Flattened Image Trees:
A powerful kernel image format

Feb 21, 2013

Joel A Fernandes <joelagnel@ti.com>

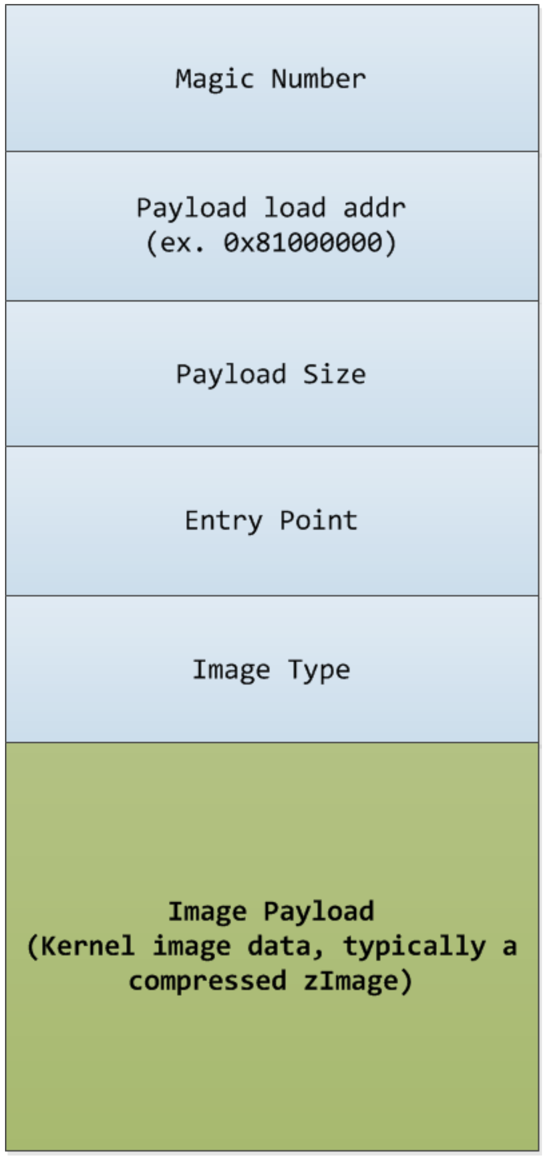
Goals of this talk

- 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

Single Component Images

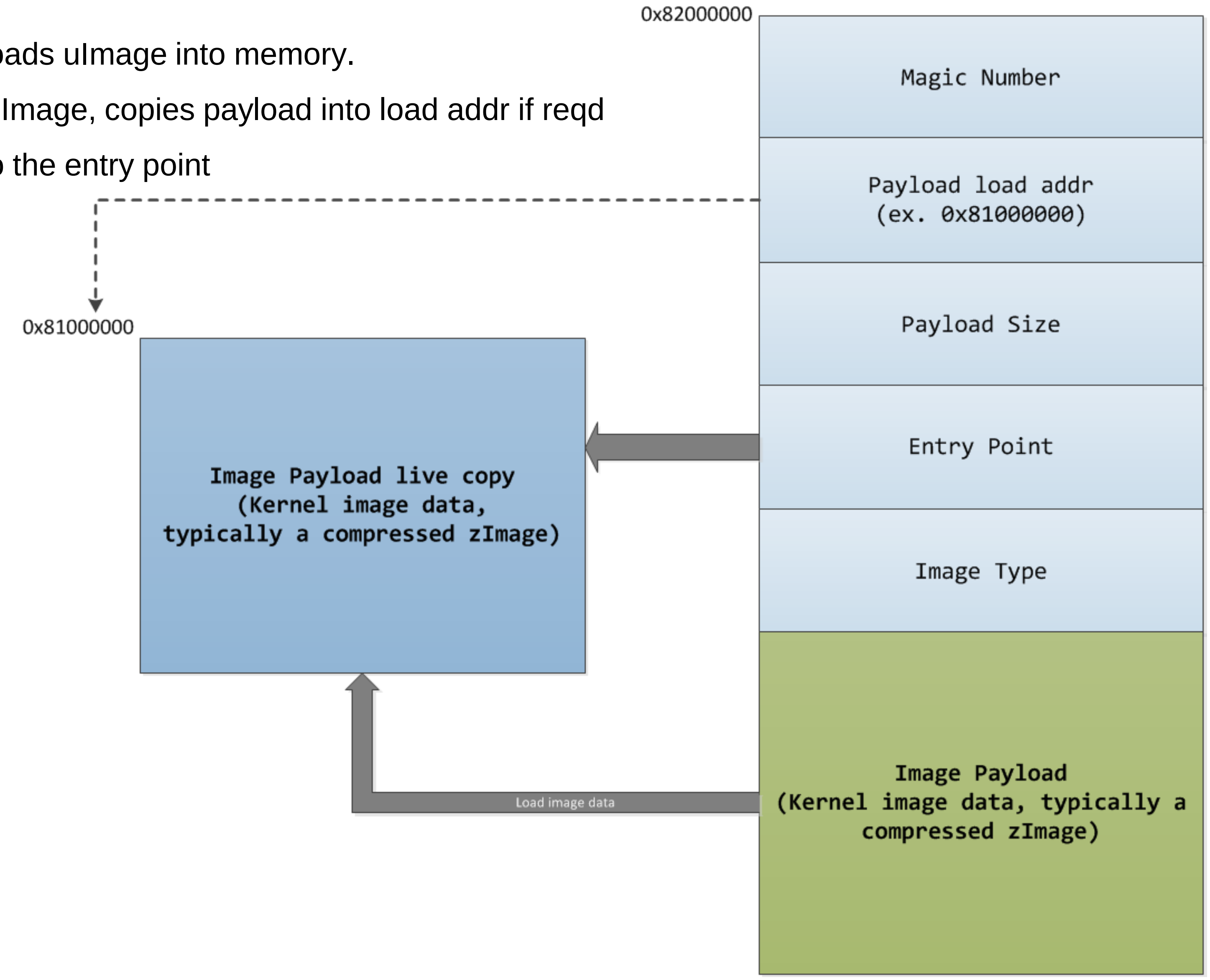
Structure of a Single Component Image

- 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


Magic Number
Payload load addr (ex. 0x81000000)
Payload Size
Entry Point
Image Type
Image Payload (Kernel image data, typically a compressed zImage)

Booting of a Single Component Image

- U-boot loads uImage into memory.
- Parses uImage, copies payload into load addr if reqd
- Jumps to the entry point
- Bootm



0x82000000
Magic Number
Payload load addr (ex. 0x81000000)
Payload Size
Entry Point
Image Type
Image Payload (Kernel image data, typically a compressed zImage)
0x81000000
Image Payload live copy (Kernel image data, typically a compressed zImage)
Load image data

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 Images

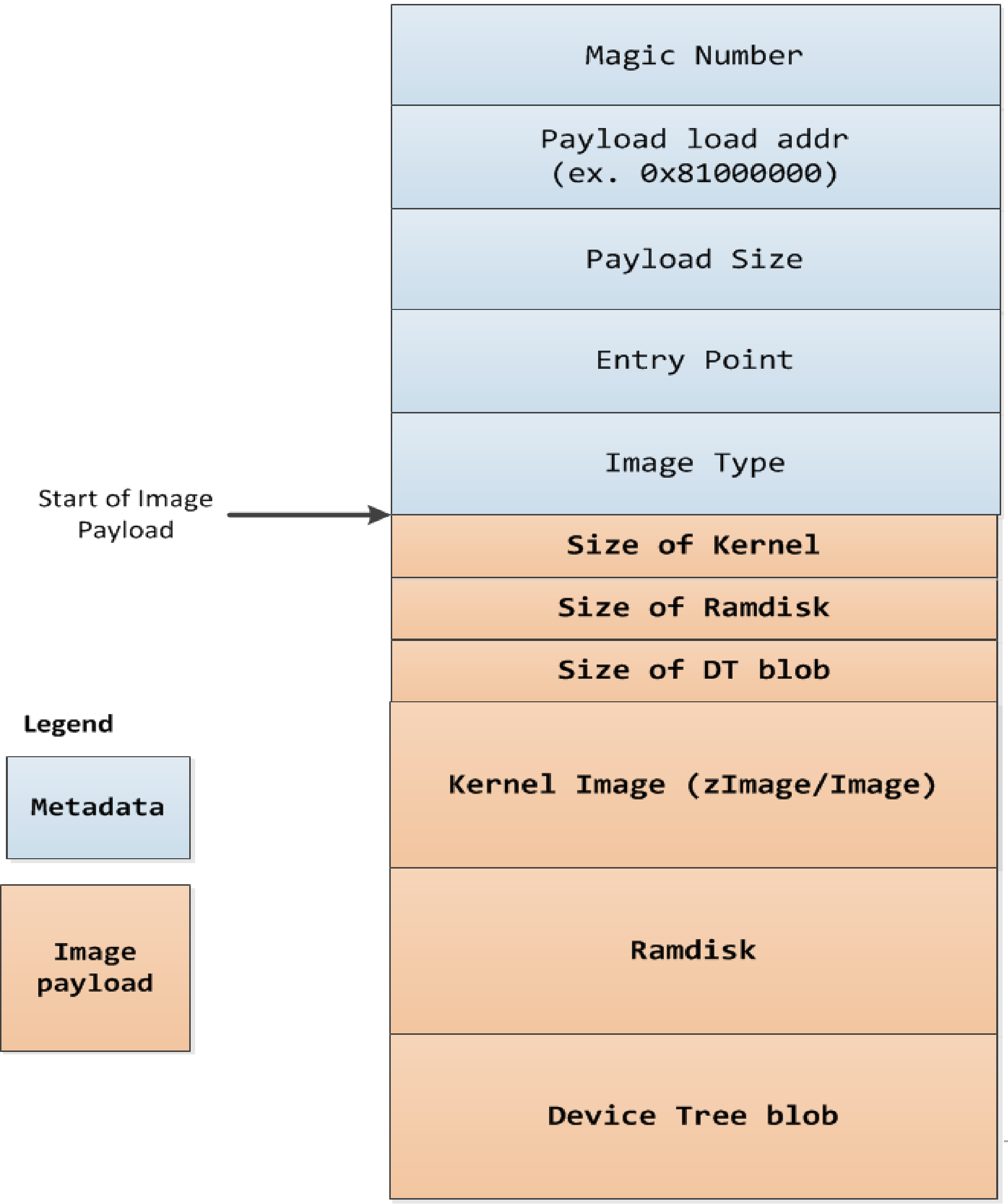
Single Component Image limitations

- 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
  - Some users found it necessary to have more than 1 component
  - Recovery of systems- where you want an initrd to give you an FS
  - Firmware ugrade where it is not easy or clean to download multiple components
  - Security- sometimes folks want to include cryptographic signatures.

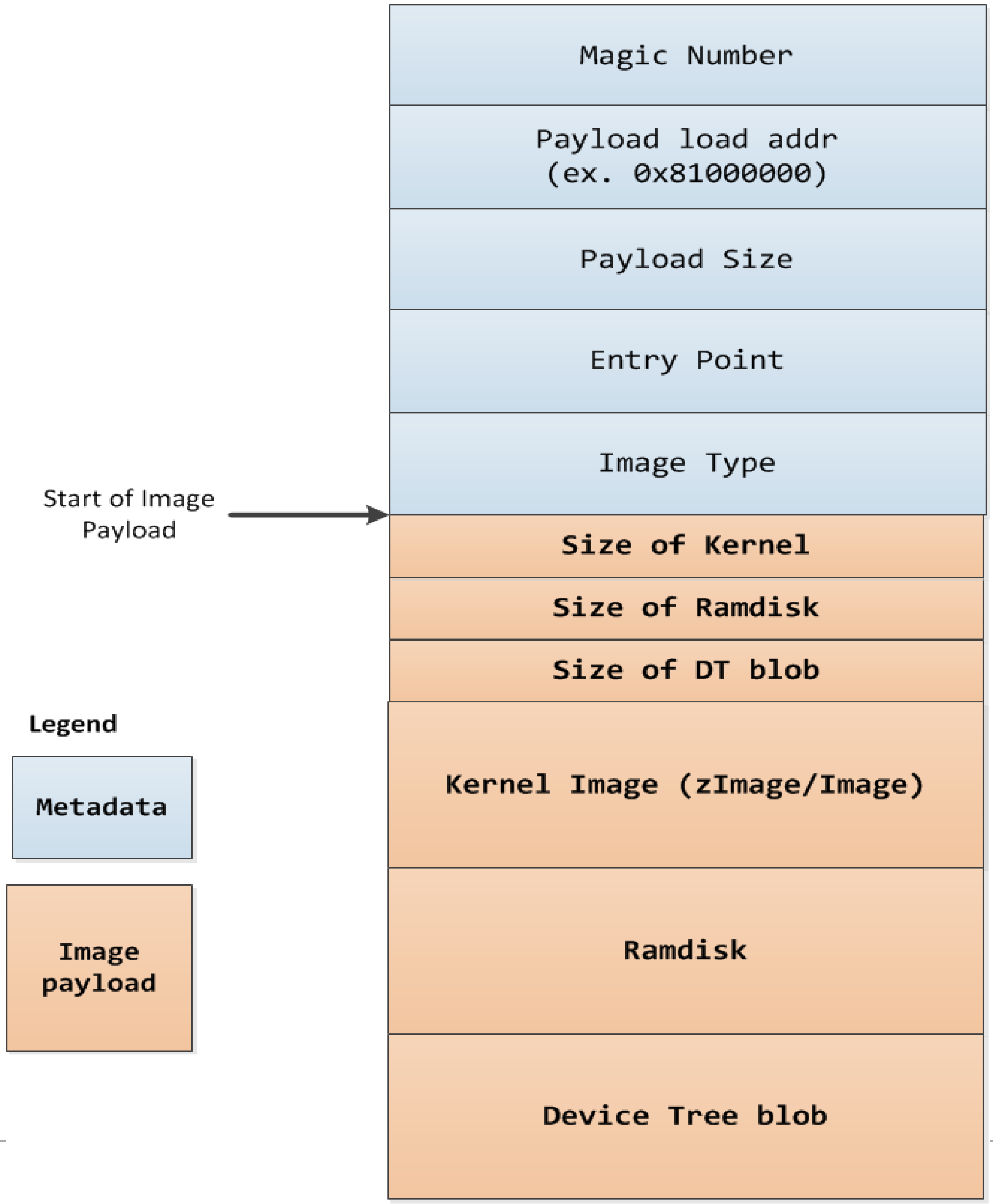
- A new image type in the “single-component” image header was introduced, called IH_MULTI which were supposed to have additional components in the image payload

Structure of a Mutli Component Image



- Embed multicomponents by Shoehorning of 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



- Each entry in the table was hard-coded to a particular pre-defined component. table id 1 was ramdisk, id 2 was chosen for device tree blob.
- Fixed mapping of id to component type. Ramdisk can’t be pushed after DT blob
- Worked.. But has drawbacks, more on that next..

Several problems with this approach..

- shoehorning meta-data into payload is not a clean method. Payload should not have to contain meta-data about an image. That's supposed to go in the headers..
- The meta-data stored in MC was limited.. No provision to load a component of the Image into a particular location of memory. Unlike the kernel which could be loaded to a particular memory address before being executed.
  - Which meant all other components had to be executed in-place.
- Hardcoding of indices of image components in the code. Remember I was talking about id 1 being kernel, id 2 being ramdisk etc.
  - Associating numbers instead of names to image components is messy and not-so-obvious about what index corresponds to what image. The 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

Several problems with this approach..

- No provision to add a component other than kernel, ramdisk, and single DT blob to a multi-component Image
  - 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?

- Sometimes one might want multiple kernel components in an image, and I'd like to select one particular kernel for debug for example, and one during a production boot. How can we represent a structure like this in a Multi-component image?

- Nice approach but doesn't scale for future designs and encourages introduction of more hacks.

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

- An image format that makes use of DT to build an image format in the format of a device 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.

Architectures and Platforms using FIT

PowerPC:

- XPedite1000 board running the PPC 440GX Embedded Processor
- MPC8544 (power pc arch) 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

Coreboot-x86:

- Acer Chromebook with Intel Celeron

Other:

Microblaze softcpu core from Xilinx

The appended DT hack to embed DTB in kernel

- Many users prefer to have DT blob embedded into kernel specially when they don't care much about multiplatform case
- 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.
- Only one DT can be appended, unlike FIT. So really makes the image a single-platform one.
- No kernel support still to build a boot loader image that has a DT appended to it. There are hacks floating that need to be applied. Rightly so… such a patch would encourage single-platform kernel

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…

One could have multiple kernels one with maybe debug options enabled, one for production. They could both have their own configuration nodes in the FIT

Then the user could boot a #debugkernel for debugging and a #production configuration for production… all using the same FIT image.

A multiplatform Kernel image

• Multiple DTBs can be embedded in a FIT; each board/platform can have their own configuration node that has their own DTB. U-boot can read the EEPROM on boards, and boot the right “configuration” node like the earlier example.

• Can combine multiple kernel images, device tree blobs and root file system images in **arbitrary combinations**; this allows for example for multibooting the same image on different boards by selecting the right DTB.

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";
                };
        };
};
```

And even more uses!

- Assume you want to **boot over DHCP** or similar, where you can provide just a single image file for download. Here it is definitely nice if you can bundle the kernel image and the DTB into one image file.

- **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..

- Kernel build support