Flattened Image Trees: A powerful kernel uImage format

Feb 21, 2013

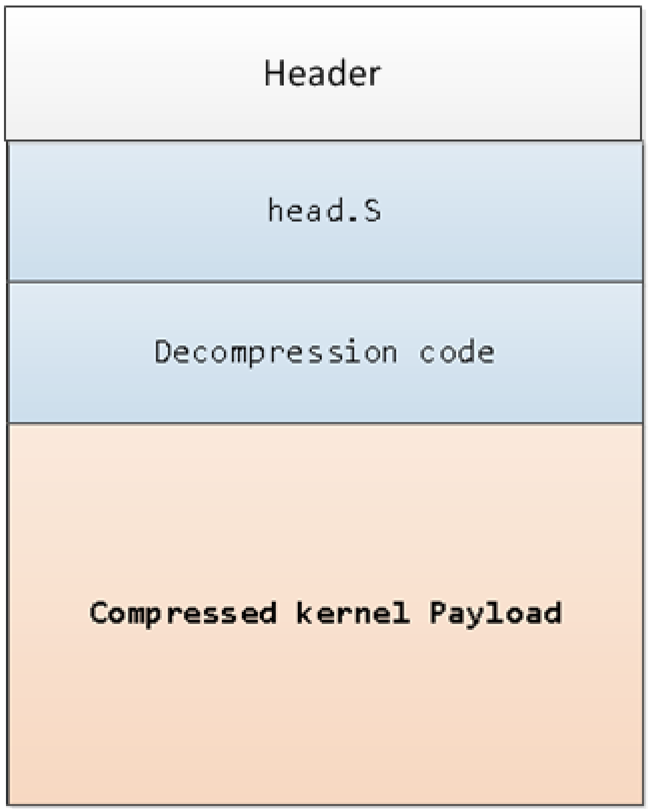
Joel A Fernandes <joelagnel@ti.com>

Goals of this talk

- Shortcomings of Legacy image formats
- To understand existing challenges in multicomponent Images
- How these have been solved
- How these can be tackled using FIT
- Recent applications (verified boot)
- Advantages of FIT
- Future work

Classical Image formats

zImage format



Very limited:

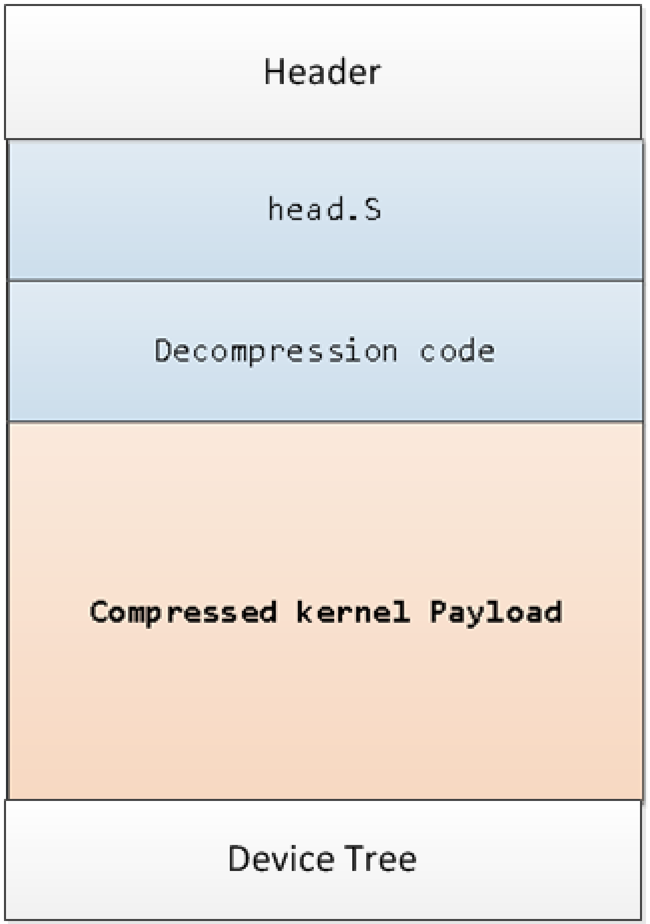
- Not much information about the kernel itself (architecture?)
- No support embedding DT
- No checksums for data integrity
- Compression format is fixed, and requires kernel recompile

Many others…

Compression is fixed by Kernel config..
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_LZMA is not set
# CONFIG_KERNEL_XZ is not set
CONFIG_KERNEL_LZO=y

Classical Image formats

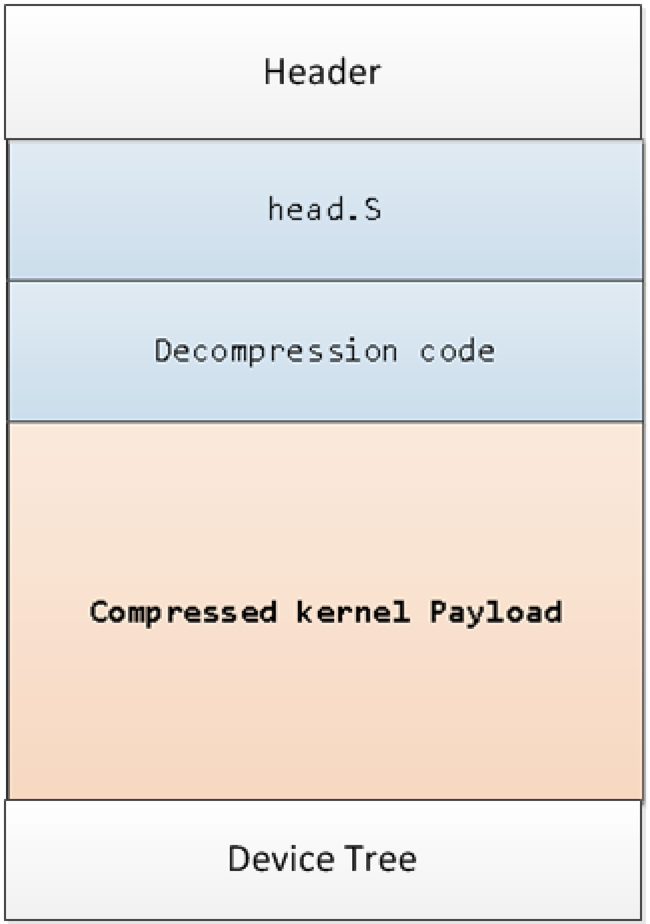
dtbImage format (PPC)



- Same like zImage, but can embed a Device tree blob
- Useful for platforms that don’t supporting passing of a DT from a bootloader.
- Same drawbacks as the zImage

Classical Image formats

simpleImage format (PPC)



- Same like dtbImage but can be executed from anywhere in memory
- Useful when Firmware cannot pass data to the kernel or kernel is expected to boot without Firmware support
- All information required for boot is present in the embedded DTB
- Again- all the earlier drawbacks in this super-simple format.

zImage hacks (ARM) to support appending of DT

- Code added to zImage head.S to support appending of DT blob

Drawbacks:

- Ugly- no real notion of what is appended.
- Only one DT. Makes the image a single-platform one.
- Still lacks kernel build support. Floating hacks.

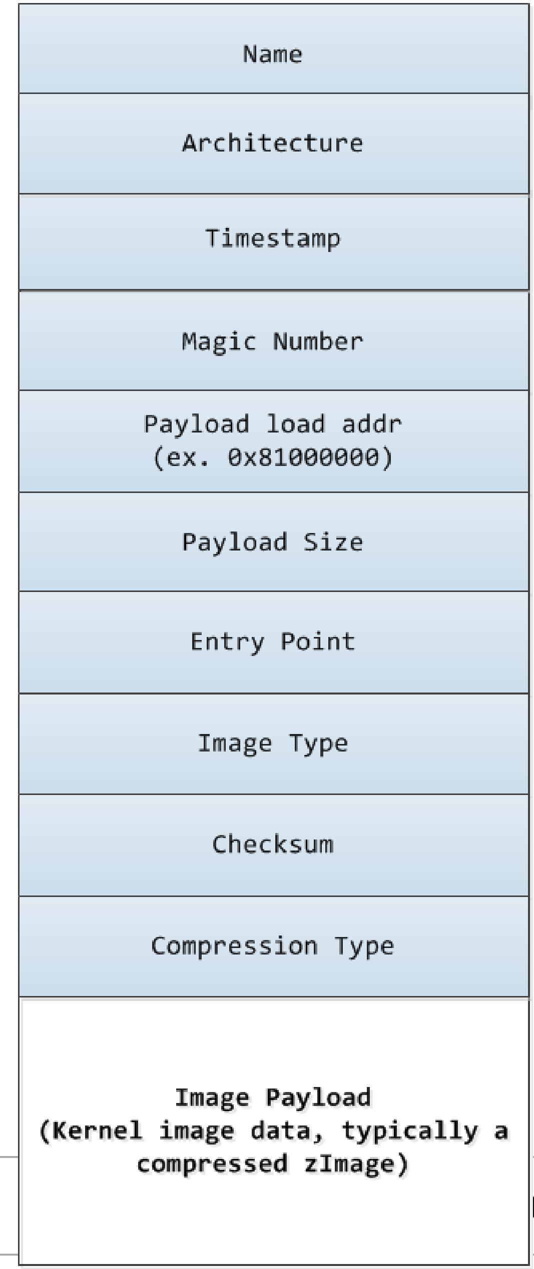
Overview of U-Boot's image format

- OS / Architecture - independent
- Multiple compression types – gzip, bzip2, lzma
- CRC checksums
- Ability to execute in place (XIP)
- Meta-data about image including name, architecture etc.
- Very efficient to parse (13 years back)

Single Component U-Boot Images

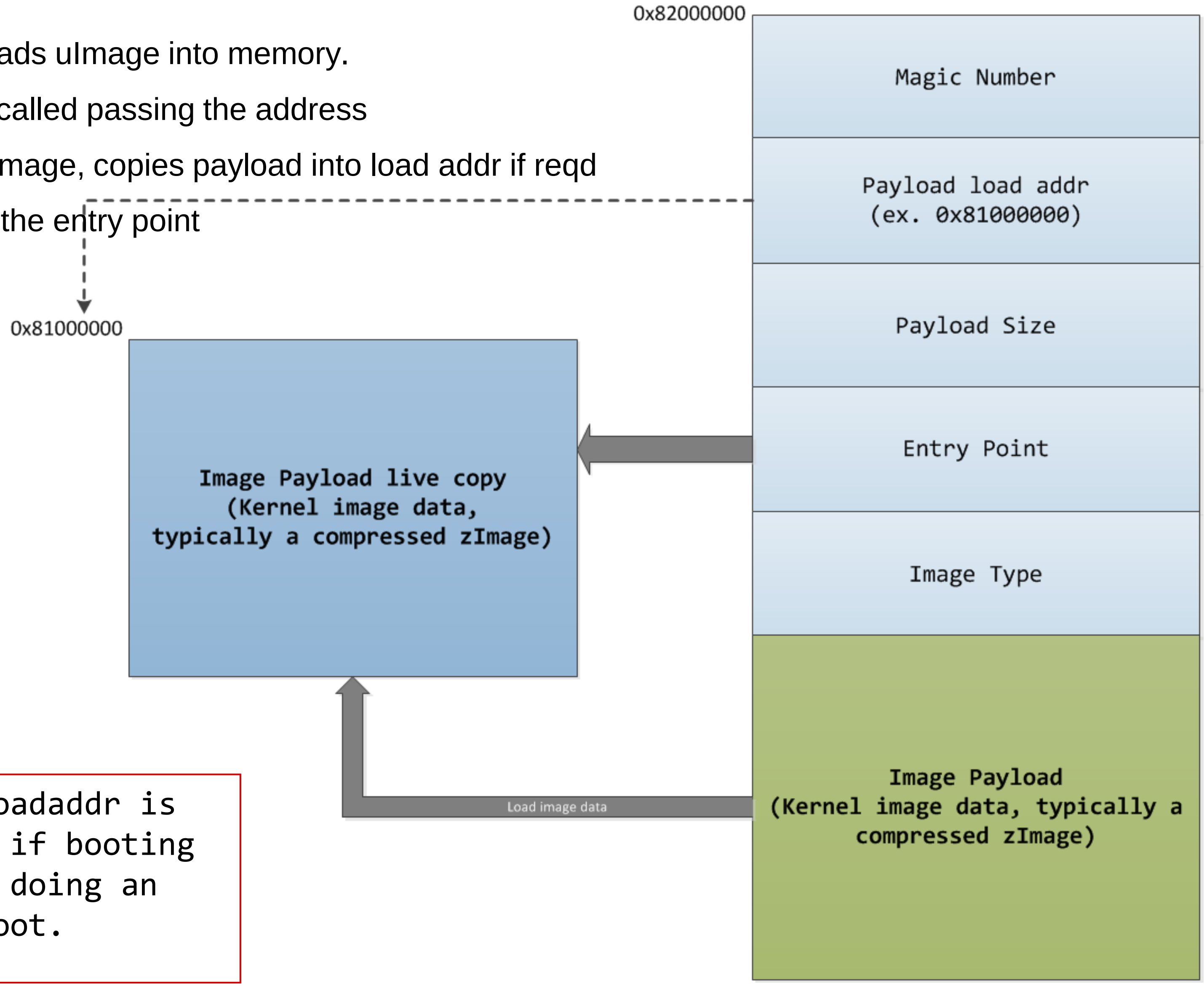
Structure of the Legacy U-Boot Image

- Only supports a single component (extended for multicomponent, more on this later)
- Architecture/OS fields exist too (not shown)
- Magic number- checks if legacy or FIT
- Payload addr- where to load in memory
- Size – how much to load
- Entry point- where should bootloader jump
- Image type- Single, Multicomponent, Inplace
- Payload- Kernel or other image payload



Booting of a Single Component Image

- U-Boot loads uImage into memory.
- Bootm is called passing the address
- Parses uImage, copies payload into load addr if reqd
- Jumps to the entry point



Copying to loadaddr is not required if booting from NOR; or doing an XIP uImage boot.

mkImage can show load addr and ep

```
# mkimage -l arch/arm/boot/uImage

Image Name:   Linux-3.7.0-26691-gea93ee1
Created:      Sat Jan 19 22:01:36 2013
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2842064 Bytes = 2775.45 kB = 2.71 MB
Load Address: 80008000
Entry Point:  80008000
```

Multi Component U-Boot Images

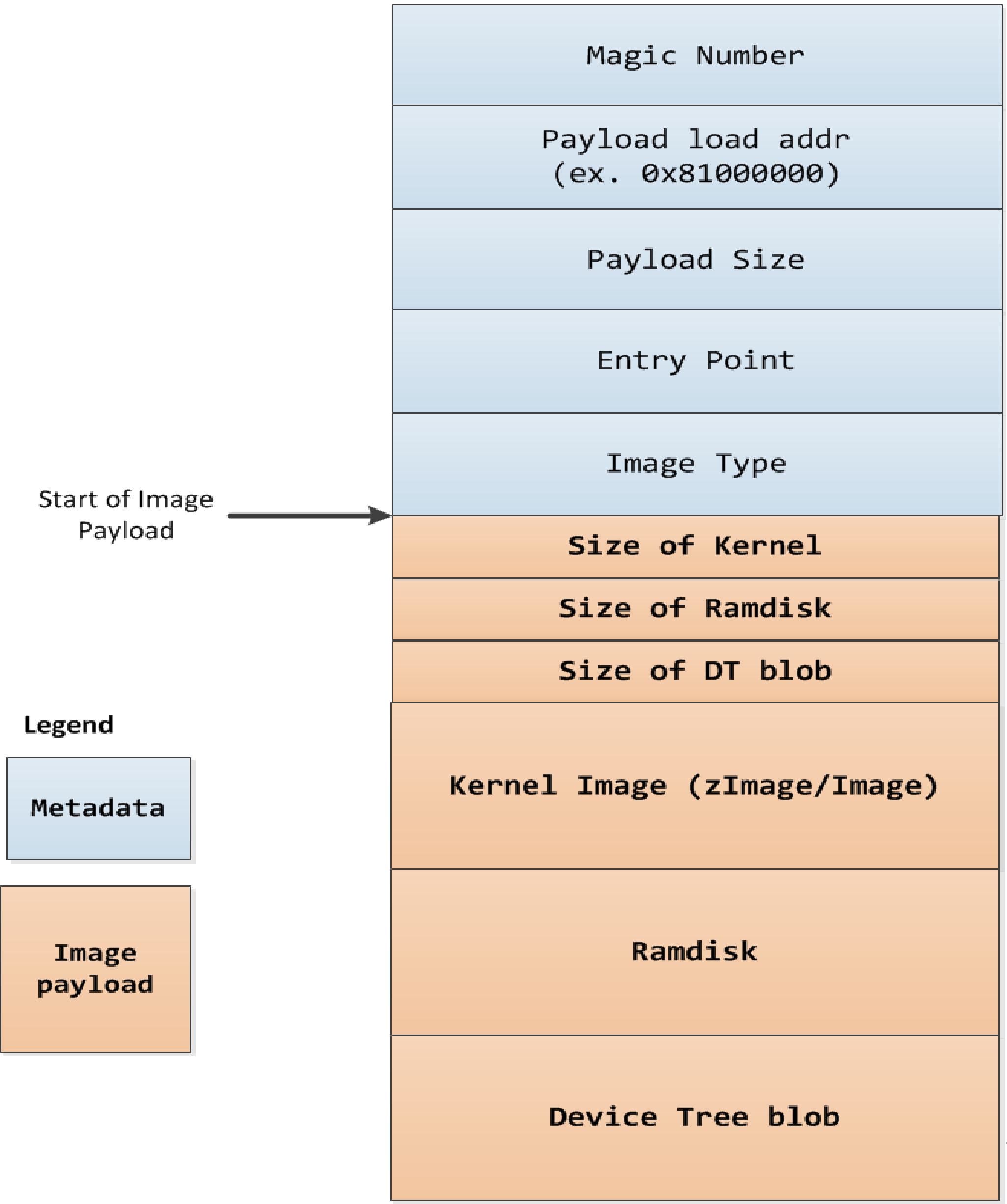
Single Component Image limitations

- Users found it necessary to have more than one component in a uImage such as Ramdisk, DT blob. Single component images limited.

- Multiple components were required to be included in some cases
  – Booting using a single image over DHCP
  – Necessity to use more than 1 component
  – Recovery of systems- where you want an initrd to give you an FS
  – Firmware upgrade where it is not easy to download multiple components
  – Security- sometimes folks want to include cryptographic signatures.

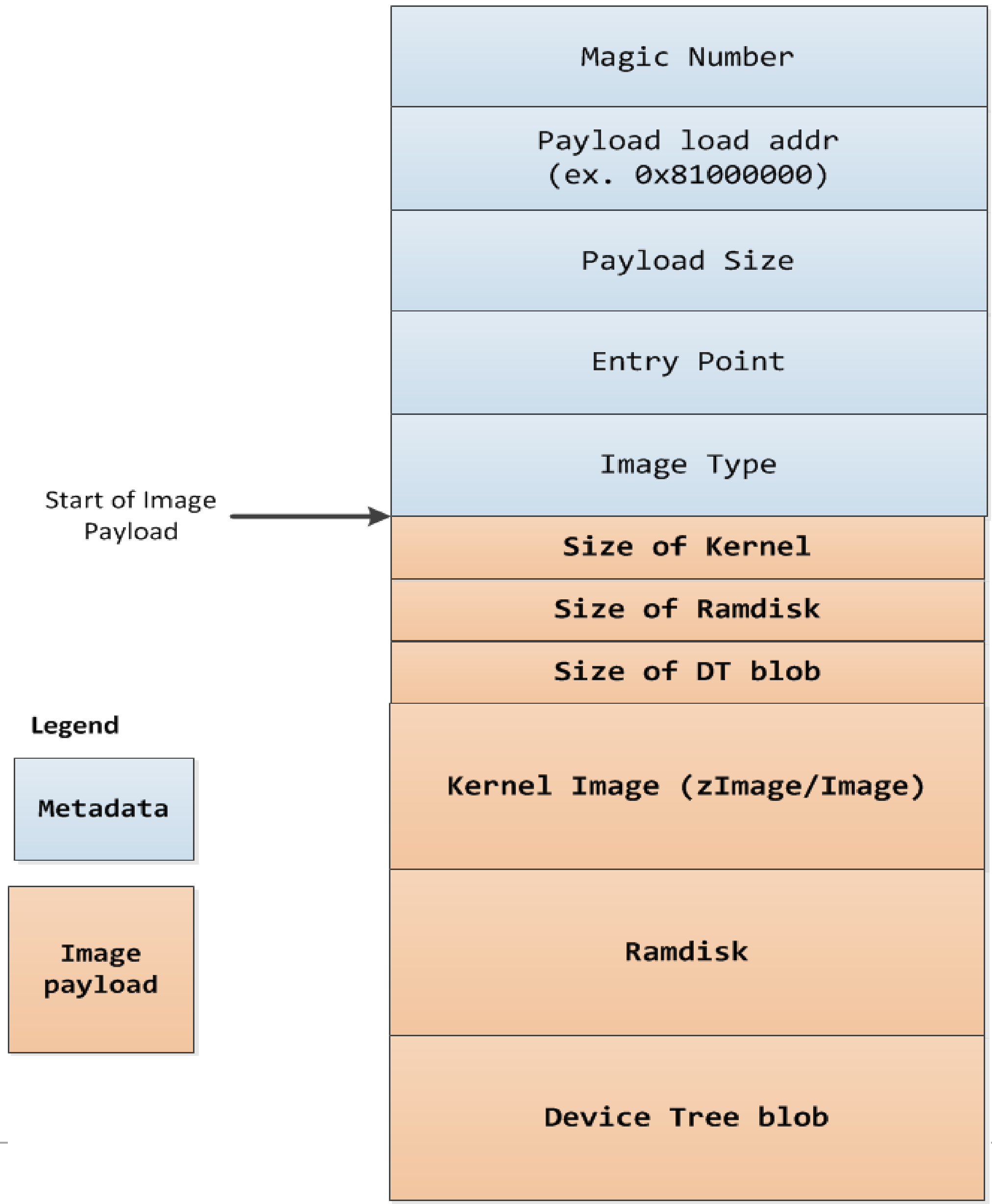
- A new image type in the "single-component" image header was introduced, called IH_MULTI with additional components in payload.
- Image header supports only CRC32, no support for other checksums

Structure of a Mutli Component Image



- Metadata into the single image payload
- A null-terminated table of component sizes was introduced.
- This table was actually a **part of the payload** that contained just the kernel image previously..

Structure of a Mutli Component Image



- Table entries hard-coded to a pre-defined component. id 1 for ramdisk, id 2 for dt.
- Fixed mapping of id to component type. Ramdisk can't be pushed after DT blob
- Worked.. But has drawbacks, more on that next..

IH_TYPE_MULTI users can DHCP a single image with kernel, ramdisk and dt. Easy!

Problems with this approach..

- The meta-data stored in MC was limited.. Can't load more than 1 position dependent component . "load address" is single.
- Hardcoding of indices of image components in the code (1=kernel, 2=ramdisk.. Not cool)
  - Associating numbers instead of names to image components is messy meta-data is not self explanatory.
  - What if in the future one image component had to be removed while another one was added? All of a sudden the component indexes of all components change and code would need to be modified.
  - Difficult to maintain code. Code is already very hacked up

Problems with this approach..

- Limited support for adding more components, only the 3 – kernel, ramdisk, and single DT blob
  - What if someone wants to add a new crypto graphic signature
  - Or a secondary ramdisk
  - Or an alternate device tree blob?
  - Or some other component that nobody thought of?

- How can multiple kernels be represented? Not possible as several fields in header are for only 1 kernel (arch, os, load addr)

- doesn't scale for future designs and encourages introduction of more hacks.
- Still no support for stronger checksums.. Nothing can be done about that even with IH_TYPE_MULTI

Introducing Tree-like structures to represent images

Add some flexibility to an image … mix meta-data with data

- Trees are a nice way to represent data with meta-data
  - Arbritrary arragement of nodes
  - Nodes can be named and can have Properties
  - Properties can even be binary images such as in the case of FIT

So wouldn't it be cool to represent a kernel image in the form:

```
kernel {
  description = “Linux kernel 3.8”
  loadaddress = “0x80200000”
  entrypoint = “0x80008000”
  data = <binary kernel image>
}
```

What is a Device Tree?

The Device Tree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time. The device tree is used both by Open Firmware, and in the standalone Flattened Device Tree (FDT) form.

- Describes functional layout
  - CPUs
  - Memory
  - Peripherals
- Describes configuration
  - Console output
  - Kernel parameters
  - Device names

Can we (re-)use the Device Tree?

- Already used in the kernel for “device tree”-based platforms
- Tools that build device trees already part of the kernel.
- Device Tree compiler has support to embed binaries in a tree property.

Flattened Image Trees

- A need for stronger checksums
- An image format that makes use of DT to build an image as a tree
- Nodes correspond to image components
- Property can have binary values using tags
- Perfect use for multicomponent images

Authored by Marian Balakowicz m8@semihalf.com

originally, for Power PC architecture.

A bit of history..

- Uboot support for pcs440ep required stronger checksums
- Old legacy header limited, couldn't support md5/sha.
- Led to looking for a new format using existing tools like dtc.

Architectures and Platforms using FIT

PowerPC:

- XPedite5400 board Freescale Eight-Core P4080 Processor-Based
  - FIT is infact supported on most if not all PowerPC based FreeScale boards
- MPC8544E PowerQUICC III based Socrates board

ARM:

- Neo Freerunner running Openmoko uses FIT
- ARM Cortex-A8 based Beaglebone. Demo follows
- Xilinx Zynq SoC (ARM Cortex-A9)
- Freescale i.MX31 based on ARM1136JF-S
- Samsung Chromebook running Samsung Exynos 5 Dual Processor

x86:

- Under review: Simon Glass has posted patches to boot a FIT on x86 and pass it a DT.

Other:

Microblaze softcpu core from Xilinx

zImage hacks to support appending of DT

- Many users prefer to have DT blob embedded into kernel
- Current way to do it is to append a DTB to kernel and build kernel with CONFIG_APPENDED_DTB .

Drawbacks..

- Ugly
- No clarity of what data is appended to the kernel for a third person who analyzes the image. Unlike FIT.
- One DT can be appended, unlike FIT. makes image single-platform.
- No kernel support still to build this. Out-of-tree hacks floating due to above drawback

Appended DT hack code ..

```
index abfce28..131558f 100644
--- a/arch/arm/boot/Makefile
+++ b/arch/arm/boot/Makefile
@@ -55,6 +55,9 @@ $(obj)/zImage:        $(obj)/compressed/vmlinux FORCE
        $(call if_changed,objcopy)
        @$(kecho) '  Kernel: $@ is ready'

+$(obj)/zImage-dtb.%:   $(obj)/%.dtb $(obj)/zImage
+       cat $(obj)/zImage $< > $@
+
 endif

+$(obj)/uImage-dtb.%:   $(obj)/zImage-dtb.% FORCE
+       $(call if_changed,uimage)
+       @echo '  Image $@ is ready'
+
```

A quick demo of FIT to show its flexibility

For the first demo, we show a FIT containing

- A Single kernel
- A single Device Tree blob

- Fit sources (.its files)
- Using mkimage to build it
- U-Boot commands to boot the image
- Boot log

- Demo uses a Beaglebone, U-Boot v2013.01-rc2, kernel 3.8

http://www.beagleboard.org/

demo 1: A simple FIT

Sources of kernel_fdt.its

```
/dts-v1/;
/ {
    description = "Simple image with single Linux kernel and FDT blob";
    #address-cells = <1>;
    images {
        kernel@1 {
            description = "Vanilla Linux kernel";
            data = /incbin/("./zImage");
            type = "kernel";
            arch = "arm";
            os = "linux";
            compression = "none";
            load = <0x80008000>;
            entry = <0x80008000>;
            hash@1 {
                algo = "crc32";
            };
            hash@2 {
                algo = "sha1";
            };
        };
[contd..]
```

dt source contd..

```
fdt@1 {
                    description = "Flattened Device Tree blob";
                    data = /incbin/("./am335x-bone.dtb");
                    type = "flat_dt";
                    arch = "arm";
                    compression = "none";
                    hash@1 {
                              algo = "crc32";
                    };
                    hash@2 {
                              algo = "sha1";
                    };
          };
    };
/* a notable concept of FIT, “configurations” */
    configurations {
          default = "conf@1";
          conf@1 {
                    description = "Boot Linux kernel with FDT blob";
                    kernel = "kernel@1";
                    fdt = "fdt@1";
          };
    };
};
```

Build the FIT using mkimage..

```
# mkimage -f kernel_fdt.its kernel_fdt.itb
FIT description: Simple image with single Linux kernel and FDT blob
Created:         Thu Jan 31 23:44:13 2013
 Image 0 (kernel@1)
  Description:  Vanilla Linux kernel
  Type:         Kernel Image
  Compression:  uncompressed
  Data Size:    2842064 Bytes = 2775.45 kB = 2.71 MB
  Architecture: ARM
  OS:           Linux
  Load Address: 0x80008000
  Entry Point:  0x80008000
  Hash algo:    crc32
  Hash value:   d4e59951
  Hash algo:    sha1
  Hash value:   933877a1fa0cad1f1dc4725918eeca4dc872e1ac
 Image 1 (fdt@1)
  Description:  Flattened Device Tree blob
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    11856 Bytes = 11.58 kB = 0.01 MB
  Architecture: ARM
  Hash algo:    crc32
  Hash value:   60fe7c97
  Hash algo:    sha1
  Hash value:   b206e49a4177ee285e1cbb225ae764815af4da7c
 Default Configuration: 'conf@1'
 Configuration 0 (conf@1)
  Description:  Boot Linux kernel with FDT blob
  Kernel:       kernel@1
  FDT:          fdt@1
```

Notice support for strong checksum algorithms like MD5, SHA1, ... Just doing a crc32 might not good enough for certain applications. Only image format that's so robust!

Boot it!

U-Boot commands to load the simple FIT

```
fitfdt=/boot/kernel_fdt.itb
setenv loadaddr 0x82000000;
run mmcargs;
ext2load mmc ${mmcdev}:2 ${loadaddr} ${fitfdt};

bootm ${loadaddr};
```

Boot it!

```
U-Boot SPL 2013.01-rc2-00174-ge56cdd7-dirty (Feb 01 2013 - 00:20:19)
..
U-Boot 2013.01-rc2-00174-ge56cdd7-dirty (Feb 01 2013 - 00:20:19)
..
## Booting kernel from FIT Image at 82000000 ...
   Using 'conf@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  Vanilla Linux kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x820000ec
     Data Size:    2842064 Bytes = 2.7 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x80008000
     Entry Point:  0x80008000
     Hash algo:    crc32
     Hash value:   d4e59951
     Hash algo:    sha1
     Hash value:   933877a1fa0cad1f1dc4725918eeca4dc872e1ac
   Verifying Hash Integrity ... crc32+ sha1+ OK

(contd…..)
```

Boot it!

(contd…)

```
## Flattened Device Tree from FIT Image at 82000000
   Using 'conf@1' configuration
   Trying 'fdt@1' FDT blob subimage
     Description:  Flattened Device Tree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x822b5fe4
     Data Size:    10568 Bytes = 10.3 KiB
     Architecture: ARM
     Hash algo:    crc32
     Hash value:   444390ae
     Hash algo:    sha1
     Hash value:   0530f3b384fb47ce796464a70ec618cf7e65b2a3
   Verifying Hash Integrity ... crc32+ sha1+ OK
   Booting using the fdt blob at 0x822b5fe4
   Loading Kernel Image ... OK
OK
   kernel loaded at 0x80008000, end = 0x802bddd0
   Loading Device Tree to 8fe44000, end 8fe49947 ... OK

Starting kernel ...
```

demo 2: Creating a FIT with a recovery configuration

Add a ramdisk node to the original FIT source. Call it kernel_fdt_rd.its

```
\ {
    images {
           kernel@1 {
             ..
           }
           fdt@1 {
             ..
           }
            ramdisk@1 {
                     description = "recovery ramdisk";
                     data = /incbin/("./ramdisk.gz");
                     type = "ramdisk";
                     arch = "arm";
                     os = "linux";
                     compression = "gzip";
                     load = <00000000>;
                     entry = <00000000>;
                     hash@1 {
                        algo = "sha1";
                     };
             };

    };
};
```

demo 2: Creating a FIT with a recovery configuration

(contd..)

```
/* Also update the configuration node – add 2 configs: default and recovery */
configurations {
      default = "defaultconf@1";
      defaultconf@1 {
         description = "Boot Linux kernel with FDT blob";
         kernel = "kernel@1";
         fdt = "fdt@1";
      };
      recoveryconf@1 {
         description = "Boot Linux kernel + fdt with ramdisk for recovery";
         kernel = "kernel@1";
         ramdisk = "ramdisk@1";
         fdt = "fdt@1";
      };
   };
};
```

demo 2: Build the FIT

```
# mkimage -f kernel_fdt_rd.its kernel_fdt_rd.itb
FIT description: Simple image with single Linux kernel and FDT blob
Created:         Sun Feb  3 17:56:05 2013
 Image 0 (kernel@1)
    .. ..
 Image 1 (fdt@1)
    .. ..
 Image 2 (ramdisk@1)
  Description:  recovery ramdisk
  Type:         RAMDisk Image
  Compression:  gzip compressed
  Data Size:    2022580 Bytes = 1975.18 kB = 1.93 MB
  Architecture: ARM
  Hash algo:    sha1
  Hash value:   2bc8b8e2064e2c0ab72dd214996c50fc2b0549da
 Default Configuration: 'defaultconf@1'
 Configuration 0 (defaultconf@1)
  Description:  Boot Linux kernel with FDT blob
  Kernel:       kernel@1
  FDT:          fdt@1
 Configuration 1 (recoveryconf@1)
  Description:  Boot Linux kernel with ramdisk for recovery and FDT blob
  Kernel:       kernel@1
  Init Ramdisk: ramdisk@1
  FDT:          fdt@1
```

demo 2: Somebody yanked the MMC card

Lets Boot the recovery configuration

```
fitfdt=/boot/kernel_fdt_rd.itb
setenv loadaddr 0x82000000;
run ramargs;
ext2load mmc ${mmcdev}:2 ${loadaddr} ${fitfdt};

bootm ${loadaddr}#recoveryconf;




/* Booting the default conf */
bootm ${loadaddr}#defaultconf;
```

Bootlog of U-Boot booting the #recoveryconf

```
U-Boot# run fitrdboot
4876960 bytes read in 980 ms (4.7 MiB/s)
## Booting kernel from FIT Image at 82000000 ...
   Using 'recoveryconf@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  Vanilla Linux kernel
     Type:         Kernel Image
     .. ..
## Loading init Ramdisk from FIT Image at 82000000 ...
   Using 'recoveryconf@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  recovery ramdisk
     Type:         RAMDisk Image
     Compression:  gzip compressed
     Data Start:   0x822b8a1c
     Data Size:    2022580 Bytes = 1.9 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x00000000
     Entry Point:  0x00000000
     Hash algo:    sha1
     Hash value:   2bc8b8e2064e2c0ab72dd214996c50fc2b0549da
   Verifying Hash Integrity ... sha1+ OK
```

Bootlog of U-Boot booting the #recoveryconf

```
## Flattened Device Tree from FIT Image at 82000000
   Using 'recoveryconf@1' configuration
   Trying 'fdt@1' FDT blob subimage
.. ..
OK
   kernel loaded at 0x80008000, end = 0x802bddd0
   Loading Ramdisk to 8fc5b000, end 8fe48cb4 ... OK
   Loading Device Tree to 8fc55000, end 8fc5a947 ... OK

Starting kernel ...

[    1.599982] VFS: Mounted root (ext2 filesystem) on device 1:0.
[    1.607883] devtmpfs: mounted
[    1.611581] Freeing init memory: 248K
Please press Enter to activate this console.

[root@arago /]#
[root@arago /]#
[root@arago /]#
[root@arago /]#
```

More use cases of FIT

Debug vs Production Kernel

- Multiple kernels one with maybe debug options enabled, one normal.
- both have their own configuration nodes in the FIT.
- User boot a #debugkernel for debugging and a #production for production

A multiplatform Kernel image

• Multiple DTBs/configuration nodes embedded in a FIT; U-Boot reads EEPROM boots correct "configuration".

• multibooting same image on different boards.

Another real world usecase…. Verified boot by Simon Glass

```
/ {
        images {
                kernel@1 {
                        data = /incbin/("...");
                        type = “kernel";
                        arch = "arm";
                        os = "linux";
                        compression = "none";
                        load = <0x111>;
                        entry = <0x222>;
                        kernel-version = <1>;
                        hash@1 {
                                algo = "sha1";
                                value = <....>;
                        };
                signature@1 {
                        algo = "sha1,rsa2048";
                        key-hint = "dev";
                        description = “Dev-signed kernel 3.8.0-33, snow FDT”;
                        signer = “mkimage”;
                        signer-version = “ v2013.01”;
                        value = <....>;
                };
                signature@2 {
                        algo = "sha1,rsa2048";
                        key-hint = “production";
                        description = “Dev-signed kernel 3.8.0-33, snow FDT”;
                        signer = “mkimage”;
                        signer-version = “ v2013.01”;
                        value = <....>;
                };};};
```

Just showing how flexible the image format is that one could extend it easily for a usecase that wasn’t even thought off! With very little “hack” code.

And extended even more for better security.. Signed configurations.

What if someone uses the same signed images, but changes the configuration?

```
configurations {
        default = "conf@1";
        conf@1 {
                kernel = "kernel@1";
                fdt = "fdt@1";
                signature@1 {
                        algo = "sha1,rsa2048";
                        key-name-hint = "dev";
                        sign-images = "fdt", "kernel";
                };
        };
};
```

A potential target for FIT
P2020RDB board – booting an AMP configuration

- Freescale P2020 dual core SoC
- Currently tedious pulls same kernel twice
- Pulls 2 DTBs
- Passes the right DTB to each kernel (i2c bus differences etc)
- Very good usecase for FIT- roll all the AMP kernels and device trees necessary into one FIT image

And even more uses!

- **Upgrade procedures for devices**, where the vendor wants to be able to distribute a single file for his target systems to avoid customers bricking their devices by choosing incompatible combinations.

Future work and challenges

- Need a simple way to extend the “make dtbs” target.
- Probably easier to FIT patches for Kbuild accepted than was before.
- Challenges in the community, U-boot hate
- U-boot currently requires a loadaddr for FIT (fix has been POC’d)
- Hardware accelerator support for Crypto operations
- Questions?

Thanks to

- Simon Glass
- Peter Tyser
- Wolfgang Denx