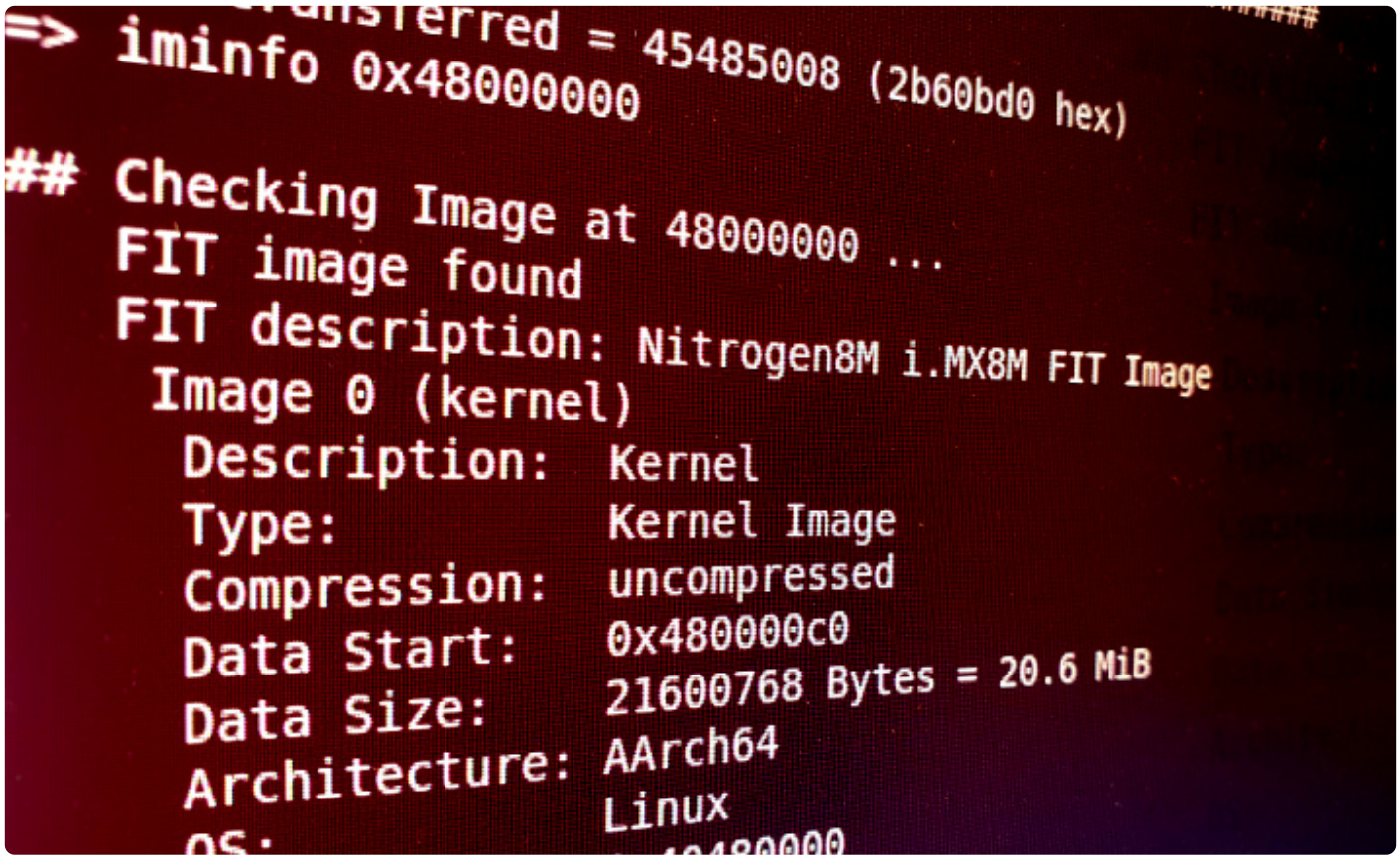


Updated on March 11, 2021 / Andrew Murray

Flattened uImage Tree (FIT) Images

Tech Blog | UBoot | Yocto

You're probably familiar with the steps required to boot Linux from U-Boot: you first load several binaries into memory, perhaps a device tree, a kernel, maybe even an initrd. You then invoke a command such as bootm or booti with arguments providing memory addresses for the binaries you've just loaded. However there is a much better way – in this post we're going to explore Flattened Image Tree (FIT) Images and show you the benefits of using them.

Let's start with the typical steps required to TFTP boot Linux on a Nitrogen8M i.MX8M reference board with an initrd.

```
=> tftp 0x40480000 Image
=> tftp 0x43000000 imx8mq-nitrogen8m.dtb
=> tftp 0x44000000 core-image-minimal-nitrogen8m.ext2.gz.u-boot
=> booti 0x40480000 0x44000000 0x43000000
```

With this approach you have to explicitly determine a load address for each component and ensure the binaries don't overlap in memory. You'll also need to make similar considerations if you have an update mechanism or store the binaries in raw flash. It's a little inconvenient and easy to make a mistake.

Fortunately, the U-Boot community extended their uImage format (typically used for wrapping kernels and ram disks with a header so that U-Boot knew what to do with it) so that it could support multiple binaries (IH_TYPE_MULTI) in a single uImage. Let's use the mkimage utility to create an multi-file uImage:

```
$ mkimage -C none -A arm64 -O linux -T multi -a 0x40480000 -e 0x40480000 -d
Image Name:
Created:      Sat Mar  6 22:04:41 2021
Image Type:   AArch64 Linux Multi-File Image (uncompressed)
Data Size:    33569915 Bytes = 32783.12 KiB = 32.01 MiB
Load Address: 40480000
Entry Point:  40480000
Contents:
   Image 0: 21600768 Bytes = 21094.50 KiB = 20.60 MiB
   Image 1: 11912092 Bytes = 11632.90 KiB = 11.36 MiB
   Image 2: 57039 Bytes = 55.70 KiB = 0.05 MiB
```

Notice how we provided multiple binaries separated by colons in the '-d' argument and set the image type as 'multi'. Let's boot it:

```
=> tftp 0x43000004 mImage ; bootm 0x43000004
Using FEC device
TFTP from server 192.168.1.198; our IP address is 192.168.1.224
Filename 'mImage'.
Load address: 0x43000004
Loading: #################################################################
         9.7 MiB/s
done
Bytes transferred = 33569979 (2003cbb hex)
## Booting kernel from Legacy Image at 43000004 ...
   Image Name:
   Image Type:   AArch64 Linux Multi-File Image (uncompressed)
   Data Size:    33569915 Bytes = 32 MiB
   Load Address: 40480000
```

```
   Entry Point:  40480000
   Contents:
      Image 0: 21600768 Bytes = 20.6 MiB
      Image 1: 11912092 Bytes = 11.4 MiB
      Image 2: 57039 Bytes = 55.7 KiB
   Verifying Checksum ... OK
## Loading init Ramdisk from multi component Legacy Image at 43000004 ...
## Flattened Device Tree from multi component Image at 43000004
   Booting using the fdt at 0x0000000044ff5df0
   Loading Multi-File Image
ERROR: reserving fdt memory region failed (addr=b8000000 size=400000)
   Using Device Tree in place at 0000000044ff5df0, end 0000000045006cbe

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.98-2.0.0-ga+yocto+gd18974845042 (oe-user@o
MP PREEMPT Fri Mar 5 17:26:18 UTC 2021
```

We now have a single multi-file uImage encapsulating all of our boot binaries. One additional benefit to this approach is that there is now checksum validation for our entire uImage.

Unfortunately this approach doesn't quite provide the flexibility we need. A requirement of ARM64 platforms is that the device tree must be located on an 8 byte boundary – unfortunately IH_TYPE_MULTI images don't provide any support for specifying a load address or alignment constraints. In fact, our boot was only successful because we tweaked the address of where we TFTP'd the uImage to until the device tree address inside it lined up. This is an example of the lack of flexibility offered by IH_TYPE_MULTI images which resulted in the Flattened uImage Tree (FIT) format being born.

Rather than reinvent the wheel the FIT format makes use of the libfdt library and the device tree compiler (dtc) – essentially a FIT image is a device tree blob (DTB) with your binaries embedded inside – but represented in a structure that allows you to provide additional information. To create a FIT image you create an 'Image Tree Source' file (akin to a DTS file) – and pass this to the mkimage command which internally uses

the dtc to construct a FIT image (akin to a DTB). Let's try this out by creating an image tree source file (.its):

```
/dts-v1/;

/ {
        description = "Nitrogen8M i.MX8M FIT Image";
        #address-cells = <1>;

        images {
                kernel {
                        description = "Kernel";
                        data = /incbin/("Image");
                        type = "kernel";
                        arch = "arm64";
                        os = "linux";
                        compression = "none";
                        load = <0x40480000>;
                        entry = <0x40480000>;
                        hash {
                                algo = "sha1";
                        };
                };
                fdt {
                        description = "DTB";
                        data = /incbin/("imx8mq-nitrogen8m.dtb");
                        type = "flat_dt";
                        arch = "arm64";
                        compression = "none";
                        load = <0x43000000>;
                        entry = <0x43000000>;
                        hash {
                                algo = "sha1";
                        };
                };
                initrd {
                        description = "Initrd";
                        data = /incbin/("core-image-minimal-nitrogen8m.ext2.
                        type = "ramdisk";
                        arch = "arm64";
                        os = "linux";
                        compression = "none";
                        hash {
```

```
                                algo = "sha1";
                        };
                };
        };

        configurations {
                default = "standard";
                standard {
                        description = "Standard Boot";
                        kernel = "kernel";
                        fdt = "fdt";
                        ramdisk = "initrd";
                        hash {
                                algo = "sha1";
                        };
                };
        };

};
```

As you can see we use a tree-like structure to describe the binaries that are to be contained in our FIT image. The sub-nodes of the 'images' node describes the individual binaries and make use of the /incbin/ directive to include those binaries. Each of the 'images' sub-nodes have some common properties including a type (e.g. kernel, flat_dt, ramdisk), information on the compression used (so that U-Boot can decompress) and hash information (such that U-Boot can perform verification). You can find more information about the supported properties and nodes here. You can see that we've added a load/entry address to the device tree sub-image to ensure that it is aligned correctly. There is also a 'configurations' node which we'll come onto shortly – however let's create a FIT image from this as follows:

```
$ mkimage -f nitrogen8M.its nitrogen8M.itb
FIT description: Nitrogen8M i.MX8M FIT Image
Created:         Wed Mar 10 00:36:52 2021
 Image 0 (kernel)
  Description:  Kernel
  Created:      Wed Mar 10 00:36:52 2021
  Type:         Kernel Image
  Compression:  uncompressed
  Data Size:    21600768 Bytes = 21094.50 KiB = 20.60 MiB
```

```
  Architecture: AArch64
  OS:           Linux
  Load Address: 0x40480000
  Entry Point:  0x40480000
  Hash algo:    sha1
  Hash value:   f6bfbebc237be13320a205a74db7d7c5ceca6823
 Image 1 (fdt)
  Description:  DTB
  Created:      Wed Mar 10 00:36:52 2021
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    57039 Bytes = 55.70 KiB = 0.05 MiB
  Architecture: AArch64
  Load Address: 0x43000000
  Hash algo:    sha1
  Hash value:   6ab42f1887c609b9a90bfab77718cc07c1d831b0
 Image 2 (initrd)
  Description:  Initrd
  Created:      Wed Mar 10 00:36:52 2021
  Type:         RAMDisk Image
  Compression:  uncompressed
  Data Size:    11912441 Bytes = 11633.24 KiB = 11.36 MiB
  Architecture: AArch64
  OS:           Linux
  Load Address: unavailable
  Entry Point:  unavailable
  Hash algo:    sha1
  Hash value:   e92aea4e83d07f1d0e6e03b485f89088d9a98e99
 Default Configuration: 'standard'
 Configuration 0 (normal)
  Description:  Standard Boot
  Kernel:       kernel
  Init Ramdisk: initrd
  FDT:          fdt
  Hash algo:    sha1
  Hash value:   unavailable
```

We now have an ‘itb’ file (also know as a FIT image) which we can boot on our board – let’s try it:

```
=> tftp 0x48000000 nitrogen8M.itb
Using FEC device
TFTP from server 192.168.1.198; our IP address is 192.168.1.224
```

```
Filename 'out.itb'.
Load address: 0x48000000
Loading: ##################################################################
         10.5 MiB/s
done
Bytes transferred = 33572236 (200458c hex)
=> bootm 0x48000000
## Loading kernel from FIT Image at 48000000 ...
   Using 'standard' configuration
   Verifying Hash Integrity ... OK
   Trying 'kernel' kernel subimage
     Description:  Kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x480000c0
     Data Size:    21600768 Bytes = 20.6 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x40480000
     Entry Point:  0x40480000
     Hash algo:    sha1
     Hash value:   f6bfbebc237be13320a205a74db7d7c5ceca6823
   Verifying Hash Integrity ... sha1+ OK
## Loading ramdisk from FIT Image at 48000000 ...
   Using 'standard' configuration
   Verifying Hash Integrity ... OK
   Trying 'initrd' ramdisk subimage
     Description:  Initrd
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x494a7b3c
     Data Size:    11912441 Bytes = 11.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
     Hash algo:    sha1
     Hash value:   e92aea4e83d07f1d0e6e03b485f89088d9a98e99
   Verifying Hash Integrity ... sha1+ OK
## Loading fdt from FIT Image at 48000000 ...
   Using 'standard' configuration
   Verifying Hash Integrity ... OK
   Trying 'fdt' fdt subimage
     Description:  DTB
```

```
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x49499b9c
     Data Size:    57039 Bytes = 55.7 KiB
     Architecture: AArch64
     Load Address: 0x43000000
     Hash algo:    sha1
     Hash value:   6ab42f1887c609b9a90bfab77718cc07c1d831b0
   Verifying Hash Integrity ... sha1+ OK
   Loading fdt from 0x49499b9c to 0x43000000
   Booting using the fdt blob at 0x43000000
   Loading Kernel Image
ERROR: reserving fdt memory region failed (addr=b8000000 size=400000)
   Using Device Tree in place at 0000000043000000, end 0000000043010ece

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.98-2.0.0-ga+yocto+gd18974845042 (oe-user@o
[    0.000000] Boot CPU: AArch64 Processor [410fd034]
[    0.000000] Machine model: Boundary Devices i.MX8MQ Nitrogen8M
```

As you can see we were able to construct a single image that contained all of our binaries which is booted with a simple command.

Let's take another look at that 'configurations' node:

```
        configurations {
                default = "standard";
                standard {
                        description = "Standard Boot";
                        kernel = "kernel";
                        fdt = "fdt";
                        ramdisk = "initrd";
                        hash {
                                algo = "sha1";
                        };
                };
        };
```

The FIT image can contain lots of binaries, more binaries than are needed for a single boot – thus this required 'configurations' node lets you group related binaries together. In our case we've created a configuration called 'standard' and indicated that it consists of a kernel, device tree and initrd. It's possible to create multiple configurations and to choose which configuration to use at boot time. This is where we begin to see the power that FIT images provide.

Let's see what happens if we add an additional sub-node of type 'ramdisk' named 'initrdRecovery' and add an additional configuration named 'recovery' that uses this initrd instead of the one used earlier. Let's tftp this new FIT image and run the U-Boot utility 'iminfo' to see what this FIT image contains:

```
=> tftp 0x48000000 nitrogen8M.itb
=> iminfo 0x48000000
## Checking Image at 48000000 ...
   FIT image found
   FIT description: Nitrogen8M i.MX8M FIT Image
    Image 0 (kernel)
     Description:  Kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x480000c0
     Data Size:    21600768 Bytes = 20.6 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x40480000
     Entry Point:  0x40480000
     Hash algo:    sha1
     Hash value:   f6bfbebc237be13320a205a74db7d7c5ceca6823
    Image 1 (fdt)
     Description:  DTB
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x49499b9c
     Data Size:    57039 Bytes = 55.7 KiB
     Architecture: AArch64
     Load Address: 0x43000000
     Hash algo:    sha1
     Hash value:   6ab42f1887c609b9a90bfab77718cc07c1d831b0
    Image 2 (initrd)
```

```
     Description:  Initrd
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x494a7b3c
     Data Size:    11912441 Bytes = 11.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
     Hash algo:    sha1
     Hash value:   e92aea4e83d07f1d0e6e03b485f89088d9a98e99
    Image 3 (initrdRecovery)
     Description:  Recovery Initrd
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x4a00410c
     Data Size:    11912441 Bytes = 11.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
     Hash algo:    sha1
     Hash value:   e92aea4e83d07f1d0e6e03b485f89088d9a98e99
    Default Configuration: 'standard'
    Configuration 0 (standard)
     Description:  Standard Boot
     Kernel:       kernel
     Init Ramdisk: initrd
     FDT:          fdt
     Hash algo:    sha1
     Hash value:   unavailable
    Configuration 1 (recovery)
     Description:  Standard Boot
     Kernel:       kernel
     Init Ramdisk: initrdRecovery
     FDT:          fdt
     Hash algo:    sha1
     Hash value:   unavailable
## Checking hash(es) for FIT Image at 48000000 ...
   Hash(es) for Image 0 (kernel): sha1+
   Hash(es) for Image 1 (fdt): sha1+
   Hash(es) for Image 2 (initrd): sha1+
   Hash(es) for Image 3 (initrdRecovery): sha1+
```

Here we can see the additional initrd and configuration. We can choose which configuration we wish to boot by specifying it via the bootm command:

```
=> bootm 0x48000000#recovery
## Loading kernel from FIT Image at 48000000 ...
   Using 'recovery' configuration
   Verifying Hash Integrity ... OK
   Trying 'kernel' kernel subimage
     Description:  Kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x480000c0
     Data Size:    21600768 Bytes = 20.6 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x40480000
     Entry Point:  0x40480000
     Hash algo:    sha1
     Hash value:   f6bfbebc237be13320a205a74db7d7c5ceca6823
   Verifying Hash Integrity ... sha1+ OK
## Loading ramdisk from FIT Image at 48000000 ...
   Using 'recovery' configuration
   Verifying Hash Integrity ... OK
   Trying 'initrdRecovery' ramdisk subimage
     Description:  Recovery Initrd
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x4a00410c
     Data Size:    11912441 Bytes = 11.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
     Hash algo:    sha1
     Hash value:   e92aea4e83d07f1d0e6e03b485f89088d9a98e99
   Verifying Hash Integrity ... sha1+ OK
## Loading fdt from FIT Image at 48000000 ...
   Using 'recovery' configuration
   Verifying Hash Integrity ... OK
   Trying 'fdt' fdt subimage
     Description:  DTB
     Type:         Flat Device Tree
     Compression:  uncompressed
```

```
     Data Start:   0x49499b9c
     Data Size:    57039 Bytes = 55.7 KiB
     Architecture: AArch64
     Load Address: 0x43000000
     Hash algo:    sha1
     Hash value:   6ab42f1887c609b9a90bfab77718cc07c1d831b0
   Verifying Hash Integrity ... sha1+ OK
   Loading fdt from 0x49499b9c to 0x43000000
   Booting using the fdt blob at 0x43000000
   Loading Kernel Image
ERROR: reserving fdt memory region failed (addr=b8000000 size=400000)
   Using Device Tree in place at 0000000043000000, end 0000000043010ece
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
```

You can see from the above output that we specified the 'recovery' configuration and that's what U-Boot loaded (e.g. notice the line "Using 'recovery' configuration").

Configurations offer lots of flexibility – you could distribute a single FIT image, yet can use that in different ways, e.g. a production/debug build, a normal/recovery build, an image with lots of DTBs which can be used on multiple hardware platforms, etc.

We've only scratched the surface and there is a lot more to love – here are some additional capabilities of FIT images:

- FIT images can contain signatures for both individual binaries and configurations – these signatures can be created by mkimage and are verified by U-Boot. Thus aiding a secure boot – read more via this LWM article, this excellent presentation and this documentation.

- FIT images can also contain arbitrary firmware – for example you may want to use it to provide an FPGA image for U-Boot to load – take a look at the imxtract command.

- Yocto can generate FIT images for you – we achieved this by adding the following to our local.conf, however you may also want to read the source as there are many variables that can have an effect.

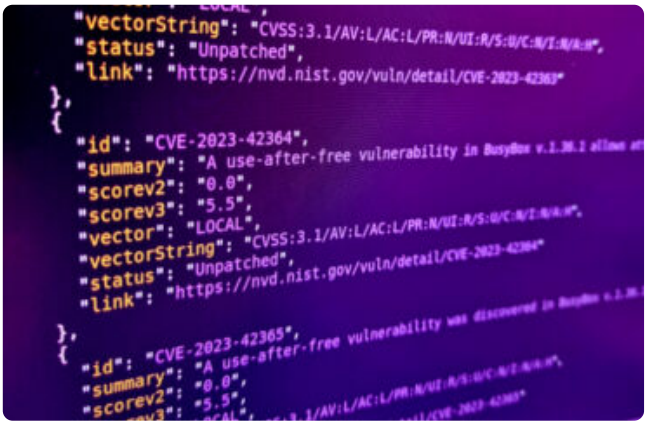
    - KERNEL_CLASSES = "kernel-fitimage"

    - KERNEL_IMAGETYPES = "fitImage"

FIT images offer many features and provide more flexibility. Given its integration with Yocto, it's also really easy to begin using it. It's certainty something to be considered for new projects and for implementations of secure boot.

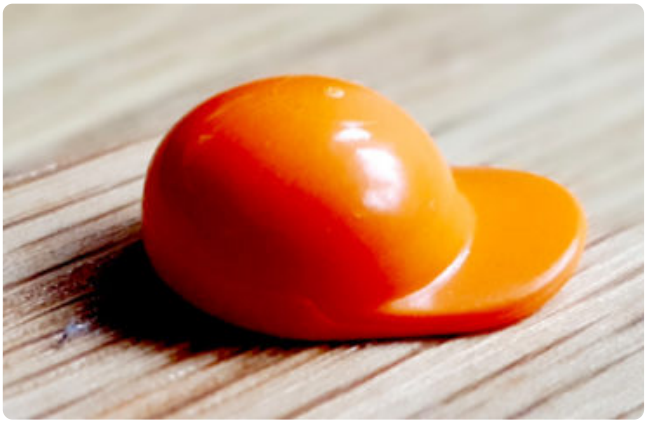
.itb .its FIT fitImage i.MX8 IH_Multi iminfo imxtract mkimage Nitrogen8M Secure Boot UBoot uImage Verified Boot Yocto

Popular Posts



Handling Security Vulnerabilities in Yocto Scarthgap

Booting ARM without an ARM



Discovering CPU features from userspace with ELF_HWCAP

i.MX6 UL Bus Encryption Engine (BEE) in the Linux Kernel