

Flattened Device Trees for Embedded FreeBSD

Rafał Jaworowski

Semihalf, The FreeBSD Project

raj@{semihalf.com, freebsd.org}

Abstract

This paper describes the development work on providing FreeBSD with the ability to use Flattened Device Tree (FDT) technology, which allows for describing hardware resources of a computer system and their dependencies in a platform-neutral and portable way.

Configuration data which cannot be self-discovered at boot-time, have to be supplied from an external source. The concept of flattened device trees is a platform and architecture independent approach to resolve such problems. The concept is inherited from Open Firmware IEEE 1275 device-tree notion, and has been successfully adopted by the embedded industry.

1 Introduction

In embedded world there is a great variety of systems based on similar silicon chips, but designed into custom boards and devices, where connections of individual components are different and there are no conventions or rules, even across members of the same family of products, for the interconnections layout and resources allocation. Furthermore, some buses and interconnects are not self-enumerable by definition (unlike PCI or USB), and there has to be some prior knowledge about how they are connected and what their unique identification is. Some of the examples of typical problems are the following:

- Memory layout (address offsets/sizes specification)
- Assignment of resources and identifica-

tion of non-enumerable devices (I^2C , SPI buses, internal on-chip resources)

- Network MAC-PHY binding
- Interrupts hierarchy and routing
- GPIO / multi-purpose pin routing and assignment

The problem is that very often embedded systems early-stage bootloaders tend to be simple and do not expose information like the above to the operating system kernel. The concept of a flattened device tree (FDT) is an established and mature way of handling such problems. Among other deployments it has been adopted as a basis for Power.org's embedded platform reference specification [EPAPR].

The idea is inherited from Open Firmware (OF) IEEE 1275 device-tree notion (part of the regular Open Firmware implementation), but it allows for using the device tree mechanism on any system (not based on OF):

- Hardware platform resources are *manually* described in a human readable text source format, where all non self-enumerating info is gathered
- This source description is converted (*compiled*) into a binary object (a flattened device tree blob), which is passed to the kernel at boot time
- The kernel (driver) learns about hardware resources details and dependencies from this [externally supplied] blob, which eliminates the need for embedding any info

about the underlying platform hardware resources in the kernel

- The flattened device tree mechanism in principle does not depend on any particular first-stage bootloader or firmware features. The only overall requirement for the environment is to provide a complete device tree description to the kernel

With such approach the operating system kernel, in particular device drivers, but also other low-level routines, can be freed from any *a priori* knowledge about the hardware resources underneath, and can be made flexible and scalable depending only on the externally supplied configuration data.

2 Definitions

The flattened device tree technology is associated with is a comprehensive environment, so before further discussion we explain the major entities.

2.1 Device tree source (DTS)

The device tree source is a text file which describes hardware resources of a computer system in a human-readable form. Example below shows a device tree source snippet featuring description of all major components (CPU, memory, system-on-chip peripherals, IRQ assignments etc.) as device tree nodes and their properties:

```
...
cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    PowerPC,8555@0 {
        device_type = "cpu";
        reg = <0x0>;
        d-cache-line-size = <32>;
        i-cache-line-size = <32>;
        d-cache-size = <0x8000>;
        i-cache-size = <0x8000>;
        timebase-frequency = <0>;
        bus-frequency = <0>;
        clock-frequency = <0>;
        next-level-cache = <&L2>;
    };
};
```

```
};

memory {
    device_type = "memory";
    reg = <0x0 0x8000000>;
};

soc8555@e0000000 {
    #address-cells = <1>;
    #size-cells = <1>;
    device_type = "soc";
    compatible = "simple-bus";
    ranges = <0x0 0xe0000000 0x100000>;
    bus-frequency = <0>;

    i2c@3000 {
        #address-cells = <1>;
        #size-cells = <0>;
        cell-index = <0>;
        compatible = "fs1-i2c";
        reg = <0x3000 0x100>;
        interrupts = <43 2>;
        interrupt-parent = <&mpic>;
        dfsrr;
    };

enet0: ethernet@24000 {
    #address-cells = <1>;
    #size-cells = <1>;
    cell-index = <0>;
    device_type = "network";
    model = "TSEC";
    compatible = "gianfar";
    reg = <0x24000 0x1000>;
    ranges = <0x0 0x24000 0x1000>;
    local-mac-address = [ 00 00 00 00 00 00 ];
    interrupts = <29 2 30 2 34 2>;
    interrupt-parent = <&mpic>;
    tbi-handle = <&tbi0>;
    phy-handle = <&phy0>;
};
...
};
```

In FreeBSD the default location for the DTS files is `sys/boot/fdt/dts` directory of the source tree.

2.2 Device tree blob (DTB)

The textual device tree description (DTS file) is first converted (*compiled*) into a binary object (the device tree blob) i.e. the DTB,

which is handed over to the final consumer (typically kernel) for parsing and processing of its contents.

2.3 Device tree compiler (DTC)

A stand-alone tool executed on the host, which transforms (*compiles*) a textual description of a device tree (DTS) into a binary object (DTB).

2.4 Device tree bindings

While the device tree textual description and the binary object are media to convey the hardware configuration info, an actual meaning of the contents are driven by the device tree *bindings*. They are certain conventions describing definitions (encoding) of particular nodes in a device tree and their properties, allowed values, ranges and so on. Such reference conventions provide the legacy Open Firmware bindings, further supplemented by the ePAPR specification [EPAPR].

3 Integration with FreeBSD

Since the FDT framework is composed of a number of various elements, the development work towards bringing this technology to the FreeBSD environment also spans several aspects. The major areas involved with the project are the following, and they will be discussed in greater details further on:

- Tools enabling the technology (dtc, libfdt)
- loader(8) support
- FreeBSD kernel support
- Long term development and maintenance of the device tree sources (DTS) for individual platforms

3.1 Baseline code

The starting point for this development was publicly available source code of the FreeBSD 9-CURRENT branch as of November 2009 timeframe. During development it was a regular practice to re-sync with an up-to-date baseline.

3.2 Build environment

A non modified tool chain bundled with the base FreeBSD (GNU binutils 2.15 and gcc 4.2.1) was used for this project.

4 Tools

When considering FDT support on any platform there are needed basic building blocks and the primary element is a *device tree compiler* tool, which lets transform the textual description into a binary object used by an end-consumer.

Since there is available a ready to use implementation of such a compiler, namely the *dtc*, developed and maintained by David Gibson and Jon Loeliger, it was natural to reuse this existing 3rd party tool for the project [DTCGIT].

Moreover, part of the *dtc* package is a stand-alone supporting library, *libfdt*, which greatly helps integrate parsing and processing of a device tree blob into any piece of software, which is supposed to retrieve data from the device tree. In case of FreeBSD environment, the *libfdt* library is used by the following entities:

- *dtc* user space tool
- last stage bootloader i.e. *loader(8)*
- FreeBSD kernel

More information about the *dtc*, *libfdt*, device tree blob format and other low-level details can be found in the [DTC PAPER].

Therefore, as a preliminary step for the development described in this paper, the *dtc* package was integrated with base FreeBSD source tree, so that both compiler and the library could be built as any other element of the system. A new build knob was added to control building of these tools when FDT support is enabled:

```
WITH_FDT
```

The typical way of building the device tree compiler is supplying the above set-

ting during buildworld procedure, either via `src.conf(5)` or manually:

```
# make buildworld -DWITH_FDT
```

Note the `dtc` tool has to be built at bootstrap tools building stage of the buildworld procedure, because the host is not guaranteed to have the compiler installed and readily available. This is similar to other elementary tools required during further stages of the build process (like `config`, `make` and others).

5 loader(8)

Systems with regular booting environment¹ include the native FreeBSD last-stage boot loader i.e. `loader(8)`, which is responsible for preparing the environment for the FreeBSD kernel before it is loaded and started.

Considering FDT support for such systems, `loader(8)` has been extended with the ability to load and manage the device tree blob, and finally hand it over to the kernel. The device tree cannot be synthesized by the `loader(8)` itself in run-time, because the loader is very much unaware of the hardware it is running on. The device tree blob has to be created beforehand and it is retrieved from permanent storage (typically `/boot` directory) at boot time.

In order to leverage existing FreeBSD booting environment, the device tree blobs are treated as raw binary *kernel modules*, which can be loaded and unloaded before the kernel is booted, in a similar way that any other kernel modules work. For example (note the `-t dtb` type specification of the module):

```
loader> load -t dtb boot/mpc8555cds.dtb
```

```
loader> lsmod
```

```
...
0x162f92c: boot/mpc8555cds.dtb (dtb, 0x1eb2)
loader>
```

In addition to using a generic mechanism for loading a device tree blob, a new dedicated

¹For discussion about booting scenarios without `loader(8)` see 6.5.1

loader command, `fdt`, has been added, to allow users inspect and manipulate the loaded blob in a number of ways. The list of available subcommands is the following:

```
fdt cd
fdt header
fdt ls
fdt mknnode
fdt mkprop
fdt prop
fdt pwd
fdt rm
```

Detailed discussion of the command usage can be found in `loader(8)` manual, but as could be inferred from the subcommands' names the user can navigate through the device tree hierarchy, as well as change its contents on demand (add, delete new nodes and properties, modify existing ones), for example:

```
loader> fdt ls
```

```
/aliases
/cpus
/cpus/PowerPC,8555@0
/memory
/soc8555@e0000000
/soc8555@e0000000/ecm-law@0
/soc8555@e0000000/ecm@1000
/soc8555@e0000000/memory-controller@2000
/soc8555@e0000000/l2-cache-controller@20000
/soc8555@e0000000/i2c@3000
/soc8555@e0000000/dma@21300
/soc8555@e0000000/dma@21300/dma-channel@0
/soc8555@e0000000/dma@21300/dma-channel@80
/soc8555@e0000000/dma@21300/dma-channel@100
/soc8555@e0000000/dma@21300/dma-channel@180
/soc8555@e0000000/ethernet@24000
...
/soc8555@e0000000/ethernet@25000
...
/soc8555@e0000000/serial@4500
/soc8555@e0000000/serial@4600
/soc8555@e0000000/crypto@30000
/soc8555@e0000000/pic@40000
...
/pci@e0008000
/pci@e0008000/i8259@19000
/pci@e0009000
```

As mentioned, FDT support in the FreeBSD loader primarily relies on the *libfdt* component, which provides basic routines for traversing the the device tree blob and managing its contents in memory. Built on this foundation is functionality of all *fdt* loader subcommands.

6 FreeBSD kernel

Because of the many different aspects of the FDT environment, bringing support for this functionality to the FreeBSD kernel is a challenging and complex undertaking spanning several subsystems. Some of the more prominent areas are highlighted below:

- Early system initialization rework to a device tree-driven model
- Integration with an existing Open Firmware framework
- Integration with FreeBSD native *newbus* device drivers hierarchy
- Conversion of individual drivers to the new conventions

- modes of booting - stand alone blob with loader - statically embedded blob into kernel image (no loader)

6.1 Basic functionality

The groundwork for the flattened device tree support in the kernel was integrating the *libfdt* library and making it part of the FreeBSD kernel image, so that data can be retrieved from the device tree and used during kernel bootstrap. This step was straightforward because *libfdt* is portable and easy to embed in other code.

6.2 Open Firmware infrastructure

FreeBSD *PowerPC*² and *Sparc64* architecture support code has been using *genuine*³ Open Firmware services from the very beginning, since this (OF) is a common case of early-stage boot loader found on these platforms.

Flattened device tree is inherited from Open Firmware device-tree notion and shares many of its design principles, so there was an incentive to reuse existing OF platform code of the FreeBSD kernel. Open Firmware however is primarily a specification, implemented by a number of vendors, and some of the internal differences have to be accounted for in the consumer code. This led to development of generic, virtualized interfaces in the kernel providing uniform access to OF services and resources regardless of the internal implementation differences:

- *OFW_** interface, provides low-level access to Open Firmware API calls. The interface hides from the user such implementation details like CPU mode of operation while executing OF code (as opposed to the consumer code) etc.
- *OFW_BUS_** interface, allows for easier retrieval of standard device node properties and translates for the user OF device tree representation to internal *newbus* device kernel objects.

Part of the OF infrastructure is also the */dev/openfirm* character device, which allows user space access to Open Firmware services.

FDT support infrastructure plugs into the OF framework in the following way:

- Provides back-end implementation of

²Using OF on PowerPC mostly concerns legacy Apple Macintosh systems. The emebded PowerPC systems very often use other boot loaders like U-Boot, for which FDT is a viable technique to provide kernel with hardware configuration data.

³Since some aspects of FDT technology is derived from or directly reusing OF concepts, we make an explicit distinction between these two approaches whenever important.

`OFW_*` methods for the device-tree retrieval, which use DTB underneath as a data source⁴.

- Uses the `OFW_BUS_*` methods to simplify nodes and properties management in higher level FDT infrastructure code.

In effect, from the consumer (client code) perspective, an FDT-driven system appeals as a genuine Open Firmware as far as device tree data retrieval is concerned.

As a demonstration of the powerful interface virtualization approach, let's bring an existing user space program, `ofwdump(8)`, which allows for enumeration of the OF device tree hierarchy: it works without modifications on FDT-enabled platforms, without even knowing there is no genuine OF underneath:

```
# ofwdump -a
Node 0xc06309a0:
  Node 0xc0630a04: aliases
  Node 0xc0630b04: cpus
    Node 0xc0630b30: PowerPC,8555@0
  Node 0xc0630bec: memory
  Node 0xc0630c24: soc8555@e0000000
    Node 0xc0630cac: ecm-law@0
    Node 0xc0630cfc: ecm@1000
    ...
  Node 0xc0630ea4: i2c@3000
  Node 0xc0630f40: dma@21300
    Node 0xc0630fd8: dma-channel@0
    Node 0xc0631074: dma-channel@80
    Node 0xc0631110: dma-channel@100
    Node 0xc06311ac: dma-channel@180
  Node 0xc063124c: ethernet@24000
    Node 0xc0631360: mdio@520
    Node 0xc06313c4: ethernet-phy@0
    Node 0xc063143c: ethernet-phy@1
    Node 0xc06314b4: tbi-phy@11
  Node 0xc0631504: ethernet@25000
    Node 0xc0631618: mdio@520
    Node 0xc0631678: tbi-phy@11
  Node 0xc06316c8: serial@4500
  Node 0xc063175c: serial@4600
  Node 0xc06317f0: crypto@30000
  Node 0xc0631898: pic@40000
```

⁴Note that non-device-tree `OFW_*` methods (e.g. device I/O, memory management) from this interface are not implemented and return error when called.

```
Node 0xc0631930: cpm@919c0
  Node 0xc06319a8: muram@80000
    Node 0xc06319f0: data@0
    Node 0xc0631a40: brg@919f0
    Node 0xc0631aa8: pic@90c00
  Node 0xc0631b58: pci@e0008000
    Node 0xc0631fa8: i8259@19000
  Node 0xc0632068: pci@e0009000
#
```

6.3 Integration with *newbus*

The FreeBSD kernel infrastructure for managing device drivers is an object-oriented framework called *newbus*. It models hardware peripherals of a computer system as a hierarchical tree with an abstract root device on top and many subordinate entities below. Hardware components are represented by object instances, which have drivers attached to them during kernel bootstrap.

It should be noted that *newbus* infrastructure is fundamental to device drivers interface definition of FreeBSD and any significant changes to its core functionality would have thorough impact on much of the kernel code. Therefore, bringing flattened device tree as a source of hardware description information for the FreeBSD kernel, requires smooth integration with *newbus* primitives and overall model.

Prior to the development described in this paper, various embedded FreeBSD architecture support implementations typically rolled out their own *newbus*-based representation of integrated peripherals, for example *obio(4)*, *ocp-bus(4)*, *mbus(4)* and so on.

In order to reduce these many similar (and incompatible) approaches, as part of the FDT kernel infrastructure, we introduce two generic and common *replacement* entities. They are both *newbus* abstract bus drivers, which do not represent any physical elements, but provide a connection between *newbus* and OF-like device tree model:

- *fdtbus(4)*
- *simplebus(4)*

Figure 1 illustrates the concept of device drivers hierarchy, with *fdtbus* and *simplebus* entities visible in the middle.

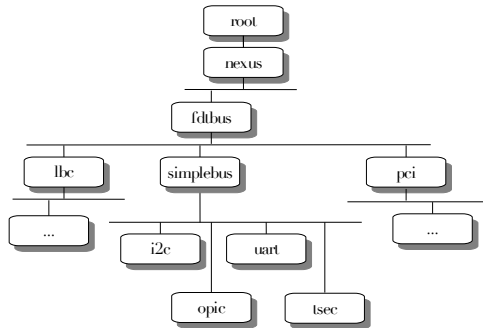


Figure 1: **Device drivers hierarchy**

This is directly reflected in the kernel view of the devices organization:

```

# devinfo
nexus0
  fdtbus0
    lbc0
      cfi0
        cfid0
    simplebus0
      i2c0
        iicbus0
          iic0
      ...
      tsec0
        miibus0
          ciphy0
      ...
      uart0
      uart1
      sec0
        openpic0
    pcib0
      pci0
        pcib1
          pci2
    pcib2
    pcib3
      pci1
        em0
#

```

6.3.1 fdtbus(4)

This abstract bus driver, newly introduced as part of the project, is the focal point of integration between flattened device tree world and FreeBSD *newbus* scheme. It is a direct replacement for various existing bus drivers representing peripherals integrated on chip. Its main responsibilities are:

- Creating *newbus children* that reflect FDT nodes counterparts
- Managing IRQ resources
- Managing MEM, I/O resources

The *fdtbus* driver provides generic, common infrastructure for all FDT-oriented device drivers. It iterates through the flattened device tree and for each first-level node⁵:

- instantiates a *newbus child* representing the node,
- retrieves resources info (memory ranges, IRQs, etc.) from the device tree blob and assigns to this *newbus child* in the form of a FreeBSD-native device resource list.

6.3.2 simplebus(4)

This bus driver is dedicated for an ePAPR-style „simple-bus” node [EPAPR], which is an umbrella node grouping integrated on-chip peripherals like interrupt controller, connectivity controllers, accelerating engines and so on.

Since the *simplebus* is akin to the *fdtbus* in that it does not represent any physical entity by itself, its driver is also generic and common for all nodes claiming „simple-bus” compatibility. In a similar way it iterates over direct descendants of the „simple-bus” node, instantiates *newbus children* accordingly and assigns resources to them, which individual device drivers can request during probing.

⁵descendants of the FDT *root* node

Note the *simplebus* does not manage device resources and passes through any requests to the *fdtbus* layer.

6.4 Conversion of existing drivers

Besides providing the infrastructure building blocks for FDT, the existing end device drivers probe and attachment code need to be adjusted to work with the new conventions.

With *fdtbus* and *simplebus* in place⁶, end device (non-bus) drivers can be made FDT-compatible fairly straightforward. Resources management does not change from the device driver perspective, so the basic conversion steps are the following:

1. Declare the driver as a child of the proper bus driver, for example:

```
DRIVER_MODULE(openpic, simplebus,  
              openpic_fdt_driver,  
              openpic_devclass, 0, 0);
```

2. Have the probe routine check for *compatible* property of the node, for example:

```
...  
if (!ofw_bus_is_compatible(dev,  
                            "chrp,open-pic"))  
    return (ENXIO);  
...
```

The above are a minimal requirement, and additional adjustments may be necessary depending on how much information a device driver has to fetch from the FDT. However, in case of simple device drivers it may be all that is needed to migrate over to the FDT approach.

Bus drivers like PCI or other (localbus type) are usually more difficult and non routine, and hence it is not easy to provide a general description of such cases, they are considered outside the scope of this paper.

⁶and with the assumption the device tree data for a particular system has already been prepared

6.5 Using FDT with FreeBSD

6.5.1 Modes of operation

When considering high-level FDT usage scenarios there appear two basic modes of operation, which depend on how the device tree blob is handled on the way to the kernel:

- stand-alone DTB file (i.e. with *loader(8)*)
- statically embedded DTB into kernel image (no *loader(8)*)

Since the first case was discussed in section 4 the details of the second scenario are described below.

6.5.2 Statically embedded blob

Some of the embedded systems cannot run the FreeBSD *loader(8)* either because it is not feasible⁷ or not desired (for example to shorten boot time).

In order for such cases to still benefit from the FDT technology, the device tree blob has to be included as integral part of the kernel image. This way the kernel is self-contained as the device tree is buried within its data segment, and it can be used for retrieval and processing exactly as when it is supplied as a stand-alone file by *loader(8)*.

6.5.3 Kernel options

The primary option for enabling the FDT support in the kernel is the following:

```
options          FDT
```

It includes all low-level and infrastructure parts of FDT kernel support, which primarily cover the *fdtbus(4)* and *simplebus(4)* drivers, as well as helper *libfdt* routines.

To specify a preferred (default) device tree source (DTS) file for a given kernel use the following build option:

⁷because the underlying firmware does not expose any programmatic access to elementary operations such as console, block device I/O or networking, which are required for *loader(8)* to work

```
makeoptions    FDT_DTS_FILE=board.dts
```

The indicated DTS file will be converted (compiled) into a binary form along with the kernel itself (note the DTS file name is relative to the default location of DTS sources i.e. `sys/boot/fdt/dts`).

Additionally, in order to statically embed a DTB file into a kernel image use the following option:

```
options        FDT_DTB_STATIC
```

Note this option requires a DTS file to be specified with the `FDT_DTS_FILE` makeoption.

6.6 Current support

6.6.1 Freescale MPC85xx

Among the early adopting platforms of the FDT technology for FreeBSD was existing MPC85xx PowerPC⁸ support which covers the family of Freescale *PowerQUICC* system-on-chip devices. The legacy *ocpbus(4)* bus driver representing integrated peripherals on these systems was removed⁹, and instead the generic FDT infrastructure (like *fdtbus(4)* and *simplebus(4)* bus drivers) was used.

- DTS files for reference development systems (MPC8555CDS and MPC8572DS), compliant with the ePAPR specification [EPAPR], were provided by the silicon vendor and were used for FreeBSD purposes with only very minor adjustments and extensions.
- All existing FreeBSD drivers for MPC85xx integrated peripherals were converted to the FDT conventions (interrupt controller, UART, Ethernet, crypto engine, PCI etc.)

6.6.2 Marvell Orion, Kirkwood, Discovery Innovation

In addition to the embedded PowerPC examples described in the previous section,

⁸based on Book-E compatible E500 core

⁹and the static hard-coded configuration tables in the kernel

a family of ARM-based system-on-chip devices were also brought to run FreeBSD with the FDT convention during the course of this project. Similarly, an existing legacy representation of integrated peripherals and resources, *mbus(4)*, was removed in favor of the FDT generic infrastructure and the newly introduced bus drivers.

- Since enabling FDT on ARM-based system was a pioneer work (no other OS was offering such support at the time of this development¹⁰, important part of this project was creating a set of DTS description files for all platforms based on Marvell ARM, which are supported by current FreeBSD.
- This covers for the following: *Orion* (DB-88F5182, DB-88F5281), *Kirkwood* (DB-88F6281, RD-88F6281, SheevaPlug) and *Discovery Innovation* (DB-78100).
- The introduced DTS files are reusing existing *ePAPR bindings* definitions, but there were some new device tree *bindings* proposed as well for device drivers and areas not previously covered by the *ePAPR* specification [EPAPR].
- All existing FreeBSD drivers for Marvell integrated peripherals were converted to the FDT conventions.

7 Summary

7.1 Benefits

The technology introduced by this project brings several advantages to the embedded FreeBSD world:

- Uniform and extensible way of representing hardware devices compliant with industry standards (ePAPR, Open Firmware), independent of architecture and platform (portable across ARM, MIPS, PowerPC, possibly others).

¹⁰Q1 2010

- Allows for and encourages code sharing and its reduction in long-term (as the common infrastructure is used by more and more platforms in favor of the legacy, incompatible but similar solutions).
- Providing hardware configuration from external source allows for multi-platform kernels (a single kernel image can be used with many configurations and hardware platform variations).

7.2 Cost

There are also some disadvantages, or maintenance costs associated with the device tree mechanism:

- Device tree compiler (*dtc*) package dependency, which needs to be included in the base source tree, followed by its maintenance like importing and updates. This seems not a significant obstacle though: the *dtc* package is stable, without dynamic development at the moment.
- Maintenance of device tree source (DTS) files; this seems like a task similar to the legacy *device.hints(5)* files maintenance and handling.

8 Acknowledgments

I would like to thank the following people:

The FreeBSD Foundation for sponsoring this development.

M. Warner Losh (The FreeBSD Project), for being the technical reviewer for the project.

Phil Brownfield (Freescale), for help and support with relicensing the MPC85xx device tree source files.

Łukasz Wójcik and Michał Hajduk (both Semihalf), for all the work on this project.

Nathan Whitehorn (The FreeBSD Project), for the work on *OF* and *OFW* interfaces, which allowed the FDT layer to smoothly plug into the Open Firmware infrastructure and benefit from existing framework and tools.

Work on this paper was sponsored by Semihalf.

9 Availability

The code described in this paper is, or will soon be, available from the FreeBSD Project Subversion repository, 9-CURRENT (HEAD) branch, and later.

References

[ASOF] Josh Boyer, Grant Likely, *A Symphony of Flavours: Using the device tree to describe embedded hardware*, Linux Symposium 2008, Ottawa

[EPAPR] Power.org, Inc., *Standard for Embedded Power Architecture™ Platform Requirements (ePAPR)*, Rev. 1.0, 23 July 2008

[DTC PAPER] David Gibson, Benjamin Herrenschmidt, *Device trees everywhere*, February 2006

[DTC GIT] David Gibson, Jon Loeliger, *The Device Tree Compiler source tree* <http://git.jdl.com/gitweb/?p=dtc.git;a=summary>