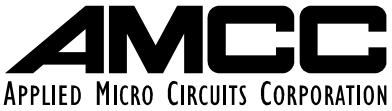
U-Boot
User's Manual

PowerPC®



Applied Micro Circuits Corporation
6290 Sequence Dr., San Diego, CA 92121
Phone: (858) 450-9333 — (800) 755-2622 — Fax: (858) 450-9885
http://www.amcc.com

Contents

1. U-Boot Overview

This section explains how to boot the AMCC 440SPe-MX evaluation board provided in the kit. This information is intended for technical personnel. For hardware-related information, such as setting DIP switches and connecting peripherals, see the *PPC440SPe Evaluation Board User's Manual* supplied in the kit.

U-Boot initializes the PowerPC microprocessor and system-level board components, and enables the loading and executing of application programs.

U-Boot is the bootloader shipped with AMCC 440SPe-MX evaluation board kit. Its main purposes are to:

- Initialize the hardware, especially the memory controller
- Provide boot parameters for the Linux kerne
- Start the Linux kernel
- Copy binary images from RAM to FLASH memory

U-Boot also acts as a monitor and provides many useful functions, such as memory dump and memory write. It allows updating U-Boot without having to boot an operating system. With the permanently storable environment variables, U-Boot is highly configurable. Thus, the boot device, initial console interface, boot parameters, and so on, are configurable.

1.1 How to Build U-Boot

U-Boot has been built in cross environments running ELDK/ Fedora core3 and Cygwin/Windows 2000.

• Linux :

It is assumed that you have the GNU cross compiling tools available in your path, named with a prefix of powerpc-linux-. If this is not the case, you must install the Embedded Linux Development Kit (ELDK) from Denx Software engineering (http://www.denx.de/) . ELDK includes the GNU cross development tools, such as the compilers, binutils, gdb, and the pre-built target tools and libraries needed to provide some functionality on the target system.

• Cygwin/Windows2k

A cross compiler is provided on the AMCC CD provided in the AMCC 440SPe-MX evaluation board kit. Cygwin software is also on the CD and can be installed if needed without any Internet connection.

1.2 How to Recompile

Create an environment variable or change the definition in makefile and set your PATH environment. For example, in a Cygwin environment with the cross compiler provided on the AMCC CD:

```
export CROSS_COMPILE=powerpc-440-linux-gnu-
PATH=$PATH:/opt/amcc-ppc-gnu-compiler/gcc-3.4.1-glibc-2.3.3/bin
```

Clear everything by typing:

```
make mrproper
```

After installing the sources you must configure U-Boot for the AMCC 440SPe-MX evaluation board. This is done by typing:

```
make EVB440SPE_config
```

Finally, type `make all`, and you will get some working U-Boot images ready for download to / installation on your system:

- "u-boot.bin" is a raw binary image
- "u-boot" is an image in ELF binary format
- "u-boot.srec" is in Motorola S-Record format

Note: The makefiles assume that you are using GNU make.

Configuration:

- Various Options can be set by Environment variables
- Environment variables are permanently stored in the 1st sector of small flash
- Environment variables are protected with a CRC32 checksum
- kernel booting:
  1. Supports compressed (gzip) kernel images
  2. Images are checked with a CRC32 checksum prior of booting
  3. Autoboot option
  4. Boot arguments can be stored in the environment variables

1.3 Initial Steps

In the default configuration, U-Boot operates in interactive mode, which provides a simple command line-oriented user interface that uses a serial console on uart0. To connect and power on your evaluation board, see the *PPC440SPe Evaluation Board User's Manual* and the *Quick Start Instructions*. Use your preferred terminal program (TTerm, minicom...) to access the U-Boot command interface. The default parameters of this serial communication are as follows:

- 115200 baud
- 8 bits
- Parity none
- 1 stop bit
- No handshake

Note that you must use a null modem cable to get a working communication link. In the simplest case, this means that U-Boot displays a prompt (default: =>) when it is ready to receive user input. You then type a command, and press *Enter*. U-Boot will run the required action(s), and then prompt for another command. After power on, a typical U-Boot init complete message such as the following is displayed on the serial console:

```
U_440SPe_V1R01 level01
AMCC PowerPC 440 SPe 3GA533C
Board: AMCC 440SPe Evaluation Board
VCO: 1066 MHz
CPU: 533 MHz
PLB: 133 MHz
OPB: 66 MHz
EPB: 66 MHz
I2C: ready
DRAM: 1024 MB
FLASH: 1 MB
PCI: Bus Dev VenId DevId Class Int
00 01 8086 1229 0200 00
In: serial
Out: serial
Err: serial
Net: ppc_440x_eth0, i82559#0
=>
```

In some cases you may see this message

```
*** Warning - bad CRC, using default environment
```

This is of no consequence and will not occur once you have initialized and saved the environment variables.

To see a list of the available U-Boot commands, you can type help (or simply ?). This will print a list of all commands that are available in your current configuration as shown. Note that U-Boot provides numerous configuration options; not all options are available and some options might simply not be selected for your configuration.

```
=> help
askenv - get environment variables from stdin
autoscr - run script from memory
base - print or set address offset
bdinfo - print Board Info structure
bootm - boot application image from memory
bootp - boot image via network using BootP/TFTP protocol
bootd - boot default, i.e., run 'bootcmd'
cmp - memory compare
coninfo - print console devices and informations
cp - memory copy
crc32 - checksum calculation
date - get/set/reset date &time
dhcp - invoke DHCP client to obtain IP/boot params
diskboot- boot from IDE device
echo - echo args to console
erase - erase FLASH memory
flinfo - print FLASH memory information
go - start application at address 'addr'
help - print online help
ide - IDE sub-system
iminfo - print header information for application image
loadb - load binary file over serial line (kermit mode)
loads - load S-Record file over serial line
loop - infinite loop on address range
md - memory display
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nm - memory modify (constant address)
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset - Perform RESET of the CPU
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
sleep - delay execution
tftpboot- boot image via network using TFTP protocol and env variables ipaddr
and serverip
version - print monitor version
? - alias for 'help'
=>
```

With the command **`help`** *`command`* (where *`command`* is the name of the command for which you need help) you can get additional information about most commands:

```
tftpboot [loadAddress] [bootfilename]
=> help setenv printenv
setenv name value ...
- set environment variable 'name' to 'value ...'
setenv name
- delete environment variable 'name'
printenv
- print values of all environment variables
printenv name ...
- print value of environment variable 'name'
=>
```

Most commands can be abbreviated if the string remains unambiguous.

```
flinfo
- print information for all FLASH memory banks
flinfo N
- print information for FLASH memory bank # N
tftpboot [loadAddress] [bootfilename]
=>
```

2. U-Boot Command Line Interface

This section describes the commands available in U-Boot. The behavior of some commands depends on the configuration of U-Boot and on the definition of certain variables in your U-Boot environment. All U-Boot commands expect numbers to be entered in hexadecimal input format. Be careful not to use edit keys other than 'Backspace', because hidden characters in things such as environment variables can be very difficult to find. The list of supported commands is as follows:

- **autoscr** - run script from memory
- **base** - print or set address offset
- **bdinfo** - print Board Info structure
- **bootm** - boot application image from memory
- **bootp** - boot image via network using BootP/TFTP protocol
- **bootd** - boot default, that is, run bootcmd
- **cmp** - memory compare
- **coninfo** - print console devices and information
- **cp** - memory copy
- **crc32** - checksum calculation
- **echo** - echo args to console
- **erase** - erase FLASH memory
- **flinfo** - print FLASH memory information
- **go** - start application at address 'addr'
- **help** - print online help
- **iminfo** - print header information for application image
- **loop** - infinite loop on address range
- **md** - memory display
- **mm** - memory modify (auto-incrementing)
- **mtest** - simple RAM test
- **mw** - memory write (fill)
- **nm** - memory modify (constant address)
- **printenv**- print environment variables
- **protect** - enable or disable FLASH write protection
- **rarpboot**- boot image via network using RARP/TFTP protocol
- **reset** - Perform RESET of the CPU
- **run** - run commands in an environment variable
- **saveenv** - save environment variables to persistent storage
- **setenv** - set environment variables
- **sleep** - delay execution
- **tftpboot**- boot image via network using TFTP protocol and env variables ipaddr and serverip
- **version** - print monitor version
- **?** - alias for 'help'

2.1 Information Commands

This section describes the information commands.

2.1.1 bdinfo - Print Board Information Structure

```
=> help bdinfo
bdinfo - No help available.
=>
```

bdinfo (short: **bdi**) displays the information that U-Boot provides about the board such as memory addresses and sizes, clock frequencies, MAC address, and so on. This information is primarily needed to pass to the Linux kernel.

```
=> bdinfo
memstart = 0x00000000
memsize = 0x40000000
flashstart = 0xFFF10000
flashsize = 0x00100000
flashoffset = 0x0002F800
sramstart = 0x00000000
sramsize = 0x00000000
bootflags = 0x00000060
intfreq = 533.333 MHz
busfreq = 133.333 MHz
ethaddr = 00:04:AC:E3:28:8A
IP addr = 192.168.2.22
baudrate = 115200 bps
=>
```

2.1.2 coninfo - Print Console Devices and Information

```
=> help conin
coninfo
=>
```

coninfo (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 in this example 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 three standard I/O streams stdin, stdout, and stderr.

2.1.3 flinfo - Print FLASH Memory Information

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

flinfo (short: **fli**) displays information about the available flash memory (see *Flash Memory Commands* on page 23 for details).

```
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 FFF10000 FFF20000 FFF30000 FFF40000
FFF50000 FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
```

2.1.4 iminfo - Print Header Information for Application Image

```
=> help iminfo
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**) displays header information for images such as Linux kernels or ramdisks, including the image name, type, and size, and verifies that the CRC32 checksums stored within the image are valid.

```
=> imi
## Checking Image at 00100000 ...
Image Name: Linux-2.4.20_mvl31-yucca_V1R01L0
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 899971 Bytes = 878.9 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
=>
```

As with many other commands, the specific operation of this command can be controlled by the settings of certain U-Boot environment variables (for this command, the **verify** environment variable). See *U-Boot Environment Variables* on page 39 for details.

2.1.5 help - Print Online Help

```
=> help 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 **?**) displays online help. If no args are provided, it 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 an arg on the **help** command.

```
=> help protect 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
```

2.2 Memory Commands

This section describes the memory commands.

2.2.1 base - Print or Set Address Offset

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

base (short: **ba**) displays or sets a base address, which is used as address offset for all memory commands. The default value of the base address is 0, so all addresses you enter are used unmodified. However, if you need to access a certain memory region (such as the internal memory of some embedded PowerPC processors) repeatedly, it is 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: aaaaaaaa aaaaaaaa 55555555 55555555 ........UUUUUUUU
00000010: aaaaaaaa aaaaaaaa 55555555 55555555 ........UUUUUUUU
00000020: 74fd3f74 bfbbffbd e94b755f b43c355b t.?t.....Ku_.<5[
=>
=> base fff10000
Base Address: 0xfff10000
=> md 0 c
fff10000: 27051956 552d426f 6f742031 2e312e32 '..VU-Boot 1.1.2
fff10010: 20284a75 6e203232 20323030 35202d20 (Jun 22 2005 -
fff10020: 30383a30 363a3139 290a555f 34343053 08:06:19).U_440S
=>
```

2.2.2 crc32 - Checksum Calculation

crc32 (short: **crc**) calculates a CRC32 checksum over a range of memory.

```
=> help crc32
crc32 address count [addr]
- compute CRC32 checksum [save at addr]
=>
CRC32 for 00100004 ... 001003ff ==> d433b05b
=>
```

If used with three args, **crc32** stores the calculated checksum at the specified address**.**

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

Thus, the CRC32 checksum was not only displayed, but also stored at address 0x100000.

2.2.3 cmp - Memory Compare

```
=> help cmp
cmp [.b, .w, .l] addr1 addr2 count
- compare memory
=>
```

cmp tests the contents of two memory areas as identical or not. **cmp** either tests the entire area specified by the third (length) arg, or stops at the first difference.

```
=> cmp 100000 200000 400
word at 0x00100004 (0x25153273) != word at 0x00200004 (0xaabbccdd)
Total of 1 word were the same
=> md 100000 c
00100000: 27051956 25153273 42b7f5d0 000dbb83 '..V%.2sB.......
00100010: 00000000 00000000 4e08f15d 05070201 ........N..]....
00100020: 4c696e75 782d322e 342e3230 5f6d766c Linux-2.4.20_mvl
=> md 200000 c
00200000: 27051956 aabbccdd 42b7f5d0 000dbb83 '..V....B.......
00200010: 00000000 00000000 4e08f15d 05070201 ........N..]....
00200020: 4c696e75 782d322e 342e3230 5f6d766c Linux-2.4.20_mvl
=>
```

As with most memory commands, **cmp** can access the memory in different sizes: as 32-bit (long word), 16-bit (word) or 8-bit (byte) data. If invoked in default form (**cmp**), the default size (32-bit or long words) is used. (The same can be selected explicitly by typing **cmp.l***.)* To access memory as 16-bit or word data, use the form **cmp.w**. To access memory as 8-bit or byte data use the form **cmp.b**. Note that the count arg specifies the number of data items to process, that is, the number of long words, words, or bytes to compare.

```
=> cmp.l 100000 200000 400
word at 0x00100004 (0x25153273) != word at 0x00200004 (0xaabbccdd)
Total of 1 word were the same
=> cmp.w 100000 200000 400
halfword at 0x00100004 (0x2515) != halfword at 0x00200004 (0xaabb)
Total of 2 halfwords were the same
=> cmp.b 100000 200000 400
byte at 0x00100004 (0x25) != byte at 0x00200004 (0xaa)
Total of 4 bytes were the same
=>
```

2.2.4 cp - Memory Copy

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

cp copies memory areas.

```
=> cp 40000000 100000 10000
=>
```

cp supports the .l, .w, and .b command forms.

2.2.5 md - Memory Display

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

md displays memory contents both as hexadecimal and ASCII data.

```
=> md fff10000 100
fff10000: 27051956 552d426f 6f742031 2e312e32 '..VU-Boot 1.1.2
fff10010: 20284a75 6e203232 20323030 35202d20 (Jun 22 2005 -
fff10020: 30383a30 363a3139 290a555f 34343053 08:06:19).U_440S
fff10030: 50655f56 31523031 206c6576 656c3031 Pe_V1R01 level01
fff10040: 00000000 00000000 00000000 00000000 ................
fff10050: 00000000 00000000 00000000 00000000 ................
fff10060: 00000000 00000000 00000000 00000000 ................
fff10070: 00000000 00000000 00000000 00000000 ................
fff10080: 00000000 00000000 00000000 00000000 ................
fff10090: 00000000 00000000 00000000 00000000 ................
fff100a0: 00000000 00000000 00000000 00000000 ................
fff100b0: 00000000 00000000 00000000 00000000 ................
fff100c0: 00000000 00000000 00000000 00000000 ................
=>
```

md supports the .l, .w, and .b command forms.

```
==> md.w fff10000
fff10000: 2705 1956 552d 426f 6f74 2031 2e31 2e32 '..VU-Boot 1.1.2
fff10010: 2028 4a75 6e20 3232 2032 3030 3520 2d20 (Jun 22 2005 -
fff10020: 3038 3a30 363a 3139 290a 555f 3434 3053 08:06:19).U_440S
```

```
fff10030: 5065 5f56 3152 3031 206c 6576 656c 3031 Pe_V1R01 level01
fff10040: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff10050: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff10060: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff10070: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff10080: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff10090: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff100a0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff100b0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
fff100c0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
=> md.b fff10000
fff10000: 27 05 19 56 55 2d 42 6f 6f 74 20 31 2e 31 2e 32 '..VU-Boot 1.1.2
fff10010: 20 28 4a 75 6e 20 32 32 20 32 30 30 35 20 2d 20 (Jun 22 2005 -
fff10020: 30 38 3a 30 36 3a 31 39 29 0a 55 5f 34 34 30 53 08:06:19).U_440S
fff10030: 50 65 5f 56 31 52 30 31 20 6c 65 76 65 6c 30 31 Pe_V1R01 level01
fff10040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff10050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff10060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff10070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff10080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff10090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
fff100f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
=>
```

The last memory address displayed and the value of the count argument are stored so if you enter **md** again without arguments it will continue automatically at the next address, and use the same count.

2.2.6 mm - Memory Modify (Auto-Incrementing)

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

mm provides a way to modify memory contents interactively. It will display the address and current contents and then prompt for user input. If you enter a legitimate hexadecimal number, this new value will be written to the address. Then it will prompt for the next address. If you do not enter a 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 hexadecimal number.

```
=> mm 100000
00100000: 27051956 ? 0
00100004: 25153273 ? aabbccdd
00100008: 42b7f5d0 ? 01234567
0010000c: 000dbb83 ? 0
00100010: 00000000 ? .
=> md 100000 10
00100000: 00000000 aabbccdd 01234567 00000000 .........#Eg....
00100010: 00000000 00000000 4e08f15d 05070201 ........N..]....
00100020: 4c696e75 782d322e 342e3230 5f6d766c Linux-2.4.20_mvl
00100030: 33312d79 75636361 5f563152 30314c30 31-yucca_V1R01L0
=>
```

mm supports the .l, .w, and .b command forms.

2.2.7 mtest - Simple RAM Test

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

mtest provides a simple memory test.

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

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

2.2.8 mw - Memory Write (Fill)

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

mw initializes (fills) memory with a specified value. If entered without a count arg, the value will be written only to the specified address. If entered with a count, then entire memory areas will be initialized with this value.

```
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 supports the .l, .w, and .b command forms.

```
=> 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 ................
=>
```

2.2.9 nm - Memory Modify (Constant Address)

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

nm (non-incrementing memory modify) interactively writes different data several times to the same address. This can be useful to access and modify device registers.

nm supports the .l, .w, and .b command forms.

2.2.10 loop - Infinite Loop on Address Range

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

loop 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. **loop** will never terminate. There is no way to stop it other than to reset the board!

2.3 Flash Memory Commands

This section describes the flash memory commands.

2.3.1 cp - Memory Copy

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

cp recognizes flash memory areas and will automatically invoke the necessary flash programming algorithm if the target area is in flash memory.

```
=> cp 100000 40000000 10000
Copy to Flash... done
=>
```

Writing to flash memory may fail if the target area has not been erased (see *erase - Erase FLASH Memory* on page 24), or if it is write protected (see *protect - Enable or Disable FLASH Write Protection* on page 25).

```
=> cp 100000 40000000 10000
Copy to Flash... Can't write to protected Flash sectors
```

Remember that the *count* arg specifies the number of items to be copied. If you specify a length instead (=byte count), use the command form **cp.b**. Otherwise you will have to calculate the correct number of items.

2.3.2 flinfo - Print FLASH Memory Information

flinfo (short: ***fli***) displays information about the available flash memory. The number of flash banks is displayed with information about the size and organization into flash sectors or erase units. For all sectors, the start addresses are displayed; some sectors may be write protected and will be marked as read-only (RO) in the output.

```
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 FFF10000 FFF20000 FFF30000 FFF40000
FFF50000 FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=>
```

2.3.3 erase - Erase FLASH Memory

```
=> help era
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
=>
```

erase (short: **era**) erases the contents of one or more sectors of flash memory. It is one of the more complex commands. The most likely usage of this command is to pass the start and end addresses of the area to be erased.

```
=> erase fff10000 fff1ffff
Erasing sector fff10000
. done
Erased 1 sectors
=>
```

Note that both the start and end addresses for this command must point exactly at the start end addresses of flash sectors, respectively. Otherwise the command will be not executed. Another way to select specific areas of flash memory for the **erase** command uses the notation of flash banks and sectors. A bank is an area of memory implemented by one or more memory chips that are connected to the same chip-select signal of the CPU. A flash sector or erase unit is the smallest area that can be erased in one operation.

For practical purposes, keep in mind that a bank is something that eventually may be erased entirely in a single operation. This may be more efficient (faster) than erasing the same area sector-by-sector. Whether such a fast bank erase algorithm exists depends upon the actual type of flash chips used on the board, and upon the implementation of the flash device driver actually used. In U-Boot, flash banks are numbered starting with 1, while flash sectors start with 0. To erase the same flash area as specified using start and end addresses in the example above you could also type:

```
=> era 1:6-8
Erase Flash Sectors 6-8 in Bank # 1
.. done
=>
```

To erase an entire bank of flash memory you can use a command such as this. Note that a warning message is displayed because some write protected sectors exist in these flash banks that were not erased. The entire flash memory (except for the write-protected sectors) can be erased with the following command:

```
=> era all
Erase Flash Bank # 1 - Warning: 1 protected sectors will not be erased!
.................. done
Erase Flash Bank # 2
......................... done
=>
```

2.3.4 protect - Enable or Disable FLASH Write Protection

```
=> help protect
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
=>
```

protect is another complex command. It sets certain parts of the flash memory to read-only mode or makes 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 readonly) in the output of the **flinfo** command.

```
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 FFF10000 FFF20000 FFF30000 FFF40000
FFF50000 FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=> protect on fff00000 fff0ffff
Protected 1 sectors
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 RO FFF10000 FFF20000 FFF30000 FFF40000 E
FFF50000 E FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=> erase fff00000 fff0ffff
- Warning: 1 protected sectors will not be erased!
```

```
done
Erased 1 sectors
> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 RO FFF10000 FFF20000 FFF30000 FFF40000 E
FFF50000 E FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=>
=> protect off 1:0
Un-Protect Flash Sectors 0-0 in Bank # 1
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 FFF10000 FFF20000 FFF30000 FFF40000 E
FFF50000 E FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=> erase 1:0
Erase Flash Sectors 0-0 in Bank # 1
Erasing sector fff00000
. done
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 E FFF10000 FFF20000 FFF30000 FFF40000 E
FFF50000 E FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=>
```

The actual level of protection depends upon the flash chips used on your hardware, and upon the implementation of the flash device driver for this board. In most cases U-Boot provides only simple software protection, that is, it prevents you from erasing or overwriting important information by accident (such as the U-Boot code itself or U-Boot's environment variables), but it cannot prevent you from circumventing these restrictions. A user who is loading and running his own flash driver code cannot be stopped by this mechanism. Also, in most cases this protection is only effective while running U-Boot; any operating system will not know about protected flash areas and will erase these if requested to do so.

2.4 Execution Control Commands

This section describes the execution control commands.

2.4.1 autoscr - Run Script from Memory

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

autoscr enables you to run shell scripts under U-Boot. You create a U-Boot script image by writing the commands you want to run into a text file. Then you must use the **mkimage** command 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, and with **autoscr** you can run the commands in such an image. For instance, the following text file can be converted into a U-Boot script image by using the **mkimage** command as shown in the example that follows it.

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

The **mkimage** command is as follows:

```
bash$ mkimage -A ppc -O linux -T script -C none -a 0 -e 0 \
> -n "autoscr example script" \
> -d /tftpboot/440SPe/example.script /tftpboot/440SPe/example.img
Image Name: autoscr example script
Created: Mon Jun 16 01:15:02 2005
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
```

Now you can load and execute this script image in U-Boot as follows:

```
=> tftp 100000 /tftpboot/440SPe/example.img
ARP broadcast 1
TFTP from server 10.0.0.2; our IP address is 10.0.0.99
Filename '/tftpboot/440SPe/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/ppc4xx/target
=>
```

2.4.2 bootm - Boot Application Image from Memory

```
=> help bootm
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
=>
```

bootm starts operating system images. From the image header the command gets information about the type of the operating system, the file compression method used (if any), the load and entry point addresses, and so on. The command will then load the image to the specified 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 its entry point.

The first argument to **bootm** is the memory address (in RAM, ROM or flash memory) at which the image is stored, followed by optional arguments that depend on the OS. For Linux, only one optional argument can be passed. If it is present, it is interpreted as the start address of an **initrd** ramdisk image (in RAM, ROM or flash memory).

In this case the *bootm* command consists of three steps:

1. The Linux kernel image is uncompressed and copied into RAM.
2. The ramdisk image is loaded to RAM.
3. Control is given to the Linux kernel, passing information about the location and size of the ramdisk image.

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

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

If a ramdisk image will be used, you can type:

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

Both examples imply that the variables used are set to correct addresses for a kernel and an **initrd** ramdisk image. When booting images that have been loaded to RAM (for example, using TFTP download) you must be careful that the locations where the (compressed) images were stored do not overlap with the memory needed to load the uncompressed kernel. Thus, 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.

2.4.3 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 in which to run. Instead they can be loaded and executed by U-Boot directly, utilizing U-Boot's service functions such as console I/O or **malloc**() and **free**(). This can be used to dynamically load and run special extensions to U-Boot such as special hardware test routines or bootstrap code to load an OS image from some file system. **go** is used to start such standalone applications. The optional args are passed to the application without modification.

2.5 Download Commands

This section describes the download commands.

2.5.1 bootp - Boot Image via Network Using BOOTP/TFTP Protocol

```
=> help bootp
bootp [loadAddress] [bootfilename]
=>
```

Assuming that Dhcpd services is correctly configured and started on your Linux box, **bootp** results in the following:

```
=> bootp
About preceding transfer (eth0):
- Sent packet number 0
- Received packet number 0
- Handled packet number 0
Waiting for PHY auto negotiation to complete. done
ENET Speed is 1000 Mbps - FULL duplex connection
BOOTP broadcast 1
DHCP client bound to address 192.168.3.22
Using ppc_440x_eth0 device
TFTP from server 192.168.3.223; our IP address is 192.168.3.22
Filename '440SPe/vmlinux.UBoot'.
Load address: 0x100000
Loading: #################################################################
#################################################################
#############################################
done
Bytes transferred = 900035 (dbbc3 hex)
=>
```

2.5.2 rarpboot - Boot Image via Network Using RARP/TFTP Protocol

```
rarpboot [loadAddress] [bootfilename]
=>
```

2.5.3 tftpboot - Boot Image via Network Using TFTP Protocol

```
=> help tftp
tftpboot [loadAddress] [bootfilename]
=>
```

2.6 Environment Variables Commands

This section describes the environment variables commands.

2.6.1 printenv - Print Environment Variables

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

printenv prints one, several, or all variables of the U-Boot environment. If args are specified, these are interpreted as the names of environment variables that will be displayed with their values.

```
=> printenv ipaddr hostname netmask
ipaddr=192.168.3.22
## Error: "hostname" not defined
netmask=255.255.255.0
=>
```

If no args are provided, **printenv** prints a list with all variables in the environment and their values, as well as statistics about the current usage and total size of the memory available for the environment.

```
=> printenv
bootargs=root=/dev/hda1
bootcmd=bootm ffc00000
baudrate=115200
loads_echo=1
ethaddr=00:04:AC:E3:28:8A
updateFlash=erase 1:1-7;erase 2:0-7;tftp 8000 d:/cygwin/opt/ppc440spe_kit/uboo}
pciconfighost=no
stdin=serial
stdout=serial
stderr=serial
ethact=ppc_440x_eth0
bootfile=440SPe/vmlinux.UBoot
filesize=dbbc3
fileaddr=100000
gatewayip=192.168.3.1
netmask=255.255.255.0
rootpath=/development/targets/440SPe/yucca1Gtarget
ipaddr=192.168.3.22
serverip=192.168.3.223
Environment size: 517/65532 bytes
=>
```

2.6.2 saveenv - Save Environment Variables to Persistent Storage

```
=> help saveenv
saveenv - No help available.
=>
```

Any 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 must use **saveenv** to write a copy of the environment settings to persistent storage, from where they are automatically loaded during start up.

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

2.6.3 setenv - Set Environment Variables

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

To modify the U-Boot environment you must use **setenv**. When invoked with one arg, 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 reclaimed automatically

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

If invoked with multiple args, the first arg will be the name of the variable, and all following args will (concatenated by single-space characters) form the value that gets stored for this variable. New variables will be created automatically, and all existing variables will be overwritten.

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

Comply with standard shell-quoting rules if the value of a variable will contain characters that have a special meaning to the command line parser (such as the $ character, which 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 (such as using backslash for quoting, space, and tab characters to separate arguments, or semicolon and new line to separate commands). Even unusual input such as "=-/|()+=" is a legal variable name in U-Boot.

A common mistake is to write:

```
setenv name=value
```

instead of

```
setenv name value
```

This will not cause an error message, which might lead you to believe that the command executed correctly, when it did not. Instead of setting the variable name to the value *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 must be separated by a space and/or tab characters!

2.6.4 run - Run Commands in an Environment Variable

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

You can use U-Boot environment variables to store commands or sequences of commands. To do this, use the **run** command as follows:

```
=> 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 args, 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#=123456789abcdef
Done.
=>
```

If a U-Boot variable contains several commands (separated by semicolons), and one of these commands fails when you run the variable, the remaining commands will be executed regardless. If you execute several variables with one call to **run**, any failing command will cause **run** to terminate. Thus, the remaining variables will not be executed.

2.6.5 bootd - Boot Default (Run 'bootcmd')

```
=> help boot
bootd - No help available.
=>
```

bootd (short: **boot**) executes the default **boot** command, which is what happens if you do not interrupt the initial countdown. This is a synonym for the run **bootcmd** command.

2.7 Miscellaneous Commands

This section describes the miscellaneous commands.

2.7.1 echo - Echo Args to Console

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

echo echoes the args to the console.

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

2.7.2 reset - Perform RESET of the CPU

```
=> help reset
reset - No help available.
=>
```

reset reboots the system.

2.7.3 sleep - Delay Execution

```
=> help sleep
sleep N
- delay execution for N seconds (N is _decimal_ !!!)
=>
```

sleep pauses execution for the number of seconds specified as the arg.

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

2.7.4 version - Print Monitor Version

```
=> help version
version - No help available.
=>
```

version (short: **vers**) prints the version and build date of the U-Boot image running on your system.

```
=> version
U-Boot 1.1.2 (Jun 22 2005 - 08:06:19)
U_440SPe_V1R01 level01
=>
```

2.7.5 ? - Alias for 'help'

? can be used as a short form for the **help** command (see *help - Print Online Help* on page 14 for details).

2.8 U-Boot Environment Variables

This section describes U-Boot environment variables.

The U-Boot environment is associated to a block of memory that is kept in persistent storage and copied to RAM when U-Boot starts. It is used to store environment variables that can be used to configure the system. The environment is protected by a CRC32 checksum. This section lists the most important environment variables, some of which have a special meaning for U-Boot. You can use these variables to configure the behavior of U-Boot to your preferences.

- **autoload:** if set to no (or any string beginning with n), the *rarpb*, *bootp,* or *dhcp* commands will perform only a configuration lookup from the BOOTP / DHCP server, but not try to load any image using TFTP.
- **autostart:** if set to yes, an image loaded using the *rarpb*, *bootp*, *dhcp*, *tftp*, commands will be automatically started (by internally calling the *bootm* command).
- **baudrate:** a decimal number that selects the console baudrate (in bps). Only a predefined list of baudrate settings is available (9600, 19200, 38400, 57600, 115200). When you change the baudrate (using the *setenv* baudrate ... command), U-Boot will switch the baudrate of the console terminal and wait for a newline, which must be entered with the new speed setting. This is to ensure that you can actually type at the new speed. If this fails, you must reset the board (which will operate at the old speed since you were not able to saveenv the new settings.) If no baudrate variable is defined, the default baudrate of 115200 is used.
- **bootargs:** The contents of this variable are passed to the Linux kernel as boot arguments (also known as command line).
- **bootcmd:** This variable defines a command string that is automatically executed when the initial countdown is not interrupted. This command is only executed when the variable bootdelay is also defined!
- **bootfile:** name of the default image to load with TFTP
- **ethaddr:** Ethernet MAC address for Ethernet 0/only ethernet interface (= eth0 in Linux). This variable can be set only once (usually during manufacturing of the board). U-Boot refuses to delete or overwrite this variable once it has been set.
- **initrd_high:** used to restrict positioning of initrd ramdisk images. If this variable is not set, initrd images will be copied to the highest possible address in RAM; this is usually what you want since it allows for maximum initrd size. If for some reason you want to make sure that the initrd image is loaded below the CFG_BOOTMAPSZ limit, you can set this environment variable to a value of no, off, or 0. Alternatively, you can set it to a maximum upper address to use (U-Boot will still check that it does not overwrite the U-Boot stack and data). For instance, when you have a system with 16 MB RAM, and want to reserve 4 MB from use by Linux, you can do this by adding "mem=12M" to the value of the bootargs variable. However, now you must make sure that the initrd image is placed in the first 12 MB as well - this can be done with:

```
=> setenv initrd_high 00c00000
```

- **ipaddr:** IP address; needed for *tftp* command
- **loadaddr:** Default load address for commands such as **tftp***,* **loads**, or **echo**. If set to 1, all characters received during a serial download (using the loads command) are echoed back. This might be needed by some terminal emulations (such as cu), but may as well just use up time on others.
- **pram:** If the Protected RAM feature is enabled in your board's configuration, this variable can be defined to enable the reservation of such protected RAM, that is, RAM that is not overwritten by U-Boot. Define this variable to hold the number of KB you want to reserve for pRAM. Note that the board info structure will still show the full amount of RAM. If pRAM is reserved, a new environment variable, mem, will automatically be defined to hold the amount of remaining RAM in a form that can be passed as a boot argument to Linux, for instance, as follows:

```
=> setenv bootargs $(bootargs) mem=\$(mem)
=> saveenv
```

In this way you can direct Linux not to use this memory, which results in a memory region that will not be affected by reboots.

- **serverip:** TFTP server IP address, needed for **tftp** command.
- **serial#:** Hardware identification information such as type string and/or serial number. This variable can be set only once (usually during manufacturing of the board). U-Boot will not delete or overwrite this variable once it has been set.
- **silent:** If the configuration option CONFIG_SILENT_CONSOLE has been enabled for your board, setting this variable to any value will suppress all console messages. See README.silent for details (doc directory, under U-Boot directory)
- **verify:** If set to n or no, disables the checksum calculation for the complete image in the **bootm** command to trade speed for safety in the boot process. Note that the header checksum is still verified.

The following environment variables may be used and automatically updated by the network boot commands (**bootp**, **dhcp**, or **tftp**), depending upon the information provided by your boot server.

- **bootfile:** See the **bootfile** description in this section
- **dnsip:** IP address of your Domain Name Server
- **gatewayip:** IP address of the Gateway (Router) to use
- **hostname:** Target hostname
- **ipaddr:** See the **ipaddr** description in this section
- **netmask:** Subnet Mask
- **rootpath:** Pathname of the root filesystem on the NFS server
- **serverip:** See the **serverip** description in this section
- **filesize:** Size (as a hexadecimal number of bytes) of the file downloaded using the last **bootp**, **dhcp**, or **tftp** command.

2.9 U-Boot Scripting Capabilities

This section describes the U-Boot scripting capabilities.

2.9.1 U-Boot Standalone Applications

U-Boot enables dynamically loading and running standalone applications, which can use some U-Boot resources, such as console I/O functions, memory allocation, or interrupt services. Several simple examples are included with the U-Boot source code, for example:

Hello World Demo:

The file examples/hello_world.c contains a small Hello World Demo application. It is automatically compiled when you build U-Boot. It is configured to run at address 0x00040004, so you can experiment with it as follows:

```
=> loads
## Ready for S-Record download ...
~>examples/hello_world.srec
1 2 3 4 5 6 7 8 9 10 11 ...
[file transfer complete]
[connected]
## Start Addr = 0x00040004
=> go 40004 Hello World! This is a test.
## Starting application at 0x00040004 ...
Hello World
argc = 7
argv[0] = "40004"
argv[1] = "Hello"
argv[2] = "World!"
argv[3] = "This"
argv[4] = "is"
argv[5] = "a"
argv[6] = "test."
argv[7] = ""
Hit any key to exit ...
## Application terminated, rc = 0x0
```

Alternatively, you can use TFTP to download the image over the network. In this case, the binary image (hello_world.bin) is used. Note that the entry point of the program is at offset 0x0004 from the start of the file, that is, the download address and the entry point address differ by four bytes.

```
=> tftp 40000 /tftpboot/hello_world.bin
...
=> go 40004 This is another test.
## Starting application at 0x00040004 ...
Hello World
argc = 5
argv[0] = "40004"
argv[1] = "This"
argv[2] = "is"
```

```
argv[3] = "another"
argv[4] = "test."
argv[5] = ""
Hit any key to exit ...
## Application terminated, rc = 0x0
```

2.10 Burning a New U-Boot Image in Flash

The AMCC 440SPe-MX evaluation board can be updated by reprogramming the flash memory with a new U-Boot kernel. U-Boot offers several commands for programming, erasing, and protecting the flash memory. To see what type of flash memory your board has, enter the **flinfo** command.

Warning: Do not reset or power off the board after the **erase flash** command. If this happens, the only way to recover is to use a JTAG debugger to reprogram the flash.

```
=> fli
Bank # 1: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF00000 FFF10000 FFF20000 FFF30000 FFF40000
FFF50000 FFF60000 E FFF70000 E
Bank # 2: AMD AM29F040 (512 Kbit, uniform sector size)
Size: 512 KB in 8 Sectors
Sector Start Addresses:
FFF80000 E FFF90000 E FFFA0000 E FFFB0000 E FFFC0000 E
FFFD0000 E FFFE0000 E FFFF0000
=>
```

The output provides a lot of information. Immediately you see the flash manufacturer, part number, and sector layout. Next you must prepare the flash sectors for programming by erasing them. Enter `erase 1:1- 4`, which tells U-Boot to erase sectors 1 through 4 of flash bank 1. This preserves the environment variables located in flash 1 sector 0. Then enter `erase 2:0-7` for the other flash bank.

```
=>erase 1:1-7
Erase Flash Sectors 1-7 in Bank # 1
.Erasing sector fff10000
.Erasing sector fff20000
.Erasing sector fff30000
.Erasing sector fff40000
.Erasing sector fff50000
.Erasing sector fff60000
.Erasing sector fff70000
. done
=> erase 2:0-7
Erase Flash Sectors 0-7 in Bank # 2
.Erasing sector fff80000
.Erasing sector fff90000
.Erasing sector fffa0000
.Erasing sector fffb0000
.Erasing sector fffc0000
.Erasing sector fffd0000
.Erasing sector fffe0000
.Erasing sector ffff0000
. done
=>
```

U-Boot supports TFTP (Trivial FTP) (a stripped-down FTP that does not require user authentication) for downloading images into the board's RAM. The **tftp** command needs two pieces of information: the name of the file to download, and where in memory to store the file, as shown in the following example:

```
=> tftp 8000 d:/cygwin/opt/ppc440SP_kit/u-boot-440SPe/u-boot.bin
About preceding transfer (eth0):
- Sent packet number 0
- Received packet number 0
- Handled packet number 0
Waiting for PHY auto negotiation to complete. done
ENET Speed is 100 Mbps - HALF duplex connection
Using ppc_440x_eth0 device
TFTP from server 192.168.2.21; our IP address is 192.168.2.22
Filename 'd:/cygwin/opt/ppc440SPe_kit/u-boot-440SPe/u-boot.bin'.
Load address: 0x8000
Loading: #################################################################
#################################################################
#################################################################
done
Bytes transferred = 983040 (f0000 hex)
=>
```

The size and location of the downloaded image are stored in the **fileaddr** and **filesize** environment variables for possible later use by other shell commands and scripts. U-Boot also supports NFS so you can download images from your existing NFS server. To program the flash memory you must copy the image from RAM to the address of flash sector 0, 0xfff10000, using the **cp** command. You will use the byte version of the command (**cp.b**) to copy the specified number of bytes. In this case you can use the **fileaddr** and **filesize** environment variables, which contain the RAM address and number of bytes loaded by the last TFTP command. Type:

```
cp.b ${fileaddr} fff10000 ${filesize}
```

at the U-Boot prompt.

```
=> cp.b ${fileaddr} fff10000 ${filesize}
Copy to Flash... ................ done
```

The U-Boot code for your board is now updated The next boot will run the newly uploaded U-Boot code. The final flash-related command is **saveenv**, which saves your current environment variables to a reserved flash sector. This enables your environment variables to persist across power cycles and reboots. You might want do this after updating the server IP address or when adding a new script. You can use U-Boot environment variables to store commands and even sequences of commands. To create such a command you must create a new environment variable. For example, to update the flash in one operation:

```
=> setenv updateFlash 'erase 1:1-7;erase 2:0-7;tftp 8000 d:/cyg-
win/opt/ppc440SPe_kit/uboot-
440SPe/u-boot.bin;cp.b ${fileaddr} fff10000 ${filesize}'
```

Then save the newly created variable:

```
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
Erasing sector fff00000
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=>
```

To update U-Boot in flash at a later time, only one command needs to be executed:

```
=> run updateFlash
```

Revision Log

Revision Date	Level	Contents of Modification
07/06/2005	1.01	Initial document creation using AS uboot information. GJG

Index