U-Boot BeagleBoneBlack Documentation

Abstract

About this Document

The documentation is written in reStructuredText and converted into a pdf document. Some parts of this document are created automatically out of the log files from the tbot build process.

Introduction

This document describes how to use the firmware U-Boot and the operating system Linux in Embedded Power Architecture®, ARM and MIPS Systems.

There are many steps along the way, and it is nearly impossible to cover them all in depth, but we will try to provide all necessary information to get an embedded system running from scratch. This includes all the tools you will probably need to configure, build and run U-Boot and Linux.

First, we describe how to install the Cross Development Tools Embedded Linux Development Kit which you probably need - at least when you use a standard x86 PC running Linux or a Sun Solaris 2.6 system as build environment.

Then we describe what needs to be done to connect to the serial console port of your target: you will have to configure a terminal emulation program like cu or kermit.

In most cases you will want to load images into your target using ethernet; for this purpose you need TFTP and DHCP / BOOTP servers. A short description of their configuration is given.

A description follows of what needs to be done to configure and build the U-Boot for a specific board, and how to install it and get it working on that board.

The configuration, building and installing of Linux in an embedded configuration is the next step. We use SELF, our Simple Embedded Linux Framework, to demonstrate how to set up both a development system (with the root filesystem mounted over NFS) and an embedded target configuration (running from a ramdisk image based on busybox).

This document does not describe what needs to be done to port U-Boot or Linux to a new hardware platform. Instead, it is silently assumed that your board is already supported by U-Boot and Linux.

Disclaimer

Use the information in this document at your own risk. DENX 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.

U-Boot

Current Versions

Das U-Boot (or just "U-Boot" for short) is Open Source Firmware for Embedded Power Architecture®, ARM, MIPS, x86 and other processors. The U-Boot project is hosted by DENX, where you can also find the project home page: http://www.denx.de/wiki/U-Boot/

The current version of the U-Boot source code can be retrieved from the DENX "git" repository.

You can browse the "git" repositories at http://git.denx.de/

The trees can be accessed through the git, HTTP, and rsync protocols. For example you can use one of the following commands to create a local clone of one of the source trees:

```
git clone git://git.denx.de/u-boot.git u-boot/
git clone http://git.denx.de/u-boot.git u-boot/
git clone rsync://git.denx.de/u-boot.git u-boot/
```

For details please see here.

Official releases of U-Boot are also available through FTP. Compressed tar archives can downloaded from the directory ftp://ftp.denx.de/pub/u-boot/.

U-Boot Command Line Interface

The following section describes the most important commands available in U-Boot. Please note that U-Boot is highly configurable, so not all of these commands may be available in the configuration of U-Boot installed on your hardware, or additional commands may exist. You can use the help command to print a list of all available commands for your configuration.

For most commands, you do not need to type in the full command name; instead it is sufficient to type a few characters. For instance, help can be abbreviated as h.

The behaviour of some commands depends on the configuration of U-Boot and on the definition of some variables in your U-Boot environment.

Almost all U-Boot commands expect numbers to be entered in hexadecimal input format. (Exception: for historical reasons, the sleep command takes its argument in decimal input format.)

Be careful not to use edit keys besides 'Backspace', as hidden characters in things like environment variables can be very difficult to find.

Information Commands

bdinfo - print Board Info structure

```
=> help bdinfo
bdinfo - print Board Info structure

Usage:
bdinfo
=>
```

The bdinfo command (bdi) prints the information that U-Boot passes about the board such as memory addresses and sizes, clock frequencies, MAC address, etc. This information is mainly needed to be passed to the Linux kernel.

```
=> bdi
arch_number = 0x00000E05
boot_params = 0x80000100
DRAM bank   = 0x00000000
-> start    = 0x80000000
-> size     = 0x20000000
baudrate    = 115200 bps
TLB addr    = 0x9FFF0000
relocaddr   = 0x9FF4F000
reloc off   = 0x1F74F000
irq_sp      = 0x9DF264E0
sp start    = 0x9DF264D0
Early malloc usage: 178 / 400
fdt_blob = 9df264f8
=>
```

coninfo - print console devices and informations

```
=> help conin
coninfo - print console devices and information

Usage:
coninfo
=>
```

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

```
=> conin
List of available devices:
serial@44e09000 00000003 IO stdin stdout stderr
serial   00000003 IO
=>
```

The output contains the device name, flags, and the current usage. For example, the output

```
serial@44e09000 00000003 IO stdin stdout stderr
```

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

flinfo - print FLASH memory information

```
=> help flinfo
Unknown command 'flinfo' - try 'help' without arguments for list of all known commands

=>
```

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

```
=> flinfo
Unknown command 'flinfo' - try 'help'
=>
```

help - print online help

```
=> help help
help - print command description/usage

Usage:
help
      - print brief description of all commands
help command ...
      - print detailed usage of 'command'
=>
```

The help command (h or ?) prints online help. Without any arguments, 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 argument to the help command:

```
=> help printenv tftp
printenv - print environment variables

Usage:
printenv [-a]
    - print [all] values of all environment variables
printenv name ...
    - print value of environment variable 'name'
tftpboot - boot image via network using TFTP protocol

Usage:
tftpboot [loadAddress] [[hostIPaddr:]bootfilename]
=>
```

Memory Commands

base - print or set address offset

```
=> help base
base - print or set address offset

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

You can use the base command (ba) to print or set a "base address" that is used as the address offset for all subsequent 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 Power Architecture® processors) it can be very convenient to set the base address to the start of this area and then use only the offsets:

ToDo

crc32 - checksum calculation

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

```
=> crc 0x80000004 0x3fc 76a37280
=>
```

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

```
=> crc 0x80000004 0x3fc 0x80000000 76a37280
=> md 0x80000000 4
80000000: 8072a376 ff551155 ffffffff 1155ffff    v.r.U.U.......U.
=>
```

As you can see, the CRC32 checksum was not only printed, but also stored at address passed in the 3th argument.

cmp - memory compare

```
=> help cmp
cmp - memory compare

Usage:
cmp [.b, .w, .l] addr1 addr2 count
=>
```

With the cmp command you can test whether the contents of two memory areas are identical or not. The command will test either the whole area as specified by the 3rd (length) argument, or stop at the first difference.

```
=> cmp 0x80000000 0x80100000 40000
Total of 262144 word(s) were the same
=> md 0x80000000 0xc
80000000: f5b5afec a76e1eb6 c2c2868a 27e177ee    ......n......w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
=> md 0x80100000 0xc
80100000: f5b5afec a76e1eb6 c2c2868a 27e177ee    ......n......w.'
80100010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80100020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
=>
```

Like most memory commands the :redtext:cmp` can access the memory in different sizes: as 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. If you want to access memory as 16 bit or word data, you can use the variant cmp.w instead; and to access memory as 8 bit or byte data please 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.

```
=> cmp.l 0x80000000 0x80100000 40000
Total of 262144 word(s) were the same
=> cmp.w 0x80000000 0x80100000 80000
Total of 524288 halfword(s) were the same
=> cmp.b 0x80000000 0x80100000 100000
Total of 1048576 byte(s) were the same
=>
```

cp - memory copy

```
=> help cp
cp - memory copy

Usage:
cp [.b, .w, .l] source target count
=>
```

The cp command is used to copy memory areas.

```
=> cp 0x80000000 0x80100000 10000
=>
```

The cp command understands the type extensions .l, .w and .b :

```
=> cp.l 0x80000000 0x80100000 10000
=> cp.w 0x80000000 0x80100000 20000
=> cp.b 0x80000000 0x80100000 40000
=>
```

md - memory display

```
=> help md
md - memory display

Usage:
md [.b, .w, .l] address [# of objects]
=>
```

The md command can be used to display memory contents both as hexadecimal and ASCII data.

```
=> md 0x80000000
80000000: f5b5afec a76e1eb6 c2c2868a 27e177ee    ......n......w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
80000040: 3be665d4 f934ed72 77f30cfe ff7d4927    .e.;r.4....w'I}.
80000050: 01938b0f 67670a77 a03e2f27 a3cddfa8    ....w.gg'/>.....
80000060: 7db4427d a8720cdc a00adede db2afc3e    }B.}..r.....>.*.
80000070: 8b68e416 5fb506a9 8a3d89e7 9155cdc2    ..h...._..=...U.
80000080: 897baab0 2df979d2 04789346 6d0794e1    ..{..y.-F.x....m
80000090: 4cf78ec4 7ef5efa3 d2079c53 56a97ebc    ...L...~S....~.V
800000a0: 1d592a82 12942c92 04204dc4 51a92c67    .*Y..,...M .g,.Q
800000b0: 8c0a5f41 9eae1126 426a3d45 efb1cdb0    A_..&...E=jB....
800000c0: 495f405b 228f37a0 558bbf15 6170078a    [@_I.7."...U..pa
800000d0: f3dd9b6f 992af779 d7a51967 cb0c7480    o...y.*.g....t..
800000e0: 0d9a0971 b232aaf7 682e70c7 8001b172    q.....2..p.hr...
800000f0: fd79317c a172ac42 46685500 b25977a8    |1y.B.r..UhF.wY.
=>
```

This command can also be used with the type extensions .l, .w and .b :

```
=> md.w 0x80000000
80000000: afec f5b5 1eb6 a76e 868a c2c2 77ee 27e1    ......n......w.'
80000010: 141f 85a9 0d29 ea0e 3b60 a3b8 bf20 9c5a    ....)...`;.. .Z.
80000020: 0728 d5f9 422c 4863 a83a 3faa fa91 375d    (...,BcH:..?..]7
80000030: c998 f708 7535 a5e2 2513 2df7 0796 c221    ....5u...%.-..!.
80000040: 65d4 3be6 ed72 f934 0cfe 77f3 4927 ff7d    .e.;r.4....w'I}.
80000050: 8b0f 0193 0a77 6767 2f27 a03e dfa8 a3cd    ....w.gg'/>.....
80000060: 427d 7db4 0cdc a872 dede a00a fc3e db2a    }B.}..r.....>.*.
80000070: e416 8b68 06a9 5fb5 89e7 8a3d cdc2 9155    ..h...._..=...U.
=> md.b 0x80000000
80000000: ec af b5 f5 b6 1e 6e a7 8a 86 c2 c2 ee 77 e1 27    ......n......w.'
80000010: 1f 14 a9 85 29 0d 0e ea 60 3b b8 a3 20 bf 5a 9c    ....)...`;.. .Z.
80000020: 28 07 f9 d5 2c 42 63 48 3a a8 aa 3f 91 fa 5d 37    (...,BcH:..?..]7
80000030: 98 c9 08 f7 35 75 e2 a5 13 25 f7 2d 96 07 21 c2    ....5u...%.-..!.
=>
```

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

```
=> md.b 0x80000000 0x20
80000000: ec af b5 f5 b6 1e 6e a7 8a 86 c2 c2 ee 77 e1 27    ......n......w.'
80000010: 1f 14 a9 85 29 0d 0e ea 60 3b b8 a3 20 bf 5a 9c    ....)...`;.. .Z.
=> md.w 0x80000000
80000000: afec f5b5 1eb6 a76e 868a c2c2 77ee 27e1    ......n......w.'
80000010: 141f 85a9 0d29 ea0e 3b60 a3b8 bf20 9c5a    ....)...`;.. .Z.
80000020: 0728 d5f9 422c 4863 a83a 3faa fa91 375d    (...,BcH:..?..]7
80000030: c998 f708 7535 a5e2 2513 2df7 0796 c221    ....5u...%.-..!.
=> md 0x80000000
80000000: f5b5afec a76e1eb6 c2c2868a 27e177ee    ......n......w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
80000040: 3be665d4 f934ed72 77f30cfe ff7d4927    .e.;r.4....w'I}.
80000050: 01938b0f 67670a77 a03e2f27 a3cddfa8    ....w.gg'/>.....
80000060: 7db4427d a8720cdc a00adede db2afc3e    }B.}..r.....>.*.
80000070: 8b68e416 5fb506a9 8a3d89e7 9155cdc2    ..h...._..=...U.
=>
```

mm - memory modify (auto-incrementing)

```
=> help mm
mm - memory modify (auto-incrementing address)

Usage:
mm [.b, .w, .l] address
=>
```

The mm command 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 (like .):

```
=> mm 0x80000000
80000000: f5b5afec ? 0
80000004: a76e1eb6 ? 0xaabbccdd
80000008: c2c2868a ? 0x01234567
8000000c: 27e177ee ? .
=> md 0x80000000 10
80000000: 00000000 aabbccdd 01234567 27e177ee    ........gE#..w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=>
```

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

```
=> mm.w 0x80000000
80000000: 0000 ? 0x0101
80000002: 0000 ? 0x0202
80000004: ccdd ? 0x4321
80000006: aabb ? 0x8765
80000008: 4567 ? .
=> md 0x80000000 10
80000000: 02020101 87654321 01234567 27e177ee    ....!Ce.gE#..w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=>
=> mm.b 0x80000000
80000000: 01 ? 0x48
80000001: 01 ? 0x65
80000002: 02 ? 0x6c
80000003: 02 ? 0x6c
80000004: 21 ? 0x6f
80000005: 43 ? 0x20
80000006: 65 ? 0x20
80000007: 87 ? 0x20
80000008: 67 ? .
=> md 0x80000000 10
80000000: 6c6c6548 2020206f 01234567 27e177ee    Hello   gE#..w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=>
```

mw - memory write (fill)

```
=> help mw
mw - memory write (fill)

Usage:
mw [.b, .w, .l] address value [count]
=>
```

The mw command 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 value, the entire memory area will be initialized with this value:

```
=> md 0x80000000 0x10
80000000: 6c6c6548 2020206f 01234567 27e177ee    Hello   gE#..w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=> mw 0x80000000 0xaabbccdd
=> md 0x80000000 0x10
80000000: aabbccdd 2020206f 01234567 27e177ee    ....o   gE#..w.'
80000010: 85a9141f ea0e0d29 a3b83b60 9c5abf20    ....)...`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=> mw 0x80000000 0 6
=> md 0x80000000 0x10
80000000: 00000000 00000000 00000000 00000000    ................
80000010: 00000000 00000000 a3b83b60 9c5abf20    ........`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=>
```

This is another command that accepts the type extensions .l, .w and .b :

```
=> mw.w 0x80000004 0x1155 6
=> md 0x80000000 0x10
80000000: 00000000 11551155 11551155 11551155    ....U.U.U.U.U.U.
80000010: 00000000 00000000 a3b83b60 9c5abf20    ........`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=> mw.b 0x80000007 0xff 7
=> md 0x80000000 0x10
80000000: 00000000 ff551155 ffffffff 1155ffff    ....U.U.......U.
80000010: 00000000 00000000 a3b83b60 9c5abf20    ........`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-..!.
=>
```

nm - memory modify (constant address)

```
=> help nm
nm - memory modify (constant address)

Usage:
nm [.b, .w, .l] 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 0x80000000
80000000: 00000000 ? 0x48
80000000: 00000048 ? 0x65
80000000: 00000065 ? 0x6c
80000000: 0000006c ? 0x6c
80000000: 0000006c ? 0x6f
80000000: 0000006f ? .
=> md 0x80000000 10
80000000: 0000006f ff551155 ffffffff 1155ffff    o...U.U.......U.
80000010: 00000000 00000000 a3b83b60 9c5abf20    ........`;.. .Z.
80000020: d5f90728 4863422c 3faaa83a 375dfa91    (...,BcH:..?..]7
80000030: f708c998 a5e27535 2df72513 c2210796    ....5u...%.-...!.
=>
```

The nm command also accepts the type .l, .w and .b

loop - infinite loop on address range

```
=> help loop
loop - infinite loop on address range

Usage:
loop [.b, .w, .l] address number_of_objects
=>
```

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!

Execution Control Commands

source - run script from memory

```
=> help source
source - run script from memory

Usage:
source [addr]
      - run script starting at addr
      - A valid image header must be present
For FIT format uImage addr must include subimage
unit name in the form of addr:<subimg_uname>
=>
```

With the source command you can run "shell" scripts under U-Boot: You create a U-Boot script image by simply writing the commands you want to run into a text file; then you will have to use the mkimage tool 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 source you can run the commands in such an image. For instance, the following text file:

```
$ cat source_example.txt
echo
echo Network Configuration:
echo ----------------------
echo Target:
printenv ipaddr hostname
echo
echo Server:
printenv serverip rootpath
echo
$
```

can be converted into a U-Boot script image using the mkimage command like this:

```
$ mkimage -A ppc -O linux -T script -C none -a 0 -e 0 -n "autoscr example script" -d\
  /work/hs/tbot/source_example.txt /var/lib/tftpboot/beagleboneblack/tbot/source.scr
Image Name:   autoscr example script
Created:      Fri Aug 18 10:24:36 2017
Image Type:   PowerPC Linux Script (uncompressed)
Data Size:    157 Bytes = 0.15 kB = 0.00 MB
Load Address: 00000000
Entry Point:  00000000
Contents:
   Image 0: 149 Bytes = 0.15 kB = 0.00 MB
$
```

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

```
=> tftp 0x80000000 beagleboneblack/tbotrandom
link up on port 0, speed 100, full duplex
Using ethernet@4a100000 device
TFTP from server 192.168.2.1; our IP address is 192.168.2.10
Filename 'beagleboneblack/tbotrandom'.
Load address: 0x80000000
Loading: *#################################################################
         #######
         4.5 MiB/s
done
Bytes transferred = 1048576 (100000 hex)
=>
=> imi 0x80000000

## Checking Image at 80000000 ...
```

```
   Legacy image found
   Image Name:   autoscr example script
   Created:      2017-08-18   8:19:02 UTC
   Image Type:   PowerPC Linux Script (uncompressed)
   Data Size:    157 Bytes = 157 Bytes
   Load Address: 00000000
   Entry Point:  00000000
   Contents:
      Image 0: 149 Bytes = 149 Bytes
   Verifying Checksum ... OK
=> source 0x80000000
## Executing script at 80000000

Network Configuration:
----------------------
Target:
ipaddr=192.168.2.10
## Error: "hostname" not defined

Server:
serverip=192.168.2.1
rootpath=/export/rootfs

=>
```

bootm - boot application image from memory

```
=> help bootm
bootm - boot application image from memory

Usage:
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
      When booting a Linux kernel which requires a flat device-tree
      a third argument is required which is the address of the
      device-tree blob. To boot that kernel without an initrd image,
      use a '-' for the second argument. If you do not pass a third
      a bd_info struct will be passed instead

For the new multi component uImage format (FIT) addresses
      must be extended to include component or configuration unit name:
      addr:<subimg_uname> - direct component image specification
      addr#<conf_uname>   - configuration specification
      Use iminfo command to get the list of existing component
      images and configurations.

Sub-commands to do part of the bootm sequence.  The sub-commands must be
issued in the order below (it's ok to not issue all sub-commands):
      start [addr [arg ...]]
      loados  - load OS image
      ramdisk - relocate initrd, set env initrd_start/initrd_end
      fdt     - relocate flat device tree
      cmdline - OS specific command line processing/setup
      bdt     - OS specific bd_t processing
      prep    - OS specific prep before relocation or go
      go      - start OS
=>
```

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.

Linux requires the flattened device tree blob to be passed at boot time, and bootm expects its third argument to be the address of the blob in memory. Second argument to bootm depends on whether an initrd initial ramdisk image is to be used. If the kernel should be booted without the initial ramdisk, the second argument should be given as "-", otherwise it is interpreted as the start address of initrd (in RAM, ROM or flash memory).

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

```
=> bootm ${kernel_addr} - ${fdt_addr}
```

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

```
=> bootm ${kernel_addr} ${ramdisk_addr} ${fdt_addr}
```

Both examples of course imply that the variables used are set to correct addresses for a kernel, fdt blob 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.

go - start application at address 'addr'

```
=> help go
go - start application at address 'addr'

Usage:
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.

TODO For more information see 5.12. U-Boot Standalone Applications.