**U-Boot Environment Variables**

Sachin Verma

Sept. 14, 2016 in [uboot](https://www.vermasachin.com/tags/uboot/)

A close look at the anatomy of the U-Boot environment.

Das U-Boot is a popular bootloader amongst embedded systems. This wide adoption of U-Boot is hardly surprising given the number of architectures and platforms supported by it. U-Boot has a flexible compile-time configuration setup. You can select different features and drivers via config options and build a custom bootloader image for your platform. U-Boot flexibility is extended at runtime as well. Using U-Boot environment variables you can influence the program execution flow.

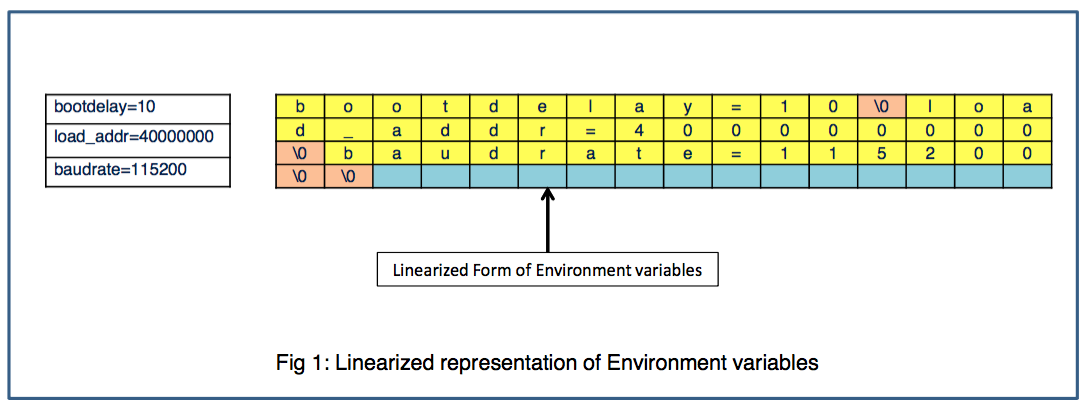
U-Boot comes with a **CLI** (*command line interpreter*) having basic scripting capabilities. This scripting ability combined with U-Boot environment variables can be used to create some powerful booting scenarios. This ability to manipulate program behavior using environment variables is beneficial for both development and production setups alike. During development we strive to test all possible paths for loading and booting images for our platform. So, you may try to load linux kernel image from a local storage (flash, SDcard, USB or eMMC etc), or access it over network (NFS, TFTP, etc.). U-Boot just makes your life easier as a developer. You just need to tweak the scripts combining environment variables in a fruitful way. Production images also need some versatility, when product OS images need upgrade, the bootloader must be configurable to fetch the images from different sources.

U-Boot has a number of system variables that you can modify to achieve the desired result. For example, on certain systems initrd images loaded at *top of DDR* may not be accessible to Linux kernel. To counter this, you can instruct U-Boot to load initrd at a *lower DDR address*. You can do this by setting the environment variable **initrd\_high**. Another common example during development is the presence of different network configurations. At your home setup you may be working on a *static IP configuration* using nfs. But, when you are out for a demo at a client location you only have a DHCP available with images kept on tftp server. U-Boot is highly configurable for such scenarios as it gives you so many options. You can change network configuration, modify IP addresses of image servers, gateway server with the help of environment variables.

You could assign console over serial port or you may use **netconsole** or **usbtty** if you may wish.

**So, What is this U-Boot environment?**

A simple answer to this question would be: "A collection of **name=value** pairs". Here, **name** refers to the name of the *environment variable* to which we wish to assign some **value**. This "value" could be of any type: string, hexadecimal, boolean etc. *Whatever be the type of the value, it is converted into string before being stored in a linearized environment data block.* Each environment variable pair ("name=value") would be stored as a **null terminated string**. So, the collection of many environment variables is nothing but *a null separated list with****double null terminator***. Figure 1 illustrates how the list of strings is actually stored. Left hand side just a logical representation of environment variables. Whereas if you see the right hand you can see that the variables have been flattened and written in a serialized form.



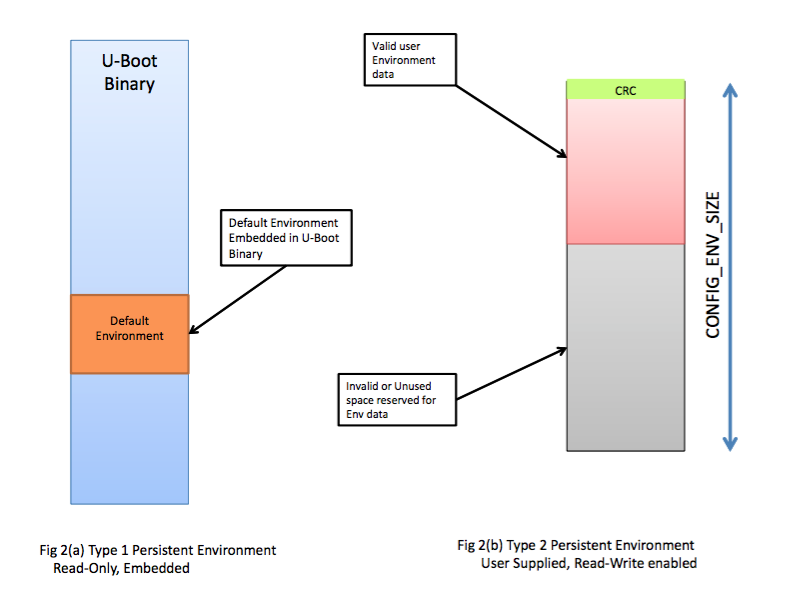
**How is the environment stored?**

U-Boot has 2 types of persistent environments

1. Default Environment (Compiled-in, Read-Only)
2. User Supplied Environment (Flashed in external storage, writable)

**Default Environment (Compiled-in, Read-Only)**

Every U-Boot binary has a default built-in environment of its own (See Fig 2(a)). During compilation, a character array called \_default *environment* is embedded in to the U-Boot image. This character array stores the environment variables as a list of null terminated strings with a double null terminator. The contents of this array are populated conditionally based on the config options selected for your board. Environment variables which are commonly used can be enabled by defining the corresponding CONFIGS's in your board config file (include/configs/<YOUR\_BOARD>.h). Fig 3 lists some commonly used options which once defined would make their way into default environment of your board.

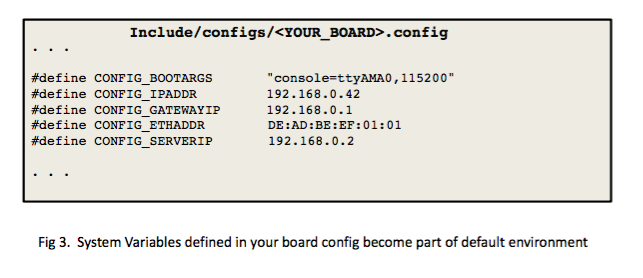


Apart from *the standard variables* used across boards you may wish to add certain environment variables specific to your board or convenient to you. You may for instance want to embed the revision number of the board in to this environment. You could do this by defining all these variables in a macro called CONFIG\_EXTRA\_ENV\_SETTINGS in your board config file.

#define CONFIG\_EXTRA\_ENV\_SETTINGS \

"board=" XSTR(BOARD) "\0" **\**

"load\_addr=" XSTR(CONFIG\_SYS\_LOAD\_ADDR) "\0"



You must remember that default environment is "**read-only**" as it is part of U-Boot image itself. Vendors normally keep some essential system variables as part of this environment.

There are good reasons to keep a default environment as part of the image:

1. Since it is read-only, you always have a default state to revert to.
2. During early bootup, user supplied environment (defined next) may be inaccessible or must not be used because of security concerns.
3. User Supplied Environment may be inaccessible due to:
   * Storage device malfunction.
   * Environment data corruption.

You must not keep too much data in to this default environment as it directly adds to the weight of the binary. Only keep critical system variables in this environment.

**User Supplied Environment (Flashed in external storage, writable)**

Typically, vendors would flash an environment data image to an external storage present on your board. The format of this prebuilt environment is again same i.e. linearized list of strings, but there is a 4 byte CRC header prefixed to it. This CRC is computed over the environment data. Figure 2(b) illustrates such an environment blob with a CRC data, followed by valid environment data and invalid one after that. Total size of this environment data is fixed to CONFIG\_ENV\_SIZE during compilation. So, if your environment usage exceeds this size you would need to recompile your U-Boot binary after increasing CONFIG\_ENV\_SIZE. If you do not increase the size, U-Boot would refuse to save the environment variables.

You may decide to keep this environment in any of the external storage, but you must configure the board config accordingly. U-Boot must know which storage and at what offset would be used to keep the user environment. U-Boot provides a number of options to configure location of Environment data. U-Boot has infrastructure to access environment stored in *serial flashes, NVRAM, NAND, dataflash, MMC and even UBI volumes*. You can obtain more information on how to use these CONFIG options from the U-Boot documentation. Since default environment size has to be minimized, most of the environment variables are stored here. Certain storage technologies like Raw NAND flashes are inherently unreliable. To combat such possibilities (including power failure) and for robustness in general, you can also configure a **Redundant user environment**. You can configure the location and size of this duplicate environment data as well in your board config.

**Saving Environment**

Out of the 2 default environments (default and User) only User is Writable. So, whenever you modify a variable and issue a saveenv command, that variable would end up in User environment. When you do saveenv, U-Boot would:

* sort the list of current environment variables
* convert them to a linearized list of strings
* compute CRC over this data and burn the environment back at its fixed location in storage.

**Creating a Pre-Built User Environment**

U-Boot provides a utility named mkenvimage which can be used to generate an environment blob suitable to be flashed.

mkenvimage needs at least 2 inputs to create the blob:

* Environment variables in a text file (only one env "name=value" string on each line)
* Size of the environment blob in bytes (Remember, this must comply with the CONFIG\_ENV\_SIZE you have defined in your board config).

For example, if my env data file is called my\_env\_data.txt, and sizeof desired env blob is 16384 (16 KiB), then i would use the following command:

$./tools/mkenvimage -s **16384** -o env\_blob my\_env\_data.txt

you can see the dump of the environment blob using the od command

$ od -t x1c env\_blob

**0000000** 0d d2 **49** **96** **62** **61** **75** **64** **72** **61** **74** **65** 3d **31** **31** **35**

**\r** **322** I **226** b a u d r a t e = **1** **1** **5**

**0000020** **32** **30** **30** **00** **62** 6f 6f **74** **64** **65** 6c **61** **79** 3d **31** **30**

**2** **0** **0** **\0** b o o t d e l a y = **1** **0**

**0000040** **00** 6c 6f **61** **64** 5f **61** **64** **64** **72** 3d **30** **78** **34** **30** **30**

**\0** l o a d \_ a d d r = **0** x **4** **0** **0**

**0000060** **30** **30** **30** **30** **30** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00**

**0** **0** **0** **0** **0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0**

**0000100** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00** **00**

**\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0** **\0**

\*

**0040000**

This environment data blob must be flashed at the predefined offset in the storage device. You can use U-Boot, Linux or any other flasher to burn this blob.

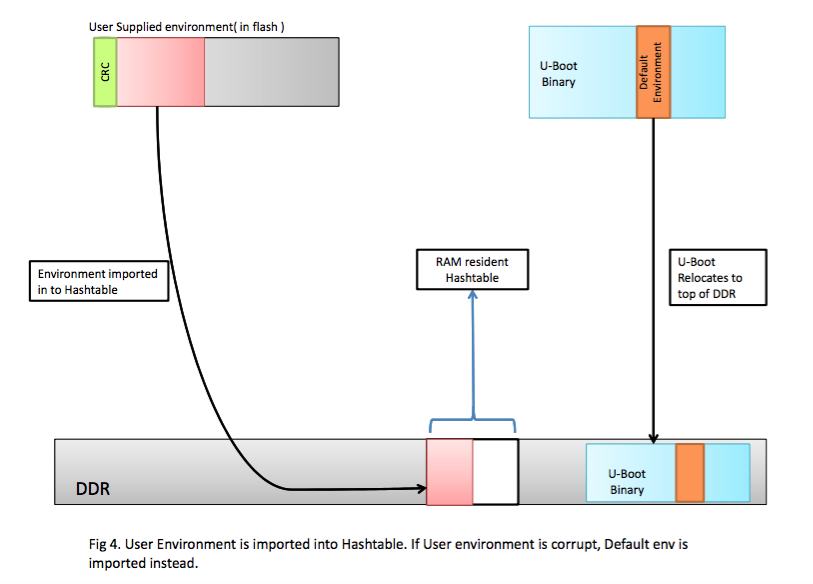
**Relocation of Environment data to RAM**

During early boot when U-Boot has not relocated to RAM, it uses the linearized form of environment data (as shown in Fig 1). But once, U-boot has relocated to RAM this linearized form is no longer used. Instead, U-Boot imports all such env data stored from persistent storage into a **RAM resident Hashtable**. If User supplied environment is good (i.e. CRC is OK), then it is imported from flash to RAM. Otherwise, U-Boot imports the default compiled-in environment to this Hashtable. Fig 4 shows how User Environment is imported in to Hashtable, whereas U-Boot along with its default environment relocates to top of RAM. In case User Environment is corrupt or inaccessible, U-boot would import the default environment in to Hashtable.

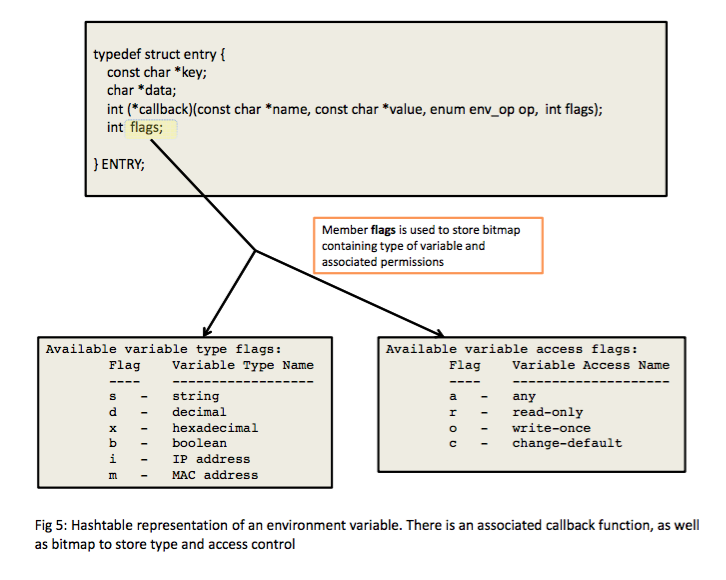
The use of Ram resident data structure (Hashtable) is important for various reasons:

1. It boosts performance as we are manipulating variables in RAM and not in flash.
2. You have to manipulate data only in RAM and need not access some slow flash driver (and deal with the associated complexity).
3. It allows U-Boot to deploy type checks and access control attributes on different variables while still keeping the persistent storage form simple (linear list).

Once environment has relocated to ram (into the Hashtable), all commands operating on environment variables would only be working on this Hashtable. U-boot would not touch the environment variables stored in the persistent storage at all (unless it needs to save the env).



Each environment variable entry inside the Hashtable is represented by a data structure called "struct entry" (See Fig 5). Apart from the members "key" and "data" which correspond to "name" and "value" in linearized representation of data, we have members called "callback" and "flags". "flags" is of integer type and is used to implement type check and access control. "callbacks" is the callback function associated with this environment variable. If defined, this callback handler would be invoked whenever any operation (like add, delete or modify) is performed on this environment variable.

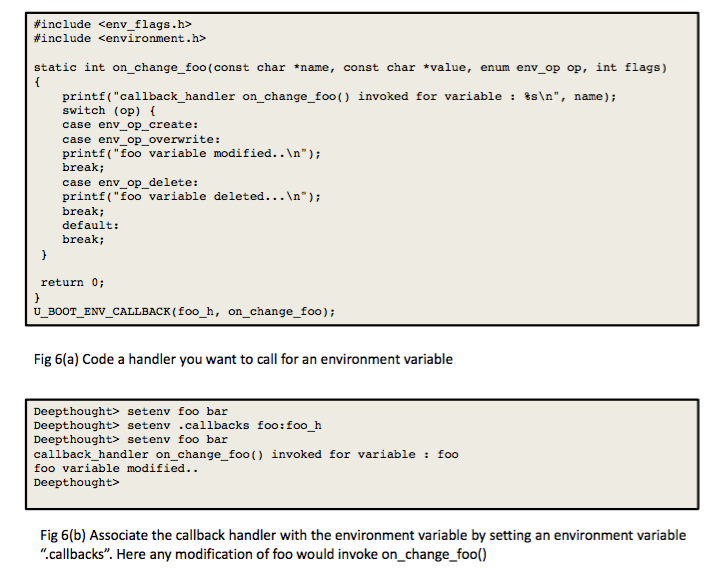


**How to Control/React to environment variable modification?**

At times you may want to do your own runtime configurations. You may want to react (accept, reject, or produce some side effects) to the changes done to some environment variable. For such use cases, U-Boot provides a mechanism of deploying callback handlers. You can associate a callback function with an environment variable. As a first step, you have to register a callback handler with U-Boot. This function would be called whenever you do any modifications to the environment variable. Fig 6(a), shows a sample code to register a callback with U-Boot. You can place such a handler code in your board specific file. The macro U\_BOOT\_ENV\_CALLBACK registers the callback function on\_change\_foo with the handler named foo\_h. Your handler is now registered with U-Boot with the name foo.

Now, you need to establish a link of this registered handler with an environment variable. If you have a look at the "struct entry" in Fig 5(a), you can see that there is a member named "callback" (a function pointer) for each environment variable. U-Boot would invoke this handler before committing modified environment variable to the Hashtable. You can make this association of callback handler with env variable either at compile time or at runtime. For doing compile-time association you need to define the config option "#define CONFIG\_ENV\_CALLBACK\_LIST\_DEFAULT foo:foo\_h" in your board config file. You can also do runtime association as depicted in Fig 6(b). Here I have created a new environment variable .callbacks, which is a standard U-Boot system variable to make such associations. I have deployed a handler named "foo\_h" for the environment variable "foo". Once this registration is done, whenever you do some modification to variable "foo", function "on\_change\_foo()" would be invoked. You can now deliver your reaction to different types of actions (env\_op\_create, env\_op\_overwrite, env\_op\_delete).

U-Boot already deploys similar handlers for managing console changes, splash image etc.



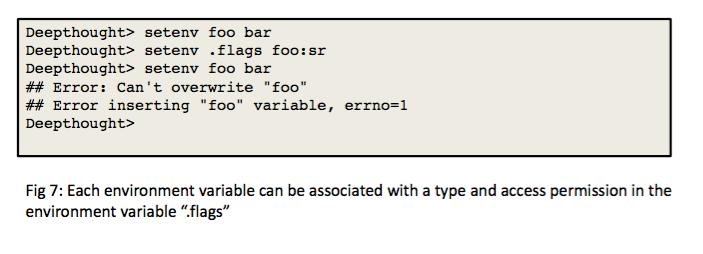
**Type and Access Control of Environment Variables**

There are certain environment variables which you want to use but do not want to do casual modifications. For example you have an environment variable "serial#", you definitely want this variable to be read-only. You want automatic rejection of any attempt to change it. Another such example is the MAC address of the device. Again you want to keep this variable as read-only or at worst write-once. U-Boot supports different access modifiers "any", "read-only", "write-once", and "change-default" (see Fig 5). U-Boot Hashtable representation of env variables has a member "int flags" (see fig 5). Member "flags" is used to keep a bitmap specifying the access permission associated with variable. So, whenever any modification attempt is done on variable, it must comply with the access permission, otherwise U-Boot would reject the changes.

Another problem faced by users is the basic sanity check of *environment variable type*. Since linearized form of environment only keeps strings, U-boot needs to make sure that it can do some kind of type checking before assigning value to a variable. To address this issue U-Boot makes use of some predefined types like "string", "decimal", "hexadecimal", "boolean", "IP Address", "MAC address". There are corresponding codes for these type modifiers: "s", "d", "x", "b", "i", "m". Again, U-Boot stores this type information of variable in "flags" as a bitmap as shown in Fig 5.

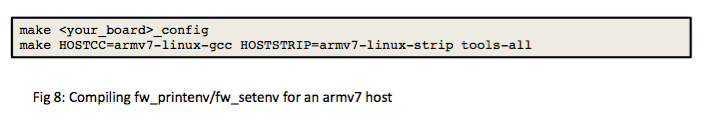
You can associate type and access control to a variable either at **compile-time or at run-time**. For compile-time association you need to define a config #define CONFIG\_ENV\_FLAGS\_LIST\_DEFAULT foo:sr in your board file. For runtime association, you can define an environment variable .flags as fig 7. Here, I am associating an environment variable foo with type s (meaning value is a string) and access control r (meaning it is read-only). Once deployed, if you try to modify variable "foo", U-Boot would reject your request. Also, if value is not of specified "type", your update to env variable would fail.

Environment variables like MAC address make use of type m. This would make U-Boot do a sanity check on the value entered by user, to confirm if the value is indeed a valid MAC address.



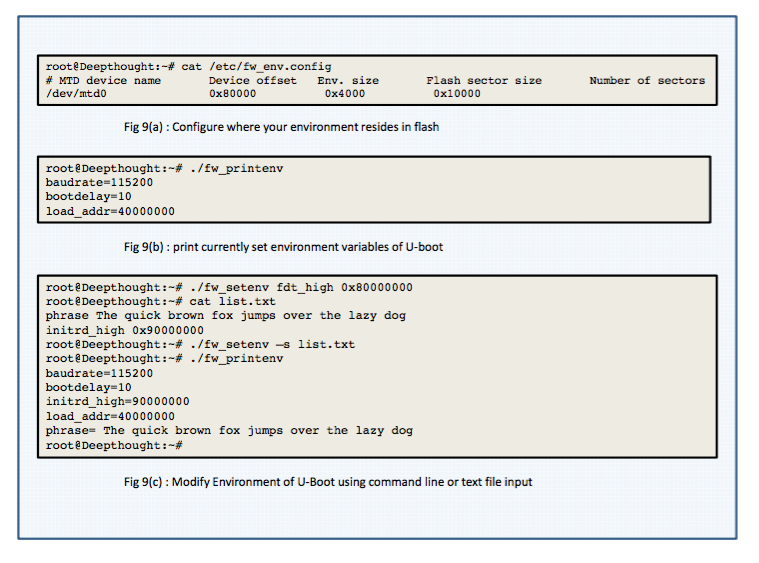
**Modifying U-Boot "User Environment" from Linux**

U-Boot environment variables can be added, modified or deleted from Linux as well. U-boot provides a set of utilities called fw\_printenv and fw\_setenv to do the job. First you need to compile these utilities for Linux. Fig 8 shows the compilation steps for the utility. Here, I am cross compiling it for an ARM platform. It is a multicall binary. So, you need to make a symlink named fw\_setenv to binary fw\_printenv.



To modify the environment, you need to first boot into linux on the target board. Now, you need to create a file /etc/fw\_env.config. This file contains all the information needed to specify the location of the Environment data blob. Fig 9 shows my configuration file. I kept my environment in a SPI flash which appeared as /dev/mtd0 to my kernel. My Environment blob was configured at an offset of 0x80000 from beginning of flash and had a size of 0x40000. The size of each sector of my flash is 0x10000. This is all the information I needed to provide in order for environment manipulation utilities to work. As soon as I keyed in the command fw\_printenv, I could see the variables that I saved in U-Boot user environment appearing on my console. You can also set the environment variables using fw\_setenv. As shown in Fig 9, I make use of a text file (list.txt) containing the variables I want to set. The format is simple, the first *whitespace* after a name acts as a delimiter, the characters till the end of line thereafter are considered value for the key.

You can verify that the variables have been set by executing fw\_printenv. These variables would now be visible from U-Boot as well.



**Restoring Default Environment**

Sometimes, after a lot of environment variable changes, you can corrupt the state. To restore sanity and get the original values of default environment you can make use of env command.

env default [-f] var [...]

Above command would forcibly reset the specified variables to a value from the default environment.

To restore complete environment from default, you may invoke the following command:

* env default -a
* env command is very powerful, you can use it import/export environment data from/to RAM.

**Final Comments**

U-Boot environment can act as a very useful runtime configuration tool. When combined with scripting it can make arduous task of development and testing of boot scenarios much simpler and fun to do.

# U-Boot Environment Variables

The U-Boot environment is a block of memory that is kept on persistent storage and copied to RAM when U-Boot starts. It is used to store environment variables which 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 to U-Boot. You can use these variables to configure the behaviour of U-Boot to your liking.

* **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](https://www.denx.de/wiki/DULG/BOOTP) / [DHCP](https://www.denx.de/wiki/DULG/DHCP) server, but not try to load any image using [TFTP](https://www.denx.de/wiki/DULG/TFTP).
* **autostart**: if set to "yes", an image loaded using the rarpb, bootp, dhcp, tftp, disk, or docb 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.  
  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 make sure you can actually type at the new speed. If this fails, you have to 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 (aka "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!
* **bootdelay**: After reset, U-Boot will wait this number of seconds before it executes the contents of the bootcmd variable. During this time a countdown is printed, which can be interrupted by pressing any key.  
  Set this variable to 0 boot without delay. Be careful: depending on the contents of your bootcmd variable, this can prevent you from entering interactive commands again forever!  
  Set this variable to -1 to disable autoboot. Set this variable to -2 to boot without delay and not check for abort.
* **bootfile**: name of the default image to load with [TFTP](https://www.denx.de/wiki/DULG/TFTP)
* **cpuclk**: (Only with MPC859 / MPC866 / MPC885 processors) On some processors, the [CPU](https://www.denx.de/wiki/DULG/CPU) clock frequency can be adjusted by the user (for example to optimize performance versus power dissipation). On such systems the cpuclk variable can be set to the desired [CPU](https://www.denx.de/wiki/DULG/CPU) clock value, in MHz. If the cpuclk variable exists and its value is within the compile-time defined limits (CONFIG\_SYS\_8xx\_CPUCLK\_MIN and CONFIG\_SYS\_8xx\_CPUCLK\_MAX = minimum resp. maximum allowed [CPU](https://www.denx.de/wiki/DULG/CPU) clock), then the specified value is used. Otherwise, the default [CPU](https://www.denx.de/wiki/DULG/CPU) clock value of CONFIG\_8xx\_CPUCLK\_DEFAULT is used.
* **ethaddr**: Ethernet [MAC](https://www.denx.de/wiki/DULG/MAC) address for first/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.
* **eth1addr**: Ethernet [MAC](https://www.denx.de/wiki/DULG/MAC) address for second ethernet interface (= eth1 in Linux).
* **eth2addr**: Ethernet [MAC](https://www.denx.de/wiki/DULG/MAC) address for third ethernet interface (= eth2 in Linux).  
  ...
* **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" or "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

Setting initrd\_high to the highest possible address in your system (0xFFFFFFFF) prevents U-Boot from copying the image to RAM at all. This allows for faster boot times, but requires a Linux kernel with zero-copy ramdisk support.

* **ipaddr**: IP address; needed for tftp command
* **loadaddr**: Default load address for commands like tftp or loads.
* **loads\_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 (like cu), but may as well just take time on others.
* **mtdparts**: This variable (usually defined using the [mtdparts](https://www.denx.de/wiki/DULG/UBootCmdGroupFlash" \l "UBootCmdFlMtdparts) command) allows to share a common [MTD](https://www.denx.de/wiki/DULG/MTD) partition scheme between U-Boot and the Linux kernel.
* **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", i. e. RAM which 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 boot argument to Linux, for instance like that:

=> setenv bootargs ${bootargs} mem=\${mem}

=> saveenv

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

* **serverip**: [TFTP](https://www.denx.de/wiki/DULG/TFTP) server IP address; needed for tftp command.
* **serial#**: contains 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 refuses to delete or overwrite this variable once it hass 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. Please see doc/README.silent for details.
* **verify**: If set to n or no disables the checksum calculation over 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 the information provided by your boot server:

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

# Basic Command Set

* **Information Commands**
  + bdinfo - print Board Info structure
  + coninfo - print console devices and informations
  + flinfo - print FLASH memory information
  + iminfo - print header information for application image
  + imls - list all images found in flash
  + help - print online help
* **Memory Commands**
  + base - print or set address offset
  + crc32 - checksum calculation
  + cmp - memory compare
  + cp - memory copy
  + md - memory display
  + mm - memory modify (auto-incrementing)
  + mtest - simple RAM test
  + mw - memory write (fill)
  + nm - memory modify (constant address)
  + loop - infinite loop on address range
* **Flash Memory Commands**
  + cp - memory copy (program flash)
  + flinfo - print FLASH memory information
  + erase - erase FLASH memory
  + protect - enable or disable FLASH write protection
* **Execution Control Commands**
  + autoscr - run script from memory
  + bootm - boot application image from memory
  + bootelf - Boot from an ELF image in memory
  + bootvx - Boot vxWorks from an ELF image
  + go - start application at address 'addr'
* **Network Commands**
  + bootp - boot image via network using BOOTP/TFTP protocol
  + cdp - Perform Cisco Discovery Protocol network configuration
  + dhcp - invoke DHCP client to obtain IP/boot params
  + loadb - load binary file over serial line (kermit mode)
  + loads - load S-Record file over serial line
  + nfs - boot image via network using NFS protocol
  + ping - send ICMP ECHO\_REQUEST to network host
  + rarpboot- boot image via network using RARP/TFTP protocol
  + tftpboot- boot image via network using TFTP protocol
* **Environment Variables Commands**
  + printenv- print environment variables
  + saveenv - save environment variables to persistent storage
  + askenv - get environment variables from stdin
  + setenv - set environment variables
  + run - run commands in an environment variable
  + bootd - boot default, i.e., run 'bootcmd'
* **Filesystem Support** (FAT, cramfs, JFFS2, Reiser)
  + chpart - change active partition
  + fsinfo - print information about filesystems
  + fsload - load binary file from a filesystem image
  + ls - list files in a directory (default /)
  + fatinfo - print information about filesystem
  + fatls - list files in a directory (default /)
  + fatload - load binary file from a dos filesystem
  + nand - NAND flash sub-system
  + reiserls- list files in a directory (default /)
  + reiserload- load binary file from a Reiser filesystem
* **Special Commands**
  + i2c - I2C sub-system
  + doc - Disk-On-Chip sub-system
  + dtt - Digital Thermometer and Themostat
  + eeprom - EEPROM sub-syste
  + fpga - FPGA sub-system
  + ide - IDE sub-system
  + kgdb - enter gdb remote debug mode
  + diskboot- boot from IDE device
  + icache - enable or disable instruction cache
  + dcache - enable or disable data cache
  + diag - perform board diagnostics (**POST** code)
  + log - manipulate logbuffer
  + pci - list and access PCI Configuraton Space
  + regdump - register dump commands
  + usb - USB sub-system
  + sspi - SPI utility commands
* **Miscellaneous Commands**
  + bmp - manipulate BMP image data
  + date - get/set/reset date & time
  + echo - echo args to console
  + exit - exit script
  + kbd - read keyboard status
  + in - read data from an IO port
  + out - write datum to IO port
  + reset - Perform RESET of the CPU
  + sleep - delay execution for some time
  + test - minimal test like /bin/sh
  + version - print monitor version
  + wd - check and set watchdog
  + ? - alias for 'help'