Proposed new DTB format

Last update: 2018-11-12
First proposed: 2018-10-28
Author: Simon Glass <sjg@chromium.org>
Reviewer: <add yourself here>
Status: IN REVIEW
canonical link: https://goo.gl/4GCyXK

Objective

- Reduce the size of the binary device tree representation
- Support faster flattree traversal
- Support storage of type information
- Consider adding support for other useful features (e.g. overlays)

Background

The device tree binary format is currently at version 17 which became the default version in 2007. Since then source language has been enhanced, e.g. to support different data sizes and allowing phandles to be referenced as a parent node. Many conventions have been added as well, such as those needed to support overlays. But there have been no changes to the binary format in this time.

The cost of changing to a new format is high:

- Agreeing what features should be supported, possibly an insurmountable task
- Required code changes to dtc and libfdt
- API changes to libfdt as needed
- Updating things like Linux and U-Boot to work with the new libfdt

The long-term stability of the current format is a significant asset to the device tree format. Changing it should not be undertaken lightly. It could be easier to change it in several small steps, each separately motivated by the desire to solve a particular problem. However it makes sense to

This document aims to provide some ideas along with some analysis of their impact in terms of binary size and code size.

Experimental approach

The ideas described in this document were tested by hacking up dtc and libfdt.

All experiments use Linux's device tree binaries for ARM devices (arm and arm64 architectures). This consists of 1267 files totalling about 32MB of binary .dtb data. Sizes of the individual files range from 8KB (zynq-zybo-z7.dtb) to 83KB (dra7-evm.dtb).

For comparison, compressing all the .dtb files with xz results in about 5.4MB of output. The source files (as produced by fdtdump, so excluding comments) total about 67MB but compress down to a similar size (5.8MB). This suggests that the entropy in the overall data is only about 17% of the .dtb size. This is not surprising for a format which is presumably designed for simplicity and ease of run-time access.

Design Ideas

Each change is presented here along with the results obtained.

A. Do nothing: status quo, version 17 format

This is the baseline. Data size is 100%, no code changes.

B. Single-cell property metadata

The v17 format has three cells for the property tag, length and name offset. This change fits this information into a single cell, thus saving two cells for each property.

With this change overall size reduces to 72.6%.

C. Use a value table

With v17 property values are added directly after the property metadata. There is no mechanism to take advantage of properties having the same value. This change adds a value table (similar to the current string table) which holds unique values. It is assumed that some bits of the 32-bit property tag can be used to store the location of the value in the value table.

With this change overall size reduces to 97.8%.

D. In-place byte

It is common to have a property with a single cell containing a single-byte value (i.e. the cell value is 0 to 0xff). This change stores the byte value in with the property tag, thus avoiding needing to store the four-byte cell.

With this change overall size reduces to 94.9%.

E. Narrow u32

It is common to have properties where each cell stores a value in the range 0 to 0xff, effectively wasting the other 3 bytes of each cell. This change stores such values as bytes instead.

The correct value must be recreated in libfdt at runtime, which requires libfdt to have access to some memory to use for this purpose. This change is therefore not very attractive.

With this change overall size reduces to 94.6%.

F. Avoid strings in pinctrl

Several SoCs use strings for pin names in their pinctrl driver. We can imagine changing the bindings for these SoCs to use phandles instead, as do many other SoCs. This change converts the string list in the following properties[1] to a zero-argument phandle list:

- marvell,pins
- nvidia,pins
- samsung,pins
- sirf,pins
- st,pins

With this change overall size reduces to 99.7%. Most files are not affected at all. If only exynos*.dts files are considered, the reduction is to 98.3%.

G. Replace compatible strings with u32

A compatible string is included with most nodes and can be quite long. This changes replaces this with a single cell encoding the vendor and device in a 32-bit value.

With this change overall size reduces to 92.4%.

H. Drop FDT_END_NODE

This tag can sometimes be replaced with a one-bit flag on the last property in the node, so long as there are not two such tags in a row. This introduces some painful special cases in the libfdt code. For example deleting a property can result in needing to add the flag to another property, or if there are no more properties, adding back anm FDT_END_NODE tag.

With this change overall size reduces to 97.6%.

I. Skip alignment to cell boundaries

With v17 all tags and property values are aligned to a cell boundary. This makes it easy to read values from the device tree without worrying about unaligned values. Of course in the case of little-endian machines, we have to do a byteswap anyway. This change drops all alignment, so that tags and values can start on any byte boundary.

With this change overall size reduces to 97.3%.

J. Use a byte-wise protocol

Emit tags and metadata as bytes, where bits 6:0 contain 7 bits of the value and bit 8 is 1 if the value continues into the next byte. This means that small values (like tags) consume only a single byte of space, but values which use all 32-bits of a cell need 5 bytes.

[1] Obtained with grep ",pins =" *.dts |grep \" |awk '{print $2}' |sort |uniq -c

This implies no alignment.

With this change overall size reduces to 97.3%.

An extension of this is J', to consider all values as strings of cells and use the bytewise protocol to emit them. This is a little odd, but is included for completeness.

Summary

Idea	**Data size (% of baseline)**	**Saving %**	**Affects**
A. Status quo, v17 format	100	0	
B. Single-cell property metadata	72.6	27.4	fdt_get_property…()
C. Use a value table	97.8	2.2	
D. In-place byte	94.9	5.1	
E. Narrow u32	94.6	5.4	fdt_getprop()
F. Avoid strings in pinctrl	99.7	0.3	Bindings
G. Replace compatible strings with u32	92.8	7.2	fdt_node_offset_by_compatible( ), etc.
H. Drop FDT_END_NODE	97.6	2.4	
I. Skip alignment to cell boundaries	97.3	2.7	Caller assumptions / code
J. Bytewise protocol	79.3	20.7	Caller assumptions / code
J'. Bytewise values too	83.4	16.6	Also fdt_getprop()
B + D + G	60.4	39.6	
B + C + D + G	59.2	40.8	
B + C + D + G + I	58.0	42.0	

Most of the above have some impact on the libfdt API. It is desirable to minimise this by avoiding options which radically change the API. Option B is clearly important for size reduction, but it does prevent use of struct fdt_property (and struct fdt_node_header which is not actually used in the API).

Since the value table (C) saves only 2.2% it does not seem worth the extra code cost. Option D does not require an API change.

Changing compatible strings provides a good win but introduces an API change to the compatible functions. This requires substantial code changes in clients.

Overall B + D + G seems like a reasonable option. It has some API impact, but nothing too great. Option G can be supported in addition to compatible strings. Clients can move over to it when desired.

Detailed design

At present when a device tree is displayed (e.g. with fdtdump or 'dtc -O dts') the output of value data generally does not mirror the input. For example the tools have to guess whether the data is a list of strings or a just set of unstructured bytes. This generally works fairly well, but it is just a heuristic.

Clients cannot obtain information about the data types provided by the property. This is defined by bindings, but there is no formal binding languages (schema) that can be read programmatically.

This section explores a possible design which supports type information, supporting the following:

- Data type: string, phandle, 8/16/32/64 int, blobs (raw bytes) or a combination
- Signed / unsigned
- Decimal / hex
- List / scalar
- Comment - a text message to help display the output
- External - a way of stored large blobs outside the device tree

Tags currently occupy a single cell. This is changed to store additional information encoded in bits.

Properties tag

Bits	**0**	**1-3**	**4**	**5**	**6**	**7**	**8-15**	**16-31**
Meaning	p (1)	type	l	s/e/i	c	d	len / val	str

- p - 1 means this is a property tag
- type - 0=string, 1=phandle, 2=multiple, 3=8bit, 4=16bit, 5=32bit, 6=64bit, 7=blob
- l - 0 for scalar, 1 for list
- s - 0 for unsigned, 1 for signed
- e - ('blob' type only) 0 for internal data, 1 for external (stored outside the device tree structure)
- i - ('byte' type only) 0 for data following tag, 1 for data encoded in val
- c - 1 to include a text comment
- d - 0 for hex, 1 for decimal
- len - length of property (0-0xfe), 0xff to store length in a separate cell
- str - string table offset for property name (0xffff to store in a separate cell)

For data stored in separate cells the ordering is:

tag	**len**	**str**	**mult_tag...**	**data**
tag (as above)	Optional length cell	Optional string offset cell	Optional multiple-type tags	Data goes here

All data types except phandle are stored as raw data starting on a cell boundary. This corresponds to the v17 format. Phandles are stored in a different format and are no-longer intended to be decoded without the help of libfdt:

For phandles, the format is:

Bits	**0-3**	**4**	**5**	**6-15**	**16-31**
Meaning	n	f	unused	phandle 0:9	offset 2:17

- n - Number of arguments (15 max, more must be stored as a 32bit list)
- f - 1 if the offset is valid, 0 if not valid
- phandle - value of phandle: 0 to 0x3fe, 0x3ff to store in the next cell
- offset - offset of the node the phandle points to (if f=1), excluding bits 0:1

Phandles values are stored as the tag followed by its 0-15 arguments:

args_tag	arg0	arg1	args_tag	arg0
args tag (as above)	argument 0	argument 1	next phandle	Argument 0

If desired we can support a 'multiple' data type. With this a multi_tag is provided which encodes the type information for each part of the value. Each cell holds an element of the data:

Bits	**0**	**1-3**	**4**	**5**	**6**	**7**	**8-15**	**16-31**
Meaning	m	type	l	s	c	d	item_len	unused

- m - 1 if there is another tag after this one, 0 if this is the last
- item_len - length of this item in bytes (0-0xfe), 0xff to store length in a separate cell

The 'multiple'-type property may contain any number of these cells, each one possibly followed by a length cell if item_len = 0xff. This allows the data to be more fully described, if desired. For example it is possible to indicate that a property contains a byte followed by a string. This is likely not useful since it is generally better to put the data in two separate properties.

Non-property tag

Bit	**0**	**1-4**	**5**	**6-15**	**16-31**
Meaning	p (0)	t	c	skip	str

- p - 0 means this is a non-property tag
- t - tag type - 1=begin_node, 2=end_node, 4=nop, 5=delete_node, 6=merge_node, 7=list element, 9=end
- c - 1 to include a text comment
- skip - stores the offset to related tags, in units of cells (0 = none):
    - For begin_node: stores the offset to the next begin_node tag at the same level
    - For end_node: stores the offset to end_node tag just before the previous begin_node tag at the same level

- str - string table offset for name (must fit in 16 bits) Store in-place

Related Work

https://elinux.org/Device_tree_future

https://www.slideshare.net/superna/linux-conference-australia-2018-device-tree-past-present-future

https://www.devicetree.org

Feedback

If you read this document please provide your short general feedback in the section below.

Username	Date	Comment