THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Embedded Linux Conference

Beyond Static Devicetrees: Implementing Runtime Hardware Dynamism in Zephyr

Wai-Hong Tam & Jason Yuan
Google

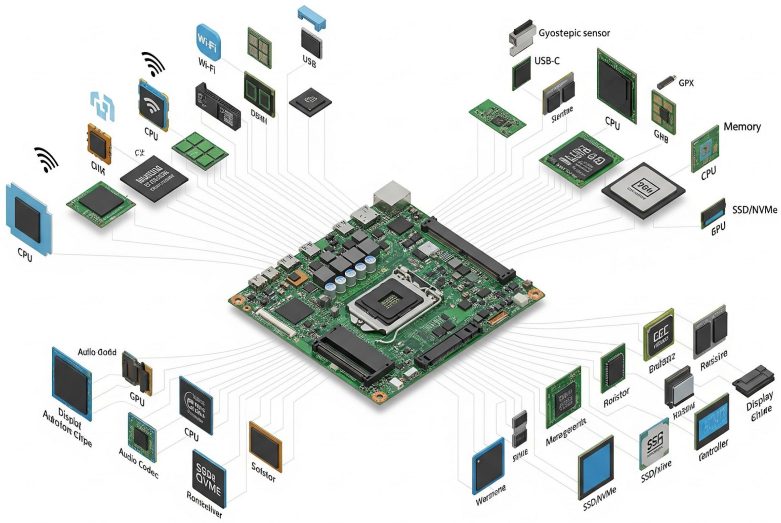
Agenda

- Mass Production Challenge
- Static Devicetree Trap
- Runtime Driver Selection
- Hardware Discovery
- Takeaways

ChromeOS EC and Zephyr Transition

- **ChromeOS Open-Source Firmware Stack:**
  - `Coreboot` -> `Depthcharge` -> Linux Kernel
  - **Embedded Controller: Zephyr Transition**
    - From a highly-coupled RTOS moving to an "Upstream-First" Zephyr RTOS.
    - *Previous talk: "Using Zephyr for Embedded Controllers" (Keith Short, ZDS 2022)*
  - **Zephyr is now everywhere.**
    - Main ECs
    - Detachable keyboards
    - Fingerprint sensor MCUs
    - Sensor hubs (e.g. Intel ISH)

Mass Production Challenge



Reference Baseboard

A single "design" turns into dozens of commercial SKUs due to supply chain volatility and ***second-sourcing***.

1 Baseboard × 3 IMUs × 4 PDCs × 2 Keyboard Layouts = 24 permutations.

The Scaling Problem:

Managing many unique binaries creates massive overhead for CI/CD, cryptographic signing, and firmware update infrastructure.

Hardware Evolution of Identity

"To have one binary, the firmware needs to know what hardware it's actually running on. We evolved this over a decade."

GPIO Straps

4 pins = 2^4 = 16 SKUs

Tri-state: 2^*N* → 3^*N*

'Pin Starvation' as complexity grew.

ADC Sensing

Voltage dividers

Limit: electrical noise & resistor tolerances

I2C EEPROM (CBI)

Data in CrOS Board Info (CBI)

Reliable but ++BOM cost

Internal Flash (CBI)

Data in CBI Region: protected 4KB block in internal flash

Zero cost

CBI Data Fields

CBI stores hardware identifiers as key-value pairs. Full doc is available in this guide.

Field Name	Tag	Type	Description
BOARD_VERSION	0	Integer	Incremental hardware version (0, 1, 2, ...)
OEM_ID	1	Integer	Unique number assigned to each OEM
SKU_ID	2	Integer	Describes specific hardware feature configurations
DRAM_PART_NUM	3	String	ASCII string identifying the DRAM inventory part
OEM_NAME	4	String	Human-readable OEM name in ASCII
...			

Legacy Approach: Hardcoded C Logic

```
static int board_is_convertible(void) {
    return sku_id == 9 || sku_id == 10 || ...
}

static int board_has_keypad(void) {
    return sku_id == 41 || sku_id == 42 || ...
}


static int board_usb_charge_mode_init(void) {
    if ((sku_id < 32 || sku_id > 39) && ...)
        return;
    ...
}
```

Magic Numbers

IDs like `sku_id == 41` carry zero semantic meaning.

High Coupling

Generic drivers become polluted with board-level logic.

Testing Gaps

Board-specific C code paths are notoriously difficult to unit-test and validate at scale.

Maintenance Burden

Every new hardware permutation requires a firmware change, code review, upver, etc.

Improvement: Bitmask (SSFC, Second Source Firmware Cache)

```
union dedede_cbi_ssfc {
  struct {
    uint32_t base_sensor : 3;
    uint32_t lid_sensor : 3;
    uint32_t tcpc_type : 2;
    uint32_t audio_codec_source : 3;
    uint32_t reserved_2 : 21;
  };
  uint32_t raw_value;
};
```

```
enum ec_ssfc_base_sensor {
  SSFC_SENSOR_BASE_DEFAULT = 0,
  SSFC_SENSOR_BMI160 = 1,
  SSFC_SENSOR_ICM426XX = 2,
  SSFC_SENSOR_LSM6DSM = 3,
  SSFC_SENSOR_ICM42607 = 4
};

...

enum ec_ssfc_tcpc_p1 {
  SSFC_TCPC_P1_DEFAULT,
  SSFC_TCPC_P1_PS8705,
  SSFC_TCPC_P1_PS8805
};
```

Improvement: Bitmask (SSFC, Second Source Firmware Cache)

```
enum ec_ssfc_base_sensor get_cbi_ssfc_base_sensor(void) {
  return (enum
ec_ssfc_base_sensor)cached_ssfc.base_sensor;
}
enum ec_ssfc_lid_sensor get_cbi_ssfc_lid_sensor (void) {
  return (enum ec_ssfc_lid_sensor)cached_ssfc.lid_sensor;
}
enum ec_ssfc_tcpc_p1 get_cbi_ssfc_tcpc_p1 (void) {
  return (enum ec_ssfc_tcpc_p1)cached_ssfc.tcpc_type;
}
```

Cleaner Semantic

Functions return enums based on bitfields.

Scalable Config

Adding components requires only new bitmask/offset definitions.

Problems still remain:

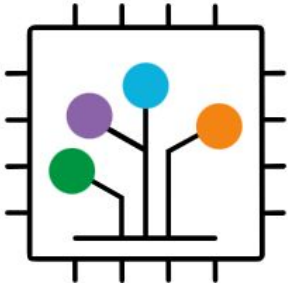
High Coupling

Testing Gaps

Maintenance Burden

Devicetree

was designed to solve exactly this problem... right?



Zephyr Static Devicetree Trap

Linux: Dynamic Runtime

The bootloader passes a `.dtb` blob to the kernel at runtime. Handles SKU sprawl trivially by swapping blobs.

Zephyr: Build-Time Static

Optimizes for RAM/Flash by compiling `.dts` into `#define` macros. Once linked, the hardware map is frozen.

Trade-off: *Flash Size* vs. *Operational Savings*

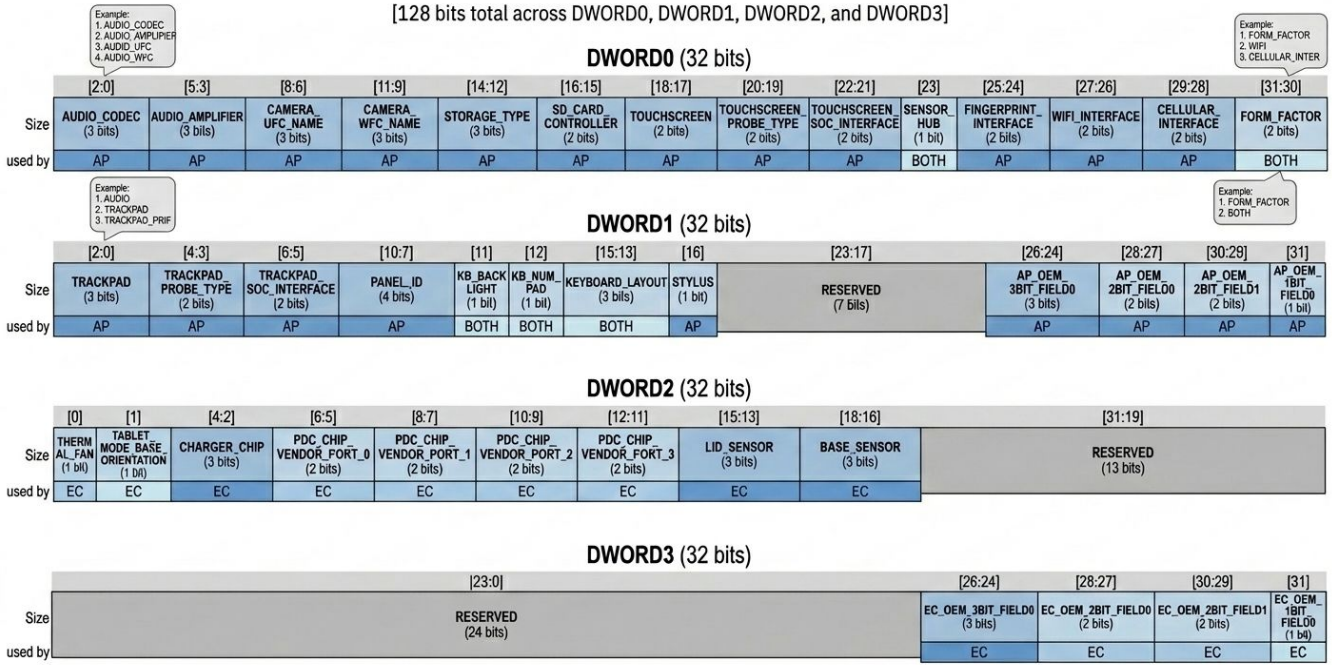
+10~30KB Flash Overhead: Accept a small read-only code size increase by compiling a superset of drivers.

Operational Savings: Consolidate 50+ build and release pipeline.

💡 **Note:** Zephyr's compile-time DT macro is brilliant for resource constraints (a useful talk by Marti Bolivar)

How do we build a dynamic driver selection around *Zephyr static Devicetree*?

128-bit UFSC (Unified Firmware & Second-Source Config) in CBI



Devicetree-Driven Schema (UFSC)

```
cbi_ufsc: cbi-ufsc {
  compatible = "cros-ec,cbi-ufsc";
  ufsc_form_factor: form-factor {
    start = <UFSC_BIT(0, 30)>; size = <2>;
  };
  ufsc_kb_backlight: kb-backlight {
    start = <UFSC_BIT(1, 11)>; size = <1>;
  };
  ufsc_pdc_port_0: pdc-chip-vendor-port-0 {
    start = <UFSC_BIT(2, 5)>; size = <2>;
  };
  ufsc_lid_sensor: lid-sensor {
    start = <UFSC_BIT(2, 13)>; size = <3>;
  };
};
```

UFSC, a 128-bit entry in the CBI (Cros Board Info)

The schema is described in Devicetree
(the common cbi_ufsc_std_schema.dtsi).

Devicetree-Driven Schema (UFSC Overlay)

```
&ufsc_kb_backlight {
   ufsc_kb_backlight_absent: kb-backlight-absent {
      compatible = "cros-ec,cbi-ufsc-value";
      value = <0>;
   };
   ufsc_kb_backlight_present: kb-backlight-present {
      compatible = "cros-ec,cbi-ufsc-value";
      value = <1>;
   };
};
&ufsc_pdc_chip_vendor_port_0 {
   ufsc_p0_rtk: rtk {
      compatible = "cros-ec,cbi-ufsc-value";
      value = <0>;
   };
   ...
};
```

The specific **values** are mapped in a **board-specific** Devicetree overlay (e.g. <board>_ec_ufsc.dtsi).

This file is auto-generated by our infrastructure.

This overlays the common schema.

The Devicetree schema simply generates the **bitmasks and offsets at compile time**.

At runtime, our macros just do a **bitwise AND** against that RAM cache.

Policy Pattern (Feature Toggles)

```
ufsc_policies: ufsc-policies {
   compatible = "cros-ec,cbi-ufsc-policies";
   kb_backlight_policy: kb-backlight {
      enable-value = <&ufsc_kb_backlight_present>;
   };
   kb_num_pad_policy: kb-num-pad {
      enable-value = <&ufsc_kb_num_pad_present>;
   };
   thermal_fan_policy: thermal-fan-pad {
      enable-value = <&ufsc_thermal_fan_present>;
   };
   ...
};
```

Global features (e.g. backlights, thermal fans) are enabled/disabled in generic init hooks, e.g. this source code:

```
/* Generic init hook */
if (cros_cbi_ufsc_check_match(
    UFSC_KB_BACKLIGHT_PRESENT )) {
   is_kblight_present = true;
} else {
   kblight_enable(0);
   is_kblight_present = false;
}
```

Improves testability via mocking frameworks (e.g. FFF).

Selector Pattern (Driver Routing)

```
pdc_selector: pdc-selector {
   compatible = "cros-ec,cbi-ufsc-pdc-selector";
   #address-cells = <1>;
   #size-cells = <0>;
   port0@0 {
      reg = <0>;
      selections = <&ufsc_p0_rtk &pdc_power_p0_rtk>,
                   <&ufsc_p0_ti &pdc_power_p0_ti>;
   };
   port1@1 {
      reg = <1>;
      selections = <&ufsc_p1_rtk &pdc_power_p1_rtk>,
                   <&ufsc_p1_ti &pdc_power_p1_ti>;
   };
};
```

Map specific UFSC values directly to Zephyr `&phandle` driver instances.

Subsystems request the driver for "Port 0" and receive the correct initialized `struct device *`.

Generating the Routing Table (Macro Magic)

```
port0@0 {
  selections = <&ufsc_p0_rtk &pdc_power_p0_rtk>,
               <&ufsc_p0_ti &pdc_power_p0_ti>;
};
```

Zero-Overhead Resolution (source code)

- **No Runtime Parsing:** zero string parsing
- Leverage **DT_DRV_COMPAT** and instance macros
- **BUILD_ASSERT** guarantees schema compliance

Compile-Time Loop (Zephyr Macro)

```
#define DT_DRV_COMPAT \
  cros_ec_cbi_ufsc_pdc_selector

DT_FOREACH_PROP_ELEM_VARGS(
  child_node, selections, CHECK_MATCH
)
```

Generated Output (optimized by the compiler)

```
switch(port) {
  case 0:
    if (cached_ufsc & MASK_RTK)  return &device_rtk;
    if (cached_ufsc & MASK_TI)  return &device_ti;
    return NULL; // Absent
  ...
}
```

Bypassing Auto-Init (`zephyr,deferred-init`)

```
&i2c3_0 {
   label = "I2C_PORT_PD";
   clock-frequency = <I2C_BITRATE_FAST_PLUS>;
   pdc_power_p0_rtk: rts54-pdc@66 {
      compatible = "realtek,rts54-pdc";
      reg = <0x66>;
      irq-gpios = <&gpioc 1 GPIO_ACTIVE_LOW>;
      zephyr,deferred-init;
      ccd;
   };
   pdc_power_p0_ti: tps6699-pdc@20 {
      compatible = "ti,tps6699-pdc";
      reg = <0x20>;
      irq-gpios = <&gpioc 1 GPIO_ACTIVE_LOW>;
      port-index-on-chip = <0x1>;
      zephyr,deferred-init;
      frs-supported;
   };
};
```

Compile all potential drivers for a given slot.

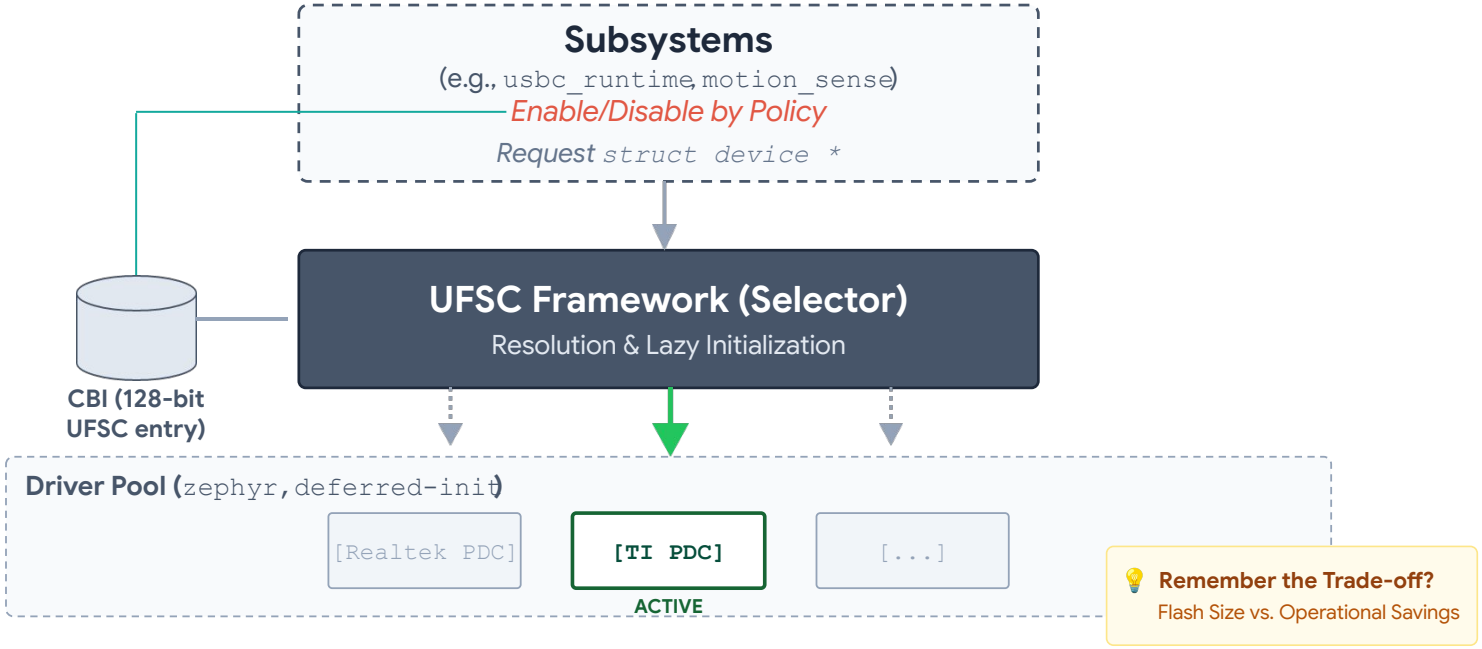
The Zephyr kernel allocates the `struct device` but skips the `init()` phase (prevents boot delays caused by I2C timeouts from missing components).

Selector logic acts as the dynamic initializer, booting *only* the chip that is selected by the UFSC.

💡 **Fun fact:**
The `zephyr,deferred-init` property was originally proposed by ChromeOS (Issue #39896) for this problem.

💡 Learn more in this talk by Ederson de Souza.

Overall Architecture Summary



Linux exposes the Devicetree in `sysfs`.

Zephyr compiles it away.

How does the Host OS find the hardware?

Hardware Discovery Gap

The Problem

The Host OS needs to identify hardware for factory provisioning and RAM, but the EC image is a static black box.

Why not probe from the EC?

- Zephyr drivers rarely support native `probe()` function.
- Implementing custom probing logic consumes RAM/Flash.

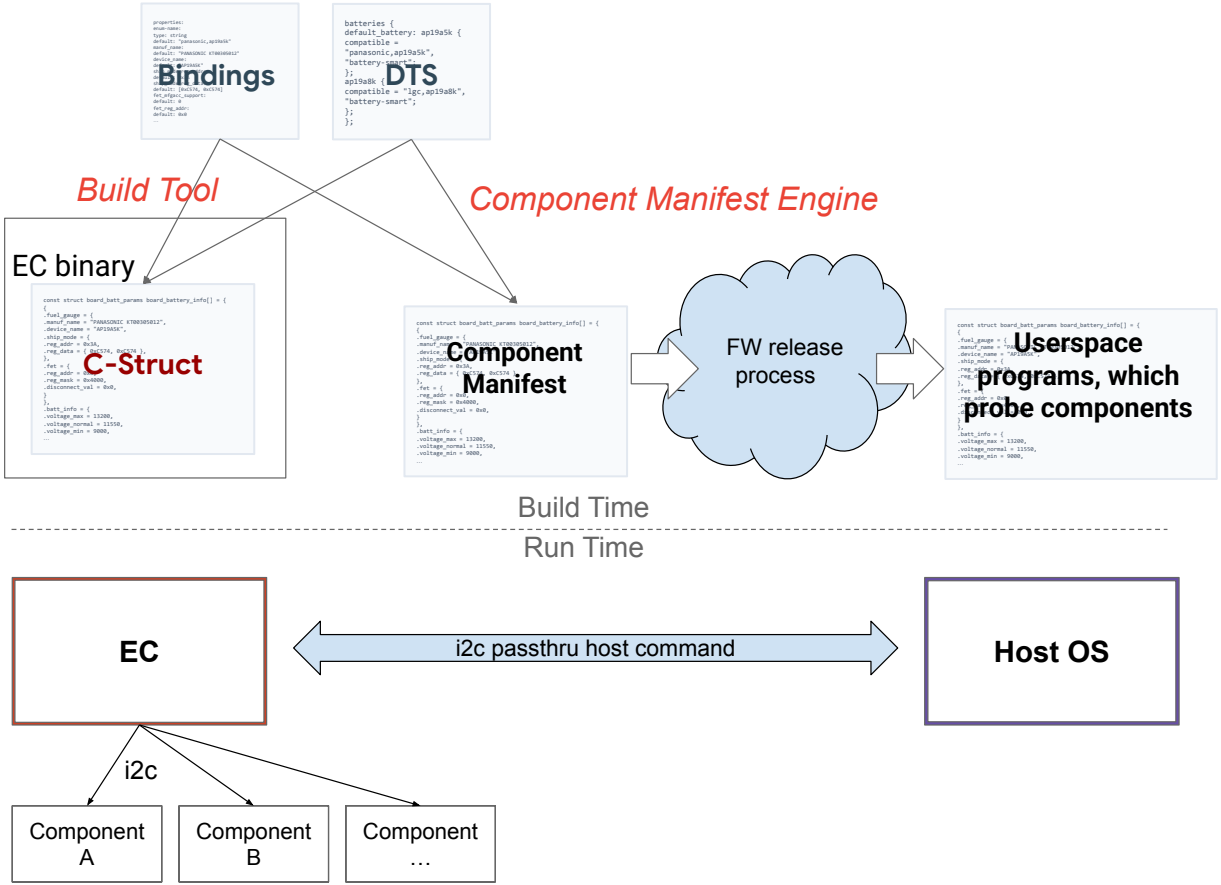
Why not probe from the Host OS?

- Our legacy approach used hard-coded rules to probe components.
- These rules often fall "out-of-sync" between the EC source and these userspace tools.

Solution: Component Manifest

Build Time: We generate a JSON file (`component_manifest.json`) from the pre-compiled Devicetree and pack it, along with the EC firmware image, into the Host OS image.

Run Time: The Host OS reads the JSON manifest and executes probe logic to discover the on-board components via I2C passthrough over the EC.


Bindings
DTS
Build Tool
Component Manifest Engine
EC binary
C-Struct
Component Manifest
FW release process
Userspace programs, which probe components
Build Time
Run Time
EC
i2c passthru host command
Host OS
i2c
Component A
Component B
Component …

What the Zephyr Compiler Does?

```
&i2c7 {
    ppc_nx20p_port1: nx20p348x@72 {
        compatible = "nxp,nx20p348x";
        status = "okay";
        reg = <0x72>;
        irq-gpios = <&gpiof 5 GPIO_ACTIVE_LOW>;
        is-alt;
    };
}
```

```
#define NX20P348X_COMPAT nxp_nx20p348x

#define PPC_CHIP_NX20P348X(id)                          \
      {                                                  \
            .i2c_port = I2C_PORT_BY_DEV(id),             \
            .i2c_addr_flags = DT_REG_ADDR(id),           \
            .drv = &nx20p348x_drv,                       \
            .irq_gpio = GPIO_DT_SPEC_GET_OR(id,          \
                        irq_gpios, {}),                  \
      }
```

What the Component Manifest Engine Does?

```
&i2c7 {
    ppc_nx20p_port1: nx20p348x@72 {
        compatible = "nxp,nx20p348x";
        status = "okay";
        reg = <0x72>;
        irq-gpios = <&gpiof 5 GPIO_ACTIVE_LOW>;
        is-alt;
    };
}
```

```
{
    "component_type": "ppc",
    "component_name": "nxp,nx20p348x",
    "i2c": {
        "port": 7,
        "addr": "0x72"
    },
    "usbc": {
        "port": 1
    }
},
```

How CME Works Internally?

- Zephyr's `gen_defines.py` creates an **EDT** (Extended Devicetree) object of the final Devicetree.
  - It's saved as a pickle file in the build directory.
- Parallel to the build process, we take the pickle and create the component manifest.
  - Create a dictionary of all the I2C ports the EC has access to.
  - Go through a list of devices we care about, for each device:
    - Check if it's on an I2C port.
    - Store the information that we care about in the JSON file.

Basic Manifest Entry

```
"component_list": [
   {
      "component_type": "ppc",
      "component_name": "nxp,nx20p348x",
      "i2c": {
          "port": 7,
          "addr": "0x72"
      },
      "usbc": {
          "port": 1
      }
   },
   ...
]
```

The components are keyed by type and name.

The `i2c` port and address are stored for probing.

Additional information (like `usbc` port mapping) is stored based on the component type.

Identifying Shared Drivers

```
"component_type": "ppc",
"component_name": "nxp,nx20p348x",
"i2c": {
    "port": 7,
    "addr": "0x72",
    "expect": [
      {
        "reg": "0x0f",
        "mask": "0xff",
        "value": "0x6a",
        "bytes": 1
      }
    ]
},
```

To identify multiple components which may share the same driver, a separate look-up table is stored with identifying information take from the spec sheets.

This is typically going to be the PID and DID of the component.

The probe-able register is stored in the “expect” field of the manifest.

Probing the Component from the Host OS

1. The Host OS parses the component manifest
2. Iterate through all the listed components
3. Ping the defined I2C address via EC I2C passthrough
4. If the component has an *expect* field, we also fetch the expected information from the register:
    - If the address responds, we know that the device exists and can print the information about the device.
    - If the address doesn't respond we return an error for not finding the device.

Integration Into UFSC

```
"component_type": "ppc",
"component_name": "nxp,nx20p348x",
"i2c": {
    "port": 7,
    "addr": "0x72"
},
"ufsc": {
    "mask": "0x70000",
    "value": 0
},
```

We take the mask and value for the components from the UFSC Devicetree and add it to the manifest.

The component manifest is not only used to verify the components, which are reachable on the board, but also used to generate the UFSC field provisioned to the CBI.

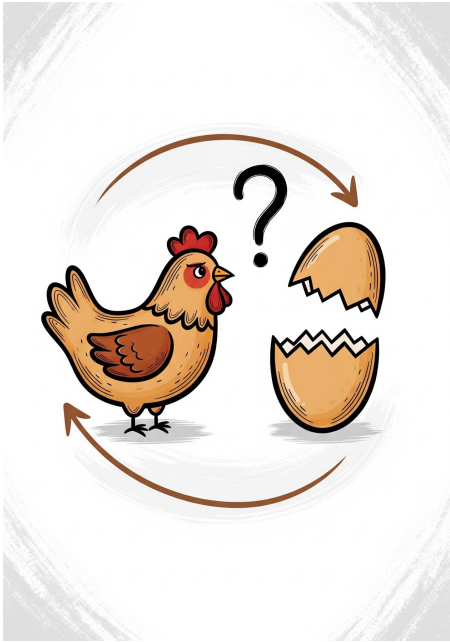
I2C Passthrough: Bypassing the Driver

Bus-Level Probing: The Host OS probes the I2C bus directly, instead of relying on a *driver-level* `probe()` function.

This solves a *Chicken-and-Egg Problem*:

- The UFSC framework skips initializing unverified components. A device-level probe would fail on an uninitialized driver.
- This probe failure makes it impossible to discover the hardware and provision the correct UFSC value.

For security reason, I2C tunneling is strictly restricted to the factory and RMA uses, and completely disabled for consumers.



Benefits and Limitations

Benefits

No memory overhead added to the EC.

No out-of-sync config between the EC and the Host.

Scalable for factory and RMA.

Limitations

Limited to simple I2C reads.

Duplicated probe methods if the driver already supports it.

Takeaways

Mass production requires dynamism

Second sourcing means the hardware is not truly static.

Compile-time macros are powerful

You can achieve dynamic routing with zero runtime cost in Zephyr.

Decouple discovery from the RTOS

Use a Component Manifest to let the Host OS safely verify hardware without driver logic.