NAME

guestfs−hacking − extending and contributing to libguestfs

DESCRIPTION

This manual page is for hackers who want to extend libguestfs itself.

THE SOURCE CODE

Libguestfs source is located in the github repository https://github.com/libguestfs/libguestfs

Large amounts of boilerplate code in libguestfs (RPC, bindings, documentation) are generated. This means that many source files will appear to be missing from a straightforward git checkout. You have to run the generator (`./configure && make −C generator`) in order to create those files.

Libguestfs uses an autotools-based build system, with the main files being *configure.ac* and *Makefile.am*. See ''THE BUILD SYSTEM''.

The *generator* subdirectory contains the generator, plus files describing the API. The*lib* subdirectory contains source for the library. The*appliance* and *daemon* subdirectories contain the source for the code that builds the appliance, and the code that runs in the appliance respectively. Other directories are covered in the section ''SOURCE CODE SUBDIRECTORIES'' below.

Apart from the fact that all API entry points go via some generated code, the library is straightforward. (In fact, even the generated code is designed to be readable, and should be read as ordinary code). Some actions run entirely in the library, and are written as C functions in files under *lib*. Others are forwarded to the daemon where (after some generated RPC marshalling) they appear as C functions in files under *daemon*.

To build from source, first read the **guestfs−building** (1).

SOURCE CODE SUBDIRECTORIES

There are a lot of subdirectories in the source tree! Which ones should you concentrate on first? *lib* and *daemon* which contain the source code of the core library. *generator* is the code generator described above, so that is important. The *Makefile.am* in the root directory will tell you in which order the subdirectories get built. And then if you are looking at a particular tool (eg. *customize*) or language binding (eg. *python*), go straight to that subdirectory, but remember that if you didn't run the generator yet, then you may find files which appear to be missing.

align
: **virt−alignment−scan** (1) command and documentation.

appliance
: The libguestfs appliance, build scripts and so on.

bash
: Bash tab-completion scripts.

build-aux
: Various build scripts used by autotools.

builder
: **virt−builder** (1) command and documentation.

bundled
: Embedded copies of other libraries, mostly for convenience (and the embedded library is not widespread enough).

  bundled/ocaml−augeas
  : Bindings for the Augeas library. These come from the ocaml-augeas library http://git.annexia.org/?p=ocaml−augeas.git

cat
: The **virt−cat** (1), **virt−filesystems** (1), **virt−log** (1), **virt−ls** (1) and **virt−tail** (1) commands and documentation.

common

Various libraries of internal code can be found in the *common* subdirectory:

common/edit
: Common code for interactively and non-interactively editing files within a libguestfs filesystem.

common/errnostring
: The communication protocol used between the library and the daemon running inside the appliance has to encode errnos as strings, which is handled by this library.

common/mlcustomize
: Library code associated with `virt-customize` but also used in other tools.

common/mlgettext
: Small, generated wrapper which allows libguestfs to be compiled with or without ocaml-gettext. This is generated by *./configure*.

common/mlpcre
: Lightweight OCaml bindings for Perl Compatible Regular Expressions (PCRE). Note this is not related in any way to Markus Mottl's ocaml-pcre library.

common/mlprogress
: OCaml bindings for the progress bar functions (see *common/progress*).

common/mlstdutils
: A library of pure OCaml utility functions used in many places.

common/mltools
: OCaml utility functions only used by the OCaml virt tools (like `virt-sysprep`, `virt-customize` etc.)

common/mlutils
: OCaml bindings for C functions in `common/utils`, and some POSIX bindings which are missing from the OCaml stdlib.

common/mlvisit
: OCaml bindings for the visit functions (see *common/visit*).

common/mlxml
: OCaml bindings for the libxml2 library.

common/options
: Common options parsing for guestfish, guestmount and some virt tools.

common/parallel
: A framework used for processing multiple libvirt domains in parallel.

common/progress
: Common code for printing progress bars.

common/protocol
: The XDR-based communication protocol used between the library and the daemon running inside the appliance is defined here.

common/qemuopts
: Mini-library for writing qemu command lines and qemu config files.

common/structs
: Common code for printing and freeing libguestfs structs, used by the library and some tools.

common/utils
: Various utility functions used throughout the library and tools.

common/visit
: Recursively visit a guestfs filesystem hierarchy.

common/windows
Utility functions for handling Windows drive letters.

contrib
Outside contributions, experimental parts.

customize
virt−customize (1) command and documentation.

daemon
The daemon that runs inside the libguestfs appliance and carries out actions.

df **virt−df** (1) command and documentation.

dib **virt−dib** (1) command and documentation.

diff **virt−diff** (1) command and documentation.

docs
Miscellaneous manual pages.

edit **virt−edit** (1) command and documentation.

examples
C API example code.

fish **guestfish** (1), the command-line shell, and various shell scripts built on top such as **virt−copy−in** (1), **virt−copy−out** (1), **virt−tar−in** (1), **virt−tar−out** (1).

format
virt−format (1) command and documentation.

fuse
guestmount (1), FUSE (userspace filesystem) built on top of libguestfs.

generator
The crucially important generator, used to automatically generate large amounts of boilerplate C code for things like RPC and bindings.

get-kernel
virt−get−kernel (1) command and documentation.

inspector
virt−inspector (1), the virtual machine image inspector.

lib Source code to the C library.

logo
Logo used on the website. The fish is called Arthur by the way.

m4 M4 macros used by autoconf. See ‘‘THE BUILD SYSTEM’’.

make-fs
virt−make−fs (1) command and documentation.

po Translations of simple gettext strings.

po-docs
The build infrastructure and PO files for translations of manpages and POD files. Eventually this will be combined with the *po* directory, but that is rather complicated.

rescue
virt−rescue (1) command and documentation.

resize
virt−resize (1) command and documentation.

sparsify
: **virt−sparsify** (1) command and documentation.

sysprep
: **virt−sysprep** (1) command and documentation.

tests
: Tests.

test-data
: Files and other test data used by the tests.

test-tool
: Test tool for end users to test if their qemu/kernel combination will work with libguestfs.

tmp
: Used for temporary files when running the tests (instead of */tmp* etc). The reason is so that you can run multiple parallel tests of libguestfs without having one set of tests overwriting the appliance created by another.

tools
: Command line tools written in Perl (**virt−win−reg** (1) and many others).

utils
: Miscellaneous utilities, such as `boot−benchmark`.

v2v
: Up to libguestfs > 1.42 this contained the **virt−v2v** (1) tool, but this has now moved into a separate repository: https://github.com/libguestfs/virt−v2v

website
: The http://libguestfs.org website files.

csharp
erlang
gobject
golang
haskell
java
lua
ocaml
php
perl
python
ruby
: Language bindings.

THE BUILD SYSTEM

Libguestfs uses the GNU autotools build system (autoconf, automake, libtool).

The *./configure* script is generated from *configure.ac* and *m4/guestfs−*.m4*. Most of the configure script is split over many m4 macro files by topic, for example *m4/guestfs−daemon.m4* deals with the dependencies of the daemon.

The job of the top level *Makefile.am* is mainly to list the subdirectories (`SUBDIRS`) in the order they should be compiled.

common−rules.mk is included in every *Makefile.am* (top level and subdirectories). *subdir−rules.mk* is included only in subdirectory *Makefile.am* files.

There are many make targets. Use this command to list them all:

```
make help
```

EXTENDING LIBGUESTFS

ADDING A NEW API

Because large amounts of boilerplate code in libguestfs are generated, this makes it easy to extend the libguestfs API.

To add a new API action there are two changes:

1. You need to add a description of the call (name, parameters, return type, tests, documentation) to *generator/actions_*.ml* and possibly *generator/proc_nr.ml*.

   There are two sorts of API action, depending on whether the call goes through to the daemon in the appliance, or is serviced entirely by the library (see ''ARCHITECTURE'' in **guestfs−internals**(1)). ''guestfs_sync'' in **guestfs**(3) is an example of the former, since the sync is done in the appliance. ''guestfs_set_trace'' in **guestfs**(3) is an example of the latter, since a trace flag is maintained in the handle and all tracing is done on the library side.

   Most new actions are of the first type, and get added to the `daemon_functions` list. Each function has a unique procedure number used in the RPC protocol which is assigned to that action when we publish libguestfs and cannot be reused. Take the latest procedure number and increment it.

   For library-only actions of the second type, add to the `non_daemon_functions` list. Since these functions are serviced by the library and do not travel over the RPC mechanism to the daemon, these functions do not need a procedure number, and so the procedure number is set to `−1`.

2. Implement the action (in C):

   For daemon actions, implement the function `do_<name>` in the `daemon/` directory.

   For library actions, implement the function `guestfs_impl_<name>` in the `lib/` directory.

   In either case, use another function as an example of what to do.

3. As an alternative to step 2: Since libguestfs 1.38, daemon actions can be implemented in OCaml. You have to set the `impl = OCaml ...` flag in the generator. Take a look at *daemon/file.ml* for an example.

After making these changes, use `make` to compile.

Note that you don't need to implement the RPC, language bindings, manual pages or anything else. It's all automatically generated from the OCaml description.

Adding tests for an API

You can supply zero or as many tests as you want per API call. The tests can either be added as part of the API description (*generator/actions_*.ml*), or in some rarer cases you may want to drop a script into `tests/*/`. Note that adding a script to`tests/*/` is slo wer, so if possible use the first method.

The following describes the test environment used when you add an API test in *actions_*.ml*.

The test environment has 4 block devices:

/dev/sda 2 GB
  General block device for testing.

/dev/sdb 2 GB
  /dev/sdb1 is an ext2 filesystem used for testing filesystem write operations.

/dev/sdc 10 MB
  Used in a few tests where two block devices are needed.

/dev/sdd
  ISO with fixed content (see *images/test.iso*).

To be able to run the tests in a reasonable amount of time, the libguestfs appliance and block devices are reused between tests. So don't try testing ''guestfs_kill_subprocess'' in **guestfs**(3) :−x

Each test starts with an initial scenario, selected using one of the `Init*` expressions, described in *generator/types.ml*. These initialize the disks mentioned above in a particular way as documented in *types.ml*. You should not assume anything about the previous contents of other disks that are not initialized.

You can add a prerequisite clause to any individual test. This is a run-time check, which, if it fails, causes the test to be skipped. Useful if testing a command which might not work on all variations of libguestfs builds. A test that has prerequisite of`Always` means to run unconditionally .

In addition, packagers can skip individual tests by setting environment variables before running `make check`.

```
 SKIP_TEST_<CMD>_<NUM>=1
```

eg: `SKIP_TEST_COMMAND_3=1` skips test #3 of "guestfs_command" in **guestfs** (3).

or:

```
 SKIP_TEST_<CMD>=1
```

eg: `SKIP_TEST_ZEROFREE=1` skips all "guestfs_zerofree" in **guestfs** (3) tests.

Packagers can run only certain tests by setting for example:

```
 TEST_ONLY="vfs_type zerofree"
```

See *tests/c−api/tests.c* for more details of how these environment variables work.

Debugging new APIs

Test new actions work before submitting them.

You can use guestfish to try out new commands.

Debugging the daemon is a problem because it runs inside a minimal environment. However you can fprintf messages in the daemon to stderr, and they will show up if you use `guestfish −v`.

ADDING A NEW LANGUAGE BINDING

All language bindings must be generated by the generator (see the *generator* subdirectory).

There is no documentation for this yet. We suggest you look at an existing binding, eg. *generator/ocaml.ml* or *generator/perl.ml*.

Adding tests for language bindings

Language bindings should come with tests. Previously testing of language bindings was rather ad-hoc, but we have been trying to formalize the set of tests that every language binding should use.

Currently only the OCaml and Perl bindings actually implement the full set of tests, and the OCaml bindings are canonical, so you should emulate what the OCaml tests do.

This is the numbering scheme used by the tests:

```
 − 000+ basic tests:

   010  load the library
   020  create
   030  create−flags
   040  create multiple handles
   050  test setting and getting config properties
   060  explicit close
   065  implicit close (in GC'd languages)
   070  optargs
   080  version
   090  retvalues

 − 100  launch, create partitions and LVs and filesystems

 − 400+ events:
   410  close event
   420  log messages
   430  progress messages

 − 800+ regression tests (specific to the language)

 − 900+ any other custom tests for the language
```

To save time when running the tests, only 100, 430, 800+, 900+ should launch the handle.

FORMATTING CODE

Our C source code generally adheres to some basic code-formatting conventions. The existing code base is not totally consistent on this front, but we do prefer that contributed code be formatted similarly. In short, use spaces-not-TABs for indentation, use 2 spaces for each indentation level, and other than that, follow the K&R style.

If you use Emacs, add the following to one of your start-up files (e.g., ˜/.emacs), to help ensure that you get indentation right:

```
;;; In libguestfs, indent with spaces everywhere (not TABs).
;;; Exceptions: Makefile and ChangeLog modes.
(add−hook 'find−file−hook
    '(lambda () (if (and buffer−file−name
                         (string−match "/libguestfs\\>"
                             (buffer−file−name))
                         (not (string−equal mode−name "Change Log"))
                         (not (string−equal mode−name "Makefile")))
                    (setq indent−tabs−mode nil))))

;;; When editing C sources in libguestfs, use this style.
(defun libguestfs−c−mode ()
  "C mode with adjusted defaults for use with libguestfs."
  (interactive)
  (c−set−style "K&R")
  (setq c−indent−level 2)
  (setq c−basic−offset 2))
(add−hook 'c−mode−hook
          '(lambda () (if (string−match "/libguestfs\\>"
                                (buffer−file−name))
                          (libguestfs−c−mode))))
```

TESTING YOUR CHANGES

Turn warnings into errors when developing to make warnings hard to ignore:

```
./configure −−enable−werror
```

Useful targets are:

`make check`

Runs the regular test suite.

This is implemented using the regular automake `TESTS` target. See the automake documentation for details.

`make check−valgrind`

Runs a subset of the test suite under valgrind.

See "VALGRIND" below.

`make check-valgrind-local-guests`
: Runs a subset of the test suite under valgrind using locally installed libvirt guests (read-only).

`make check-direct`
: Runs all tests using default appliance back-end. This only has any effect if a non-default backend was selected using `./configure --with-default-backend=...`

`make check-valgrind-direct`
: Run a subset of the test suite under valgrind using the default appliance back-end.

`make check-uml`
: Runs all tests using the User-Mode Linux backend.

  As there is no standard location for the User-Mode Linux kernel, you *have* to set `LIBGUESTFS_HV` to point to the kernel image, eg:

```
 make check-uml LIBGUESTFS_HV=~/d/linux-um/vmlinux
```

`make check-valgrind-uml`
: Runs all tests using the User-Mode Linux backend, under valgrind.

  As above, you have to set `LIBGUESTFS_HV` to point to the kernel.

`make check-with-upstream-qemu`
: Runs all tests using a local qemu binary. It looks for the qemu binary in QEMUDIR (defaults to *$HOME/d/qemu*), but you can set this to another directory on the command line, eg:

```
 make check-with-upstream-qemu QEMUDIR=/usr/src/qemu
```

`make check-with-upstream-libvirt`
: Runs all tests using a local libvirt. This only has any effect if the libvirt backend was selected using `./configure --with-default-backend=libvirt`

  It looks for libvirt in LIBVIRTDIR (defaults to *$HOME/d/libvirt*), but you can set this to another directory on the command line, eg:

```
 make check-with-upstream-libvirt LIBVIRTDIR=/usr/src/libvirt
```

`make check-slow`
: Runs some slow/long−running tests which are not run by default.

  To mark a test as slow/long−running:

  - Add it to the list of `TESTS` in the *Makefile.am*, just like a normal test.
  - Modify the test so it checks if the `SLOW=1` environment variable is set, and if *not* set it skips (ie. returns with exit code 77). If using `$TEST_FUNCTIONS`, you can call the function `slow_test` for this.
  - Add a variable `SLOW_TESTS` to the *Makefile.am* listing the slow tests.
  - Add a rule to the *Makefile.am*:

```
check-slow:
  $(MAKE) check TESTS="$(SLOW_TESTS)" SLOW=1
```

`sudo make check-root`
: Runs some tests which require root privileges. These are supposed to be safe, but take care. You have to run this as root (eg. using **sudo** (8) explicitly).

  To mark a test as requiring root:

  - Add it to the list of `TESTS` in the *Makefile.am*, just like a normal test.
  - Modify the test so it checks if euid == 0, and if *not* set it skips (ie. returns with exit code 77). If using `$TEST_FUNCTIONS`, you can call the function `root_test` for this.

- Add a variable `ROOT_TESTS` to the *Makefile.am* listing the root tests.
- Add a rule to the *Makefile.am*:

```
check-root:
  $(MAKE) check TESTS="$(ROOT_TESTS)"
```

`make check-all`
Equivalent to running all `make check*` rules except `check-root`.

`make check-release`
Runs a subset of `make check*` rules that are required to pass before a tarball can be released. Currently this is:

- check
- check-valgrind
- check-direct
- check-valgrind-direct
- check-slow

`make installcheck`
Run `make check` on the installed copy of libguestfs.

The version of installed libguestfs being tested, and the version of the libguestfs source tree must be the same.

Do:

```
./configure
make clean ||:
make
make installcheck
```

VALGRIND

When you do `make check-valgrind`, it searches for any *Makefile.am* in the tree that has a `check-valgrind:` target and runs it.

Writing the *Makefile.am* and tests correctly to use valgrind and working with automake parallel tests is subtle.

If your tests are run via a shell script wrapper, then in the wrapper use:

```
$VG virt-foo
```

and in the *Makefile.am* use:

```
check-valgrind:
    make VG="@VG@" check
```

However, if your binaries run directly from the `TESTS` rule, you have to modify the *Makefile.am* like this:

```
LOG_COMPILER = $(VG)

check-valgrind:
    make VG="@VG@" check
```

In either case, check that the right program is being tested by examining the *tmp/valgrind** log files carefully.

SUBMITTING PATCHES

Submit patches to the mailing list: http://www.redhat.com/mailman/listinfo/libguestfs and CC to rjones@redhat.com.

You do not need to subscribe to the mailing list if you don't want to. There may be a short delay while your message is moderated.

DAEMON CUSTOM PRINTF FORMATTERS

In the daemon code we have created custom printf formatters `%Q` and `%R`, which are used to do shell quoting.

`%Q` Simple shell quoted string. Any spaces or other shell characters are escaped for you.

`%R` Same as `%Q` except the string is treated as a path which is prefixed by the sysroot.

For example:

```
 asprintf (&cmd, "cat %R", path);
```

would produce `cat /sysroot/some\ path\ with\ spaces`

Note: Do *not* use these when you are passing parameters to the `command{,r,v,rv}()` functions. These parameters do NOT need to be quoted because they are not passed via the shell (instead, straight to exec). You probably want to use the `sysroot_path()` function however.

INTERNATIONALIZATION (I18N) SUPPORT

We support i18n (gettext anyhow) in the library.

However many messages come from the daemon, and we don't translate those at the moment. One reason is that the appliance generally has all locale files removed from it, because they take up a lot of space. So we'd have to readd some of those, as well as copying our PO files into the appliance.

Debugging messages are never translated, since they are intended for the programmers.

MISCELLANEOUS TOPICS

HOW OCAML PROGRAMS ARE COMPILED AND LINKED

Mostly this section is ''how we make automake & ocamlopt work together'' since OCaml programs themselves are easy to compile.

Automake has no native support for OCaml programs, ocamlc nor ocamlopt. What we do instead is to treat OCaml programs as C programs which happen to contain these ''other objects'' (`"DEPENDENCIES"` in automake-speak) that happen to be the OCaml objects. This works because OCaml programs usually have C files for native bindings etc.

So a typical program is described as just its C sources:

```
 virt_customize_SOURCES = ... crypt-c.c perl_edit-c.c
```

For programs that have no explicit C sources, we create an empty *dummy.c* file, and list that instead:

```
 virt_resize_SOURCES = dummy.c
```

The OCaml objects which contain most of the code are listed as automake dependencies (other dependencies may also be listed):

```
 virt_customize_DEPENDENCIES = ... customize_main.cmx
```

The only other special thing we need to do is to provide a custom link command. This is needed because automake won't assemble the ocamlopt command, the list of objects and the `-cclib` libraries in the correct order otherwise.

```
 virt_customize_LINK = \
     $(top_builddir)/ocaml-link.sh -cclib '-lutils' -- ...
```

The actual rules, which you can examine in *customize/Makefile.am*, are a little bit more complicated than this because they have to handle:

- Compiling for byte code or native code.
- The pattern rules needed to compile the OCaml sources to objects.

  These are now kept in *subdir-rules.mk* at the top level, which is included in every subdirectory *Makefile.am*.

- Adding OCaml sources files to `EXTRA_DIST`.

  Automake isn't aware of the complete list of sources for a binary, so it will not add them all automatically.

MAINTAINER TASKS

MAINTAINER MAKEFILE TARGETS

These `make` targets probably won't work and aren't useful unless you are a libguestfs maintainer.

make maintainer-commit

This commits everything in the working directory with the commit message `Version $(VERSION).`. You must update *configure.ac*, clean and rebuild first.

make maintainer-tag

This tags the current HEAD commit with the tag `v$(VERSION)` and one of the messages:

```
 Version $(VERSION) stable

 Version $(VERSION) development
```

(See "LIBGUESTFS VERSION NUMBERS" in **guestfs** (3) for the difference between a stable and development release.)

make maintainer-check-authors

Check that all authors (found in git commit messages) are included in the *generator/authors.ml* file.

make maintainer-check-extra-dist

This rule must be run after `make dist` (so there is a tarball in the working directory). It compares the contents of the tarball with the contents of git to ensure that no files have been missed from *Makefile.am* `EXTRA_DIST` rules.

make maintainer-upload-website

This is used by the software used to automate libguestfs releases to copy the libguestfs website to another git repository before it is uploaded to the web server.

MAKING A STABLE RELEASE

When we make a stable release, there are several steps documented here. See "LIBGUESTFS VERSION NUMBERS" in **guestfs** (3) for general information about the stable branch policy.

- Check `make && make check` works on at least:

  Fedora (x86−64)
  Debian (x86−64)
  Ubuntu (x86−64)
  Fedora (aarch64)
  Fedora (ppc64)
  Fedora (ppc64le)
- Check `./configure --without-libvirt` works.
- Finalize *guestfs−release−notes.pod*
- Create new stable and development directories under http://libguestfs.org/download.
- Edit *website/index.html.in*.
- Set the version (in *configure.ac*) to the new *stable* version, ie. 1.XX.0, and commit it:

```
./localconfigure
make distclean -k
./localconfigure
make && make dist
make maintainer-commit
make maintainer-tag
```

- Create the stable branch in git:

```
git branch stable-1.XX
git push origin stable-1.XX
```

- Do a full release of the stable branch.
- Set the version to the next development version and commit that. Optionally do a full release of the development branch.

INTERNAL DOCUMENTATION

This section documents internal functions inside libguestfs and various utilities. It is intended for libguestfs developers only.

This section is autogenerated from `/**` comments in source files, which are marked up in POD format.

These functions are not publicly exported, and may change or be removed at any time.

Subdirectory *lib*

File lib/actions−support.c

Helper functions for the actions code in *lib/actions−*.c*.

File lib/appliance−cpu.c

The appliance choice of CPU model.

Function `lib/appliance-cpu.c:guestfs_int_get_cpu_model`

```
const char *
guestfs_int_get_cpu_model (int kvm)
```

Return the right CPU model to use as the qemu `−cpu` parameter or its equivalent in libvirt. This returns:

`‘‘host’’`
: The literal string `"host"` means use `−cpu host`.

`‘‘max’’`
: The literal string `"max"` means use `−cpu max` (the best possible). This requires awkward translation for libvirt.

some string
: Some string such as `"cortex−a57"` means use `−cpu cortex−a57`.

`NULL`
: `NULL` means no `−cpu` option at all. Note returning `NULL` does not indicate an error.

This is made unnecessarily hard and fragile because of two stupid choices in QEMU:

- The default for `qemu−system−aarch64 −M virt` is to emulate a `cortex−a15` (WTF?).
- We don’t know for sure if KVM will work, but `−cpu host` is broken with TCG, so we almost always pass a broken `−cpu` flag if KVM is semi-broken in any way.

File lib/appliance−kcmdline.c

The appliance kernel command line.

Definition `lib/appliance-kcmdline.c:VALID_TERM`

```
#define VALID_TERM
```

Check that the `$TERM` environment variable is reasonable before we pass it through to the appliance.

Function `lib/appliance−kcmdline.c:get_root_uuid_with_file`

```
static char *
get_root_uuid_with_file (guestfs_h *g, const char *appliance)
```

Given a disk image containing an extX filesystem, return the UUID.

Function `lib/appliance−kcmdline.c:run_qemu_img_dd`

```
static int
run_qemu_img_dd (guestfs_h *g, const char *in_file, char *out_file)
```

Read the first 256k bytes of the in_file with **qemu−img** (1) command and write them into the out_file. That may be useful to get UUID of the QCOW2 disk image with `get_root_uuid_with_file`.

The function returns zero if successful, otherwise −1.

Function `lib/appliance−kcmdline.c:get_root_uuid`

```
static char *
get_root_uuid (guestfs_h *g, const char *appliance)
```

Get the UUID from the appliance disk image.

Function `lib/appliance−kcmdline.c:guestfs_int_appliance_command_line`

```
char *
guestfs_int_appliance_command_line (guestfs_h *g,
                                     const char *appliance,
                                    int flags)
```

Construct the Linux command line passed to the appliance. This is used by the `direct` and `libvirt` backends, and is simply located in this file because it's a convenient place for this common code.

The `appliance` parameter is the filename of the appliance (could be NULL) from which we obtain the root UUID.

The `flags` parameter can contain the following flags logically or'd together (or 0):

`APPLIANCE_COMMAND_LINE_IS_TCG`
: If we are launching a qemu TCG guest (ie. KVM is known to be disabled or unavailable). If you don't know, don't pass this flag.

Note that this function returns a newly allocated buffer which must be freed by the caller.

File lib/appliance−uefi.c

Find the UEFI firmware needed to boot the appliance.

See also *lib/uefi.c* (autogenerated file) containing the firmware file locations.

Function `lib/appliance−uefi.c:guestfs_int_get_uefi`

```
int
guestfs_int_get_uefi (guestfs_h *g, char *const *firmwares,
                      const char **firmware, char **code, char **vars,
                      int *flags)
```

Return the location of firmware needed to boot the appliance. This is aarch64 only currently, since that's the only architecture where UEFI is mandatory (and that only for RHEL).

`firmwares` is an optional list of allowed values for the firmware autoselection of libvirt. It is `NULL` to indicate it is not supported. `*firmware` is set to one of the strings in `firmwares` in case one can be used.

`*code` is initialized with the path to the read-only UEFI code file. `*vars` is initialized with the path to a copy of the UEFI vars file (which is cleaned up automatically on exit).

In case a UEFI firmware is available, either `*firmware` is set to a non−`NULL` value, or `*code` and `*vars` are.

`*code` and `*vars` should be freed by the caller, and `*firmware` **must** not.

If the function returns `−1` then there was a real error which should cause appliance building to fail (no UEFI firmware is not an error).

See also *virt−v2v.git/v2v/utils.ml*:find_uefi_firmware

File lib/appliance.c

This file deals with building the libguestfs appliance.

Function `lib/appliance.c:guestfs_int_build_appliance`

```
int
guestfs_int_build_appliance (guestfs_h *g,
                             char **kernel_rtn,
                             char **initrd_rtn,
                             char **appliance_rtn)
```

Locate or build the appliance.

This function locates or builds the appliance as necessary, handling the supermin appliance, caching of supermin-built appliances, or using either a fixed or old-style appliance.

The return value is `0` = good, `−1` = error. Returned in`appliance.kernel` will be the name of the kernel to use, `appliance.initrd` the name of the initrd, `appliance.image` the name of the ext2 root filesystem. `appliance.image` can be `NULL`, meaning that we are using an old-style (non−e xt2) appliance. All three strings must be freed by the caller. However the referenced files themselves must *not* be deleted.

The process is as follows:

1. Look in `path` which contains a supermin appliance skeleton. If no element has this, skip straight to step 3.
2. Call `supermin −−build` to build the full appliance (if it needs to be rebuilt). If this is successful, return the full appliance.
3. Check `path`, looking for a fixed appliance. If one is found, return it.
4. Check `path`, looking for an old-style appliance. If one is found, return it.

The supermin appliance cache directory lives in *$TMPDIR/.guestfs−$UID/* and consists of up to four files:

```
$TMPDIR/.guestfs−$UID/lock             − the supermin lock file
$TMPDIR/.guestfs−$UID/appliance.d/kernel − the kernel
$TMPDIR/.guestfs−$UID/appliance.d/initrd − the supermin initrd
$TMPDIR/.guestfs−$UID/appliance.d/root   − the appliance
```

Multiple instances of libguestfs with the same UID may be racing to create an appliance. However (since supermin ≥ 5) supermin provides a *−−lock* flag and atomic update of the *appliance.d* subdirectory.

Function `lib/appliance.c:locate_or_build_appliance`

```
static int
locate_or_build_appliance (guestfs_h *g,
                           struct appliance_files *appliance,
                           const char *path)
```

Check `path`, looking for one of appliances: supermin appliance, fixed appliance or old-style appliance. If one of the fixed appliances is found, return it. If the supermin appliance skeleton is found, build and return appliance.

Return values:

```
  1 = appliance is found, returns C<appliance>,
  0 = appliance not found,
 -1 = error which aborts the launch process.
```

Function `lib/appliance.c:search_appliance`

```
static int
search_appliance (guestfs_h *g, struct appliance_files *appliance)
```

Search elements of `g->path`, returning the first `appliance` element which matches the predicate function `locate_or_build_appliance`.

Return values:

```
  1 = a path element matched, returns C<appliance>,
  0 = no path element matched,
 -1 = error which aborts the launch process.
```

Function `lib/appliance.c:build_supermin_appliance`

```
static int
build_supermin_appliance (guestfs_h *g,
                          const char *supermin_path,
                          struct appliance_files *appliance)
```

Build supermin appliance from `supermin_path` to *$TMPDIR/.guestfs−$UID*.

Returns: `0` = built or `-1` = error (aborts launch).

Function `lib/appliance.c:run_supermin_build`

```
static int
run_supermin_build (guestfs_h *g,
                    const char *lockfile,
                    const char *appliancedir,
                    const char *supermin_path)
```

Run `supermin --build` and tell it to generate the appliance.

Function `lib/appliance.c:dir_contains_file`

```
static int
dir_contains_file (guestfs_h *g, const char *dir, const char *file)
```

Returns true iff `file` is contained in `dir`.

Function `lib/appliance.c:dir_contains_files`

```
static int
dir_contains_files (guestfs_h *g, const char *dir, ...)
```

Returns true iff every listed file is contained in `dir`.

File lib/command.c

A wrapper for running external commands, loosely based on libvirt's `virCommand` interface.

In outline to use this interface you must:

1. Create a new command handle:

   ```
   struct command *cmd;
   cmd = guestfs_int_new_command (g);
   ```

2. *Either* add arguments:

```
guestfs_int_cmd_add_arg (cmd, "qemu-img");
guestfs_int_cmd_add_arg (cmd, "info");
guestfs_int_cmd_add_arg (cmd, filename);
```

(**NB:** You don't need to add a `NULL` argument at the end.)

3. *Or* construct a command using a mix of quoted and unquoted strings. (This is useful for **system** (3)/`popen("r")`−style shell commands, with the added safety of allowing args to be quoted properly).

```
guestfs_int_cmd_add_string_unquoted (cmd, "qemu-img info ");
guestfs_int_cmd_add_string_quoted (cmd, filename);
```

4. Set various flags, such as whether you want to capture errors in the regular libguestfs error log.
5. Run the command. This is what does the **fork** (2) call, optionally loops over the output, and then does a **waitpid** (3) and returns the exit status of the command.

```
r = guestfs_int_cmd_run (cmd);
if (r == −1)
  // error
// else test r using the WIF* functions
```

6. Close the handle:

```
guestfs_int_cmd_close (cmd);
```

(or use `CLEANUP_CMD_CLOSE`).

Function `lib/command.c:guestfs_int_new_command`

```
struct command *
guestfs_int_new_command (guestfs_h *g)
```

Create a new command handle.

Function `lib/command.c:guestfs_int_cmd_add_arg`

```
void
guestfs_int_cmd_add_arg (struct command *cmd, const char *arg)
```

Add single arg (for `execv`−style command execution).

Function `lib/command.c:guestfs_int_cmd_add_arg_format`

```
void
guestfs_int_cmd_add_arg_format (struct command *cmd, const char *fs, ...)
```

Add single arg (for `execv`−style command execution) using a **printf** (3)−style format string.

Function `lib/command.c:guestfs_int_cmd_add_string_unquoted`

```
void
guestfs_int_cmd_add_string_unquoted (struct command *cmd, const char *str)
```

Add a string (for **system** (3)−style command execution).

This variant adds the strings without quoting them, which is dangerous if the string contains untrusted content.

Function `lib/command.c:guestfs_int_cmd_add_string_quoted`

```
void
guestfs_int_cmd_add_string_quoted (struct command *cmd, const char *str)
```

Add a string (for **system** (3)−style command execution).

The string is enclosed in double quotes, with any special characters within the string which need escaping done. This is used to add a single argument to a **system** (3)−style command string.

Function `lib/command.c:guestfs_int_cmd_set_stdout_callback`

```
void
guestfs_int_cmd_set_stdout_callback (struct command *cmd,
                                     cmd_stdout_callback stdout_callback,
                                     void *stdout_data, unsigned flags)
```

Set a callback which will capture stdout.

If flags contains `CMD_STDOUT_FLAG_LINE_BUFFER` (the default), then the callback is called line by line on the output. If there is a trailing `\n` then it is automatically removed before the callback is called. The line buffer is `\0`−terminated.

If flags contains `CMD_STDOUT_FLAG_UNBUFFERED`, then buffers are passed to the callback as it is received from the command. Note in this case the buffer is *not* `\0`−terminated, so you need to may attention to the length field in the callback.

If flags contains `CMD_STDOUT_FLAG_WHOLE_BUFFER`, then the callback is called exactly once, with the entire buffer. Note in this case the buffer is *not* `\0`−terminated, so you need to may attention to the length field in the callback.

Function `lib/command.c:guestfs_int_cmd_set_stderr_to_stdout`

```
void
guestfs_int_cmd_set_stderr_to_stdout (struct command *cmd)
```

Equivalent to adding `2>&1` to the end of the command. This is incompatible with the `capture_errors` flag, because it doesn't make sense to combine them.

Function `lib/command.c:guestfs_int_cmd_clear_capture_errors`

```
void
guestfs_int_cmd_clear_capture_errors (struct command *cmd)
```

Clear the `capture_errors` flag. This means that any errors will go to stderr, instead of being captured in the event log, and that is usually undesirable.

Function `lib/command.c:guestfs_int_cmd_clear_close_files`

```
void
guestfs_int_cmd_clear_close_files (struct command *cmd)
```

Don't close file descriptors after the fork.

XXX Should allow single fds to be sent to child process.

Function `lib/command.c:guestfs_int_cmd_set_child_callback`

```
void
guestfs_int_cmd_set_child_callback (struct command *cmd,
                                    cmd_child_callback child_callback,
                                    void *data)
```

Set a function to be executed in the child, right before the execution. Can be used to setup the child, for example changing its current directory.

Function `lib/command.c:guestfs_int_cmd_set_child_rlimit`

```
void
guestfs_int_cmd_set_child_rlimit (struct command *cmd, int resource, long limit)
```

Set up child rlimits, in case the process we are running could consume lots of space or time.

Function `lib/command.c:finish_command`

```
static void
finish_command (struct command *cmd)
```

Finish off the command by either `NULL`−terminating the argv array or adding a terminating `\0` to the string, or die with an internal error if no command has been added.

Function `lib/command.c:loop`

```
static int
loop (struct command *cmd)
```

The loop which reads errors and output and directs it either to the log or to the stdout callback as appropriate.

Function `lib/command.c:guestfs_int_cmd_run`

```
int
guestfs_int_cmd_run (struct command *cmd)
```

Fork, run the command, loop over the output, and waitpid.

Returns the exit status. Test it using `WIF*` macros.

On error: Calls `error` and returns `−1`.

Function `lib/command.c:guestfs_int_cmd_pipe_run`

```
int
guestfs_int_cmd_pipe_run (struct command *cmd, const char *mode)
```

Fork and run the command, but don't wait. Roughly equivalent to `popen(...,"r"|"w")`.

Returns the file descriptor of the pipe, connected to stdout (`"r"`) or stdin (`"w"`) of the child process.

After reading/writing to this pipe, call `guestfs_int_cmd_pipe_wait` to wait for the status of the child.

Errors from the subcommand cannot be captured to the error log using this interface. Instead the caller should call `guestfs_int_cmd_get_pipe_errors` (after `guestfs_int_cmd_pipe_wait` returns an error).

Function `lib/command.c:guestfs_int_cmd_pipe_wait`

```
int
guestfs_int_cmd_pipe_wait (struct command *cmd)
```

Wait for a subprocess created by `guestfs_int_cmd_pipe_run` to finish. On error (eg. failed syscall) this returns `−1` and sets the error. If the subcommand fails, then use `WIF*` macros to check this, and call `guestfs_int_cmd_get_pipe_errors` to read the error messages printed by the child.

Function `lib/command.c:guestfs_int_cmd_get_pipe_errors`

```
char *
guestfs_int_cmd_get_pipe_errors (struct command *cmd)
```

Read the error messages printed by the child. The caller must free the returned buffer after use.

Function `lib/command.c:guestfs_int_cmd_close`

```
void
guestfs_int_cmd_close (struct command *cmd)
```

Close the `cmd` object and free all resources.

Function `lib/command.c:process_line_buffer`

```
static void
process_line_buffer (struct command *cmd, int closed)
```

Deal with buffering stdout for the callback.

File lib/conn−socket.c

This file handles connections to the child process where this is done over regular POSIX sockets.

Function `lib/conn−socket.c:handle_log_message`

```
static int
handle_log_message (guestfs_h *g,
                    struct connection_socket *conn)
```

This is called if `conn->console_sock` becomes ready to read while we are doing one of the connection operations above. It reads and deals with the log message.

Returns:

`1`	log message(s) were handled successfully
`0`	connection to appliance closed
`-1`	error

Function `lib/conn-socket.c:guestfs_int_new_conn_socket_listening`

```
struct connection *
guestfs_int_new_conn_socket_listening (guestfs_h *g,
                                       int daemon_accept_sock,
                                       int console_sock)
```

Create a new socket connection, listening.

Note that it's OK for `console_sock` to be passed as `-1`, meaning there's no console available for this appliance.

After calling this, `daemon_accept_sock` is owned by the connection, and will be closed properly either in `accept_connection` or `free_connection`.

Function `lib/conn-socket.c:guestfs_int_new_conn_socket_connected`

```
struct connection *
guestfs_int_new_conn_socket_connected (guestfs_h *g,
                                       int daemon_sock,
                                       int console_sock)
```

Create a new socket connection, connected.

As above, but the caller passes us a connected `daemon_sock` and promises not to call `accept_connection`.

File lib/create.c

APIs for creating empty disks.

Mostly this consists of wrappers around the **qemu-img** (1) program.

Definition `lib/create.c:VALID_FORMAT`

```
#define VALID_FORMAT
```

Check for valid backing format. Allow any `^[[:alnum]]+$` (in C locale), but limit the length to something reasonable.

File lib/drives.c

Drives added are stored in an array in the handle. Code here manages that array and the individual `struct drive`data.

Function `lib/drives.c:create_overlay`

```
static int
create_overlay (guestfs_h *g, struct drive *drv)
```

For readonly drives, create an overlay to protect the original drive content. Note we never need to clean up these overlays since they are created in the temporary directory and deleted when the handle is closed.

Function `lib/drives.c:create_drive_file`

```
static struct drive *
create_drive_file (guestfs_h *g,
                   const struct drive_create_data *data)
```

Create and free the `struct drive`.

Function `lib/drives.c:create_drive_dev_null`

```
static struct drive *
create_drive_dev_null (guestfs_h *g,
                       struct drive_create_data *data)
```

Create the special */dev/null* drive.

Traditionally you have been able to use */dev/null* as a filename, as many times as you like. Ancient KVM (RHEL 5) cannot handle adding */dev/null* readonly. qemu 1.2 + virtio-scsi segfaults when you use any zero-sized file including */dev/null*.

Because of these problems, we replace */dev/null* with a non-zero sized temporary file. This shouldn't make any difference since users are not supposed to try and access a null drive.

Function `lib/drives.c:drive_to_string`

```
static char *
drive_to_string (guestfs_h *g, const struct drive *drv)
```

Convert a `struct drive` to a string for debugging. The caller must free this string.

Function `lib/drives.c:add_drive_to_handle_at`

```
static void
add_drive_to_handle_at (guestfs_h *g, struct drive *d, size_t drv_index)
```

Add `struct drive` to the `g->drives` vector at the given index `drv_index`. If the array isn't large enough it is reallocated. The index must not contain a drive already.

Function `lib/drives.c:add_drive_to_handle`

```
static void
add_drive_to_handle (guestfs_h *g, struct drive *d)
```

Add struct drive to the end of the `g->drives` vector in the handle.

Function `lib/drives.c:guestfs_int_add_dummy_appliance_drive`

```
void
guestfs_int_add_dummy_appliance_drive (guestfs_h *g)
```

Called during launch to add a dummy slot to `g->drives`.

Function `lib/drives.c:guestfs_int_free_drives`

```
void
guestfs_int_free_drives (guestfs_h *g)
```

Free up all the drives in the handle.

Definition `lib/drives.c:VALID_FORMAT_IFACE`

```
#define VALID_FORMAT_IFACE
```

Check string parameter matches regular expression `^[-_[:alnum:]]+$` (in C locale).

Definition `lib/drives.c:VALID_DISK_LABEL`

```
#define VALID_DISK_LABEL
```

Check the disk label is reasonable. It can't contain certain characters, eg. `'/'`, `','`. However be stricter here and ensure it's just alphabetic and ≤ 20 characters in length.

Definition `lib/drives.c:VALID_HOSTNAME`

```
#define VALID_HOSTNAME
```

Check the server hostname is reasonable.

Function `lib/drives.c:valid_port`

```
static int
valid_port (int port)
```

Check the port number is reasonable.

Function `lib/drives.c:valid_blocksize`

```
static int
valid_blocksize (int blocksize)
```

Check the block size is reasonable. It can't be other then 512 or 4096.

Function `lib/drives.c:guestfs_impl_remove_drive`

```
int
guestfs_impl_remove_drive (guestfs_h *g, const char *label)
```

This function implements "guestfs_remove_drive" in **guestfs** (3).

Depending on whether we are hotplugging or not, this function does slightly different things: If not hotplugging, then the drive just disappears as if it had never been added. The later drives "move up" to fill the space. When hotplugging we have to do some complex stuff, and we usually end up leaving an empty (`NULL`) slot in the `g->drives` vector.

Function `lib/drives.c:guestfs_int_checkpoint_drives`

```
size_t
guestfs_int_checkpoint_drives (guestfs_h *g)
```

Checkpoint and roll back drives, so that groups of drives can be added atomically. Only used by "guestfs_add_domain" in **guestfs** (3).

Function `lib/drives.c:guestfs_impl_debug_drives`

```
char **
guestfs_impl_debug_drives (guestfs_h *g)
```

Internal function to return the list of drives.

File lib/errors.c

This file handles errors, and also debug, trace and warning messages.

Errors in libguestfs API calls are handled by setting an error message and optional errno in the handle. The caller has the choice of testing API calls to find out if they failed and then querying the last error from the handle, and/or getting a callback.

From the point of view of the library source, generally you should use the `error` or `perrorf` macros along error paths, eg:

```
if (something_bad) {
  error (g, "something bad happened");
  return -1;
}
```

Make sure to call the `error` or `perrorf` macro exactly once along each error path, since the handle can only store a single error and the previous error will be overwritten.

Function `lib/errors.c:guestfs_int_warning`

```
void
guestfs_int_warning (guestfs_h *g, const char *fs, ...)
```

Print a warning.

Code should *not* call this function directly. Use the`warning(g,fs,...)` macro.

Warnings are printed unconditionally. We try to make these rare: Generally speaking, a warning should either be an error, or if it's not important for end users then it should be a debug message.

Function `lib/errors.c:guestfs_int_debug`

```
 void
 guestfs_int_debug (guestfs_h *g, const char *fs, ...)
```

Print a debug message.

Code should *not* call this function directly. To add debug messages in the library, use the `debug(g,fs,...)` macro. The macro checks if `g->verbose` is false and avoids the function call, meaning the macro is more efficient.

Function `lib/errors.c:guestfs_int_trace`

```
 void
 guestfs_int_trace (guestfs_h *g, const char *fs, ...)
```

Print a trace message.

Do not call this function. All calls are generated automatically.

Function `lib/errors.c:guestfs_int_error_errno`

```
 void
 guestfs_int_error_errno (guestfs_h *g, int errnum, const char *fs, ...)
```

Set the last error and errno in the handle, and optionally raise the error callback if one is defined.

If you don't need to set errno, use the `error(g,fs,...)` macro instead of calling this directly. If you need to set errno then there is no macro wrapper, so calling this function directly is fine.

Function `lib/errors.c:guestfs_int_perrorf`

```
 void
 guestfs_int_perrorf (guestfs_h *g, const char *fs, ...)
```

Similar to **perror**(3), but it sets the last error in the handle, raises the error callback if one is defined, and supports format strings.

You should probably use the `perrorf(g,fs,...)` macro instead of calling this directly.

Function `lib/errors.c:guestfs_int_launch_failed_error`

```
 void
 guestfs_int_launch_failed_error (guestfs_h *g)
```

Raise a launch failed error in a standard format.

Since this is the most common error seen by people who have installation problems, buggy qemu, etc, and since no one reads the FAQ, describe in this error message what resources are available to debug launch problems.

Function `lib/errors.c:guestfs_int_unexpected_close_error`

```
 void
 guestfs_int_unexpected_close_error (guestfs_h *g)
```

Raise an error if the appliance unexpectedly crashes after launch.

Function `lib/errors.c:guestfs_int_launch_timeout`

```
 void
 guestfs_int_launch_timeout (guestfs_h *g)
```

Raise an error if the appliance hangs during launch.

Function `lib/errors.c:guestfs_int_external_command_failed`

```
void
guestfs_int_external_command_failed (guestfs_h *g, int status,
                                     const char *cmd_name, const char *extra)
```

Raise an error if an external command fails.

`status` is the status code of the command (eg. returned from **waitpid**(2) or **system**(3)). This function turns the status code into an explanatory string.

File lib/events.c

Function `lib/events.c:replace_old_style_event_callback`

```
static void
replace_old_style_event_callback (guestfs_h *g,
                                  guestfs_event_callback cb,
                                  uint64_t event_bitmask,
                                  void *opaque,
                                  void *opaque2)
```

Emulate old-style callback API.

There were no event handles, so multiple callbacks per event were not supported. Calling the same `guestfs_set_*_callback` function would replace the existing event. Calling it with`cb == NULL` meant that the caller wanted to remove the callback.

File lib/guestfs−internal−all.h

This header contains definitions which are shared by all parts of libguestfs, ie. the daemon, the library, language bindings and virt tools (ie. *all* C code).

If you need a definition used by only the library, put it in *lib/guestfs−internal.h* instead.

If a definition is used by only a single tool, it should not be in any shared header file at all.

File lib/guestfs−internal.h

This header file is included in the libguestfs library (*lib/*) only.

See also *lib/guestfs−internal−all.h*.

Structure `lib/guestfs−internal.h:event`

```
struct event {
  uint64_t event_bitmask;
  guestfs_event_callback cb;
  void *opaque;

  /* opaque2 is not exposed through the API, but is used internally to
   * emulate the old−style callback API.
   */
  void *opaque2;
};
```

This struct is used to maintain a list of events registered against the handle. See `g−>events` in the handle.

Structure `lib/guestfs−internal.h:drive`

```
struct drive {
  /* Original source of the drive, eg. file:..., http:... */
  struct drive_source src;

  /* If the drive is readonly, then an overlay [a local file] is
   * created before launch to protect the original drive content, and
   * the filename is stored here.  Backends should open this file if
   * it is non−NULL, else consult the original source above.
```

```
     *
     * Note that the overlay is in a backend−specific format, probably
     * different from the source format.  eg. qcow2, UML COW.
     */
    char *overlay;

    /* Various per−drive flags. */
    bool readonly;
    char *iface;
    char *name;
    char *disk_label;
    char *cachemode;
    enum discard discard;
    bool copyonread;
    int blocksize;
  };
```

There is one `struct drive` per drive, including hot-plugged drives.

Structure `lib/guestfs−internal.h:backend_ops`

```
  struct backend_ops {
    /* Size (in bytes) of the per−handle data structure needed by this
     * backend.  The data pointer is allocated and freed by libguestfs
     * and passed to the functions in the 'void *data' parameter.
     * Inside the data structure is opaque to libguestfs.  Any strings
     * etc pointed to by it must be freed by the backend during
     * shutdown.
     */
    size_t data_size;

    /* Create a COW overlay on top of a drive.  This must be a local
     * file, created in the temporary directory.  This is called when
     * the drive is added to the handle.
     */
    char *(*create_cow_overlay) (guestfs_h *g, void *data, struct drive *drv);

    /* Launch and shut down. */
    int (*launch) (guestfs_h *g, void *data, const char *arg);
    int (*shutdown) (guestfs_h *g, void *data, int check_for_errors);

    /* Miscellaneous. */
    int (*get_pid) (guestfs_h *g, void *data);
    int (*max_disks) (guestfs_h *g, void *data);

    /* Hotplugging drives. */
    int (*hot_add_drive) (guestfs_h *g, void *data, struct drive *drv, size_t drv_
    int (*hot_remove_drive) (guestfs_h *g, void *data, struct drive *drv, size_t d
  };
```

Backend operations.

Each backend (eg. libvirt, direct) defines some functions which get run at various places in the handle lifecycle (eg. at launch, shutdown). The backend defines this struct pointing to those functions.

Structure `lib/guestfs−internal.h:connection`

```
struct connection {
  const struct connection_ops *ops;

  /* In the real struct, private data used by each connection module
   * follows here.
   */
};
```

Connection module.

A `connection` represents the appliance console connection plus the daemon connection. It hides the underlying representation (POSIX sockets, `virStreamPtr`).

Structure `lib/guestfs−internal.h:cached_feature`

```
struct cached_feature {
  char *group;
  int result;
};
```

Cache of queried features.

Used to cache the appliance features (see *lib/available.c*).

Structure `lib/guestfs−internal.h:guestfs_h`

```
struct guestfs_h {
  struct guestfs_h *next;      /* Linked list of open handles. */
  enum state state;             /* See the state machine diagram in guestfs(3)*/

  /* Lock acquired when entering any public guestfs_* function to
   * protect the handle.
   */
  pthread_mutex_t lock;

  /**** Configuration of the handle. ****/
  bool verbose;                 /* Debugging. */
  bool trace;                   /* Trace calls. */
  bool autosync;                /* Autosync. */
  bool direct_mode;             /* Direct mode. */
  bool recovery_proc;           /* Create a recovery process. */
  bool enable_network;          /* Enable the network. */
  bool selinux;                 /* selinux enabled? */
  bool pgroup;                  /* Create process group for children? */
  bool close_on_exit;           /* Is this handle on the atexit list? */

  int smp;                      /* If > 1, −smp flag passed to hv. */
  int memsize;                 /* Size of RAM (megabytes). */

  char *path;                  /* Path to the appliance. */
  char *hv;                    /* Hypervisor (HV) binary. */
  char *append;                        /* Append to kernel command line. */

  struct hv_param *hv_params;   /* Extra hv parameters. */

  char *program;                /* Program name. */
  char *identifier;             /* Handle identifier. */

  /* Array of drives added by add−drive* APIs.
```

```
     *
     * Before launch this list can be empty or contain some drives.
     *
     * During launch, a dummy slot may be added which represents the
     * slot taken up by the appliance drive.
     *
     * When hotplugging is supported by the backend, drives can be
     * added to the end of this list after launch.  Also hot-removing a
     * drive causes a NULL slot to appear in the list.
     *
     * During shutdown, this list is deleted, so that each launch gets a
     * fresh set of drives (however callers: don't do this, create a new
     * handle each time).
     *
     * Always use ITER_DRIVES macro to iterate over this list!
     */
    struct drive **drives;
    size_t nr_drives;

  #define ITER_DRIVES(g,i,drv)              \
    for (i = 0; i < (g)->nr_drives; ++i)    \
      if (((drv) = (g)->drives[i]) != NULL)

    /* Backend.  NB: Use guestfs_int_set_backend to change the backend. */
    char *backend;                /* The full string, always non-NULL. */
    char *backend_arg;            /* Pointer to the argument part. */
    const struct backend_ops *backend_ops;
    void *backend_data;           /* Per-handle data. */
    char **backend_settings;      /* Backend settings (can be NULL). */

    /**** Runtime information. ****/
    /* Temporary and cache directories. */
    /* The actual temporary directory - this is not created with the
     * handle, you have to call guestfs_int_lazy_make_tmpdir.
     */
    char *tmpdir;
    char *sockdir;
    /* Environment variables that affect tmpdir/cachedir/sockdir locations. */
    char *env_tmpdir;             /* $TMPDIR (NULL if not set) */
    char *env_runtimedir;         /* $XDG_RUNTIME_DIR (NULL if not set)*/
    char *int_tmpdir;   /* $LIBGUESTFS_TMPDIR or guestfs_set_tmpdir or NULL */
    char *int_cachedir; /* $LIBGUESTFS_CACHEDIR or guestfs_set_cachedir or NULL */

    /* Error handler, plus stack of old error handlers. */
    pthread_key_t error_data;

    /* Linked list of error_data structures allocated for this handle,
     * plus a mutex to protect the linked list.
     */
    pthread_mutex_t error_data_list_lock;
    struct error_data *error_data_list;

    /* Out of memory error handler. */
    guestfs_abort_cb           abort_cb;
    /* Events. */
    struct event *events;
    size_t nr_events;

    /* Private data area. */
    struct hash_table *pda;
    struct pda_entry *pda_next;

    /* User cancelled transfer.  Not signal-atomic, but it doesn't
     * matter for this case because we only care if it is != 0.
     */
    int user_cancel;

    struct timeval launch_t;      /* The time that we called guestfs_launch. */

    /* Used by bindtests. */
    FILE *test_fp;

    /* Used to generate unique numbers, eg for temp files.  To use this,
     * '++g->unique'.  Note these are only unique per-handle, not
     * globally unique.
     */
    int unique;

    /*** Protocol. ***/
    struct connection *conn;              /* Connection to appliance. */
    int msg_next_serial;

  #if HAVE_FUSE
    /**** Used by the mount-local APIs. ****/
    char *localmountpoint;
    struct fuse *fuse;                    /* FUSE handle. */
    int ml_dir_cache_timeout;             /* Directory cache timeout. */
    Hash_table *lsc_ht, *xac_ht, *rlc_ht; /* Directory cache. */
    int ml_read_only;                     /* If mounted read-only. */
    int ml_debug_calls;        /* Extra debug info on each FUSE call. */
  #endif

  #ifdef HAVE_LIBVIRT_BACKEND
    /* Used by lib/libvirt-auth.c. */
  #define NR_CREDENTIAL_TYPES 9
    unsigned int nr_supported_credentials;
    int supported_credentials[NR_CREDENTIAL_TYPES];
    const char *saved_libvirt_uri; /* Doesn't need to be freed. */
    bool wrapper_warning_done;
    unsigned int nr_requested_credentials;
    virConnectCredentialPtr requested_credentials;
  #endif

    /* Cached features. */
    struct cached_feature *features;
    size_t nr_features;
```

```
   /* Used by lib/info.c.  −1 = not tested or error; else 0 or 1. */
   int qemu_img_supports_U_option;
 };
```

The libguestfs handle.

Structure `lib/guestfs−internal.h:version`

```
 struct version {
   int v_major;
   int v_minor;
   int v_micro;
 };
```

Used for storing major.minor.micro version numbers. See *lib/version.c* for more information.

File lib/guid.c

Function `lib/guid.c:guestfs_int_validate_guid`

```
 int
 guestfs_int_validate_guid (const char *str)
```

Check whether a string supposed to contain a GUID actually contains it. It can recognize strings either as `{21EC2020−3AEA−1069−A2DD−08002B30309D}` or `21EC2020−3AEA−1069−A2DD−08002B30309D`.

File lib/handle.c

This file deals with the `guestfs_h` handle, creating it, closing it, and initializing/setting/getting fields.

Function `lib/handle.c:init_libguestfs`

```
 static void
 init_libguestfs (void)
```

No initialization is required by libguestfs, but libvirt and libxml2 require initialization if they might be called from multiple threads. Hence this constructor function which is called when libguestfs is first loaded.

Function `lib/handle.c:shutdown_backend`

```
 static int
 shutdown_backend (guestfs_h *g, int check_for_errors)
```

This function is the common path for shutting down the backend qemu process.

`guestfs_shutdown` calls `shutdown_backend` with `check_for_errors=1`. `guestfs_close` calls `shutdown_backend` with `check_for_errors=0`.

`check_for_errors` is a hint to the backend about whether we care about errors or not. In the libvirt case it can be used to optimize the shutdown for speed when we don't care.

Function `lib/handle.c:close_handles`

```
 static void
 close_handles (void)
```

Close all open handles (called from **atexit** (3)).

Function `lib/handle.c:guestfs_int_get_backend_setting_bool`

```
 int
 guestfs_int_get_backend_setting_bool (guestfs_h *g, const char *name)
```

This is a convenience function, but we might consider exporting it as an API in future.

File lib/info.c

Function `lib/info.c:qemu_img_supports_U_option`

```
static int
qemu_img_supports_U_option (guestfs_h *g)
```

Test if the qemu-img info command supports the `-U` option to disable locking. The result is memoized in the handle.

Note this option was added in qemu 2.11. We can remove this test when we can assume everyone is using qemu >= 2.11.

File lib/inspect−icon.c

Function `lib/inspect-icon.c:guestfs_int_download_to_tmp`

```
char *
guestfs_int_download_to_tmp (guestfs_h *g, const char *filename,
                             const char *extension,
                             uint64_t max_size)
```

Download a guest file to a local temporary file.

The name of the temporary (downloaded) file is returned. The caller must free the pointer, but does *not* need to delete the temporary file. It will be deleted when the handle is closed.

The name of the temporary file is randomly generated, but an extension can be specified using `extension` (or pass `NULL` for none).

Refuse to download the guest file if it is larger than `max_size`. On this and other errors,`NULL` is returned.

File lib/launch−direct.c

Implementation of the `direct` backend.

For more details see "BACKENDS" in **guestfs**(3).

Function `lib/launch-direct.c:add_drive_standard_params`

```
static int
add_drive_standard_params (guestfs_h *g, struct backend_direct_data *data,
                           struct qemuopts *qopts,
                           size_t i, struct drive *drv)
```

Add the standard elements of the `-drive` parameter.

Function `lib/launch-direct.c:add_device_blocksize_params`

```
static int
add_device_blocksize_params (guestfs_h *g, struct qemuopts *qopts,
                           struct drive *drv)
```

Add the physical_block_size and logical_block_size elements of the `-device` parameter.

File lib/launch−libvirt.c

Function `lib/launch-libvirt.c:get_source_format_or_autodetect`

```
static char *
get_source_format_or_autodetect (guestfs_h *g, struct drive *drv)
```

Return `drv->src.format`, but if it is `NULL`, autodetect the format.

libvirt has disabled the feature of detecting the disk format, unless the administrator sets `allow_disk_format_probing=1` in */etc/libvirt/qemu.conf*. There is no way to detect if this option is set, so we have to do format detection here using `qemu-img` and pass that to libvirt.

This can still be a security issue, so in most cases it is recommended the users pass the format to libguestfs which will faithfully pass that straight through to libvirt without doing autodetection.

Caller must free the returned string. On error this function sets the error in the handle and returns `NULL`.

Function `lib/launch-libvirt.c:make_qcow2_overlay`

```
static char *
make_qcow2_overlay (guestfs_h *g, const char *backing_drive,
                    const char *format)
```

Create a qcow2 format overlay, with the given `backing_drive` (file). The `format` parameter is the backing file format. The `format` parameter can be NULL, in this case the backing format will be determined automatically. This is used to create the appliance overlay, and also for read-only drives.

File lib/launch.c

This file implements "guestfs_launch" in **guestfs**(3).

Most of the work is done by the backends (see "BACKEND" in **guestfs**(3)), which are implemented in *lib/launch−direct.c*, *lib/launch−libvirt.c* etc, so this file mostly passes calls through to the current backend.

Function `lib/launch.c:guestfs_int_launch_send_progress`

```
void
guestfs_int_launch_send_progress (guestfs_h *g, int perdozen)
```

This function sends a launch progress message.

Launching the appliance generates approximate progress messages. Currently these are defined as follows:

```
   0 / 12: launch clock starts
   3 / 12: appliance created
   6 / 12: detected that guest kernel started
   9 / 12: detected that /init script is running
  12 / 12: launch completed successfully
```

Notes:

1. This is not a documented ABI and the behaviour may be changed or removed in future.
2. Messages are only sent if more than 5 seconds has elapsed since the launch clock started.
3. There is a hack in *lib/proto.c* to make this work.

Function `lib/launch.c:guestfs_int_timeval_diff`

```
int64_t
guestfs_int_timeval_diff (const struct timeval *x, const struct timeval *y)
```

Compute `y - x` and return the result in milliseconds.

Approximately the same as this code: http://www.mpp.mpg.de/~huber/util/timevaldiff.c

Function `lib/launch.c:guestfs_int_unblock_sigterm`

```
void
guestfs_int_unblock_sigterm (void)
```

Unblock the `SIGTERM` signal. Call this after **fork**(2) so that the parent process can send `SIGTERM` to the child process in case `SIGTERM` is blocked. See https://bugzilla.redhat.com/1460338.

Function `lib/launch.c:guestfs_impl_max_disks`

```
int
guestfs_impl_max_disks (guestfs_h *g)
```

Returns the maximum number of disks allowed to be added to the backend (backend dependent).

Function `lib/launch.c:guestfs_impl_wait_ready`

```
int
guestfs_impl_wait_ready (guestfs_h *g)
```

Implementation of "guestfs_wait_ready" in **guestfs**(3). You had to call this function after launch in versions ≤ 1.0.70, but it is now an (almost) no-op.

Function `lib/launch.c:guestfs_int_create_socketname`

```
int
guestfs_int_create_socketname (guestfs_h *g, const char *filename,
                               char (*sockpath)[UNIX_PATH_MAX])
```

Create the path for a socket with the selected filename in the tmpdir.

Function `lib/launch.c:guestfs_int_register_backend`

```
void
guestfs_int_register_backend (const char *name, const struct backend_ops *ops)
```

When the library is loaded, each backend calls this function to register itself in a global list.

Function `lib/launch.c:guestfs_int_set_backend`

```
int
guestfs_int_set_backend (guestfs_h *g, const char *method)
```

Implementation of "guestfs_set_backend" in **guestfs**(3).

- Callers must ensure this is only called in the config state.
- This shouldn't call `error` since it may be called early in handle initialization. It can return an error code however.

File lib/private−data.c

Implement a private data area where libguestfs C API users can attach arbitrary pieces of data to a `guestfs_h` handle.

For more information see "PRIVATE DATA AREA" in **guestfs**(3).

Language bindings do not generally expose this, largely because in non-C languages it is easy to associate data with handles in other ways (using hash tables or maps).

Structure `lib/private−data.c:pda_entry`

```
struct pda_entry {
  char *key;                    /* key */
  void *data;                   /* opaque user data pointer */
};
```

The private data area is internally stored as a gnulib hash table containing `pda_entry` structures.

Note the private data area is allocated lazily, since the vast majority of callers will never use it. This means `g−>pda` is likely to be `NULL`.

File lib/proto.c

This is the code used to send and receive RPC messages and (for certain types of message) to perform file transfers. This code is driven from the generated actions (*lib/actions−*.c*). There are five different cases to consider:

1. A non-daemon function (eg. "guestfs_set_verbose" in **guestfs**(3)). There is no RPC involved at all, it's all handled inside the library.
2. A simple RPC (eg. "guestfs_mount" in **guestfs**(3)). We write the request, then read the reply. The sequence of calls is:

   ```
   guestfs_int_send
   guestfs_int_recv
   ```

3. An RPC with `FileIn` parameters (eg. "guestfs_upload" in **guestfs**(3)). We write the request, then write the file(s), then read the reply. The sequence of calls is:

```
guestfs_int_send
guestfs_int_send_file  (possibly multiple times)
guestfs_int_recv
```

4. An RPC with `FileOut` parameters (eg. "guestfs_download" in **guestfs**(3)). We write the request, then read the reply, then read the file(s). The sequence of calls is:

   ```
   guestfs_int_send
   guestfs_int_recv
   guestfs_int_recv_file  (possibly multiple times)
   ```

5. Both `FileIn` and `FileOut` parameters. There are no calls like this in the current API, but they would be implemented as a combination of cases 3 and 4.

All read/write/etc operations are performed using the current connection module (`g−>conn`). During operations the connection module transparently handles log messages that appear on the console.

Function `lib/proto.c:child_cleanup`

```
static void
child_cleanup (guestfs_h *g)
```

This is called if we detect EOF, ie. qemu died.

Function `lib/proto.c:guestfs_int_progress_message_callback`

```
void
guestfs_int_progress_message_callback (guestfs_h *g,
                                       const guestfs_progress *message)
```

Convenient wrapper to generate a progress message callback.

Function `lib/proto.c:guestfs_int_log_message_callback`

```
void
guestfs_int_log_message_callback (guestfs_h *g, const char *buf, size_t len)
```

Connection modules call us back here when they get a log message.

Function `lib/proto.c:check_daemon_socket`

```
static ssize_t
check_daemon_socket (guestfs_h *g)
```

Before writing to the daemon socket, check the read side of the daemon socket for any of these conditions:

error
: return −1

daemon cancellation message
: return −2

progress message
: handle it here

end of input or appliance exited unexpectedly
: return 0

anything else
: return 1

Function `lib/proto.c:guestfs_int_send_file`

```
int
guestfs_int_send_file (guestfs_h *g, const char *filename)
```

Send a file.

Returns `0` on success, `−1` for error, `−2` if the daemon cancelled (we must read the error message).

Function `lib/proto.c:send_file_data`

```
static int
send_file_data (guestfs_h *g, const char *buf, size_t len)
```

Send a chunk of file data.

Function `lib/proto.c:send_file_cancellation`

```
static int
send_file_cancellation (guestfs_h *g)
```

Send a cancellation message.

Function `lib/proto.c:send_file_complete`

```
static int
send_file_complete (guestfs_h *g)
```

Send a file complete chunk.

Function `lib/proto.c:recv_from_daemon`

```
static int
recv_from_daemon (guestfs_h *g, uint32_t *size_rtn, void **buf_rtn)
```

This function reads a single message, file chunk, launch flag or cancellation flag from the daemon. If something was read, it returns `0`, otherwise `−1`.

Both `size_rtn` and `buf_rtn` must be passed by the caller as non-NULL.

`*size_rtn` returns the size of the returned message or it may be `GUESTFS_LAUNCH_FLAG` or `GUESTFS_CANCEL_FLAG`.

`*buf_rtn` is returned containing the message (if any) or will be set to `NULL`. `*buf_rtn` must be freed by the caller.

This checks for EOF (appliance died) and passes that up through the child_cleanup function above.

Log message, progress messages are handled transparently here.

Function `lib/proto.c:guestfs_int_recv`

```
int
guestfs_int_recv (guestfs_h *g, const char *fn,
                  guestfs_message_header *hdr,
                  guestfs_message_error *err,
                  xdrproc_t xdrp, char *ret)
```

Receive a reply.

Function `lib/proto.c:guestfs_int_recv_discard`

```
int
guestfs_int_recv_discard (guestfs_h *g, const char *fn)
```

Same as `guestfs_int_recv`, but it discards the reply message.

Notes (XXX):

- This returns an int, but all current callers ignore it.
- The error string may end up being set twice on error paths.

Function `lib/proto.c:guestfs_int_recv_file`

```
int
guestfs_int_recv_file (guestfs_h *g, const char *filename)
```

Returns `−1` = error, `0` = EOF, `>0` = more data

Function `lib/proto.c:receive_file_data`

```
static ssize_t
receive_file_data (guestfs_h *g, void **buf_r)
```

Receive a chunk of file data.

Returns `-1` = error, `0` = EOF, `>0` = more data

File lib/qemu.c

Functions to handle qemu versions and features.

Function `lib/qemu.c:guestfs_int_test_qemu`

```
struct qemu_data *
guestfs_int_test_qemu (guestfs_h *g)
```

Test that the qemu binary (or wrapper) runs, and do `qemu -help` and other commands so we can find out the version of qemu and what options this qemu supports.

This caches the results in the cachedir so that as long as the qemu binary does not change, calling this is effectively free.

Function `lib/qemu.c:cache_filename`

```
static char *
cache_filename (guestfs_h *g, const char *cachedir,
                const struct stat *statbuf, const char *suffix)
```

Generate the filenames, for the stat file and the other cache files.

By including the size and mtime in the filename we also ensure that the same user can use multiple versions of qemu without conflicts.

Function `lib/qemu.c:parse_qemu_version`

```
static void
parse_qemu_version (guestfs_h *g, const char *qemu_help,
                    struct version *qemu_version)
```

Parse the first line of `qemu_help` into the major and minor version of qemu, but don't fail if parsing is not possible.

Function `lib/qemu.c:parse_json`

```
static void
parse_json (guestfs_h *g, const char *json, json_t **treep)
```

Parse the json output from QMP. But don't fail if parsing is not possible.

Function `lib/qemu.c:parse_has_kvm`

```
static void
parse_has_kvm (guestfs_h *g, const char *json, bool *ret)
```

Parse the json output from QMP query-kvm to find out if KVM is enabled on this machine. Don't fail if parsing is not possible, assume KVM is available.

The JSON output looks like: {"return": {"enabled": true, "present": true}}

Function `lib/qemu.c:generic_read_cache`

```
static int
generic_read_cache (guestfs_h *g, const char *filename, char **strp)
```

Generic functions for reading and writing the cache files, used where we are just reading and writing plain text strings.

Function `lib/qemu.c:generic_qmp_test`

```
static int
generic_qmp_test (guestfs_h *g, struct qemu_data *data,
                  const char *qmp_command,
                  char **outp)
```

Run a generic QMP test on the QEMU binary.

Function `lib/qemu.c:guestfs_int_qemu_version`

```
struct version
guestfs_int_qemu_version (guestfs_h *g, struct qemu_data *data)
```

Return the parsed version of qemu.

Function `lib/qemu.c:guestfs_int_qemu_supports`

```
int
guestfs_int_qemu_supports (guestfs_h *g, const struct qemu_data *data,
                           const char *option)
```

Test if option is supported by qemu command line (just by grepping the help text).

Function `lib/qemu.c:guestfs_int_qemu_supports_device`

```
int
guestfs_int_qemu_supports_device (guestfs_h *g,
                                  const struct qemu_data *data,
                                  const char *device_name)
```

Test if device is supported by qemu (currently just greps the `qemu -device ?` output).

Function `lib/qemu.c:guestfs_int_qemu_mandatory_locking`

```
int
guestfs_int_qemu_mandatory_locking (guestfs_h *g,
                                    const struct qemu_data *data)
```

Test if the qemu binary uses mandatory file locking, added in QEMU >= 2.10 (but sometimes disabled).

Function `lib/qemu.c:guestfs_int_qemu_escape_param`

```
char *
guestfs_int_qemu_escape_param (guestfs_h *g, const char *param)
```

Escape a qemu parameter.

Every `,` becomes `,,`. The caller must free the returned string.

XXX This functionality is now only used when constructing a qemu-img command in *lib/create.c*. We should extend the qemuopts library to cover this use case.

Function `lib/qemu.c:guestfs_int_drive_source_qemu_param`

```
char *
guestfs_int_drive_source_qemu_param (guestfs_h *g,
                                     const struct drive_source *src)
```

Useful function to format a drive + protocol for qemu.

Note that the qemu parameter is the bit after `"file="`. It is not escaped here, but would usually be escaped if passed to qemu as part of a full −drive parameter (but not for **qemu−img** (1)).

Function `lib/qemu.c:guestfs_int_discard_possible`

```
bool
guestfs_int_discard_possible (guestfs_h *g, struct drive *drv,
                              const struct version *qemu_version)
```

Test if discard is both supported by qemu AND possible with the underlying file or device. This returns`1` if discard is possible. It returns `0` if not possible and sets the error to the reason why.

This function is called when the user set `discard == "enable"`.

Function `lib/qemu.c:guestfs_int_free_qemu_data`

```
void
guestfs_int_free_qemu_data (struct qemu_data *data)
```

Free the `struct qemu_data`.

File lib/rescue.c

Support for **virt−rescue** (1).

File lib/stringsbuf.c

An expandable NULL-terminated vector of strings (like `argv`).

Use the `DECLARE_STRINGSBUF` macro to declare the stringsbuf.

Note: Don't confuse this with stringsbuf in the daemon which is a different type with different methods.

Function `lib/stringsbuf.c:guestfs_int_add_string_nodup`

```
void
guestfs_int_add_string_nodup (guestfs_h *g, struct stringsbuf *sb, char *str)
```

Add a string to the end of the list.

This doesn't call **strdup** (3) on the string, so the string itself is stored inside the vector.

Function `lib/stringsbuf.c:guestfs_int_add_string`

```
void
guestfs_int_add_string (guestfs_h *g, struct stringsbuf *sb, const char *str)
```

Add a string to the end of the list.

This makes a copy of the string.

Function `lib/stringsbuf.c:guestfs_int_add_sprintf`

```
void
guestfs_int_add_sprintf (guestfs_h *g, struct stringsbuf *sb,
                         const char *fs, ...)
```

Add a string to the end of the list.

Uses an sprintf-like format string when creating the string.

Function `lib/stringsbuf.c:guestfs_int_end_stringsbuf`

```
void
guestfs_int_end_stringsbuf (guestfs_h *g, struct stringsbuf *sb)
```

Finish the string buffer.

This adds the terminating NULL to the end of the vector.

Function `lib/stringsbuf.c:guestfs_int_free_stringsbuf`

```
void
guestfs_int_free_stringsbuf (struct stringsbuf *sb)
```

Free the string buffer and the strings.

File lib/tmpdirs.c

Handle temporary directories.

Function `lib/tmpdirs.c:set_abs_path`

```
static int
set_abs_path (guestfs_h *g, const char *ctxstr,
              const char *tmpdir, char **tmpdir_ret)
```

We need to make all tmpdir paths absolute because lots of places in the code assume this. Do it at the time we set the path or read the environment variable (https://bugzilla.redhat.com/882417).

The `ctxstr` parameter is a string displayed in error messages giving the context of the operation (eg. name of environment variable being used, or API function being called).

Function `lib/tmpdirs.c:guestfs_impl_get_tmpdir`

```
char *
guestfs_impl_get_tmpdir (guestfs_h *g)
```

Implements the `guestfs_get_tmpdir` API.

Note this actually calculates the tmpdir, so it never returns `NULL`.

Function `lib/tmpdirs.c:guestfs_impl_get_cachedir`

```
char *
guestfs_impl_get_cachedir (guestfs_h *g)
```

Implements the `guestfs_get_cachedir` API.

Note this actually calculates the cachedir, so it never returns `NULL`.

Function `lib/tmpdirs.c:guestfs_impl_get_sockdir`

```
char *
guestfs_impl_get_sockdir (guestfs_h *g)
```

Implements the `guestfs_get_sockdir` API.

Note this actually calculates the sockdir, so it never returns `NULL`.

Function `lib/tmpdirs.c:guestfs_int_lazy_make_tmpdir`

```
int
guestfs_int_lazy_make_tmpdir (guestfs_h *g)
```

The `g->tmpdir` (per-handle temporary directory) is not created when the handle is created. Instead we create it lazily before the first time it is used, or during launch.

Function `lib/tmpdirs.c:guestfs_int_make_temp_path`

```
char *
guestfs_int_make_temp_path (guestfs_h *g,
                            const char *name, const char *extension)
```

Generate unique temporary paths for temporary files.

Returns a unique path or NULL on error.

Function `lib/tmpdirs.c:guestfs_int_lazy_make_supermin_appliance_dir`

```
char *
guestfs_int_lazy_make_supermin_appliance_dir (guestfs_h *g)
```

Create the supermin appliance directory under cachedir, if it does not exist.

Sanity-check that the permissions on the cachedir are safe, in case it has been pre-created maliciously or tampered with.

Returns the directory name which the caller must free.

Function `lib/tmpdirs.c:guestfs_int_recursive_remove_dir`

```
void
guestfs_int_recursive_remove_dir (guestfs_h *g, const char *dir)
```

Recursively remove a temporary directory. If removal fails, just return (it's a temporary directory so it'll eventually be cleaned up by a temp cleaner).

This is implemented using `rm -rf` because that's simpler and safer.

File lib/umask.c

Return current umask in a thread-safe way.

glibc documents, but does not actually implement, a "**getumask** (3)" call.

We use `Umask` from */proc/self/status* for Linux ≥ 4.7. For older Linux and other Unix, this file implements an expensive but thread-safe way to get the current process's umask.

Thanks to: Josh Stone, Jiri Jaburek, Eric Blake.

Function `lib/umask.c:guestfs_int_getumask`

```
int
guestfs_int_getumask (guestfs_h *g)
```

Returns the current process's umask. On failure, returns `−1` and sets the error in the guestfs handle.

Function `lib/umask.c:get_umask_from_proc`

```
static int
get_umask_from_proc (guestfs_h *g)
```

For Linux ≥ 4.7 get the umask from */proc/self/status*.

On failure this returns `−1`. However if we could not open the */proc* file or find the `Umask` entry in it, return `−2` which causes the fallback path to run.

Function `lib/umask.c:get_umask_from_fork`

```
static int
get_umask_from_fork (guestfs_h *g)
```

Fallback method of getting the umask using fork.

File lib/unit−tests.c

Unit tests of internal functions.

These tests may use a libguestfs handle, but must not launch the handle. Also, avoid long-running tests.

Function `lib/unit−tests.c:test_split`

```
static void
test_split (void)
```

Test `guestfs_int_split_string`.

Function `lib/unit−tests.c:test_concat`

```
static void
test_concat (void)
```

Test `guestfs_int_concat_strings`.

Function `lib/unit−tests.c:test_join`

```
static void
test_join (void)
```

Test `guestfs_int_join_strings`.

Function `lib/unit−tests.c:test_validate_guid`

```
static void
test_validate_guid (void)
```

Test `guestfs_int_validate_guid`.

Function `lib/unit-tests.c:test_drive_name`

```
static void
test_drive_name (void)
```

Test `guestfs_int_drive_name`.

Function `lib/unit-tests.c:test_drive_index`

```
static void
test_drive_index (void)
```

Test `guestfs_int_drive_index`.

Function `lib/unit-tests.c:test_getumask`

```
static void
test_getumask (void)
```

Test `guestfs_int_getumask`.

Function `lib/unit-tests.c:test_command`

```
static void
test_command (void)
```

Test `guestfs_int_new_command` etc.

XXX These tests could be made much more thorough. So far we simply test that it's not obviously broken.

Function `lib/unit-tests.c:test_qemu_escape_param`

```
static void
test_qemu_escape_param (void)
```

Test `guestfs_int_qemu_escape_param`

XXX I wanted to make this test run qemu, passing some parameters which need to be escaped, but I cannot think of a way to do that without launching a VM.

Function `lib/unit-tests.c:test_timeval_diff`

```
static void
test_timeval_diff (void)
```

Test `guestfs_int_timeval_diff`.

File lib/version.c

This file provides simple version number management.

Function `lib/version.c:guestfs_int_version_from_x_y`

```
int
guestfs_int_version_from_x_y (guestfs_h *g, struct version *v, const char *str)
```

Parses a version from a string, looking for a `X.Y` pattern.

Returns `-1` on failure (like failed integer parsing), `0` on missing match, and `1` on match and successful parsing. `v` is changed only on successful match.

Function `lib/version.c:guestfs_int_version_from_x_y_re`

```
int
guestfs_int_version_from_x_y_re (guestfs_h *g, struct version *v,
                                 const char *str, const pcre2_code *re)
```

Parses a version from a string, using the specified `re` as regular expression which *must* provide (at least) two matches.

Returns `−1` on failure (like failed integer parsing), `0` on missing match, and `1` on match and successful parsing. `v` is changed only on successful match.

Function `lib/version.c:guestfs_int_version_from_x_y_or_x`

```
int
guestfs_int_version_from_x_y_or_x (guestfs_h *g, struct version *v,
                                   const char *str)
```

Parses a version from a string, either looking for a `X.Y` pattern or considering it as whole integer.

Returns `−1` on failure (like failed integer parsing), `0` on missing match, and `1` on match and successful parsing. `v` is changed only on successful match.

Function `lib/version.c:guestfs_int_parse_unsigned_int`

```
int
guestfs_int_parse_unsigned_int (guestfs_h *g, const char *str)
```

Parse small, unsigned ints, as used in version numbers.

This will fail with an error if trailing characters are found after the integer.

Returns ≥ `0` on success, or `−1` on failure.

File lib/wait.c

Function `lib/wait.c:guestfs_int_waitpid`

```
int
guestfs_int_waitpid (guestfs_h *g, pid_t pid, int *status, const char *errmsg)
```

A safe version of **waitpid** (3) which retries if `EINTR` is returned.

Note: this only needs to be used in the library, or in programs that install a non-restartable `SIGCHLD` handler (which is not the case for any current libguestfs virt tools).

If the main program installs a SIGCHLD handler and sets it to be non-restartable, then what can happen is the library is waiting in a wait syscall, the child exits, `SIGCHLD` is sent to the process, and the wait syscall returns `EINTR`. Since the library cannot control the signal handler, we have to instead restart the wait syscall, which is the purpose of this wrapper.

Function `lib/wait.c:guestfs_int_waitpid_noerror`

```
void
guestfs_int_waitpid_noerror (pid_t pid)
```

Like `guestfs_int_waitpid`, but ignore errors.

Function `lib/wait.c:guestfs_int_wait4`

```
int
guestfs_int_wait4 (guestfs_h *g, pid_t pid, int *status,
                   struct rusage *rusage, const char *errmsg)
```

A safe version of **wait4** (2) which retries if `EINTR` is returned.

File lib/whole−file.c

Function `lib/whole−file.c:guestfs_int_read_whole_file`

```
int
guestfs_int_read_whole_file (guestfs_h *g, const char *filename,
                             char **data_r, size_t *size_r)
```

Read the whole file `filename` into a memory buffer.

The memory buffer is initialized and returned in `data_r`. The size of the file in bytes is returned in `size_r`. The return buffer must be freed by the caller.

On error this sets the error in the handle and returns `-1`.

For the convenience of callers, the returned buffer is NUL-terminated (the NUL is not included in the size).

The file must be a **regular**, **local**, **trusted** file. In particular, do not use this function to read files that might be under control of an untrusted user since that will lead to a denial-of-service attack.

Subdirectory *common/edit*

File common/edit/file−edit.c

This file implements common file editing in a range of utilities including **guestfish** (1), **virt−edit** (1), **virt−customize** (1) and **virt−builder** (1).

It contains the code for both interactive−(editor−)based editing and non-interactive editing using Perl snippets.

Function `common/edit/file-edit.c:edit_file_editor`

```
int
edit_file_editor (guestfs_h *g, const char *filename, const char *editor,
                  const char *backup_extension, int verbose)
```

Edit `filename` using the specified `editor` application.

If `backup_extension` is not null, then a copy of `filename` is saved with `backup_extension` appended to its file name.

If `editor` is null, then the `$EDITOR` environment variable will be queried for the editor application, leaving `vi` as fallback if not set.

Returns `-1` for failure, `0` on success, `1` if the editor did not change the file (e.g. the user closed the editor without saving).

Function `common/edit/file-edit.c:edit_file_perl`

```
int
edit_file_perl (guestfs_h *g, const char *filename, const char *perl_expr,
                const char *backup_extension, int verbose)
```

Edit `filename` running the specified `perl_expr` using Perl.

If `backup_extension` is not null, then a copy of `filename` is saved with `backup_extension` appended to its file name.

Returns `-1` for failure, `0` on success.

Subdirectory *common/options*

File common/options/config.c

This file parses the guestfish configuration file, usually *~/.libguestfs−tools.rc* or */etc/libguestfs−tools.conf*.

Note that `parse_config` is called very early, before command line parsing, before the `verbose` flag has been set, even before the global handle `g` is opened.

File common/options/decrypt.c

This file implements the decryption of disk images, usually done before mounting their partitions.

Function `common/options/decrypt.c:make_mapname`

```
static void
make_mapname (const char *device, char *mapname, size_t len)
```

Make a LUKS map name from the partition name, eg. `"/dev/vda2" => "cryptvda2"`

Function `common/options/decrypt.c:inspect_do_decrypt`

```
void
inspect_do_decrypt (guestfs_h *g, struct key_store *ks)
```

Simple implementation of decryption: look for any encrypted partitions and decrypt them, then rescan for VGs.

File common/options/display−options.c

This file contains common code used to implement *−−short−options* and *−−long−options* in C virt tools. (The equivalent for OCaml virt tools is implemented by *common/mltools/getopt.ml*).

These "hidden" options are used to implement bash tab completion.

Function `common/options/display−options.c:display_short_options`

```
void
display_short_options (const char *format)
```

Implements the internal `tool --short-options` flag, which just lists out the short options available. Used by bash completion.

Function `common/options/display−options.c:display_long_options`

```
void
display_long_options (const struct option *long_options)
```

Implements the internal `tool --long-options` flag, which just lists out the long options available. Used by bash completion.

File common/options/domain.c

Implements the guestfish (and other tools) *−d* option.

Function `common/options/domain.c:add_libvirt_drives`

```
int
add_libvirt_drives (guestfs_h *g, const char *guest)
```

This function is called when a user invokes `guestfish−dguest`.

Returns the number of drives added (`>0`), or `−1` for failure.

File common/options/inspect.c

This file implements inspecting the guest and mounting the filesystems found in the right places. It is used by the **guestfish**(1) *−i* option and some utilities such as **virt−cat**(1).

Function `common/options/inspect.c:inspect_mount_handle`

```
void
inspect_mount_handle (guestfs_h *g, struct key_store *ks)
```

This function implements the *−i* option.

Function `common/options/inspect.c:print_inspect_prompt`

```
void
print_inspect_prompt (void)
```

This function is called only if `inspect_mount_root` was called, and only after we've printed the prompt in interactive mode.

File common/options/keys.c

Function `common/options/keys.c:read_key`

```
char *
read_key (const char *param)
```

Read a passphrase ('Key') from */dev/tty* with echo off.

The caller (*fish/cmds.c*) will call free on the string afterwards. Based on the code in cryptsetup file *lib/utils.c*.

File common/options/options.c

This file contains common options parsing code used by guestfish and many other tools which share a common options syntax.

For example, guestfish, virt-cat, virt-ls etc all support the *−a* option, and that is handled in all of those tools using a macro `OPTION_a` defined in *fish/options.h*.

There are a lot of common global variables used, `drvs` accumulates the list of drives, `verbose` for the *−v* flag, and many more.

Function `common/options/options.c:option_a`

```
void
option_a (const char *arg, const char *format, int blocksize,
          struct drv **drvsp)
```

Handle the guestfish *−a* option on the command line.

Function `common/options/options.c:option_d`

```
void
option_d (const char *arg, struct drv **drvsp)
```

Handle the *−d* option when passed on the command line.

Function `common/options/options.c:display_mountpoints_on_failure`

```
static void
display_mountpoints_on_failure (const char *mp_device,
                                const char *user_supplied_options)
```

If the *−m* option fails on any command, display a useful error message listing the mountpoints.

File common/options/uri.c

This file implements URI parsing for the *−a* option, in many utilities including **guestfish** (1), **virt−cat** (1), **virt−builder** (1), **virt−customize** (1), etc.

Subdirectory *common/parallel*

File common/parallel/domains.c

This file is used by `virt-df` and some of the other tools when they are implicitly asked to operate over all libvirt domains (VMs), for example when `virt-df` is called without specifying any particular disk image.

It hides the complexity of querying the list of domains from libvirt.

Function `common/parallel/domains.c:free_domains`

```
void
free_domains (void)
```

Frees up everything allocated by `get_all_libvirt_domains`.

Function `common/parallel/domains.c:get_all_libvirt_domains`

```
void
get_all_libvirt_domains (const char *libvirt_uri)
```

Read all libguest guests into the global variables `domains` and `nr_domains`. The guests are ordered by name. This exits on any error.

File common/parallel/estimate−max−threads.c

Function `common/parallel/estimate-max-threads.c:estimate_max_threads`

```
size_t
estimate_max_threads (void)
```

This function uses the output of `free -m` to estimate how many libguestfs appliances could be safely started in parallel. Note that it always returns ≥ 1.

Function `common/parallel/estimate-max-threads.c:read_line_from`

```
static char *
read_line_from (const char *cmd)
```

Run external command and read the first line of output.

File common/parallel/parallel.c

This file is used by `virt-df` and some of the other tools when they need to run multiple parallel libguestfs instances to operate on a large number of libvirt domains efficiently.

It implements a multithreaded work queue. In addition it reorders the output so the output still appears in the same order as the input (ie. still ordered alphabetically).

Function `common/parallel/parallel.c:start_threads`

```
int
start_threads (size_t option_P, guestfs_h *options_handle, work_fn work)
```

Run the threads and work through the global list of libvirt domains.

`option_P` is whatever the user passed in the *−P* option, or `0` if the user didn't use the *−P* option (in which case the number of threads is chosen heuristically).

`options_handle` (which may be `NULL`) is the global guestfs handle created by the options mini-library.

The work function (`work`) should do the work (inspecting the domain, etc.) on domain index `i`. However it *must not* print out any result directly. Instead it prints anything it needs to the supplied `FILE *`. The work function should return `0` on success or `−1` on error.

The `start_threads` function returns `0` if all work items completed successfully, or `−1` if there was an error.

Subdirectory *common/progress*

File common/progress/progress.c

This file implements the progress bar in **guestfish** (1), **virt−resize** (1) and **virt−sparsify** (1).

Function `common/progress/progress.c:progress_bar_init`

```
struct progress_bar *
progress_bar_init (unsigned flags)
```

Initialize a progress bar struct.

It is intended that you can reuse the same struct for multiple commands (but only in a single thread). Call `progress_bar_reset` before each new command.

Function `common/progress/progress.c:progress_bar_free`

```
void
progress_bar_free (struct progress_bar *bar)
```

Free a progress bar struct.

Function `common/progress/progress.c:progress_bar_reset`

```
void
progress_bar_reset (struct progress_bar *bar)
```

This function should be called just before you issue any command.

Function `common/progress/progress.c:estimate_remaining_time`

```
static double
estimate_remaining_time (struct progress_bar *bar, double ratio)
```

Return remaining time estimate (in seconds) for current call.

This returns the running mean estimate of remaining time, but if the latest estimate of total time is greater than two s.d.'s from the running mean then we don't print anything because we're not confident that the estimate is meaningful. (Returned value is <0.0 when nothing should be printed).

Function `common/progress/progress.c:progress_bar_set`

```
void
progress_bar_set (struct progress_bar *bar,
                  uint64_t position, uint64_t total)
```

Set the position of the progress bar.

This should be called from a `GUESTFS_EVENT_PROGRESS` event callback.

Subdirectory *common/qemuopts*

File common/qemuopts/qemuopts−tests.c

Unit tests of internal functions.

These tests may use a libguestfs handle, but must not launch the handle. Also, avoid long-running tests.

File common/qemuopts/qemuopts.c

Mini-library for writing qemu command lines and qemu config files.

There are some shortcomings with the model used for qemu options which aren't clear until you try to convert options into a configuration file. However if we attempted to model the options in more detail then this library would be both very difficult to use and incompatible with older versions of qemu. Hopefully the current model is a decent compromise.

For reference here are the problems:

- There's inconsistency in qemu between options and config file, eg. `−smp 4` becomes:

  ```
  [smp−opts]
    cpus = "4"
  ```

- Similar to the previous point, you can write either `−smp 4` or `−smp cpus=4` (although this won't work in very old qemu). When generating a config file you need to know the implicit key name.
- In `−opt key=value,...` the `key` is really a tree/array specifier. The way this works is complicated but hinted at here: http://git.qemu.org/?p=qemu.git;a=blob;f=util/keyval.c;h=93d5db6b590427e412dfb172f1c406d6dd8958c1;hb=HEA
- Some options are syntactic sugar. eg. `−kernel foo` is sugar for `−machine kernel=foo`.

Function `common/qemuopts/qemuopts.c:qemuopts_create`

```
struct qemuopts *
qemuopts_create (void)
```

Create an empty list of qemu options.

The caller must eventually free the list by calling `qemuopts_free`.

Returns `NULL` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_free`

```
void
qemuopts_free (struct qemuopts *qopts)
```

Free the list of qemu options.

Function `common/qemuopts/qemuopts.c:qemuopts_add_flag`

```
int
qemuopts_add_flag (struct qemuopts *qopts, const char *flag)
```

Add a command line flag which has no argument. eg:

```
qemuopts_add_flag (qopts, "-no-user-config");
```

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_add_arg`

```
int
qemuopts_add_arg (struct qemuopts *qopts, const char *flag, const char *value)
```

Add a command line flag which has a single argument. eg:

```
qemuopts_add_arg (qopts, "-m", "1024");
```

Don't use this if the argument is a comma-separated list, since quoting will not be done properly. See `qemuopts_add_arg_list`.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_add_arg_format`

```
int
qemuopts_add_arg_format (struct qemuopts *qopts, const char *flag,
                         const char *fs, ...)
```

Add a command line flag which has a single formatted argument. eg:

```
qemuopts_add_arg_format (qopts, "-m", "%d", 1024);
```

Don't use this if the argument is a comma-separated list, since quoting will not be done properly. See `qemuopts_add_arg_list`.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_add_arg_noquote`

```
int
qemuopts_add_arg_noquote (struct qemuopts *qopts, const char *flag,
                          const char *value)
```

This is like `qemuopts_add_arg` except that no quoting is done on the value.

For `qemuopts_to_script` and `qemuopts_to_channel`, this means that neither shell quoting nor qemu comma quoting is done on the value.

For `qemuopts_to_argv` this means that qemu comma quoting is not done.

`qemuopts_to_config*` will fail.

You should use this with great care.

Function `common/qemuopts/qemuopts.c:qemuopts_start_arg_list`

```
int
qemuopts_start_arg_list (struct qemuopts *qopts, const char *flag)
```

Start an argument that takes a comma-separated list of fields.

Typical usage is like this (with error handling omitted):

```
qemuopts_start_arg_list (qopts, "-drive");
qemuopts_append_arg_list (qopts, "file=foo");
qemuopts_append_arg_list_format (qopts, "if=%s", "ide");
qemuopts_end_arg_list (qopts);
```

which would construct `-drive file=foo,if=ide`

See also `qemuopts_add_arg_list` for a way to do simple cases in one call.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_add_arg_list`

```
int
qemuopts_add_arg_list (struct qemuopts *qopts, const char *flag,
                       const char *elem0, ...)
```

Add a command line flag which has a list of arguments. eg:

```
qemuopts_add_arg_list (qopts, "-drive", "file=foo", "if=ide", NULL);
```

This is turned into a comma-separated list, like: `-drive file=foo,if=ide`. Note that this handles qemu quoting properly, so individual elements may contain commas and this will do the right thing.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_set_binary`

```
int
qemuopts_set_binary (struct qemuopts *qopts, const char *binary)
```

Set the qemu binary name.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_set_binary_by_arch`

```
int
qemuopts_set_binary_by_arch (struct qemuopts *qopts, const char *arch)
```

Set the qemu binary name to `qemu-system-[arch]`.

As a special case if `arch` is `NULL`, the binary is set to the KVM binary for the current host architecture:

```
qemuopts_set_binary_by_arch (qopts, NULL);
```

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_to_script`

```
int
qemuopts_to_script (struct qemuopts *qopts, const char *filename)
```

Write the qemu options to a script.

`qemuopts_set_binary*` must be called first.

The script file will start with `#!/bin/sh` and will be chmod to mode `0755`.

Returns `0` on success. Returns `-1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:shell_quote`

```
static void
shell_quote (const char *str, FILE *fp)
```

Print `str` to `fp`, shell-quoting it if necessary.

Function `common/qemuopts/qemuopts.c:shell_and_comma_quote`

```
static void
shell_and_comma_quote (const char *str, FILE *fp)
```

Print `str` to `fp` doing both shell and qemu comma quoting.

Function `common/qemuopts/qemuopts.c:qemuopts_to_channel`

```
int
qemuopts_to_channel (struct qemuopts *qopts, FILE *fp)
```

Write the qemu options to a `FILE *fp`.

`qemuopts_set_binary*` must be called first.

Only the qemu command line is written. The caller may need to add `#!/bin/sh` and may need to chmod the resulting file to `0755`.

Returns `0` on success. Returns `−1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_to_argv`

```
char **
qemuopts_to_argv (struct qemuopts *qopts)
```

Return a NULL-terminated argument list, of the kind that can be passed directly to **execv** (3).

`qemuopts_set_binary*` must be called first. It will be returned as `argv[0]` in the returned list.

The list of strings and the strings themselves must be freed by the caller.

Returns `NULL` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_to_config_file`

```
int
qemuopts_to_config_file (struct qemuopts *qopts, const char *filename)
```

Write the qemu options to a qemu config file, suitable for reading in using `qemu −readconfig filename`.

Note that qemu config files have limitations on content and quoting, so not all qemuopts structs can be written (this function returns an error in these cases). For more information see https://habkost.net/posts/2016/12/qemu−apis−qemuopts.html
https://bugs.launchpad.net/qemu/+bug/1686364

Also, command line argument names and config file sections sometimes have different names. For example the equivalent of `−m 1024` is:

```
[memory]
  size = "1024"
```

This code does *not* attempt to convert between the two forms. You just need to know how to do that yourself.

Returns `0` on success. Returns `−1` on error, setting `errno`.

Function `common/qemuopts/qemuopts.c:qemuopts_to_config_channel`

```
int
qemuopts_to_config_channel (struct qemuopts *qopts, FILE *fp)
```

Same as `qemuopts_to_config_file`, but this writes to a `FILE *fp`.

Subdirectory *common/utils*

File common/utils/cleanups.c

Libguestfs uses `CLEANUP_*` macros to simplify temporary allocations. They are implemented using the `__attribute__((cleanup))` feature of gcc and clang. Typical usage is:

```
fn ()
{
  CLEANUP_FREE char *str = NULL;
  str = safe_asprintf (g, "foo");
  // str is freed automatically when the function returns
}
```

There are a few catches to be aware of with the cleanup mechanism:

- If a cleanup variable is not initialized, then you can end up calling **free** (3) with an undefined value, resulting in the program crashing. For this reason, you should usually initialize every cleanup variable with something, eg. `NULL`
- Don’t mark variables holding return values as cleanup variables.

- The `main()` function shouldn't use cleanup variables since it is normally exited by calling **exit**(3), and that doesn't call the cleanup handlers.

The functions in this file are used internally by the `CLEANUP_*` macros. Don't call them directly.

File common/utils/gnulib−cleanups.c

Libguestfs uses `CLEANUP_*` macros to simplify temporary allocations. They are implemented using the `__attribute__((cleanup))` feature of gcc and clang. Typical usage is:

```
fn ()
{
  CLEANUP_FREE char *str = NULL;
  str = safe_asprintf (g, "foo");
  // str is freed automatically when the function returns
}
```

There are a few catches to be aware of with the cleanup mechanism:

- If a cleanup variable is not initialized, then you can end up calling **free**(3) with an undefined value, resulting in the program crashing. For this reason, you should usually initialize every cleanup variable with something, eg. `NULL`
- Don't mark variables holding return values as cleanup variables.
- The `main()` function shouldn't use cleanup variables since it is normally exited by calling **exit**(3), and that doesn't call the cleanup handlers.

The functions in this file are used internally by the `CLEANUP_*` macros. Don't call them directly.

File common/utils/guestfs−utils.h

This header file is included in all ''frontend'' parts of libguestfs, namely the library, non-C language bindings, virt tools and tests.

The daemon does **not** use this header. If you need a place to put something shared with absolutely everything including the daemon, put it in *lib/guestfs−internal−all.h*

If a definition is only needed by a single component of libguestfs (eg. just the library, or just a single virt tool) then it should **not** be here!

File common/utils/libxml2−writer−macros.h

These macros make it easier to write XML. To use them correctly you must be aware of these assumptions:

- The `xmlTextWriterPtr` is called `xo`. It is used implicitly by all the macros.
- On failure, a function called `xml_error` is called which you must define (usually as a macro). You must use `CLEANUP_*` macros in your functions if you want correct cleanup of local variables along the error path.
- All the ''bad'' casting is hidden inside the macros.

Definition `common/utils/libxml2−writer−macros.h:start_element`

```
#define start_element
```

To define an XML element use:

```
start_element ("name") {
  ...
} end_element ();
```

which produces `<name>...</name>`

Definition `common/utils/libxml2−writer−macros.h:empty_element`

```
#define empty_element
```

To define an empty element:

```
 empty_element ("name");
```

which produces `<name/>`

Definition `common/utils/libxml2-writer-macros.h:single_element`

```
 #define single_element
```

To define a single element with no attributes containing some text:

```
 single_element ("name", text);
```

which produces `<name>text</name>`

Definition `common/utils/libxml2-writer-macros.h:single_element_format`

```
 #define single_element_format
```

To define a single element with no attributes containing some text using a format string:

```
 single_element_format ("cores", "%d", nr_cores);
```

which produces `<cores>4</cores>`

Definition `common/utils/libxml2-writer-macros.h:attribute`

```
 #define attribute
```

To define an XML element with attributes, use:

```
 start_element ("name") {
   attribute ("foo", "bar");
   attribute_format ("count", "%d", count);
   ...
 } end_element ();
```

which produces `<name foo="bar" count="123">...</name>`

Definition `common/utils/libxml2-writer-macros.h:attribute_ns`

```
 #define attribute_ns
```

`attribute_ns (prefix, key, namespace_uri, value)`defines a namespaced attrib ute.

Definition `common/utils/libxml2-writer-macros.h:string`

```
 #define string
```

To define a verbatim string, use:

```
 string ("hello");
```

Definition `common/utils/libxml2-writer-macros.h:string_format`

```
 #define string_format
```

To define a verbatim string using a format string, use:

```
 string ("%s, world", greeting);
```

Definition `common/utils/libxml2-writer-macros.h:base64`

```
 #define base64
```

To write a string encoded as base64:

```
 base64 (data, size);
```

Definition `common/utils/libxml2-writer-macros.h:comment`

```
 #define comment
```

To define a comment in the XML, use:

```
  comment ("number of items = %d", nr_items);
```

File common/utils/stringlists−utils.c

Utility functions used by the library, tools and language bindings.

These functions *must not* call internal library functions such as `safe_*`, `error` or `perrorf`, or any `guestfs_int_*`.

Function `common/utils/stringlists−utils.c:guestfs_int_split_string`

```
char **
guestfs_int_split_string (char sep, const char *str)
```

Split string at separator character `sep`, returning the list of strings. Returns `NULL` on memory allocation failure.

Note (assuming `sep` is `:`):

`str == NULL`
    aborts

`str == ''''`
    returns `[]`

`str == ''abc''`
    returns `["abc"]`

`str == '':''`
    returns `["", ""]`

File common/utils/utils.c

Utility functions used by the library, tools and language bindings.

These functions *must not* call internal library functions such as `safe_*`, `error` or `perrorf`, or any `guestfs_int_*`.

Function `common/utils/utils.c:guestfs_int_replace_string`

```
char *
guestfs_int_replace_string (const char *str, const char *s1, const char *s2)
```

Replace every instance of `s1` appearing in `str` with `s2`. A newly allocated string is returned which must be freed by the caller. If allocation fails this can return `NULL`.

For example:

```
replace_string ("abcabb", "ab", "a");
```

would return `"acab"`.

Function `common/utils/utils.c:guestfs_int_exit_status_to_string`

```
char *
guestfs_int_exit_status_to_string (int status, const char *cmd_name,
                                   char *buffer, size_t buflen)
```

Translate a wait/system exit status into a printable string.

Function `common/utils/utils.c:guestfs_int_random_string`

```
int
guestfs_int_random_string (char *ret, size_t len)
```

Return a random string of characters.

Notes:

- The `ret` buffer must have length `len+1` in order to store the final `\0` character.
- There is about 5 bits of randomness per output character (so about `5*len` bits of randomness in the resulting string).

Function `common/utils/utils.c:guestfs_int_drive_name`

```
char *
guestfs_int_drive_name (size_t index, char *ret)
```

This turns a drive index (eg. `27`) into a drive name (eg. `"ab"`).

Drive indexes count from `0`. The return buffer has to be large enough for the resulting string, and the returned pointer points to the *end* of the string.

https://rwmj.wordpress.com/2011/01/09/how−are−linux−drives−named−beyond−drive−26−devsdz/

Function `common/utils/utils.c:guestfs_int_drive_index`

```
ssize_t
guestfs_int_drive_index (const char *name)
```

The opposite of `guestfs_int_drive_name`. Take a string like `"ab"` and return the index (eg `27`).

Note that you must remove any prefix such as `"hd"`, `"sd"` etc, or any partition number before calling the function.

Function `common/utils/utils.c:guestfs_int_is_true`

```
int
guestfs_int_is_true (const char *str)
```

Similar to `Tcl_GetBoolean`.

Function `common/utils/utils.c:guestfs_int_string_is_valid`

```
bool
guestfs_int_string_is_valid (const char *str,
                             size_t min_length, size_t max_length,
                             int flags, const char *extra)
```

Check a string for validity, that it contains only certain characters, and minimum and maximum length. This function is usually wrapped in a VALID_* macro, see *lib/drives.c* for an example.

`str` is the string to check.

`min_length` and `max_length` are the minimum and maximum length checks. `0` means no check.

The flags control:

`VALID_FLAG_ALPHA`
    7−bit ASCII-only alphabetic characters are permitted.

`VALID_FLAG_DIGIT`
    7−bit ASCII-only digits are permitted.

`extra` is a set of extra characters permitted, in addition to alphabetic and/or digits. (`extra = NULL`for no extra).

Returns boolean `true` if the string is valid (passes all the tests), or `false` if not.

Function `common/utils/utils.c:guestfs_int_fadvise_normal`

```
void
guestfs_int_fadvise_normal (int fd)
```

Hint that we will read or write the file descriptor normally.

On Linux, this clears the `FMODE_RANDOM` flag on the file [see below] and sets the per-file number of readahead pages to equal the block device readahead setting.

It’s OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_fadvise_sequential`

```
void
guestfs_int_fadvise_sequential (int fd)
```

Hint that we will read or write the file descriptor sequentially.

On Linux, this clears the `FMODE_RANDOM` flag on the file [see below] and sets the per-file number of readahead pages to twice the block device readahead setting.

It's OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_fadvise_random`

```
void
guestfs_int_fadvise_random (int fd)
```

Hint that we will read or write the file descriptor randomly.

On Linux, this sets the `FMODE_RANDOM` flag on the file. The effect of this flag is to:

- Disable normal sequential file readahead.
- If any read of the file is done which misses in the page cache, 2MB are read into the page cache. [I think − I'm not sure I totally understand what this is doing]

It's OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_fadvise_noreuse`

```
void
guestfs_int_fadvise_noreuse (int fd)
```

Hint that we will access the data only once.

On Linux, this does nothing.

It's OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_fadvise_dontneed`

```
void
guestfs_int_fadvise_dontneed (int fd)
```

Hint that we will not access the data in the near future.

On Linux, this immediately writes out any dirty pages in the page cache and then invalidates (drops) all pages associated with this file from the page cache. Apparently it does this even if the file is opened or being used by other processes. This setting is not persistent; if you subsequently read the file it will be cached in the page cache as normal.

It's OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_fadvise_willneed`

```
void
guestfs_int_fadvise_willneed (int fd)
```

Hint that we will access the data in the near future.

On Linux, this immediately reads the whole file into the page cache. This setting is not persistent; subsequently pages may be dropped from the page cache as normal.

It's OK to call this on a non-file since we ignore failure as it is only a hint.

Function `common/utils/utils.c:guestfs_int_shell_unquote`

```
char *
guestfs_int_shell_unquote (const char *str)
```

Unquote a shell-quoted string.

Augeas passes strings to us which may be quoted, eg. if they come from files in */etc/sysconfig*. This function can do simple unquoting of these strings.

Note this function does not do variable substitution, since that is impossible without knowing the file context and indeed the environment under which the shell script is run. Configuration files should not use complex quoting.

`str` is the input string from Augeas, a string that may be single− or double-quoted or may not be quoted. The returned string is unquoted, and must be freed by the caller. `NULL` is returned on error and `errno` is set accordingly.

For information on double-quoting in bash, see https://www.gnu.org/software/bash/manual/html_node/Double−Quotes.html

Function `common/utils/utils.c:guestfs_int_is_reg`

```
int
guestfs_int_is_reg (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a regular file.

Function `common/utils/utils.c:guestfs_int_is_dir`

```
int
guestfs_int_is_dir (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a directory.

Function `common/utils/utils.c:guestfs_int_is_chr`

```
int
guestfs_int_is_chr (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a char device.

Function `common/utils/utils.c:guestfs_int_is_blk`

```
int
guestfs_int_is_blk (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a block device.

Function `common/utils/utils.c:guestfs_int_is_fifo`

```
int
guestfs_int_is_fifo (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a named pipe (FIFO).

Function `common/utils/utils.c:guestfs_int_is_lnk`

```
int
guestfs_int_is_lnk (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a symbolic link.

Function `common/utils/utils.c:guestfs_int_is_sock`

```
int
guestfs_int_is_sock (int64_t mode)
```

Return true if the `guestfs_statns` or `guestfs_lstatns` `st_mode` field represents a Unix domain socket.

Function `common/utils/utils.c:guestfs_int_full_path`

```
char *
guestfs_int_full_path (const char *dir, const char *name)
```

Concatenate `dir` and `name` to create a path. This correctly handles the case of concatenating `"/" + "filename"` as well as `"/dir" + "filename"`. `name` may be `NULL`.

The caller must free the returned path.

This function sets `errno` and returns `NULL` on error.

Function `common/utils/utils.c:guestfs_int_hexdump`

```
void
guestfs_int_hexdump (const void *data, size_t len, FILE *fp)
```

Hexdump a block of memory to `FILE *`, used for debugging.

Function `common/utils/utils.c:guestfs_int_strerror`

```
const char *
guestfs_int_strerror (int errnum, char *buf, size_t buflen)
```

Thread-safe strerror_r.

This is a wrapper around the two variants of **strerror_r**(3) in glibc since it is hard to use correctly (RHBZ#2030396).

The buffer passed in should be large enough to store the error message (256 chars at least) and should be non-static. Note that the buffer might not be used, use the return value.

Subdirectory *common/visit*

File common/visit/visit.c

This file contains a recursive function for visiting all files and directories in a guestfs filesystem.

Adapted from https://rwmj.wordpress.com/2010/12/15/tip−audit−virtual−machine−for−setuid−files/

Function `common/visit/visit.c:visit`

```
int
visit (guestfs_h *g, const char *dir, visitor_function f, void *opaque)
```

Visit every file and directory in a guestfs filesystem, starting at `dir`.

`dir` may be `"/"` to visit the entire filesystem, or may be some subdirectory. Symbolic links are not followed.

The visitor function `f` is called once for every directory and every file. The parameters passed to `f` include the current directory name, the current file name (or `NULL` when we're visiting a directory), the `guestfs_statns` (file permissions etc), and the list of extended attributes of the file. The visitor function may return `−1` which causes the whole recursion to stop with an error.

Also passed to this function is an `opaque` pointer which is passed through to the visitor function.

Returns `0` if everything went OK, or `−1` if there was an error. Error handling is not particularly well defined. It will either set an error in the libguestfs handle or print an error on stderr, but there is no way for the caller to tell the difference.

Subdirectory *common/windows*

File common/windows/windows.c

This file implements `win:` Windows file path support in **guestfish**(1).

Function `common/windows/windows.c:is_windows`

```
int
is_windows (guestfs_h *g, const char *root)
```

Checks whether `root` is a Windows installation.

This relies on an already being done introspection.

Function `common/windows/windows.c:windows_path`

```
char *
windows_path (guestfs_h *g, const char *root, const char *path, int readonly)
```

Resolves `path` as possible Windows path according to `root`, giving a new path that can be used in libguestfs API calls.

Notes:

- `root` must be a Windows installation
- relies on an already being done introspection
- will unmount all the existing mount points and mount the Windows root (according to `readonly`)
- calls **exit** (3) on memory allocation failures

Subdirectory *daemon*

File daemon/command.c

This file contains a number of useful functions for running external commands and capturing their output.

Function `daemon/command.c:commandf`

```
int
commandf (char **stdoutput, char **stderror, unsigned flags,
          const char *name, ...)
```

Run a command. Optionally capture stdout and stderr as strings.

Returns `0` if the command ran successfully, or `−1` if there was any error.

For a description of the `flags` see `commandrvf`.

There is also a macro `command(out,err,name,...)` which calls `commandf` with `flags=0`.

Function `daemon/command.c:commandrf`

```
int
commandrf (char **stdoutput, char **stderror, unsigned flags,
           const char *name, ...)
```

Same as `command`, but we allow the status code from the subcommand to be non-zero, and return that status code.

We still return `−1` if there was some other error.

There is also a macro `commandr(out,err,name,...)` which calls `commandrf` with `flags=0`.

Function `daemon/command.c:commandvf`

```
int
commandvf (char **stdoutput, char **stderror, unsigned flags,
           char const *const *argv)
```

Same as `command`, but passing in an argv array.

There is also a macro `commandv(out,err,argv)` which calls `commandvf` with `flags=0`.

Function `daemon/command.c:commandrvf`

```
int
commandrvf (char **stdoutput, char **stderror, unsigned flags,
            char const* const *argv)
```

This is a more sane version of **system** (3) for running external commands. It uses fork/execvp, so we don't need to worry about quoting of parameters, and it allows us to capture any error messages in a buffer.

If `stdoutput` is not `NULL`, then `*stdoutput` will return the stdout of the command as a string.

If `stderror` is not `NULL`, then `*stderror` will return the stderr of the command. If there is a final \n character, it is removed so you can use the error string directly in a call to `reply_with_error`.

Flags are:

`COMMAND_FLAG_FOLD_STDOUT_ON_STDERR`
: For broken external commands that send error messages to stdout (hello, parted) but that don't have any useful stdout information, use this flag to capture the error messages in the `*stderror` buffer. If using this flag, you should pass `stdoutput=NULL` because nothing could ever be captured in that buffer.

`COMMAND_FLAG_CHROOT_COPY_FILE_TO_STDIN`
: For running external commands on chrooted files correctly (see https://bugzilla.redhat.com/579608) specifying this flag causes another process to be forked which chroots into sysroot and just copies the input file to stdin of the specified command. The file descriptor is ORed with the flags, and that file descriptor is always closed by this function. See *daemon/hexdump.c* for an example of usage.

There is also a macro `commandrv(out,err,argv)` which calls `commandrvf` with `flags=0`.

File daemon/device−name−translation.c

Function `daemon/device−name−translation.c:device_name_translation_init`

```
void
device_name_translation_init (void)
```

Cache daemon disk mapping.

When the daemon starts up, populate a cache with the contents of /dev/disk/by−path. It's easiest to use `ls −lv` here since the names are sorted awkwardly.

Function `daemon/device−name−translation.c:device_name_translation`

```
char *
device_name_translation (const char *device)
```

Perform device name translation.

Libguestfs defines a few standard formats for device names. (see also "BLOCK DEVICE NAMING" in **guestfs** (3) and "guestfs_canonical_device_name" in **guestfs** (3)). They are:

/dev/sdX[N]
/dev/hdX[N]
/dev/vdX[N]
: These mean the Nth partition on the Xth device. Because Linux no longer enumerates devices in the order they are passed to qemu, we must translate these by looking up the actual device using /dev/disk/by−path/

/dev/mdX
/dev/VG/LV
/dev/mapper/...
/dev/dm−N
: These are not translated here.

It returns a newly allocated string which the caller must free.

It returns `NULL` on error. **Note** it does *not* call `reply_with_*`.

We have to open the device and test for `ENXIO`, because the device nodes may exist in the appliance.

File daemon/guestfsd.c

This is the guestfs daemon which runs inside the guestfs appliance. This file handles start up and connecting back to the library.

Function `daemon/guestfsd.c:print_shell_quote`

```
static int
print_shell_quote (FILE *stream,
                   const struct printf_info *info ATTRIBUTE_UNUSED,
                   const void *const *args)
```

printf helper function so we can use `%Q` (“quoted”) and `%R` to print shell-quoted strings. See **guestfs−hacking** (1) for more details.

File daemon/internal.c

Internal functions that are not part of the public API.

File daemon/utils−c.c

Bindings for utility functions.

Note that functions called from OCaml code **must never** call any of the `reply*` functions.

File daemon/utils.c

Miscellaneous utility functions used by the daemon.

Function `daemon/utils.c:is_root_device_stat`

```
static int
is_root_device_stat (struct stat *statbuf)
```

Return true iff device is the root device (and therefore should be ignored from the point of view of user calls).

Function `daemon/utils.c:is_device_parameter`

```
int
is_device_parameter (const char *device)
```

Parameters marked as `Device`, `Dev_or_Path`, etc can be passed a block device name. This function tests if the parameter is a block device name.

It can also be used in daemon code to test if the string passed as a `Dev_or_Path` parameter is a device or path.

Function `daemon/utils.c:sysroot_path`

```
char *
sysroot_path (const char *path)
```

Turn `"/path"` into `"/sysroot/path"`.

Returns `NULL` on failure. The caller*must* check for this and call `reply_with_perror("malloc")`. The caller must also free the returned string.

See also the custom `%R` printf formatter which does shell quoting too.

Function `daemon/utils.c:sysroot_realpath`

```
char *
sysroot_realpath (const char *path)
```

Resolve path within sysroot, calling `sysroot_path` on the resolved path.

Returns `NULL` on failure. The caller*must* check for this and call `reply_with_perror("malloc")`. The caller must also free the returned string.

See also the custom `%R` printf formatter which does shell quoting too.

Function `daemon/utils.c:is_power_of_2`

```
int
is_power_of_2 (unsigned long v)
```

Returns true if `v` is a power of 2.

Uses the algorithm described at http://graphics.stanford.edu/˜seander/bithacks.html#DetermineIfPowerOf2

Function `daemon/utils.c:split_lines_sb`

```
struct stringsbuf
split_lines_sb (char *str)
```

Split an output string into a NULL-terminated list of lines, wrapped into a stringsbuf.

Typically this is used where we have run an external command which has printed out a list of things, and we want to return an actual list.

The corner cases here are quite tricky. Note in particular:

`''''`
: returns `[]`

`''\n''`
: returns `[""]`

`''a\nb''`
: returns `["a"; "b"]`

`''a\nb\n''`
: returns `["a"; "b"]`

`''a\nb\n\n''`
: returns `["a"; "b"; ""]`

The original string is written over and destroyed by this function (which is usually OK because it's the 'out' string from `command*()`). You can free the original string, because `add_string()` strdups the strings.

`argv` in the `struct stringsbuf` will be `NULL` in case of errors.

Function `daemon/utils.c:filter_list`

```
char **
filter_list (bool (*p) (const char *str), char **strs)
```

Filter a list of strings. Returns a newly allocated list of only the strings where `p (str) == true`.

Note it does not copy the strings, be careful not to double-free them.

Function `daemon/utils.c:trim`

```
void
trim (char *str)
```

Skip leading and trailing whitespace, updating the original string in-place.

Function `daemon/utils.c:parse_btrfsvol`

```
int
parse_btrfsvol (const char *desc_orig, mountable_t *mountable)
```

Parse the mountable descriptor for a btrfs subvolume. Don't call this directly; it is only used from the stubs.

A btrfs subvolume is given as:

```
btrfsvol:/dev/sda3/root
```

where */dev/sda3* is a block device containing a btrfs filesystem, and root is the name of a subvolume on it. This function is passed the string following `"btrfsvol:"`.

On success, `mountable->device` and `mountable->volume` must be freed by the caller.

Function `daemon/utils.c:mountable_to_string`

```
char *
mountable_to_string (const mountable_t *mountable)
```

Convert a `mountable_t` back to its string representation

This function can be used in an error path, so must not call `reply_with_error`.

Function `daemon/utils.c:prog_exists`

```
int
prog_exists (const char *prog)
```

Check program exists and is executable on `$PATH`.

Function `daemon/utils.c:random_name`

```
int
random_name (char *template)
```

Pass a template such as `"/sysroot/XXXXXXXX.XXX"`. This updates the template to contain a randomly named file. Any `'X'` characters after the final `'/'` in the template are replaced with random characters.

Notes: You should probably use an 8.3 path, so it's compatible with all filesystems including basic FAT. Also this only substitutes lowercase ASCII letters and numbers, again for compatibility with lowest common denominator filesystems.

This doesn't create a file or check whether or not the file exists (it would be extremely unlikely to exist as long as the RNG is working).

If there is an error, `−1` is returned.

Function `daemon/utils.c:udev_settle_file`

```
void
udev_settle_file (const char *file)
```

LVM and other commands aren't synchronous, especially when udev is involved. eg. You can create or remove some device, but the `/dev` device node won't appear until some time later. This means that you get an error if you run one command followed by another.

Use `udevadm settle` after certain commands, but don't be too fussed if it fails.

Function `daemon/utils.c:make_exclude_from_file`

```
char *
make_exclude_from_file (const char *function, char *const *excludes)
```

Turn list `excludes` into a temporary file, and return a string containing the temporary file name. Caller must unlink the file and free the string.

`function` is the function that invoked this helper, and it is used mainly for errors/debugging.

Function `daemon/utils.c:read_whole_file`

```
char *
read_whole_file (const char *filename, size_t *size_r)
```

Read whole file into dynamically allocated array. If there is an error, DON'T call reply_with_perror, just return NULL. Returns a`\0`−terminated string. `size_r` can be specified to get the size of the returned data.

File daemon/xattr.c

Function `daemon/xattr.c:split_attr_names`

```
static char **
split_attr_names (char *buf, size_t len)
```

listxattr(2) returns the string `"foo\0bar\0baz"` of length `len`. (The last string in the list is \0−terminated but the \0 is not included in `len`).

This function splits it into a regular list of strings.

Note that the returned list contains pointers to the original strings in `buf` so be careful that you do not double-free them.

Subdirectory *fish*

File fish/alloc.c

This file implements the guestfish `alloc` and `sparse` commands.

Function `fish/alloc.c:alloc_disk`

```
int
alloc_disk (const char *filename, const char *size_str, int add, int sparse)
```

This is the underlying allocation function. It's called from a few other places in guestfish.

File fish/copy.c

This file implements the guestfish commands `copy−in` and `copy−out`.

File fish/destpaths.c

The file handles tab-completion of filesystem paths in guestfish.

File fish/display.c

The file implements the guestfish `display` command, for displaying graphical files (icons, images) in disk images.

File fish/echo.c

The file implements the guestfish `echo` command.

File fish/edit.c

guestfish `edit` command, suggested by Ján Ondrej.

File fish/events.c

This file implements the guestfish event-related commands, `event`, `delete−event` and `list−events`.

File fish/fish.c

guestfish, the guest filesystem shell. This file contains the main loop and utilities.

Function `fish/fish.c:parse_command_line`

```
static struct parsed_command
parse_command_line (char *buf, int *exit_on_error_rtn)
```

Parse a command string, splitting at whitespace, handling `'!'`, `'#'` etc. This destructively updates `buf`.

`exit_on_error_rtn` is used to pass in the global `exit_on_error` setting and to return the local setting (eg. if the command begins with `'−'`).

Returns in `parsed_command.status`:

`1` got a guestfish command (returned in `cmd_rtn`/`argv_rtn`/`pipe_rtn`)

`0` no guestfish command, but otherwise OK

`−1` an error

Function `fish/fish.c:parse_quoted_string`

```
static ssize_t
parse_quoted_string (char *p)
```

Parse double-quoted strings, replacing backslash escape sequences with the true character. Since the string is returned in place, the escapes must make the string shorter.

Function `fish/fish.c:execute_and_inline`

```
static int
execute_and_inline (const char *cmd, int global_exit_on_error)
```

Used to handle `<!` (execute command and inline result).

Function `fish/fish.c:issue_command`

```
int
issue_command (const char *cmd, char *argv[], const char *pipecmd,
               int rc_exit_on_error_flag)
```

Run a command.

`rc_exit_on_error_flag` is the `exit_on_error` flag that we pass to the remote server (when issuing `--`*remote* commands). It does not cause `issue_command` itself to exit on error.

Function `fish/fish.c:extended_help_message`

```
void
extended_help_message (void)
```

Print an extended help message when the user types in an unknown command for the first command issued. A common case is the user doing:

```
 guestfish disk.img
```

expecting guestfish to open *disk.img* (in fact, this tried to run a non-existent command `disk.img`).

Function `fish/fish.c:error_cb`

```
static void
error_cb (guestfs_h *g, void *data, const char *msg)
```

Error callback. This replaces the standard libguestfs error handler.

Function `fish/fish.c:free_n_strings`

```
static void
free_n_strings (char **str, size_t len)
```

Free strings from a non-NULL terminated `char**`.

Function `fish/fish.c:decode_ps1`

```
static char *
decode_ps1 (const char *str)
```

Decode `str` into the final printable prompt string.

Function `fish/fish.c:win_prefix`

```
char *
win_prefix (const char *path)
```

Resolve the special `win:...` form for Windows-specific paths. The generated code calls this for all device or path arguments.

The function returns a newly allocated string, and the caller must free this string; else display an error and return `NULL`.

Function `fish/fish.c:file_in`

```
char *
file_in (const char *arg)
```

Resolve the special `FileIn` paths (`-` or `-<<END` or filename).

The caller (*fish/cmds.c*) will call `free_file_in` after the command has run which should clean up resources.

Function `fish/fish.c:file_out`

```
char *
file_out (const char *arg)
```

Resolve the special `FileOut` paths (`-` or filename).

The caller (*fish/cmds.c*) will call `free(str)` after the command has run.

Function `fish/fish.c:progress_callback`

```
void
progress_callback (guestfs_h *g, void *data,
                   uint64_t event, int event_handle, int flags,
                   const char *buf, size_t buf_len,
                   const uint64_t *array, size_t array_len)
```

Callback which displays a progress bar.

File fish/glob.c

This file implements the guestfish `glob` command.

Function `fish/glob.c:expand_devicename`

```
static char **
expand_devicename (guestfs_h *g, const char *device)
```

Glob-expand device patterns, such as `/dev/sd*` (https://bugzilla.redhat.com/635971).

There is no `guestfs_glob_expand_device` function because the equivalent can be implemented using functions like `guestfs_list_devices`.

It's not immediately clear what it means to expand a pattern like `/dev/sd*`. Should that include device name translation? Should the result include partitions as well as devices?

Should `"/dev/"` + `"*"` return every possible device and filesystem? How about VGs? LVs?

To solve this what we do is build up a list of every device, partition, etc., then glob against that list.

Notes for future work (XXX):

- This doesn't handle device name translation. It wouldn't be too hard to add.
- Could have an API function for returning all device-like things.

Function `fish/glob.c:add_strings_matching`

```
static int
add_strings_matching (char **pp, const char *glob,
                      char ***ret, size_t *size_r)
```

Using POSIX **fnmatch**(3), find strings in the list `pp` which match pattern `glob`. Add strings which match to the `ret` array. `*size_r` is the current size of the `ret` array, which is updated with the new size.

Function `fish/glob.c:single_element_list`

```
static char **
single_element_list (const char *element)
```

Return a single element list containing `element`.

File fish/help.c

The file implements the guestfish `help` command.

Function `fish/help.c:display_help`

```
int
display_help (const char *cmd, size_t argc, char *argv[])
```

The `help` command.

This used to just list all commands, but that's not very useful. Instead display some useful context-sensitive help. This could be improved if we knew how many drives had been added already, and whether anything was mounted.

File fish/hexedit.c

This file implements the guestfish `hexedit` command.

File fish/lcd.c

Function `fish/lcd.c:run_lcd`

```
int
run_lcd (const char *cmd, size_t argc, char *argv[])
```

guestfish `lcd` command (similar to the `lcd` command in BSD ftp).

File fish/man.c

Function `fish/man.c:run_man`

```
int
run_man (const char *cmd, size_t argc, char *argv[])
```

guestfish `man` command

File fish/more.c

This file implements the guestfish `more` command.

File fish/prep.c

This file implements the guestfish *−N* option for creating pre-prepared disk layouts.

File fish/rc.c

This file implements guestfish remote (command) support.

Function `fish/rc.c:rc_listen`

```
void
rc_listen (void)
```

The remote control server (ie. `guestfish --listen`).

Function `fish/rc.c:rc_remote`

```
int
rc_remote (int pid, const char *cmd, size_t argc, char *argv[],
           int exit_on_error)
```

The remote control client (ie. `guestfish --remote`).

File fish/reopen.c

This file implements the guestfish `reopen` command.

File fish/setenv.c

This file implements the guestfish `setenv` and `unsetenv` commands.

File fish/supported.c

This file implements the guestfish `supported` command.

File fish/tilde.c

This file implements tilde (˜) expansion of home directories in **guestfish** (1).

Function `fish/tilde.c:try_tilde_expansion`

```
char *
try_tilde_expansion (char *str)
```

This is called from the script loop if we find a candidate for `˜username` (tilde-expansion).

Function `fish/tilde.c:expand_home`

```
static char *
expand_home (char *orig, const char *append)
```

Return `$HOME` + append string.

Function `fish/tilde.c:find_home_for_username`

```
static const char *
find_home_for_username (const char *username, size_t ulen)
```

Lookup `username` (of length `ulen`), return home directory if found, or `NULL` if not found.

File fish/time.c

This file implements the guestfish `time` command.

Subdirectory *python*

File python/handle.c

This file contains a small number of functions that are written by hand. The majority of the bindings are generated (see *python/actions−*.c*).

SEE ALSO

guestfs (3), **guestfs−building** (1), **guestfs−examples** (3), **guestfs−internals** (1), **guestfs−performance** (1), **guestfs−release−notes** (1), **guestfs−testing** (1), **libguestfs−test−tool** (1), **libguestfs−make−fixed−appliance** (1), http://libguestfs.org/.

AUTHORS

Richard W.M. Jones (`rjones at redhat dot com`)

COPYRIGHT

LICENSE

BUGS

To get a list of bugs against libguestfs, use this link: https://bugzilla.redhat.com/buglist.cgi?component=libguestfs&product=Virtualization+Tools

To report a new bug against libguestfs, use this link: https://bugzilla.redhat.com/enter_bug.cgi?component=libguestfs&product=Virtualization+Tools

When reporting a bug, please supply:

- The version of libguestfs.
- Where you got libguestfs (eg. which Linux distro, compiled from source, etc)
- Describe the bug accurately and give a way to reproduce it.
- Run **libguestfs−test−tool** (1) and paste the **complete, unedited** output into the bug report.