IBM Linux Technology Center

QIDL: An Embedded Language to Serialize Guest Data Structures for Live Migration

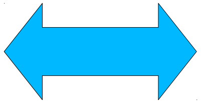
Michael Roth
mdroth@linux.vnet.ibm.com

QIDL in a nutshell

- **QEMU Interface Description Language**
- **Facilitates device state serialization**
- Annotations for struct fields (similar to GCC attributes)
  - describe how to serialize a field
  - describe whether a field should/shouldn't be serialized
- QIDL parser processes annotations and generates QAPI schemas for device state
- Existing QAPI code generator creates serialization/deserialization routines

Serializing/Deserializing device state

```
typedef struct RTCState {
    ...
    uint8_t cmos_data[128];
    uint8_t cmos_index;
    uint64_t base_rtc;
    uint64_t last_update;
    ...
} RTCState;
```



```
{
    "cmos_data": [
       57,
       0,
       …
    ],
    "cmos_index": 15,
    "base_rtc": 1351877119,
    "last_update":
       1351877119938261000,
     …
}
```

- **Useful for introspection**
- **Device testing**
- **Migration (more on that later)**

QIDL in a nutshell

- QEMU Interface Description Language
- Facilitates device state serialization
- **Annotations for struct fields (similar to GCC attributes)**
  - **describe how to serialize a field**
  - describe whether a field should/shouldn't be serialized
- QIDL parser processes annotations and generates QAPI schemas for device state
- Existing QAPI code generator creates serialization/deserialization routines

Disambiguating C types for serialization

- Can't always infer the proper way to serialize a field:
  - Arrays
    - size_t data_len;
    - uint32_t *data;
  - Is *data an array ptr? If so, how many elements?
    - size_t data_len;
    - uint32_t q_size(data_len) *data;
  - Character arrays vs. null-terminated strings
    - char my_char_array[64];
    - char q_string my_string[64]

QIDL in a nutshell

- QEMU Interface Description Language
- Facilitates device state serialization
- **Annotations for struct fields (similar to GCC attributes)**
    - describe how to serialize a field
    - **describe whether a field should/shouldn't be serialized**
- QIDL parser processes annotations and generates QAPI schemas for device state
- Existing QAPI code generator creates serialization/deserialization routines

Determining what to serialize

- **Serialize everything by default**
- **Strict conditions for exempting fields from serialization (rarely needed)**
- **Handful of annotations to handle this:**
  - q_immutable
  - q_derived
  - q_elsewhere

QIDL in a nutshell

- QEMU Interface Description Language
- Facilitates device state serialization
- Annotations for struct fields (similar to GCC attributes)
  - describe how to serialize a field
  - describe whether a field should/shouldn't be serialized
- **QIDL parser processes annotations and generates QAPI schemas for device state**
- **Existing QAPI code generator creates serialization/deserialization routines**

Converts Annotated Devices to QAPI Schemas

```
QIDL_DECLARE(RTCState) {
    ...
    uint8_t cmos_data[128];
    uint8_t cmos_index;
    uint64_t base_rtc;
    QEMUTimer *periodic_timer;
    ...
};
```

```
{
    'type': 'RTCState',
    'data': {
        'cmos_data': {
            '<annotated>': 'true',
            'type': ['uint8'],
            'array_size': 128,
        },
        'cmos_index': 'uint8',
        'base_rtc': 'uint64',
        'periodic_timer': 'QEMUTimer',
        ...
    },
}
```

- **Same schema format used for:**
- **QMP**
- **Guest Agent**
- **Netdev options (QemuOpts->C)**

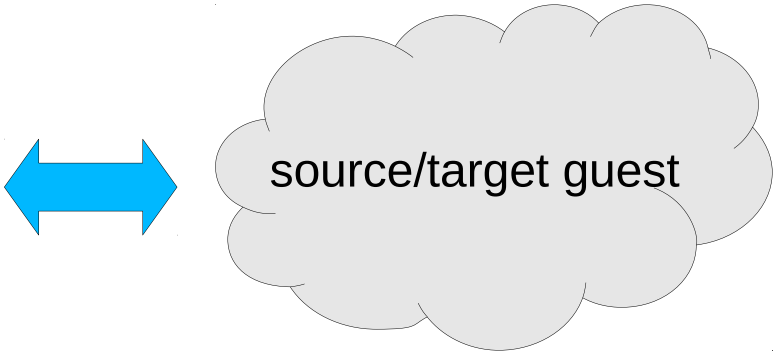
QIDL and Migration

- Currently we mostly use VMState to handle migration
  - **Associates wire fields with struct fields**
  - **Per-device/and per-field versioning**
  - Post-load functions can handle old->new translations (if we keep legacy fields, or legacy fields proved unrequired to begin with)
  - Subsections can avoid the need for new->old translations (if we don't make use of new fields)
  - Pre-save functions can handle new->old translations (if we keep legacy fields, no exceptions)
  - But often we don't keep legacy fields around...

Migration via VMState

```
static const VMStateDescription vmstate_rtc = {
    .name = "mc146818rtc",
    .version_id = 3,
    .minimum_version_id = 1,
    .minimum_version_id_old = 1,
    .post_load = rtc_post_load,
    .fields      = (VMStateField []) {
        VMSTATE_BUFFER(cmos_data, RTCState),
        VMSTATE_UINT8(cmos_index, RTCState),
        VMSTATE_UINT64_V(base_rtc, RTCState, 3),
        VMSTATE_UINT64_V(last_update, RTCState, 3),
        ...
        VMSTATE_END_OF_LIST()
    }
};
```

```
typedef struct RTCState {
    ...
    uint8_t cmos_data[128];
    uint8_t cmos_index;
    uint64_t base_rtc;
    uint64_t last_update;
    ...
} RTCState;
```



QIDL and Migration

- Currently we mostly use VMState to handle migration
  - Associates wire fields with struct fields
  - Per-device/and per-field versioning
  - Post-load functions can handle old->new translations (**if we keep legacy fields, or legacy fields proved unrequired to begin with**)
  - Subsections can avoid the need for new->old translations (**if we don't make use of new fields**)
  - Pre-save functions can handle new->old translations (**if we keep legacy fields, no exceptions**)
  - **But often we don't keep legacy fields around...**

Legacy fields tend to get dropped over time

```
mdroth@loki:~/w/qemu.git$ grep -r VMSTATE hw | grep UNUSED
hw/e1000.c:        VMSTATE_UNUSED_TEST(is_version_1, 4), /* was instance id */
hw/e1000.c:        VMSTATE_UNUSED(4), /* Was mmio_base.  */
hw/pxa2xx_dma.c:        VMSTATE_UNUSED_TEST(is_version_0, 4),
hw/mc146818rtc.c:        VMSTATE_UNUSED(7*4),
hw/mc146818rtc.c:        VMSTATE_UNUSED(3*8),
hw/eeprom93xx.c:        VMSTATE_UNUSED_TEST(is_old_eeprom_version, 1),
hw/zaurus.c:        VMSTATE_UNUSED_TEST(is_version_0, 2),
hw/stellaris.c:        VMSTATE_UNUSED(8),
hw/spitz.c:        VMSTATE_UNUSED_TEST(is_version_0, 5),
hw/ne2000.c:        VMSTATE_UNUSED(4), /* was irq */
hw/pcnet.c:        VMSTATE_UNUSED_TEST(is_version_2, 4),
hw/kvmvapic.c:        VMSTATE_UNUSED(8),     /* signature */
hw/rtl8139.c:        VMSTATE_UNUSED(4),
hw/ac97.c:        VMSTATE_UNUSED_TEST (is_version_2, 3),
hw/eepro100.c:        VMSTATE_UNUSED(32),
hw/eepro100.c:        VMSTATE_UNUSED(3*4),
hw/eepro100.c:        VMSTATE_UNUSED(19*4),
hw/ioapic_common.c:        VMSTATE_UNUSED_V(2, 8), /* to account for qemu-kvm's v2 format */
```

QIDL and Migration

- **Goal: Long-term, same-machine-level migration compatibility**
  - Lock in the wire protocol for pc-X after each release
  - Documented, stable wire protocol for pc-1.0, pc-1.1, etc.
  - During migration, translate internal device representation to the appropriate wire protocol based on the current machine level.
  - Basically, do what we do for -M pc-X for VMState as well.
  - What does QIDL have to do with any of this?

QIDL and Migration

- **Could do better now**
  - **Move legacy fields into compat structs**
  - **Add version-aware pre_save routines to derive legacy values from current device representation**
  - **Allow use of older vmstate version for outgoing migration**
- **Still skirting around the main issue**
  - **VMState is too tightly coupled to our internal device representations**
  - **Ideally: a VMState describes the API for instantiating a device for -M 1.0, or -M 1.1, etc**
  - **Our input is something we generate dynamically**

Leveraging QIDL for Migration

- **QIDL serializes device state to arbitrary formats, including QObjects**
- **Paths to fields in serialized objects correspond closely to struct fields**
- Legacy fields can be computed and added to object dynamically
- VMStateDescriptions can use stringified fields to key into the translated object

Serializing/Deserializing device state

```
typedef struct RTCState {
    ...
    uint8_t cmos_data[128];
    uint8_t cmos_index;
    uint64_t base_rtc;
    uint64_t last_update;
    ...
} RTCState;
```

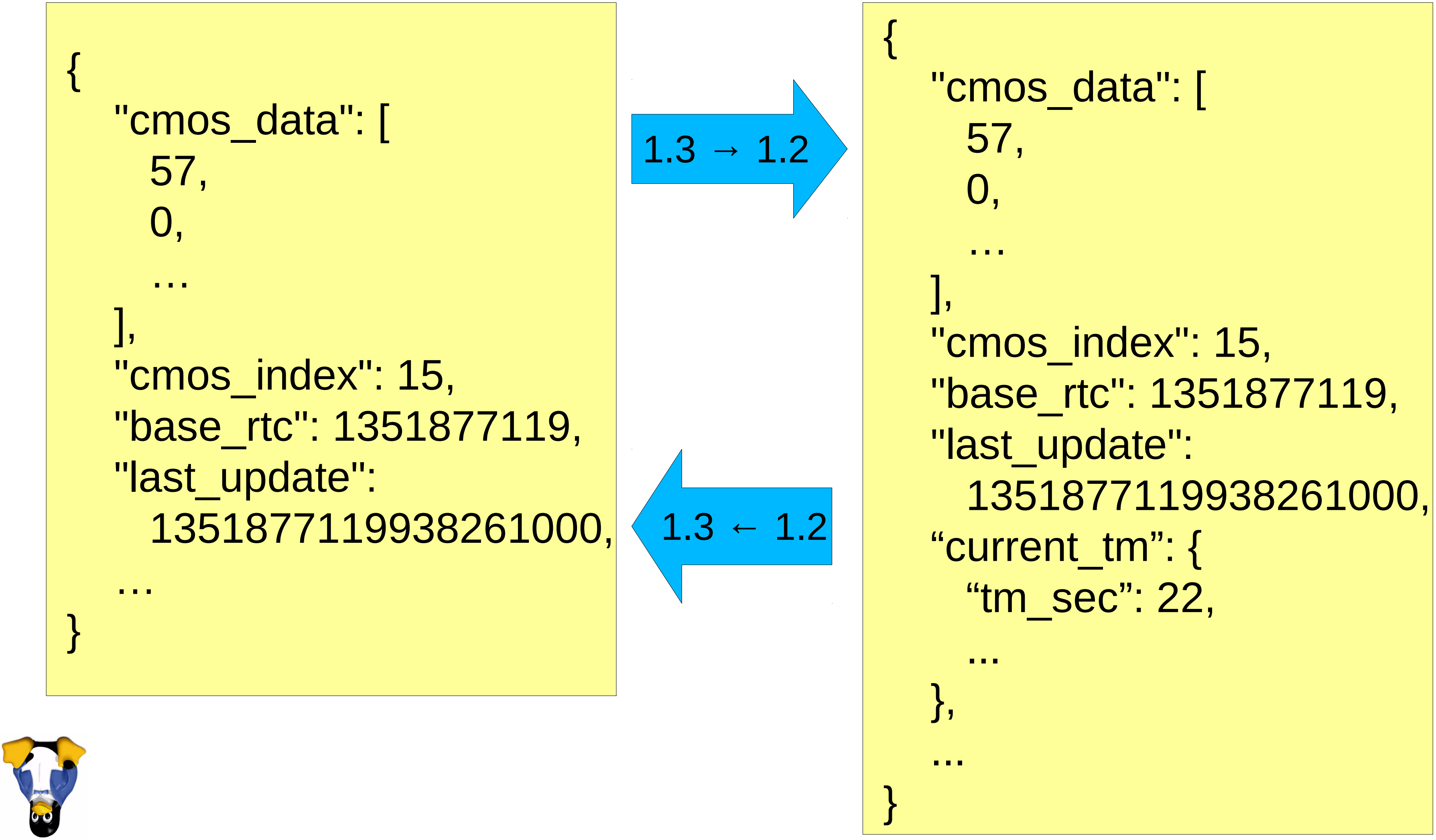
serialize

```
{
    "cmos_data": [
       57,
       0,
       …
    ],
    "cmos_index": 15,
    "base_rtc": 1351877119,
    "last_update":
       1351877119938261000,
    …
}
```

Leveraging QIDL for Migration

- QIDL serializes device state to arbitrary formats, including QObjects
- Paths to fields in serialized objects correspond closely to struct fields
- **Transformations on qobject can compute legacy fields and add them dynamically**
  - Can chain transformations to reduce maintenance (similar to how we handle qdev properties)
- VMStateDescriptions can use stringified fields to key into the translated object

Compatibility Transformations



```
{
   "cmos_data": [
     57,
     0,
     …
   ],
   "cmos_index": 15,
   "base_rtc": 1351877119,
   "last_update":
     1351877119938261000,
   …
}
```

1.3 → 1.2

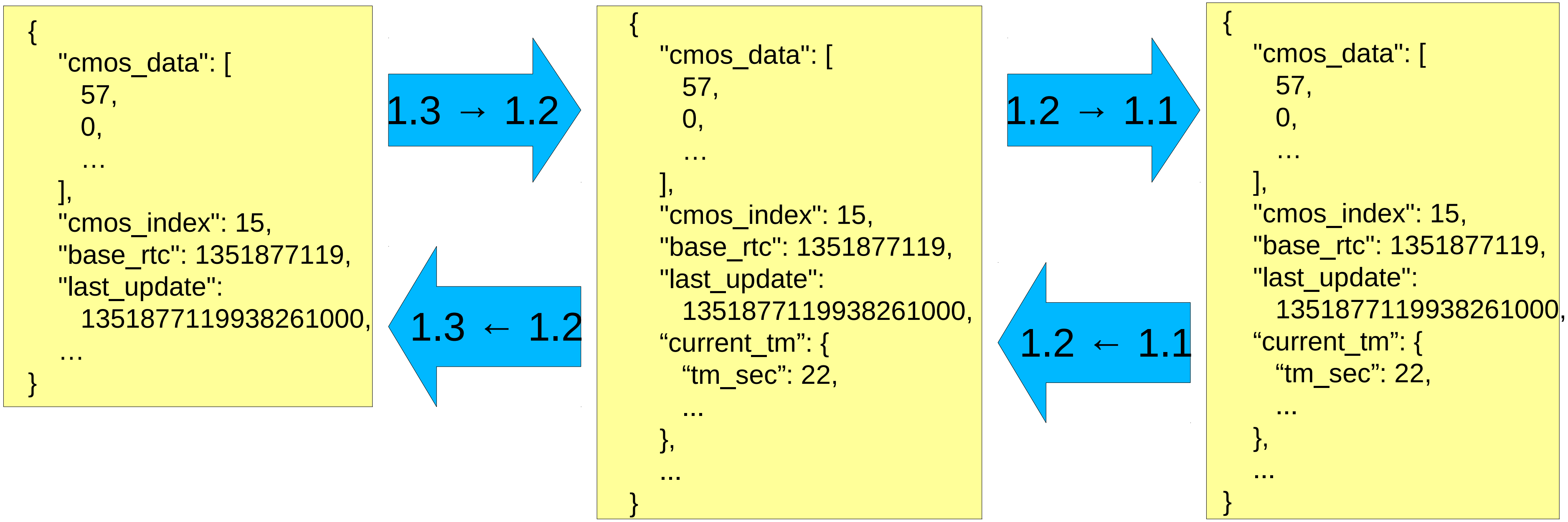
1.3 ← 1.2

```
{
   "cmos_data": [
     57,
     0,
     …
   ],
   "cmos_index": 15,
   "base_rtc": 1351877119,
   "last_update":
     1351877119938261000,
   “current_tm”: {
     “tm_sec”: 22,
     ...
   },
   ...
}
```

Leveraging QIDL for Migration

- QIDL serializes device state to arbitrary formats, including QObjects
- Paths to fields in serialized objects correspond closely to struct fields
- **Transformations on qobject can compute legacy fields and add them dynamically**
  - **Can chain transformations to reduce maintenance (similar to how we handle qdev properties)**
- VMStateDescriptions can use stringified fields to key into the translated object

Chained Compatibility Transformations



```
{
    "cmos_data": [
      57,
      0,
      …
    ],
    "cmos_index": 15,
    "base_rtc": 1351877119,
    "last_update":
      1351877119938261000,
    …
}
```

1.3 → 1.2

1.3 ← 1.2

```
{
    "cmos_data": [
      57,
      0,
      …
    ],
    "cmos_index": 15,
    "base_rtc": 1351877119,
    "last_update":
      1351877119938261000,
    “current_tm”: {
      “tm_sec”: 22,
      ...
    },
    ...
}
```

1.2 → 1.1

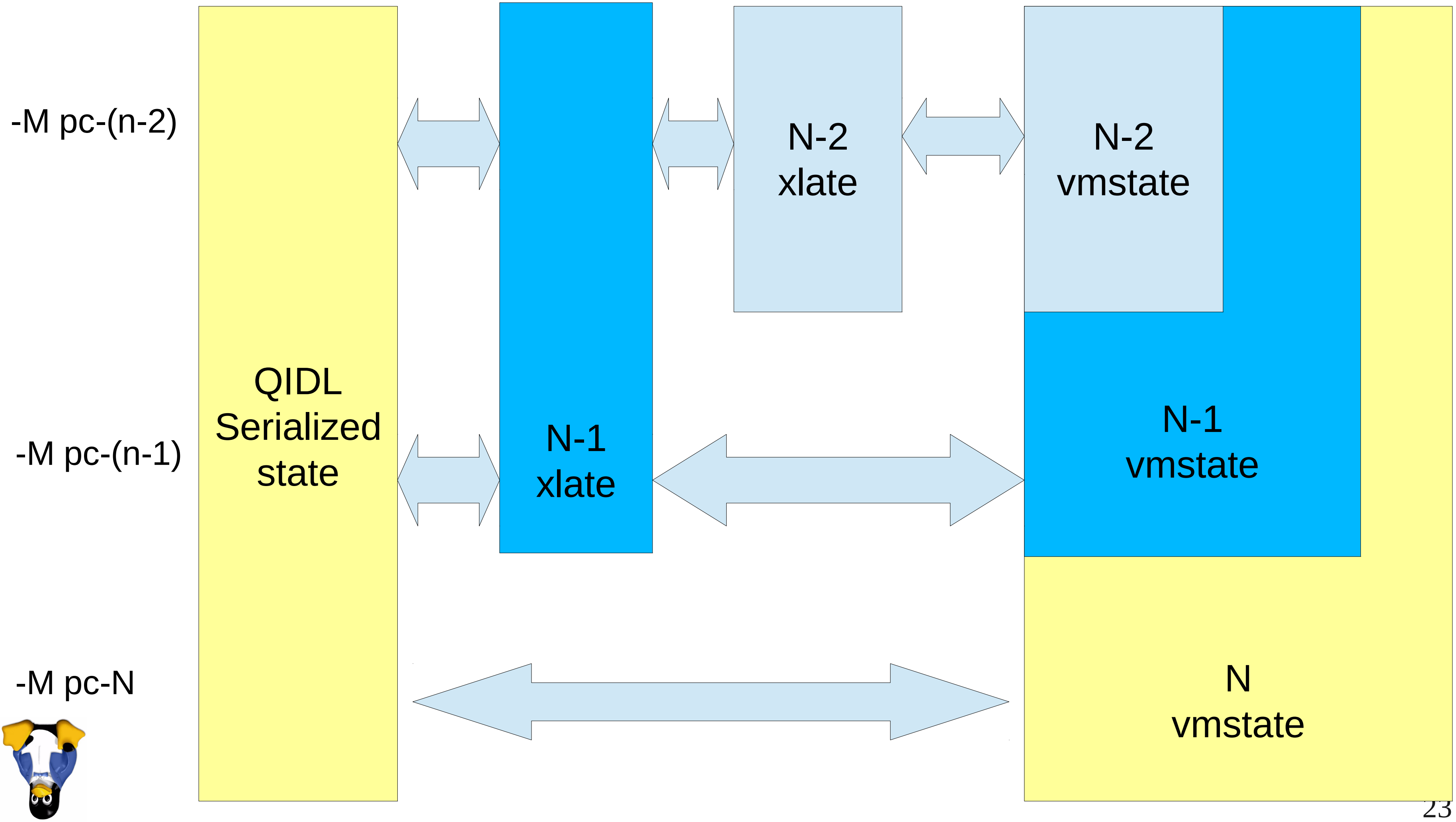
1.2 ← 1.1

```
{
    "cmos_data": [
      57,
      0,
      …
    ],
    "cmos_index": 15,
    "base_rtc": 1351877119,
    "last_update":
      1351877119938261000,
    “current_tm”: {
      “tm_sec”: 22,
      ...
    },
    ...
}
```

Leveraging QIDL for Migration

- QIDL serializes device state to arbitrary formats, including QObjects
- Paths to fields in serialized objects correspond closely to struct fields
- Transformations on qobject can compute legacy fields and add them dynamically
    - Can chain transformations to reduce maintenance (similar to how we handle qdev properties)
- **VMStateDescriptions can use stringified fields to key into the translated object**

Putting it all Together



Status and Future Plans

- Patches on the list for base infrastructure
- Patches on the list for first set of device conversions:
  - PCI, piix3-ide, mc146818rtc, hpet, cirrus-vga, PIIX3, i440FX, pci-bridge
- Standard PC devices by 1.4, underway
- QIDL-compatible VMState by 1.4, depending community feedback
- Convert individual devices to using QIDL for migration on an as-needed basis

- Questions?