NAME

buildkernel-b2 − build bootable Linux kernel for the Excito B2

SYNOPSIS

buildkernel-b2 [*options*]

DESCRIPTION

buildkernel-b2 is a script that builds a Gentoo Linux kernel image suitable for booting either from the B2’s internal disk (default), or from a USB key (when the **--usb** option is specified).

It is useful when targeting a B2 system with:

- the default U-Boot settings in flash memory; and
- a payload kernel that has size > 4MiB uncompressed (or > 2MiB compressed).

You can run **buildkernel-b2** either on the B2 directly, or on a Gentoo PC (with an appropriately configured **crossdev** setup). If running on a PC, a directory *deploy_root* will be created as a result of running the script, with the necessary files for you to copy across to your B2. If running on the B2 natively, the files (kernel, modules, DTBs, config and System.map) will be moved to the correct positions for you (and prior copies of all but the modules backed up).

buildkernel-b2 should be run as root, in the top level kernel source directory. You will need an appropriate *.config* file in place when building the kernel (however, if running on the B2, the running kernel’s config will be used as a fallback, if one is available via */proc/config.gz*, and no other *.config* is found).

ALGORITHM DETAIL (DISK VARIANT)

When power is applied to the Excito B2 *without* the rear button depressed, its U-Boot bootloader by default executes the **diskboot** command, which attempts to:

1) load */boot/uImage* to 0x00400000 (4MiB), then
2) load */boot/8313E21.dtb* to 0x00600000 (6MiB) (rev 1 B2s load */boot/bubba.dtb* instead), then
3) **bootm** 0x00400000 - 0x00600000

Unfortunately, a fairly standard 3.18+ kernel has a size >2MiB, even when compressed (assuming reasonable built-in options, such as ext4 support etc), so when U-Boot loads the DTB to the 6MiB address in step 2, it ends up overwriting some of it. Even if this problem were to be avoided, there is a further issue: the uncompressed kernel is usually >4MiB, but must be loaded to address 0x00000000, so U-Boot will refuse run it after decompression (as the uImage header will be detected to have been damaged)! In either case, the image will fail to boot.

This can of course be worked around by changing U-Boot’s flash environment (to set safer load addresses), but that risks a user bricking her device, so we’d like to avoid it.

Accordingly, in this script, we build a modified uImage, which can be booted successfully with the default U-Boot settings. This contains a ’shim’ (built using the PowerPC assembly code from *reloc_shim.S*) uImage, followed by enough zero padding to reach the 16MiB boundary, followed by a 32-bit big endian quantity holding the length of the ’real’ kernel, and followed finally by the uncompressed ’real’ kernel itself (the payload).

As such, when diskboot loads our augmented uImage to 0x00400000 in step 1, we end up with the following memory map:

Address	Contents
0x00400000	valid (small) uImage of shim, load/exec address 0x020000000
... zero padding ...	
0x01000000	length of kernel (4 bytes big endian unsigned)
0x01000004	uncompressed 'real' kernel

And after it loads the DTB in step 2, we have:

Address	Contents
0x00400000	valid (small) uImage of shim, load/exec address 0x020000000
... zero padding ...	
0x00600000	valid image of (initial) DTB
... zero padding ...	
0x01000000	length of kernel (4 bytes big endian unsigned)
0x01000004	uncompressed 'real' kernel

Then **bootm** is run by U-Boot in step 3, which copies our uImage 'shim' payload to its target address (which we set via mkimage to be 0x02000000), patches up the DTB (with additional info, such as (U-Boot's) kernel command line) then relocates it to somewhere below the 8MiB boundary that U-Boot thinks is safe, sets up the various registers required to invoke a PowerPC kernel (see the file *arch/powerpc/lib/bootm.c*, function **boot_jump_linux()**, in the U-Boot source code), and then starts the 'kernel' (our shim, in this case), at its execution address (which we set also to be 0x02000000).

The map is then:

Address	Contents
0x00400000	valid (small) uImage of shim, load/exec address 0x020000000
... zero padding ...	
0x00600000	valid image of (initial) DTB
... zero padding ...	
0x00??????	U-Boot's modified copy of the DTB (with command line etc.), somewhere 'safe' (it thinks) below the 8MiB boundary; the address of this is in r3 when the kernel (actually, our shim) is called
... zero padding ...	
0x01000000	length of kernel (4 bytes big endian unsigned)
0x01000004	uncompressed 'real' kernel
0x02000000	our shim (which U-Boot has just started)

The shim will then:

1) copy the (U-Boot modified and relocated) DTB up to a genuinely safe memory location (0x00f00000, the 15MiB boundary), where the uncompressed (real) kernel will definitely not overwrite it
2) patch up the r3 register with the new DTB address
3) copy the (real) uncompressed kernel to 0x00000000, then
4) jump to 0x00000000 to start the (real) kernel.

Just prior to jumping into the real kernel, we therefore have:

Address	**Contents**
0x00000000	the real kernel image (note that the uImage at 0x00400000, the original DTB at 0x00600000, and possibly even U-Boot's copy of the relocated DTB will probably be overwritten by this, but we don't care at this point)
... zero padding ...	
0x00f00000	valid copy of (U-Boot's modified) DTB
... zero padding ...	
0x01000000	length of kernel (4 bytes big endian unsigned)
0x01000004	uncompressed 'real' kernel
0x02000000	our shim (currently executing)

Once the real kernel is started, all the extra memory used for these copies is of course reclaimed, and boot proceeds as normal.

Please note that the uImage created by this script cannot be tested directly by **mkimage -l**, as that command will attempt to account for all the data in the file; however, it *does* work with U-Boot itself, since that simply loads the file into memory, and subsequently tries to figure out if the data it finds *starting* at 0x00400000 is a valid uImage (which it is).

ALGORITHM DETAIL (USB VARIANT)

When power is applied to the Excito B2 *with* the rear button depressed, its U-Boot bootloader by default executes the **usbboot** command, which attempts to:

1) load */install/8313E21.itb* (from USB partition 1) to 0x00400000 (4MiB) (rev 1 B2s load */install/install.itb* instead), then
2) **bootm**

The .itb is a flat image tree (FIT) file, which contains a kernel and DTB, together with some metadata for both.

Unfortunately, a fairly standard 3.18+ kernel has a size >4MiB, when decompressed (assuming reasonable built-in options, such as ext4 support etc), and because it has to reside at address 0x00000000, U-Boot will fail to start it after decompression (as it will detect the corrupted FIT image header).

This can of course be worked around by changing U-Boot's flash environment (to set a safer load address), but that risks a user bricking her device, so we'd like to avoid it.

Accordingly, in this script, we build a modified FIT, which can be booted successfully with the default U-

Boot settings. This FIT contains a ’shim’ (built using the PowerPC assembly code from *reloc_shim_itb.S*) with the real kernel appended, saved as a raw uncompressed binary, plus of course a standard DTB file. The FIT contents and metadata are specified by *reloc_shim.its*, in which we specify that our ’kernel’ should be moved (by U-Boot) to 0x02000000 (the 32MiB boundary), and executed from that address. The DTB has no specified deployment address.

As such, when **usbboot** loads our augmented FIT to 0x00400000 in step 1, we end up with the following memory map:

Address	**Contents**
0x00400000	valid FIT image

When it runs **bootm** (step 2), U-Boot will copy the kernel (with our prepended relocation shim) up to address 0x02000000 (it will not complain about this, as there is plenty of memory up there, and the copy will not overwrite the original FIT). It then modifies the DTB in the FIT image (setting additional information, such as the kernel command line), and relocates it to somewhere below the 8MiB boundary that it thinks is safe. Then, it sets up the various registers required to invoke a PowerPC kernel (see the file *arch/powerpc/lib/bootm.c*, function **boot_jump_linux()**, in the U-Boot source code), and then starts the ’kernel’ (actually, in this case, our prepended shim), at 0x02000000.

The map is then:

Address	**Contents**
0x00400000	valid FIT image
0x00??????	U-Boot’s modified copy of the DTB (with command line etc.), somewhere ’safe’ (it thinks) below the 8MiB boundary; the address of this is in r3 when the kernel (actually, our shim) is called
0x02000000	our shim (which U-Boot has just started), with the real (’payload’) kernel appended

The shim will then:

1) copy the (U-Boot modified) DTB up to a genuinely safe memory location (0x00f00000, the 15MiB boundary), where the uncompressed (real) kernel will definitely not overwrite it
2) patch up the r3 register with the new DTB address
3) copy the (real) uncompressed kernel to 0x00000000, then
4) jump to 0x00000000 to start the (real) kernel

Just prior to jumping into the real kernel, we therefore have:

Address	**Contents**
0x00000000	copy of the real kernel image (note that the FIT at 0x00400000, and possibly even U-Boot's copy of the relocated DTB will probably be overwritten by this, but we don't care at this point)
0x00f00000	valid copy of (U-Boot's modified) DTB
0x02000000	our shim (currently executing), with copy of the real kernel image appended

Once the real kernel is started, all the extra memory used for these copies is of course reclaimed, and boot proceeds as normal.

NB - you must use uncompressed images for this trick to work.

OPTIONS

−c, **−−clean**
Specifies that a **make clean** should be carried out in the kernel source directory prior to building (this will leave the *.config* file intact). Most of the time, it is fine not to **make clean**.

−h, **−−help**
Displays a short help screen, and exits.

−m, **−−menuconfig**
Specifies that the GUI-based kernel configuration tool (**make menuconfig**) should be invoked at the start of the build.

−n, **−−no−pump**
Normally, when building on the B2, this script will invoke **make**(1) with the **pump**(1) prefix (to distribute compilation and pre-processing), if the **distcc-pump Portage**(5) feature is detected. Specify this option to force a local build instead.

−u, **−−usb**
Instructs **buildkernel-b2** to create USB-bootable FIT images (install.itb and 8313E21.itb), rather than the default uImage.

−v, **−−verbose**
Provides more verbose output from invoked tools, where possible.

−V, **−−version**
Displays the version number of **buildkernel-b2**, and exits.

BUGS

- **buildkernel-b2** currently executes the kernel build process as the root user. It would be a little more hygienic to build as a non-privileged user, and then install as root. Also, this script should really be integrated into the *arch/powerpc/boot* wrapper build process, rather than be shipped standalone.
- Currently, you must override the bootloader-provided command line in your kernel *.config* (since the former will otherwise specify an incorrect root, unless you reflash the U-Boot environment... which we are trying to avoid).
- It should in theory be possible to use **kexec()** to have a small kernel chainload the real one, but I haven't had any success getting this to work cleanly on the B2. If you have, please let me know!

COPYRIGHT

AUTHORS

sakaki — send bug reports or comments to <sakaki@deciban.com>

SEE ALSO

make(1), **pump**(1), **portage**(5).