YAML and Devicetree

October 21, 2017/0 Comments/in Technical /by Pantelis Antoniou

Introduction

This document attempts to explain the rationale behind using a YAML based data model instead of the standard devicetree source (DTS). It assumes a working knowledge of devicetree, so readers are expected to have perused the devicetree specification located here.

Devicetree and its underlying concepts

While device tree deals with describing hardware devices, at its core it is a method of declaring a hierarchical structure as defined in the Devicetree Specification:

“A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.”

This structure is familiar to anyone with a passing knowledge of programming languages with rich data structures: nodes can be hashes keyed by their name, properties can be either scalars or sequences of scalars, and labels of nodes and phandles can be references/pointers.

Unfortunately, device tree is not orthogonal enough for this mapping to work. Namely, properties are irregular in the following ways:

1. Boolean values cannot be part of a sequence, since a named property is defined as false if it doesn't exist in a node and as true otherwise.

2. Phandles are encoded as integer (cell) scalar values and are allowed in any property that contains cell values.

3. While properties are defined either as a single value or a sequence of values, their type information is thrown away. The importance of this is that the property accessors must have an out-of-band way to be informed of the type(s) used in the property, i.e. property type information is not discoverable.

Lifecycle of a Devicetree.

The purpose of the device tree is (or at least was until recently) to be provided to an operating system at boot time. This was done by the following steps.

1. Device Tree source files (DTS) are processed by a compiler to generate an in-memory tree structure. This structure is dynamically created at compile time by editing operations of the the compile tree sources which are:
    - Device tree sources are usually now pre-processed using the C proprocessor, but the built-in source include directive is still supported. Note that mixing them is permitted although it can lead to the unexpected behaviour of the base source file being preprocessed while the included one is not.
    - Declaration of a device node results in the creation of new in-memory device node if it doesn't exist, or reusing it if it does.
    - Declaration of a property results in the creation of a new in-memory property containing the new property values if it doesn't exist or replacing it if it does.
    - Node and property removal directives remove nodes and properties of the runtime tree structure as appropriate.
    - Node labels and references to them in properties are tracked. Note that references are the only scalar values that are tracked in the in-memory property data structure.
    - phandle references editing operations of the form '&label' & '&{/path}' are processed. These reference nodes with labels declared earlier in the main tree source. This form is typically used when compiling a device tree comprised of a main source file and a number of included files because it lends well to the a pattern of incremental change.
    - The special /memreserve/ directive is parsed and processed.
2. The in-memory tree structure is 'flattened', i.e. it is serialized to create a device tree blob (DTB). It is in this stage that the symbolic references to node labels are resolved to integer/cell phandle values, with references to them being replaced by a cell value of the node's phandle. Special 'automatic' properties (named `phandle`) containing the assigned phandle values are created for nodes that are referenced.

3. This device tree blob file is placed in the applicable device and the bootloader is informed about how to retrieve it. This may be done by placing it in non-volatile storage at a specific byte offset, or being put in a boot-loader accesible filesystem with a specific name, etc.

4. The bootloader starts, retrieves the device tree blob, and either passes it unchanged to the operating system or performs minor modification (i.e. altering the boot command line in the chosen node or enabling/disabling devices by modifying the status properties of some nodes). The bootloader typically does not create an in-memory tree structure at this step, it operates on the DTB blob level.

5. The operating system starts and 'unflattens' the device tree blob which the bootloader has passed to it using the agreed upon architecture specific interface. The in-memory tree structure created is the same as the one created at the end of the compilation step, but with any changes that the boot-loader performed. The kernel at this point starts using the in-memory data structure, and it is referred to as the `live-tree` going forward. Note that

while node phandles are discovered and tracked by the ‘phandle’ properties, their references cannot be deduced at this time.

6. The operating system (including any device drivers) scans the live-tree and performs initialization and configuration of the hardware described there. Note that the operating system must have complete knowledge of the nodes and properties of an active node. This is evident by the use of access methods that include type information (e.g. of_property_read_u32()), node references needing to be explicitly discovered by converting cell phandle value to a reference to a node, etc. Unfortunately, this is very error-prone since the type information has been discarded. There is no way to disallow access to a property using a different method than what was declared in the original source file.

The steps above are applicable to the simple case of a single platform, and up to a few years ago used to be the norm. In contemporary systems the situation is more complex for the following reasons:

1. A common requirement is for a single image to be used for a number of different (but sufficiently similar) platforms. The number of stored DTBs would match the number of supported platforms, even if their changes are minimal.

2. Hardware is no longer static. The proliferation of FPGAs and add-on expansion boards requires runtime device tree modification using device tree overlays. Those overlays are extremely similar to the way in-memory tree modification is performed at compile time but it is different in subtle ways.

3. The device tree lifecycle expects perfect coordination across all the steps without the possibility of errors. This is troublesome in practice since every step in the sequence is part of a different project (compiler, bootloader, operating system). Errors can easily creep in and are usually not detected until the last step of the sequence, the operating system boot process. In case of an error, the result is usually a hung system without any indication what might have gone wrong.

YAML as a source format alternative

YAML is a human-readable data serialization language which is expressive enough to cover all DTS features. Simple YAML files are just key-value pairs that are very easy to parse, even without using a formal YAML parser. YAML streams are containing documents separated with a — marker. This model is a good fit for device tree since one may simply append a few lines of text to a given YAML stream to modify it.

YAML parsers are very mature, as YAML was first released in 2001. It is currently in wide-spread use and schema validation tools are available and common. Additionally, YAML support is available for many major programming languages.

Mapping of DTS constructs to YAML

The mapping of DTS constructs to YAML is relatively straightforward since they are both key-value declaration languages.

- Comments in YAML are done using the # character instead of the C-like comments that DTS uses.

```
/*

                dts comment */

#
```

```
                YAML comment
```

- DTS is a free form language using braces for denoting nest level while YAML is indentation sensitive in standard YAML encoding. Fortunately YAML is a superset of JSON which can be used as a valid free form.

```
node {

   property
                = "foo";

}

node:



                 property: "foo"

{

                "node": { "property": "foo" } }
```

- There is no explict root in YAML encoding. Top level nodes & properties are taken to be located in the root.

```
/ {

    property;

    subnode
                {

       another-property;

    };

};

property: true

  subnode:



                    another-property: true
```

- Sequences in YAML may be denoted either by a single line starting with a hyphen ‘-‘, or bracketed JSON form. The following are equivalent.

```
property

                = "a", "b", "c";

property:

  -
                "a"
```

```
  -
                "b"


                 - "c"
property:
                [ "a", "b", "c" ]
```

- Values that may be evaluated as numeric scalars are used as cells.

```
property
                = <10>;
property:
                10
```

Note that this includes integer expressions as well

```
property
                = <(5 + 5)>;
property:
                5+5
```

- String property values are enclosed in double quotes, although this is optional if the value cannot be expressed as a numeric scalar.

```
property
                = "string";
property:
                "string"
```

- Boolean values are encoded as true and false. This is not implicit like in DTS.

```
property;
property:
                true
```

Note that it is possible to declare a property as `false` but you will get a warning about it being removed when generating the DTB.

```
property:
                false
```

- It is possible to explictly declare the type of a scalar using the standard ‘!’ method of YAML. For instance this is how byte properties are supported.

```
property
                = [0124AB];

property:
                !int8 [ 0x01, 0x24, 0xab ]
```

- Similarly the /bits/ directive is supported by explicit tagging.

```
property
                = /bits/ 64 <100>;

property:
                !int64 100
```

- Labels are named anchors and are referenced by a ‘*’. Note that references are typed as such in YAML, they are transformed to phandle cells only on DTB generation.

```
label: node {
   property;
};



ref = <&label>;



&label {
    foo;
};
node: &label
  property:
                true



ref: *label
```

```
*label:


                 foo: true
```

- The delete node and properties directives are replaced with assignment to null/~. It works the same for both properties and nodes.

```
/ {

    node
                {

        property;

    };

};




/ {

    /delete-node/
                node;

};

node:

  property:
                true




node:

                ~
```

- There is no source /include/ directive in YAML. It is expected that thet C preprocessor will be used as is the norm with DTS.

- Similarly there is no /include-bin/ directive, YAML can relatively easily include binary data as base64 string properties.

- To easily support pre-processor macros from a DTS environment, scalars that are detected to be space separated integer expressions are transparently converted to scalar integer sequences.

```
#define MACRO(x, y) x y
                (x + y)

property

                = ;
```

```
#define MACRO(x, y) x y
                (x + y)

property:

                MACRO(10, 5)
```

Will result in

```
property:

                [ 10, 5, 15 ]
```

The YAML advantage

Radical changes are seldom worth it without bringing in significant benefits. Switching to YAML instead of DTS is indeed a radical change, but it does carry benefits, namely:

1. YAML is a well known and mature technology which is supported by many programming languages and environments.

2. YAML's original purpose was data serialization. Therefore it is orthogonal and supports high-level language data structures well.

3. It is suited for the description of graph structures, since it supports references and anchors.

4. With its mature parsers and tools, it easily supports the human edit and compile cycle that is now common with device tree development. Since all property values are potentially typed, it is possible to track type information in order to perform thorough validation and checking against device tree bindings (once the bindings are converted to a machine readable format, preferably YAML). As well, this allows the reporting of accurate error messages and warnings at any stage of the compilation process.

5. It is possible to generate YAML as an intermediate format with references not resolved, in a similar way that object files are used. Those intermediate files can them be compiled/linked again to generate the final DTB/YAML file. For example, instead of compiling into a single output file, one could generate intermediate YAML files, similar in every way to device tree overlays, and then perform the final 'linking' step at either compile time or the bootloader.

6. It is relatively easy to parse, and a resource limited parser that can be included in bootloaders or the kernel is possible.

7. Data in YAML can easily be converted to and from other formats making it convertable to formats which future tools may understand.

The yamldt compiler

`yamldt` is a YAML/DTS to DT blob generator/compiler and validator. The YAML schema is functionally equivalent to DTS and supports all DTS features, while as a DTS compiler it is bit-exact compatible with DTC. `yamldt` parses a device tree description (source) file in YAML/DTS format and outputs a device tree blob (which can be bit-exact to the one generated from the reference dtc compiler if the -C option is used).

Validation is performed against a YAML schema that defines properties and constraints. A checker uses this schema to

generate small code fragments that are compiled to eBPF and executed for the specific validation of each DT node the rule selects in the output tree.

Validation

As mentioned above, `yamldt` is capable of performing validation of DT constructs using a C-based eBPF checker. eBPF code fragments are assembled that can perform type checking of properties and enforce arbitrary value constraints while fully supporting inheritance.

As an example, here's how the validation of a given fragment works using on a jedec,spi-nor node:

```
m25p80@0:

  compatible:
                "s25fl256s1"

  spi-max-frequency:
                76800000

  reg:
                0

  spi-tx-bus-width:
                1

  spi-rx-bus-width:
                4

  "#address-cells":
                1



                 "#size-cells": 1
```

The binding for this is:

```
%YAML 1.1

---

jedec,spi-nor:

  version:
                1




  title:
                >

    SPI
                NOR flash: ST M25Pxx (and similar) serial flash chips
```

```
maintainer:

  name:
              Unknown




inherits:
              *spi-slave




properties:

  reg:

    category:
              required

    type:
              int

    description:
              chip select address of device




  compatible:
              &jedec-spi-nor-compatible

    category:
              required

    type:
              strseq

    description:
              >

      May
              include a device-specific string consisting of the

      manufacturer
              and name of the chip. A list of supported chip

      names
              follows.

      Must
              also include "jedec,spi-nor" for any SPI NOR flash that
              can

      be
              identified by the JEDEC READ ID opcode (0x9F).

    constraint:
```

```
        |

anystreq(v,
        "at25df321a") ||

anystreq(v,
         "at25df641") ||

anystreq(v,
        "at26df081a") ||

anystreq(v,
          "mr25h256") ||

anystreq(v,
           "mr25h10") ||

anystreq(v,
           "mr25h40") ||

anystreq(v,
        "mx25l4005a") ||

anystreq(v,
        "mx25l1606e") ||

anystreq(v,
        "mx25l6405d") ||


        anystreq(v,"mx25l12805d") ||


        anystreq(v,"mx25l25635e") ||

anystreq(v,
           "n25q064") ||

anystreq(v,
        "n25q128a11") ||

anystreq(v,
        "n25q128a13") ||

anystreq(v,
          "n25q512a") ||

anystreq(v,
        "s25fl256s1") ||

anystreq(v,
         "s25fl512s") ||

anystreq(v,
        "s25sl12801") ||

anystreq(v,
         "s25fl008k") ||
```

```
    anystreq(v,
             "s25fl064k") ||


            anystreq(v,"sst25vf040b") ||

    anystreq(v,
                "m25p40") ||

    anystreq(v,
                "m25p80") ||

    anystreq(v,
                "m25p16") ||

    anystreq(v,
                "m25p32") ||

    anystreq(v,
                "m25p64") ||

    anystreq(v,
               "m25p128") ||

    anystreq(v,
                "w25x80") ||

    anystreq(v,
                "w25x32") ||

    anystreq(v,
                "w25q32") ||

    anystreq(v,
                "w25q64") ||

    anystreq(v,
              "w25q32dw") ||

    anystreq(v,
              "w25q80bl") ||

    anystreq(v,
               "w25q128") ||

    anystreq(v,
               "w25q256")



spi-max-frequency:

  category:
            required

  type:
            int
```

```
    description:
              Maximum frequency of the SPI bus the chip can operate at

    constraint:
              |

      v
              > 0 && v < 100000000




  m25p,fast-read:

    category:
              optional

    type:
              bool

    description:
              >

      Use
              the "fast read" opcode to read data from the chip instead

      of
              the usual "read" opcode. This opcode is not supported by

      all
              chips and support for it can not be detected at runtime.

      Refer
              to your chips' datasheet to check if this is supported

      by
              your chip.



example:

  dts:
              |

    flash:
              m25p80@0 {

        #address-cells
              = <1>;

        #size-cells
              = <1>;

        compatible
              = "spansion,m25p80", "jedec,spi-nor";

        reg
```

```
                = <0>;

                spi-max-frequency = <40000000>;

                m25p,fast-read;

      };

    yaml:
                |

      m25p80@0:
                &flash

                "#address-cells": 1

        "#size-cells":
                1

        compatible:
                [ "spansion,m25p80", "jedec,spi-nor" ]

        reg:
                0;

                spi-max-frequency: 40000000

                       m25p,fast-read: true
```

Note the constraint rule matches on any compatible string in the given list. This binding inherits from spi-slave as indicated by the line: `inherits: *spi-slave`

`*spi-slave` is standard YAML reference notation which points to the spi-slave binding, pasted here for convenience:

```
%YAML 1.1

---

spi-slave: &spi-slave

  version:
                1



  title:
                SPI Slave Devices
```

```
maintainer:

  name:
              Mark Brown <broonie@kernel.org>



inherits:
              *device-compatible



class:
              spi-slave

virtual:
              true



description:
              >

  SPI
              (Serial Peripheral Interface) slave bus devices are children of

  a
              SPI master bus device.



#
              constraint: |+

#
               class_of(parent(n), "spi")



properties:

  reg:

    category:
              required

    type:
              int

    description:
              chip select address of device
```

```
compatible:

  category:
            required

  type:
            strseq

  description:
            compatible strings




spi-max-frequency:

  category:
            required

  type:
            int

  description:
            Maximum SPI clocking speed of device in Hz




spi-cpol:

  category:
            optional

  type:
            bool

  description:
            >

    Boolean
            property indicating device requires

    inverse
            clock polarity (CPOL) mode




spi-cpha:

  category:
            optional

  type:
            bool

  description:
            >
```

```
    Boolean
            property indicating device requires

    shifted
            clock phase (CPHA) mode



spi-cs-high:

  category:
            optional

  type:
            bool

  description:
            >

    Boolean
            property indicating device requires

    chip
            select active high



spi-3wire:

  category:
            optional

  type:
            bool

  description:
            >

    Boolean
            property indicating device requires

    3-wire
            mode.



spi-lsb-first:

  category:
            optional

  type:
            bool

  description:
            >
```

```
      Boolean
              property indicating device requires

      LSB
              first mode.



  spi-tx-bus-width:

    category:
              optional

    type:
              int

    constraint:
              v == 1 || v == 2 || v == 4

    description:
              >

      The
              bus width(number of data wires) that

      used
              for MOSI. Defaults to 1 if not present.



  spi-rx-bus-width:

    category:
              optional

    type:
              int

    constraint:
              v == 1 || v == 2 || v == 4

    description:
              >

      The
              bus width(number of data wires) that

      used
              for MISO. Defaults to 1 if not present.



notes:
              >

  Some
```

SPI controllers and devices support Dual and Quad SPI transfer mode.

It allows data in the SPI system to be transferred in 2 wires(DUAL) or 4 wires(QUAD).

Now the value that spi-tx-bus-width and spi-rx-bus-width can receive is only 1(SINGLE), 2(DUAL) and 4(QUAD). Dual/Quad mode is not allowed when 3-wire mode is used.

If a gpio chipselect is used for the SPI slave the gpio number will be passed via the SPI master node cs-gpios property.

example:

dts:

```
|

  spi@f00
            {

            ethernet-switch@0 {

          compatible
            = "micrel,ks8995m";

            spi-max-frequency = <1000000>;

          reg
            = <0>;

      };



      codec@1
            {

          compatible
            = "ti,tlv320aic26";
```

```
                spi-max-frequency = <100000>;

              reg
                = <1>;

          };

      };

    yaml:
                |

      spi@f00:


                ethernet-switch@0:

          compatible:
                "micrel,ks8995m"


                spi-max-frequency: 1000000

          reg:
                0




        codec@1:

          compatible:
                "ti,tlv320aic26"


                spi-max-frequency: 100000



                          reg: 1
```

Note the `&spi-slave` anchor, this is what it is used to refer to other parts of the schema.

The SPI slave binding defines a number of properties that all inherited bindings include. This in turn inherits from `device-compatible`, which is this:

```
%YAML 1.1

---

device-compatible:
                &device-compatible

  title:
                Contraint for devices with compatible properties

  #
                select node for checking when the compatible constraint and
```

```
  #
                the device status enable constraint are met.

  selected:
                [ "compatible", *device-status-enabled ]



  class:
                constraint



                 virtual: true
```

Note that device-compatible is a binding that all devices defined with the DT schema will inherit from.

The `selected` property will be used to generate a select test that will be used to to find out whether a node should be checked against a given rule.

The `selected` rule defines two constraints. The first one is the name of a variable in a derived binding that all its constraints must satisfy; in this case it's the jedec,spi-nor compatible constraint in the binding above. The selected constraint is a reference to the `device-status-enabled` constraint defined at:

```
%YAML 1.1

---

device-enabled:

  title:
                Contraint for enabled devices



  class:
                constraint

  virtual:
                true



  properties:

    status:
                &device-status-enabled

      category:
                optional

      type:
                str

      description:
```

```
                Marks device state as enabled

      constraint:
                |



                       !exists || streq(v, "okay")

                || streq(v, "ok")
```

The `device-enabled` constraint checks where the node is enabled in DT parlance.

Taking those two constraints together, `yamldt` generates an enable method filter which triggers on an enable device node that matches any of the compatible strings defined in the jedec,spi-nor binding.

The check method will be generated by collecting all the property constraints (category, type and explicit value constraints).

Note how in the above example a variable (v) is used as the current property value. The generated methods will provide it, initialized to the current value to the constraint.

Note that custom, manually written select and check methods are possible but their usage is not recommended for simple types.

Installation

Install libyaml-dev and the standard autoconf/automake generation tools, then compile with the standard `./autogen.sh`, `./configure`, and `make` cycle. Note that the bundled validator requires a working eBPF compiler and libelf. Known working clang versions with eBPF support are 4.0 and higher.

For a complete example of a port of a board DTS file to YAML take a look in the `port/` directory

Usage

The `yamldt` options available are:

```
yamldt [options]
                <input-file>

 options
                are:

   -q,
                --quiet           Suppress; -q (warnings) -qq (errors) -qqq
                (everything)

   -I,
                --in-format=X     Input format type X=[auto|yaml|dts]

   -O,
                --out-format=X    Output format type X=[auto|yaml|dtb|dts|null]

   -o,
                --out=X           Output file
```

```
-c
                              Don't resolve references (object mode)

-g,
            --codegen         Code generator configuration file

    --schema
                     Use schema (all yaml files in dir/)

    --save-temps
                 Save temporary files

    --schema-save
                Save schema to given file

    --color
                      [auto|off|on]

    --debug
                      Debug messages

-h,
            --help            Help

-v,
            --version         Display version



DTB
            specific options



-V,
            --out-version=X   DTB blob version to produce (only 17 supported)

-C,
            --compatible      Bit-exact DTC compatibility mode

-@,
            --symbols         Generate symbols node

-A,
            --auto-alias      Generate aliases for all labels

-R,
            --reserve=X       Make space for X reserve map entries

-S,
            --space=X         Make the DTB blob at least X bytes long

-a,
            --align=X         Make the DTB blob align to X bytes

-p,
            --pad=X           Pad the DTB blob with X bytes
```

```
   -H,
                --phandle=X       Set phandle format [legacy|epapr|both]

   -W,
                --warning=X       Enable/disable warning (NOP)

   -E,
                --error=X         Enable/disable error (NOP)


                  -b, --boot-cpu=X      Force boot

                cpuid to X
```

`-q/--quiet` suppresses message output.

The `-I/--in-format` option selects the input format type. By default it is set to auto which is capable of selecting based on file extension and input format source patterns.

The `-O/--out-format` option selects the output format type. By default it is set to auto which uses the output file extension.

`-o/--out` sets the output file.

The `-c` option causes unresolved references to remain in the output file resuling in an object file. If the output format is set to DTB/DTS it will generate an overlay, if set to yaml it results in a YAML file which can be subsequently recompiled as an intermediate object file.

The `-g/--codegen` option will use the given YAML file(s) (or dir/ as in the schema option) as input for the code generator.

The `--schema` option will use the given file(s) as input for the checker. As an extension, if given a directory name with a terminating slash (i.e. dir/) it will recursively collect and use all YAML files within.

The `--save-temps` option will save all intermediate files/blobs.

`--schema-save` will save the processed schema and codegen file including all compiled validation filters. Using it speeds validation of multiple files since it can be used as an input via the –schema option.

`--color` controls color output in the terminal, while `--debug` enables the generation of a considerable amount of debugging messages.

The following DTB specific options are supported:

`-V/--out-version` selects the DTB blob version; currently only version 17 is supported.

The `-C/--compatible` option generates a bit-exact DTB file as the DTC compiler.

The `-@/--symbols` and `-A/--auto-alias` options generate a **symbols** and alias entries for all the defined labels in the source files.

The `-R/--reserve`, `-S/--space`, `-a/--align` and `-p/--pad` options work the same way as in DTC. `-R` add reserve memreserve entries, `-S` adds extra space, `-a` aligns and `-p` pads extra space end of the DTB blob.

The `-H/--phandle` option selects either legacy/epapr or both phandle styles.

The `-W/--warning` and `-E/--error` options are there for command line compatibility with dtc and are ignored.

Finally `-d/--boot-cpu` forces the boot cpuid.

Automatic suffix detection does what you expect, i.e. an output file ending in .dtb if selecting the DTB generation option, .yaml if selecting the yaml generation option, and so on.

Given a source file in YAML `foo.yaml`, you generate a DTB file with:

```
# foo.yaml

foo: &foo

  bar:
                true

  baz:

   -
                12

   -
                8

   -
                *foo



                 frob: [ "hello", "there"

                ]
```

To process it with yamldt:

```
$ yamldt -o foo.dtb
                foo.yaml

$ ls -l foo.dtb

-rw-rw-r-- 1 panto panto
                153 Jul 27 18:50 foo.dtb

$ fdtdump foo.dtb

/dts-v1/;

// magic:
                0xd00dfeed

// totalsize:       0xe1
                (225)

// off_dt_struct:   0x38
```

```
// off_dt_strings:  0xc8

// off_mem_rsvmap:  0x28

// version:     17

// last_comp_version:
                16

// boot_cpuid_phys: 0x0

// size_dt_strings: 0x19

// size_dt_struct:  0x90



/ {

    foo
                {

        bar;

        baz
                = <0x0000000c 0x00000008 0x00000001>;

        frob
                = "hello", "there";

        phandle
                = <0x00000001>;

    };

    __symbols__
                {

        foo
                = "/foo";

    };

};
```

dts2yaml

`dts2yaml` is an automatic DTS to YAML conversion tool, that works on standard DTS files which use the preprocessor. It is capable of detecting macro usage and advanced DTS concepts, like property/node deletes, etc. Conversion is accurate as long as the source file still looks like DTS source (i.e. it is not using extremely complex macros).

```
dts2yaml [options]
                [input-file]

 options
                are:

   -o,
                --output        Output file

   -t,
                --tabs       Set tab size (default 8)

   -s,
                --shift      Shift when outputing YAML (default 2)

   -l,
                --leading    Leading space for output

   -d,
                --debug      Enable debug messages

       --silent
                        Be really silent

       --color
                         [auto|off|on]

   -r,
                --recursive     Generate DTS/DTSI included files

   -h,
                --help       Help



                      --color         [auto|off|on]
```

All the input files will be converted to yaml format. If no output option is given, the output will be named according to the input filename. So foo.dts will yield foo.yaml and foo.dtsi will yield foo.yamli.

The recursive option converts all included files that have a dts/dtsi extension as well.

Test suite

To run the test-suite you will need a relatively recent DTC compiler, YAML patches are no longer required.

The test-suite first converts all the DTS files in the Linux kernel for all architectures to YAML format using `dts2yaml`. Afterwards, it compiles the YAML files with `yamldt` and the DTS files with `dtc`. The resulting dtb files are bit-exact because the `-C` option is used.

Run `make check` to run the test suite.
Run `make validate` to run the test suite and perform schema validation checks. It is recommended to use the `--keep-going` flag to continue checking even in the presence of validation errors.

Currently out of 1379 DTS files, only 6 fail conversion:

`exynos3250-monk`

```
                exynos4412-trats2 exynos3250-rinato exynos5433-tm2

exynos5433-tm2e
```

All 6 use a complex pin mux macro declaration that it is not possible to automatically convert.

Workflow

It is expected that the first thing a user of `yamldt` would want to do is to convert an existing DTS configuration to YAML.

The following example uses the beaglebone black and the am335x-boneblack.dts source as located in the port/ directory.

Compile the original DTS source with DTC

```
$  cc -E  -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input


    -nostdinc
                -undef -x assembler-with-cpp -D__DTS__ am335x-boneblack.dts




                   | dtc -@ -q -I dts -O dtb - -o

                am335x-boneblack.dtc.dtb
```

Use `dts2yaml` to convert to yaml

```
$ dts2yaml -r
                am335x-boneblack.dts

$ ls *.yaml*

am335x-boneblack-common.yamli
                 am335x-bone-common.yamli  am33xx-clocks.yamli

am33xx.yamli

                 tps65217.yamli
```

Note the recursive option automatically generates the dependent include files.

```
$ cc -I ./ -I ../../port
                -I ../../include -I ../../include/dt-bindings/input


    -nostdinc
                -undef -x assembler-with-cpp -D__DTS__ am335x-boneblack.yaml |
```

```
                  ../../yamldt -C -@ - -o

                am335x-boneblack.dtb



$ ls -l *.dtb

-rw-rw-r-- 1 panto panto
                50045 Jul 27 19:10 am335x-boneblack.dtb

-rw-rw-r-- 1 panto panto
                50045 Jul 27 19:07 am335x-boneblack.dtc.dtb

$ md5sum *.dtb

3bcf838dc9c32c196f66870b7e6dfe81
                 am335x-boneblack.dtb

3bcf838dc9c32c196f66870b7e6dfe81

                 am335x-boneblack.dtc.dtb
```

Compiling without the -C option results in a file with the same functionality, but it is slightly smaller due to better string table optimization.

```
$ yamldt
                am335x-boneblack.dtc.yaml -o am335x-boneblack.dtb

$ ls -l *.dtb

-rw-rw-r-- 1 panto panto
                50003 Jul 27 19:12 am335x-boneblack.dtb

-rw-rw-r--

                1 panto panto 50045 Jul 27 19:07 am335x-boneblack.dtc.dtb
```

Note that the CPP command line is the same, so no changes to header files are required. `dts2yaml` will detect macro usage and convert from the space delimited form that DTC uses to the comma delimted form used by YAML.

yamldt as a DTC compiler

`yamldt` supports all `dtc` options, so using it as a `dtc` replacement is straightforward.

Using it for compiling the Linux Kernel DTS files is as simple as:

```
$

                make DTC=yamldt dtbs
```

Note that by default the compatibility option (-C) is not used, so if you need to be bit-compatible with DTC pass the -C flag as follows:

```
$
```

```
                make DTC=yamldt DTC_FLAGS="-C"
```

Generally, `yamldt` is a little bit faster than `dtc` and generates somewhat smaller DTB files (if not using the -C option). However, due to internally tracking all parsed tokens and their locations in files, it is capable of generating accurate error messages that are parseable by text editors for automatic movement to the error.

For example, with this file containing an error:

```
/* duplicate label */

/dts-v1/;

/ {

    a:
                foo { foo; };

    a:
                bar { bar; };

};
```

`yamldt` will generate the following error:

```
$ yamldt -I dts -o dts
                -C duplabel.dts

duplabel.dts:8:2: error:
                duplicate label a at "/bar"

  a:
                bar {

  ^

duplabel.dts:4:2: error:
                duplicate label a is defined also at "/foo"

  a:
                foo {



                 ^
```

while `dtc` will generate:

```
$ yamldt -I dts -o dts
                -C duplabel.dts

dts: ERROR
                (duplicate_label): Duplicate label 'a' on /bar and /foo

ERROR:

                Input tree has errors, aborting (use -f to force output)
```

Known features of DTC that are not available are:

- Only version 17 DT blobs are supported. Passing a -V argument requesting a different one will result in error.
- Assembly output is not supported.
- Assembly and filesystem inputs are not supported.
- The warning and error options are accepted, but they don’t do anything. yamldt uses a validation schema for application specific errors and warnings so those options are superfluous.

Notes on DTS to DTS conversion

The conversion from DTS is straight forward:

For example:

```
/* foo.dts */

/ {

    foo
                = "bar";

    #cells
                = <2>;

    phandle-ref
                = <&ref 1>;

    ref:
                refnode { baz; };

};

# foo.yaml

foo: "bar"

"#cells": 2

phandle-ref: [ *ref 1 ]

refnode: &ref


                 baz: true
```

Major differences between DTS & YAML:

- YAML is using # as a comment marker, therefore properties with a # prefix get converted to explicit string literals:

```
#cells

                = <0>;
```

to YAML

```
"#cells":
                0
```

- YAML is indentation sensitive, but it is a JSON superset. Therefore the following are equivalent:

```
foo:
                [ 1, 2 ]
foo:
 -
                1
                - 2
```

- The labels in DTS are defined and used as:

```
foo: node { baz; };
bar
                = <&foo>;
```

In YAML the equivalent methods are called anchors and are defined as follows:

```
node: &foo
  baz:
                true
bar:
                *foo
```

- Explicit tags in YAML are using !, so the following:

```
mac
                = [ 0 1 2 3 4 5 ];
```

is used like this in YAML:

```
mac:
                !int8 [ 0, 1, 2, 3, 4, 5 ]
```

- DTS uses spaces to seperate array elements, YAML uses either indentation or commas in JSON form. Note that yamldt is smart enough to detect the DTS form and automatically convert in most cases:

```
pinmux
                = <0x00 0x01>;
```

In YAML:

```
pinmux:

  -
                0x00



                 - 0x01
```

or:

```
pinmux:

                [ 0x00, 0x01 ]
```

- Path references () automatically are converted to pseudo YAML anchors (of the form yaml_pseudo__n__):

```
/ {

    foo
                { bar; };

};

ref

                = <&/foo>;
```

In YAML:

```
foo: &yaml_pseudo__0__

ref:

                *foo
```

- Integer expression evaluation, similar in manner to that which the CPP preprocessor performs, is available. This is required in order for macros to work. For example, given the following two files:

```
/* add.h */

#define

                ADD(x, y) ((x) + (y))

# macro-use.yaml



#include "add.h"
```

```
result:

                ADD(10, 12)
```

The output after the cpp preprocessor pass:

```
result:

                ((10) + (12))
```

Parsing with `yamldt` to DTB will generate a property:

```
result

                = <22>;
```

Validation example

For this example we're going to use port/am335x-boneblack-dev/. An extra rule-check.yaml file has been added where validation tests can be performed.

That file contains a single jedec,spi-nor device:

```
*spi0:

  m25p80@0:

    compatible:
                "s25fl256s1"

    spi-max-frequency:
                76800000

    reg:
                0

    spi-tx-bus-width:
                1

    spi-rx-bus-width:
                4

    "#address-cells":
                1



                   "#size-cells": 1
```

This is a valid device node, so running validate produces the following:

```
$ make validate

cc -E -MT
                am33xx.cpp.yaml -MMD -MP -MF am33xx.o.Yd -I ./ -I ../../port -I
```

```
                ../../include -I ../../include/dt-bindings/input -nostdinc -undef
                -x assembler-with-cpp -D__DTS__ -D__YAML__ am33xx.yaml
                >am33xx.cpp.yaml

cc -E -MT
                am33xx-clocks.cpp.yaml -MMD -MP -MF am33xx-clocks.o.Yd -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                am33xx-clocks.yaml >am33xx-clocks.cpp.yaml

cc -E -MT
                am335x-bone-common.cpp.yaml -MMD -MP -MF am335x-bone-common.o.Yd -I
                ./ -I ../../port -I ../../include -I
                ../../include/dt-bindings/input -nostdinc -undef -x
                assembler-with-cpp -D__DTS__ -D__YAML__ am335x-bone-common.yaml
                >am335x-bone-common.cpp.yaml

cc -E -MT
                am335x-boneblack-common.cpp.yaml -MMD -MP -MF
                am335x-boneblack-common.o.Yd -I ./ -I ../../port -I ../../include
                -I ../../include/dt-bindings/input -nostdinc -undef -x
                assembler-with-cpp -D__DTS__ -D__YAML__
                am335x-boneblack-common.yaml >am335x-boneblack-common.cpp.yaml

cc -E -MT
                am335x-boneblack.cpp.yaml -MMD -MP -MF am335x-boneblack.o.Yd -I ./
                -I ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                am335x-boneblack.yaml >am335x-boneblack.cpp.yaml

cc -E -MT
                rule-check.cpp.yaml -MMD -MP -MF rule-check.o.Yd -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                rule-check.yaml >rule-check.cpp.yaml

../../yamldt  -g
                ../../validate/schema/codegen.yaml -S ../../validate/bindings/ -y
                am33xx.cpp.yaml am33xx-clocks.cpp.yaml am335x-bone-common.cpp.yaml
                am335x-boneblack-common.cpp.yaml am335x-boneblack.cpp.yaml
                rule-check.cpp.yaml -o am335x-boneblack-rules.pure.yaml

jedec,spi-nor:

                /ocp/spi@48030000/m25p80@0 OK
```

Note the last line. It means the node was checked and was found OK.

Editing the rule-check.yaml file, let's introduce a couple of errors. The following output is generated by commenting out the reg property `# reg: 0`:

```
$ make validate

cc -E -MT
                rule-check.cpp.yaml -MMD -MP -MF rule-check.o.Yd -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                rule-check.yaml &gt;rule-check.cpp.yaml

../../yamldt  -g
                ../../validate/schema/codegen.yaml -S ../../validate/bindings/ -y
```

```
                am33xx.cpp.yaml am33xx-clocks.cpp.yaml am335x-bone-common.cpp.yaml
                am335x-boneblack-common.cpp.yaml am335x-boneblack.cpp.yaml
                rule-check.cpp.yaml -o am335x-boneblack-rules.pure.yaml

jedec,spi-nor:
                /ocp/spi@48030000/m25p80@0 FAIL (-2004)

../../validate/bindings/jedec,spi-nor.yaml:15:5:
                error: missing property: property was defined at
                /jedec,spi-nor/properties/reg

     reg:



                    ^~~~
```

Note the descriptive error and the pointer to the missing property in the schema.

Making another error, assign a string to the reg property `reg: "string"`:

```
$ make validate

$ make validate

cc -E -MT
                rule-check.cpp.yaml -MMD -MP -MF rule-check.o.Yd -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                rule-check.yaml &gt;rule-check.cpp.yaml

../../yamldt  -g
                ../../validate/schema/codegen.yaml -S ../../validate/bindings/ -y
                am33xx.cpp.yaml am33xx-clocks.cpp.yaml am335x-bone-common.cpp.yaml
                am335x-boneblack-common.cpp.yaml am335x-boneblack.cpp.yaml
                rule-check.cpp.yaml -o am335x-boneblack-rules.pure.yaml

jedec,spi-nor:
                /ocp/spi@48030000/m25p80@0 FAIL (-3004)

rule-check.yaml:8:10:
                error: bad property type

     reg:
                "string"

          ^~~~~~~

../../validate/bindings/jedec,spi-nor.yaml:15:5:
                error: property was defined at /jedec,spi-nor/properties/reg

     reg:



                    ^~~~
```

Note the message about the type error, and the pointer to the location where the reg property was defined.

Finally, let’s make an error that violates a constraint.

Change the `spi-tx-bus-width` value to 3:

```
$ make validate

cc -E -MT
                rule-check.cpp.yaml -MMD -MP -MF rule-check.o.Yd -I ./ -I
                ../../port -I ../../include -I ../../include/dt-bindings/input
                -nostdinc -undef -x assembler-with-cpp -D__DTS__ -D__YAML__
                rule-check.yaml &gt;rule-check.cpp.yaml

../../yamldt  -g
                ../../validate/schema/codegen.yaml -S ../../validate/bindings/ -y
                am33xx.cpp.yaml am33xx-clocks.cpp.yaml am335x-bone-common.cpp.yaml
                am335x-boneblack-common.cpp.yaml am335x-boneblack.cpp.yaml
                rule-check.cpp.yaml -o am335x-boneblack-rules.pure.yaml

jedec,spi-nor:
                /ocp/spi@48030000/m25p80@0 FAIL (-1018)

rule-check.yaml:9:23:
                error: constraint rule failed

     spi-tx-bus-width:
                3

                       ^

../../validate/bindings/spi/spi-slave.yaml:77:19:
                error: constraint that fails was defined here

       constraint:
                v == 1 || v == 2 || v == 4


                ^~~~~~~~~~~~~~~~~~~~~~~~~

../../validate/bindings/spi/spi-slave.yaml:74:5:
                error: property was defined at
                /spi-slave/properties/spi-tx-bus-width



                    spi-tx-bus-width:
```

Note how the offending value is highlighted. The offending constraint and property definition are aslo listed.