Towards a more expressive and introspectable QEMU command line

Markus Armbruster <armbru@redhat.com>
KVM Forum 2017
Part I
Things we want from the command line
A simple example

You could run QEMU like this:

```bash
$ qemu -s -machine usb=on,accel=kvm disk.qcow2
```

Observe:

- Options, with and without arguments, as usual
- Complex argument of the form `key=value`,…
A real-world example

We clearly push CLI beyond its intended use...
Wanted: config files

Some use cases are better served by config files:

$ qemu-system-x86_64 -readconfig vm1.cfg

Everything CLI should also work in config files
Another config interface: QMP

QEMU Monitor Protocol (QMP):

QMP> {"execute": "blockdev-add",
    "arguments": {"node-name": "node1",
                  "driver": "file",
                  "filename": "tmp.img"}}

{"return": {}}

Observe:

- Commands and responses are JSON objects
Why *two* config interfaces?

Run-time reconfiguration must use QMP

Much initial configuration uses CLI, because...
Why *two* config interfaces?

Run-time reconfiguration must use QMP

Much initial configuration uses CLI, because

- we’ve always done it this way (and we turn like a tanker)
Why *two* config interfaces?

Run-time reconfiguration must use QMP

Much initial configuration uses CLI, because

- we’ve always done it this way (and we turn like a tanker)
- we’re *devoted* to backward compatibility
Wanted: equality

Some configuration is needed both in CLI and QMP (e.g. -chardev & chardev-add, -object & object-add)

Our infrastructure should support this:

- CLI and QMP need to be equally expressive
  - QMP needs to express CLI’s key=value,…
  - CLI needs to express QMP’s JSON objects
Wanted: equality

Some configuration is needed both in CLI and QMP (e.g. -chardev & chardev-add, -object & object-add)

Our infrastructure should support this:
- CLI and QMP need to be equally expressive
  - QMP needs to express CLI’s key=value,…
  - CLI needs to express QMP’s JSON objects
- Want to drive single C interface with equal ease
CLI evolves constantly

![Graph showing the evolution of CLI lines over time with a 5% annual growth rate](image-url)
Wanted: interface introspection

Programs interfacing with QEMU need to know:

- What options are available?
  Example: does this QEMU support `-blockdev`?
- What keys does an option support?
  Example: does `-spice` support `unix`?
- What values does a key support?
  Example: does `-blockdev` support `driver=gluster`?

⇝ CLI needs to support introspection
Part II
Are these needs met?
(TLDR: nope)
CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    
    
    "-msg timestamp[=on|off]\n"
    
    " change the format of messages\n"
    
    " on|off controls leading timestamps\n",
    
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
.prepend a timestamp to each log message.
ETEXI
```
Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "on|off controls leading timestamps\n", QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```
CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    " change the format of messages\n"
    " on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)
STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```
CLI option definition

Options are defined like this:

DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    " change the format of messages\n"
    " on|off controls leading timestamps\n",
    QEMU_ARCH_ALL)

prepend a timestamp to each log message.
CLI option definition

Options are defined like this:

```
DEF("msg", HAS_ARG, QEMU_OPTION_msg,
    "-msg timestamp[=on|off]\n"
    " change the format of messages\n"
    "
    QEMU_ARCH_ALL)

STEXI
@item -msg timestamp[=on|off]
@findex -msg
prepend a timestamp to each log message.
ETEXI
```

Optarg format in help and manual text, but not in code.
How we parse CLI

Get next option and optarg (if any)

Simple optarg
- help, -hda FILE

Execute e.g. print help

Record for later in global state

More options?
How we parse CLI

Get next option and optarg (if any)

- Parse optarg
- Record for later in global state

Complex optarg
- `cpu CPU,[-+]FLAG,...`

Execute e.g. print help

20+ ad hoc parsers

More options?
How we parse CLI

- Get next option and optarg (if any)
- Parse optarg
- Record for later in global state
- More options?
- Execute e.g. print help

Complex optarg

-drive KEY=VAL,...

QemuOpts parse optarg & record for later
generic parser for key=value,...
How we parse CLI

1. Get next option and optarg (if any)
2. Parse optarg
3. Record for later in global state
4. QemuOpts parse optarg & record for later
5. Extra actions
6. More options?
7. Execute e.g. print help
8. Record for later in global state
How we parse CLI

- Get next option and optarg (if any)
- Parse optarg
- Record for later in global state
- Execute e.g. print help
- Record for later in global state
- More options?
- Extra actions

QemuOpts bypassed

20+ ad hoc parsers

QemuOpts parse optarg & record for later
How we parse CLI

QemuOpts bypassed

Get next option and optarg (if any)

Parse optarg

Record for later in global state

Extra actions

QemuOpts parse optarg & record for later

More options?

Execute e.g. print help

Impure
How we parse CLI

Get next option and optarg (if any)

Parse optarg

Record for later in global state

More options?

QemuOpts bypassed

Execute e.g. print help

QemuOpts parse optarg & record for later

Extra actions

OK

Impure
Impact on CLI config files

Config files apply to QemuOpts state:
- `writeconfig` writes it out
- `readconfig` reads it in

Impact of QemuOpts’ sad state on config files:
- **Okay:** works
- **Impure:** broken (extra actions skipped)
- **Bypassed:** not supported

Config files work for one out of five options
Impact on introspection

Introspection is based on QemuOpts:

- QMP’s query-command-line-options has no other source of truth

Impact on introspection:

- Okay: works
- Impure: works anyway
- Bypassed: not supported

Can introspect one out of five options
Fix by QemuOpts taking over?

Get next option
and optarg (if any)

Plan A since forever

More options?

QemuOpts
parse optarg &
record for later
Fix by QemuOpts taking over?

Get next option and optarg (if any)

Plan A since forever
Problem: stay compatible to 20+ ad hoc parsers

More options?
Fix by QemuOpts taking over?

Get next option and optarg (if any)

Plan A since forever
Problem: stay compatible to 20+ ad hoc parsers
Also: QemuOpts has issues

QemuOpts parse optarg & record for later

More options?
QemuOpts’ data model

Derived from abstract \textit{key=\textit{value},...} syntax:
- Keyword parameters, all optional
- Parameter \textit{values} are \textit{strings}

Encapsulated in type \texttt{QemuOpts}:
- Parameters are \textit{dynamically typed}

Thrown in for convenience:
- Can restrict values to integer or bool

Stupidest model that could possibly work
Simple example: -msg

-msg has just one bool parameter timestamp

QemuOpts declaration:

```c
static QemuOptsList qemu_msg_opts = {
  .name = "msg",
  [boilerplate omitted...]
  .desc = {
    {
      .name = "timestamp",
      .type = QEMU_OPT_BOOL,
    },
    { 0 }
  },
};
```
Simple example: -msg

msg has just one bool parameter timestamp

QemuOpts declaration:

```c
static QemuOptsList qemu_msg_opts = {
  .name = "msg",
  .desc = {
    {
      .name = "timestamp",
      .type = QEMU_OPT_BOOL,
    },
    { 0 }
  },
};
```
Simple example: `-msg`

`-msg` has just one bool parameter `timestamp`

QemuOpts declaration:

```c
static QemuOptsList qemu_msg_opts = {
  .name = "msg",
  [boilerplate omitted...]
  .desc = {
    {
      .name = "timestamp",
      .type = QEMU_OPT_BOOL,
    },
    { 0 }
  },
};
```

Note: option argument definition is separate from option definition
Next example: -numa

-numa has a mandatory parameter type. Additional parameters depend on value of type (e.g. with type=node, we have parameter nodeid)

QemuOpts declaration:

```c
static QemuOptsList qemu_numa_opts = {
    .name = "numa",
    [boilerplate omitted...]
    .desc = {
        { 0 }
        { 0 }
    },
};
```
Next example: -numa

-numa has a mandatory parameter type
Additional parameters depend on value of type
(e.g. with type=node, we have parameter nodeid)

QemuOpts declaration:

```c
static QemuOptsList qemu_numa_opts = {
    .name = "numa",
    .desc = {
        { 0 }
    },
};
```

Best QemuOpts can do:
accept any key, with string value
(bye, bye introspection)
Code to parse -numa’s optarg

QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
    [details...]);

[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
Code to parse -numa’s optarg

QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
    [details...]);

[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
Code to parse -numa’s optarg

```c
QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg,
                                 [details...]);

[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
```
Code to parse -numa’s optarg

QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg, [details...]);

Get and check parameters

const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
Code to parse -numa’s optarg

QemuOptsList *list = qemu_find_opts("numa");
QemuOpts *opts = qemu_opts_parse(list, optarg, 
[details...]);

[error out if !opts...]
const char *type = qemu_opt_get(opts, "type");
if (!type) {
    [error out...]
} else if (!strcmp(type, "node")) {
    const char *s = qemu_opt_get(opts, "nodeid");
    uint64_t nodeid = qemu_strtou64(s, [...]);
    [more checking...]
} else [more cases...]
Third example: -blockdev

-blockdev is like QMP blockdev-add

QMP> {"execute": "blockdev-add", "arguments": {
    "driver": "qcow2", "node-name": "node1",
    "file": {
        "driver": "file",
        "filename": "disk1.qcow2"
    }
}}

"arguments" is a tree
Third example: -blockdev

-blockdev is like QMP blockdev-add

QMP> {"execute": "blockdev-add", "arguments": {
    "driver": "qcow2", "node-name": "node1",
    "file": { "driver": "file",
        "filename": "disk1.qcow2"}}}

But QemuOpts is by design flat...
Third example: -blockdev

-blockdev is like QMP blockdev-add

QMP> {"execute": "blockdev-add", "arguments": {
    "driver": "qcow2", "node-name": "node1",
    "file":{"driver": "file",
      "filename": "disk1.qcow2"}}}

But QemuOpts is by design flat...
Flatten the arguments tree with dotted keys:

-blockdev driver=qcow2,node-name=node1,\
  file.driver=file,file.filename=tmp.qcow2
Dotted keys in a nutshell

Basic idea:
- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

```
{ ... }

{ ... }

"driver": "file",
"filename": "disk1.qcow2"
```
Dotted keys in a nutshell

Basic idea:
- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:
- Bolted onto QemuOpts
  → Bye, bye introspection, hello extra code
Dotted keys in a nutshell

Basic idea:
- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:
- Bolted onto QemuOpts
  → Bye, bye introspection, hello extra code
- Inconsistent with other workarounds
Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  ~→ Bye, bye introspection, hello extra code
- Inconsistent with other workarounds
  (to be kept for backward compatibility)
Dotted keys in a nutshell

Basic idea:
- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:
- Bolted onto QemuOpts
  - Bye, bye introspection, hello extra code
- Inconsistent with other workarounds
  (to be kept for backward compatibility)
- Can express most, but not all trees
Dotted keys in a nutshell

Basic idea:

- Specify tree by listing its (string-valued) leaves
- Dotted key encodes path to leaf

Clever, but has issues:

- Bolted onto QemuOpts
  - Bye, bye introspection, hello extra code
- Inconsistent with *other* workarounds
  (to be kept for backward compatibility)
- Can express most, but not all trees
- More, but we’re running out of time
Needs our CLI fails to meet

**Config files:** incomplete & inadequate
**Expressive power:** weaker than QMP
**Single C interface:** tedious glue code needed
**Introspection:** anemic compared to QMP
Part III
Solutions
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
    case NUMA_OPTIONS_TYPE_NODE:
        uint16_t nodeid = numa->u.node.nodeid;
        [more...]
        [more cases...]
}
Actual code to parse `--numa`

```c
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
   [error out...]
}
switch (numa->type) {
   case NUMA_OPTIONS_TYPE_NODE:
      uint16_t nodeid = numa->u.node.nodeid;
      [more...]
      [more cases...]
```
Actual code to parse -numa

Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeidid;
    [more...]
    [more cases...]
Actual code to parse `-numa`

```c
Visitor *v;
NumaOptions *numa;

v = opts_visitor_new(opts);
visit_type_NumOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}
switch (numa->type) {
  case NUMA_OPTIONS_TYPE_NODE:
    uint16_t nodeid = numa->u.node.nodeid;
    [more...]
[more cases...]
```

Look Ma, no strings!
Okay, but what’s a QAPI type?

QAPI types are defined in a QAPI schema like this:

```json
{
  'union': 'NumaOptions',
  'base': {
    'type': 'NumaOptionsType'
  },
  'discriminator': 'type',
  'data': {
    'node': 'NumaNodeOptions',
    [more variants...]
  }
}
{
  'enum': 'NumaOptionsType',
  'data': [
    'node', [more values...]
  ]
}
```

QAPI compiles them to C types plus code for serializing to/from JSON, introspection, ...
QAPI C type NumaOptions

typedef enum NumaOptionsType {
    NUMA_OPTIONS_TYPE_NODE = 0,
    [more type values...]
} NumaOptionsType;

struct NumaOptions {
    NumaOptionsType type;
    union { /* union tag is @type */
        NumaNodeOptions node;
        [more variants...]
    } u;
};
Where QAPI beats QemuOpts

- More expressive type system
  - enumerations
  - algebraic types
  - arbitrarily nested

- More precise introspection

- Generated C types beat QemuOpts & strings
  - Interfaces made for QMP use nice C types
  - Interfaces made for CLI use QemuOpts (and suck)
  - Driving single interface requires conversion
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

⇝ Plan B: QemuOpts mapped to QAPI takes over

However:
Option is then defined in three places
Assumes flat, conflicts with dotted keys
Compatibility headaches
Fixable, but too much stuff bolted onto QemuOpts

⇝ Plan C: QAPI takes over
Status: cooking
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

⇒ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

⇝ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys
- Compatibility headaches
Fix by mapping to QAPI types?

Mapping QemuOpts to QAPI looks like progress

⇝ Plan B: QemuOpts mapped to QAPI takes over

However:

- Option is then defined in *three* places
- Assumes flat, conflicts with dotted keys
- Compatibility headaches

Fixable, but too much stuff bolted onto QemuOpts

⇝ Plan C: **QAPI** takes over

Status: cooking 😊
QAPIfy CLI option definition

```
##
# @--msg:
# prepend a timestamp to each log message
##
{
  'option': '-msg',
  'data': { '*timestamp': 'bool' },
  'help': [ 
    "-msg timestamp[=on|off]",
    "  change the format of messages",
    "  on|off controls leading timestamps"]
}
```

Observe:
- Option defined in *one* place: QAPI schema
- Like QMP command less 'returns' plus 'help'
QAPI-generated code

All option definitions together compile to:

```c
QAPIOption *qapi_options_parse(int argc,
                                  char *argv[]);
```

Takes argument vector

Returns array of parsed options QAPIOption[]
QAPIOption is tagged union of option arguments
New code to parse -numa

Visitor *v;
NumaOptions *numa = qopt[i].u.numa;

v = opts_visitor_new(opts);
visit_type_NumaOptions(v, NULL, &numa, &err);
visit_free(v);
if (!numa) {
    [error out...]
}

switch (numa->type) {
    case NUMA_OPTIONS_TYPE_NODE:
        uint16_t nodeid = numa->u.node.nodeid;
        [more...]
    [more cases...]
Alternative optarg syntax: JSON

qapi_options_parse() supports JSON in addition to dotted key=value,…

Dotted keys are
- nicer for simple cases
- needed for backward compatibility

JSON is
- more general
- recommended for programmatic use
Can this meet our CLI needs?

Config files ✓ (JSON)
  can read config just like we read QMP
Can this meet our CLI needs?

**Config files** ✓ (JSON)
  can read config just like we read QMP

**Expressive power** ✓
  same as QMP with JSON
  close with dotted keys
Can this meet our CLI needs?

**Config files** ✓ (JSON)
- can read config just like we read QMP

**Expressive power** ✓
- same as QMP with JSON
- close with dotted keys

**Single C interface** ✓
- types shared with QMP
Can this meet our CLI needs?

**Config files** ✓ (JSON)
- can read config just like we read QMP

**Expressive power** ✓
- same as QMP with JSON
- close with dotted keys

**Single C interface** ✓
- types shared with QMP

**Introspection** ✓
- can do just like QMP
Can this meet our CLI needs?

**Config files** ✓ (JSON)
- can read config just like we read QMP

**Expressive power** ✓
- same as QMP with JSON
- close with dotted keys

**Single C interface** ✓
- types shared with QMP

**Introspection** ✓
- can do just like QMP

**Backward compatibility** ?
Backward compatibility

Existing 20+ ad hoc parsers
- Short term: make optarg a string, pass to parser
- (bye, bye introspection)

Existing bolted-on dotted keys
- I think we're good there

Existing bolted-on conversion to QAPI
- Replicate its extra features and corner cases: flattening, integer lists, ...

QemuOpts eccentricities
- Replicate those too: syntactic sugar, trickery around repeated keys, special key id, ...
Backward compatibility

_existing 20+ ad hoc parsers_

- Short term: make optarg a string, pass to parser
  (bye, bye introspection)
- Long term: support alternate parsers
Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers

- Existing bolted-on dotted keys
  - I think we’re good there
Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers
- Existing bolted-on dotted keys
  - I think we’re good there
- Existing bolted-on conversion to QAPI
  - Replicate its extra features and corner cases: flattening, integer lists, ...
Backward compatibility

- Existing 20+ ad hoc parsers
  - Short term: make optarg a string, pass to parser (bye, bye introspection)
  - Long term: support alternate parsers
- Existing bolted-on dotted keys
  - I think we’re good there
- Existing bolted-on conversion to QAPI
  - Replicate its extra features and corner cases: flattening, integer lists, ...
- QemuOpts eccentricities
  - Replicate those too: syntactic sugar, trickery around repeated keys, special key id, ...
Must it be?

Backward compatibility is a choice
We choose how much pain to inflict on ourselves to avoid inconveniencing users
Up to what point are the opportunity costs worth it?

Muss es sein?
Status: cooking, needs work

Posted: [RFC PATCH] Command line QAPIfication

Highlights and (some of the) lowlights:

- All options QAPIfied
- Most option arguments not fully QAPIfied, yet (backward compatibility work hiding here)
- Introspection works
- Config file not yet implemented
- Generated docs look more terrible than usual
Questions?
Thanks & good bye

You might find these links useful:

- [RFC PATCH] Command line QAPIfication

- QEMU interface introspection: from hacks to solutions

No rubber chickens were harmed in the making of this presentation