop [.b, .w, .l] address number_of_objects
    - loop on a set of addresses
```

The **loop** command reads in a tight loop from a range of memory. This is intended as a special form of a memory test, since this command tries to read the memory as fast as possible. This command will never terminate. There is no way to stop it but to reset the board!

```
=> loop 100000 8
```

4.39 MD

Memory display:

```
md [.b, .w, .l] address [# of objects]
    - memory display
```

```
=> md 100000 10
00100000: 48616c6c 6f202020 01234567 312e312e    Hallo   .#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
```

4.40 MM

MEMORY MODIFY (AUTO-INCREMENTING)

```
mm [.b, .w, .l] address
    - memory modify, auto increment address
```

The **mm** is a method to interactively modify memory contents. It will display the address and current contents and then prompt for user input. If you enter a legal hexadecimal number, this new value will be written to the address. Then the next address will be prompted. If you don't enter any value and just press **Enter**, then the contents of this address will remain unchanged. The command stops as soon as you enter any data that is not a hex number:

```
=> mm 100000
00100000: 27051956 ? 0
00100004: 50504342 ? AABBCCDD
00100008: 6f6f7420 ? 01234567
0010000c: 312e312e ? .
=> md 100000 10
00100000: 00000000 aabbccdd 01234567 312e312e    .........#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
=>
```

Again this command can be used with the type extensions **.l**, **.w** and **.b** :

```
=> mm.w 100000
00100000: 0000 ? 0101
00100002: 0000 ? 0202
00100004: aabb ? 4321
00100006: ccdd ? 8765
00100008: 0123 ? .
=> md 100000 10
00100000: 01010202 43218765 01234567 312e312e    ....C!.e.#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
=>
=> mm.b 100000
00100000: 01 ? 48
00100001: 01 ? 61
00100002: 02 ? 6c
00100003: 02 ? 6c
00100004: 43 ? 6f
00100005: 21 ? 20
00100006: 87 ? 20
00100007: 65 ? 20
00100008: 01 ? .
```

4.41 MTEST

Simple ram test:

```
mtest [start [end [pattern]]]
    - simple RAM read/write test
```

The **mtest** provides a simple memory test.

```
=> mtest 100000 200000
Testing 00100000 ... 00200000:
Pattern 0000000F  Writing...  Reading...
=>
```

This tests writes to memory, thus modifying the memory contents. It will fail when applied to ROM or flash memory. This command may crash the system when the tested memory range includes areas that are needed for the operation of the U-Boot firmware (like exception vector code, or U-Boot's internal program code, stack or heap memory areas).

4.42 MW

Memory write (fill):

```
mw [.b, .w, .l] address value [count]
    - write memory
```

The **mw** is a way to initialize (fill) memory with some value. When called without a count argument, the value will be written only to the specified address. When used with a count, then a whole memory areas will be initialized with this value:

```
=> md 100000 10
00100000: 0000000f 00000010 00000011 00000012    ................
00100010: 00000013 00000014 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw 100000 aabbccdd
=> md 100000 10
00100000: aabbccdd 00000010 00000011 00000012    ................
00100010: 00000013 00000014 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw 100000 0 6
=> md 100000 10
00100000: 00000000 00000000 00000000 00000000    ................
00100010: 00000000 00000000 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=>
=> mw.w 100004 1155 6
=> md 100000 10
00100000: 00000000 11551155 11551155 11551155    .....U.U.U.U.U.U
00100010: 00000000 00000000 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw.b 100007 ff 7
=> md 100000 10
00100000: 00000000 115511ff ffffffff ffff1155    .....U.........U
00100010: 00000000 00000000 00000015 00000016    ................
```

```
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=>
```

4.43 NFS

Boot image via network using nfs protocol:

```
nfs [loadAddress] [host ip addr:bootfilename]
```

4.44 NM

Memory modify (constant address):

```
nm [.b, .w, .l] address
    - memory modify, read and keep address
```

The **nm** command (non-incrementing memory modify) can be used to interactively write different data several times to the same address. This can be useful for instance to access and modify device registers:

```
=> nm.b 100000
00100000: 00 ? 48
00100000: 48 ? 61
00100000: 61 ? 6c
00100000: 6c ? 6c
00100000: 6c ? 6f
00100000: 6f ? .
=> md 100000 8
00100000: 6f000000 115511ff ffffffff ffff1155    o....U.........U
00100010: 00000000 00000000 00000015 00000016    ................
=>
```

4.45 PCI

List and access pci configuration space:

```
pci [bus] [long]
    - short or long list of PCI devices on bus 'bus'
pci header b.d.f
    - show header of PCI device 'bus.device.function'
pci display[.b, .w, .l] b.d.f [address] [# of objects]
    - display PCI configuration space (CFG)
pci next[.b, .w, .l] b.d.f address
    - modify, read and keep CFG address
pci modify[.b, .w, .l] b.d.f address
    -  modify, auto increment CFG address
pci write[.b, .w, .l] b.d.f address value
- write to CFG address
```

4.46 PING

Send icmp echo_request to network host:

```
ping pingAddress
```

4.47 PRINTENV

Print environment variables:

```
printenv
    - print values of all environment variables
printenv name ...
    - print value of environment variable 'name'
```

The **printenv** command prints one, several or all variables of the U-Boot environment. When arguments are given, these are interpreted as the names of environment variables which will be printed with their values:

```
=> printenv ipaddr hostname netmask
ipaddr=10.0.0.99
hostname=tqm
netmask=255.0.0.0
=>
```

Without arguments, **printenv** prints all a list with all variables in the environment and their values, plus some statistics about the current usage and the total size of the memory available for the environment.

```
=> printenv
baudrate=115200
serial#=TQM860LDDBA3-P50.203 10226122 4
ethaddr=00:D0:93:00:28:81
bootdelay=5
loads_echo=1
clocks_in_mhz=1
load=tftp 100000 /tftpboot/ppcboot.bin
update=protect off all;era 1:0-4;cp.b 100000 40000000 $(filesize);setenv filesize;saveenv
rtai=tftp 100000 /tftpboot/pImage.rtai;run nfsargs;run addip;bootm
preboot=echo;echo Type "run flash_nfs" to mount root filesystem over NFS;echo
nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=$(serverip):$(rootpath)
addip=setenv bootargs $(bootargs)
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname):$(netdev):off panic=1
flash_nfs=run nfsargs;run addip;bootm $(kernel_addr)
kernel_addr=40040000
netdev=eth0
hostname=tqm
rootpath=/opt/hardhat/devkit/ppc/8xx/target
ramargs=setenv bootargs root=/dev/ram rw
flash_self=run ramargs;run addip;bootm $(kernel_addr) $(ramdisk_addr)
ramdisk_addr=40100000
bootcmd=run flash_self
stdin=serial
stderr=serial
stdout=serial
filesize=dd
netmask=255.0.0.0
ipaddr=10.0.0.99
serverip=10.0.0.2

Environment size: 992/16380 bytes
=>
```

4.48 PROTECT

Enable or disable flash write protection:

```
protect on  start end
    - protect FLASH from addr 'start' to addr 'end'
protect on  N:SF[-SL]
    - protect sectors SF-SL in FLASH bank # N
protect on  bank N
    - protect FLASH bank # N
protect on  all
    - protect all FLASH banks
protect off start end
    - make FLASH from addr 'start' to addr 'end' writable
protect off N:SF[-SL]
    - make sectors SF-SL writable in FLASH bank # N
protect off bank N
    - make FLASH bank # N writable
protect off all
    - make all FLASH banks writable

=>
```

The **protect** command is another complex one. It is used to set certain parts of the flash memory to read-only mode or to make them writable again. Flash memory that is "protected" (= read-only) cannot be written (with the **cp** command) or erased (with the **erase** command). Protected areas are marked as (RO) (for "read-only") in the output of the **flinfo** command:

```
=> fli

Bank # 1: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40000000 (RO) 40008000 (RO) 4000C000 (RO) 40010000 (RO) 40020000 (RO)
    40040000      40060000      40080000      400A0000      400C0000
    400E0000      40100000      40120000      40140000      40160000
    40180000      401A0000      401C0000      401E0000      40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=> protect on 40100000 401FFFFF
Protected 8 sectors
=> fli

Bank # 1: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
```

```
    40000000 (RO) 40008000 (RO) 4000C000 (RO) 40010000 (RO) 40020000 (RO)
    40040000      40060000      40080000      400A0000      400C0000
    400E0000      40100000 (RO) 40120000 (RO) 40140000 (RO) 40160000 (RO)
    40180000 (RO) 401A0000 (RO) 401C0000 (RO) 401E0000 (RO) 40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=> era 40100000 401FFFFF
Erase Flash from 0x40100000 to 0x401fffff - Warning: 8 protected sectors will not be
erased!
 done
Erased 8 sectors
=> protect off 1:11
Un-Protect Flash Sectors 11-11 in Bank # 1
=> fli

Bank # 1: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40000000 (RO) 40008000 (RO) 4000C000 (RO) 40010000 (RO) 40020000 (RO)
    40040000      40060000      40080000      400A0000      400C0000
    400E0000      40100000      40120000 (RO) 40140000 (RO) 40160000 (RO)
    40180000 (RO) 401A0000 (RO) 401C0000 (RO) 401E0000 (RO) 40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=> era 1:11
Erase Flash Sectors 11-11 in Bank # 1
. done
=>
```

The actual level of protection depends on the flash chips used on your hardware, and on the implementation of the flash device driver for this board. In most cases U-Boot provides just a simple software-protection, i.e. it prevents you from erasing or overwriting important stuff by accident (like the U-Boot code itself or U-Boot's environment variables), but it cannot prevent you from circumventing these

restrictions - a nasty user who is loading and running his own flash driver code cannot and will not be stopped by this mechanism. Also, in most cases this protection is only effective while running U-Boot, i.e. any operating system will not know about "protected" flash areas and will happily erase these if requested to do so.

4.49 RARPBOOT

Boot image via network using RARP/TFTP protocol:

```
rarpboot [loadAddress] [bootfilename]
```

4.50 REGINFO

Print register information.

4.51 RESET

Perform reset of the cpu.

The **reset** command reboots the system.

4.52 RUN

RUn commands in an environment variable:

```
run var [...]  - run the commands in the environment variable(s) 'var'
```

You can use U-Boot environment variables to store commands and even sequences of commands. To execute such a command, you use the **run** command:

```
=> setenv test echo This is a test\;printenv ipaddr\;echo Done.=> printenv test
test=echo This is a test;printenv ipaddr;echo Done.
=> run test
This is a test ipaddr=10.0.0.99
Done.
=>
```

You can call **run** with several variables as arguments, in which case these commands will be executed in sequence:

```
=> setenv test2 echo This is another Test\;printenv serial#\;echo Done.
=> printenv test test2
test=echo This is a test;printenv ipaddr;echo Done.
test2=echo This is another Test;printenv serial#;echo Done.
=> run test test2
This is a test
ipaddr=10.0.0.99
Done.
This is another Test
serial#=TQM860LDDBA3-P50.203 10226122 4
Done.
=>
```

If a U-Boot variable contains several commands (separated by semicolon), and one of these commands fails when you "run" this variable, the remaining commands will be executed anyway. If you execute

several variables with one call to run, any failing command will cause "run" to terminate, i.e. the remaining variables are not executed.

4.53 SAVEENV

Save environment variables to persistent storage.

All changes you make to the U-Boot environment are made in RAM only. They are lost as soon as you reboot the system. If you want to make your changes permanent you have to use the **saveenv** command to write a copy of the environment settings to persistent storage, from where they are automatically loaded during startup:

```
=> saveenv
Saving Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=>
```

4.54 SETENV

Set environment variables:

```
setenv name value ...
    - set environment variable 'name' to 'value ...'
setenv name
    - delete environment variable 'name'
```

To modify the U-Boot environment you have to use the **setenv** command. When called with exactly one argument, it will delete any variable of that name from U-Boot's environment, if such a variable exists. Any storage occupied for such a variable will be automatically reclaimed:

```
=> printenv foo
foo=This is an example value.
=> setenv foo
=> printenv foo
## Error: "foo" not defined
=>
```

When called with more arguments, the first one will again be the name of the variable, and all following arguments will (concatenated by single space characters) form the value that gets stored for this variable. New variables will be automatically created, existing ones overwritten.

```
=> printenv bar
## Error: "bar" not defined
=> setenv bar This is a new example.
=> printenv bar
bar=This is a new example.
=>
```

Remember standard shell quoting rules when the value of a variable shall contain characters that have a special meaning to the command line parser (like the **$** character that is used for variable substitution or

the semicolon which separates commands). Use the backslash (\) character to escape such special characters.

```
=> setenv cons_opts console=tty0 console=ttyS0,\$(baudrate)
=> printenv cons_opts
cons_opts=console=tty0 console=ttyS0,$(baudrate)
=>
```

There is no restriction on the characters that can be used in a variable name except the restrictions imposed by the command line parser (like using backslash for quoting, space and tab characters to separate arguments, or semicolon and newline to separate commands). Even strange input like "=-/|()+=" is a perfectly legal variable name in U-Boot.

A common mistake is to write

```
setenv name=value
```

instead of

```
setenv name value
```

There will be no error message, which lets you believe everything went OK, but it didn't: instead of setting the variable name to the value you tried to delete a variable with the name name=value. This is probably not what you intended. Always remember that name and value have to be separated by space and/or tab characters.

4.55 SLEEP

Delay execution for some time:

```
sleep N - delay execution for N seconds (N is _decimal_ #)
```

The **sleep** command pauses execution for the number of seconds given as the argument:

```
=> date ; sleep 5 ; date
Date: 2002-04-07 (Sunday)    Time: 23:15:40
Date: 2002-04-07 (Sunday)    Time: 23:15:45
=>
```

4.56 TFTPBOOT

Boot image via network using TFTP protocol:

```
tftpboot [loadAddress] [bootfilename]
```

4.57 USB

USB sub-system:

```
usb reset - reset (rescan) USB controller
usb stop [f]  - stop USB [f]=force stop
usb tree  - show USB device tree
usb info [dev] - show available USB devices
usb scan  - (re-)scan USB bus for storage devices
usb device [dev] - show or set current USB storage device
usb part [dev] - print partition table of one or all USB storage devices
usb read addr blk# cnt - read `cnt' blocks starting at block `blk#'
```

```
        to memory address `addr'
```

4.58 USBBOOT

Boot from USB device:

```
usbboot loadAddr dev:part
```

4.59 VERSION

Print monitor version:

You can print the version and build date of the U-Boot image running on your system using the **version** command (short: **vers**):

```
=> version

U-Boot 1.1.3 (Apr 25 2006 - 22:09:33)
=>
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: LITE5200BUBPG
Rev. 0
5/2006

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

freescale™
semiconductor