ioctl based interfaces

ioctl() is the most common way for applications to interface with device drivers. It is flexible and easily extended by adding new commands and can be passed through character devices, block devices as well as sockets and other special file descriptors.

However, it is also very easy to get ioctl command definitions wrong, and hard to fix them later without breaking existing applications, so this documentation tries to help developers get it right.

Command number definitions

The command number, or request number, is the second argument passed to the ioctl system call. While this can be any 32-bit number that uniquely identifies an action for a particular driver, there are a number of conventions around defining them.

`include/uapi/asm-generic/ioctl.h` provides four macros for defining ioctl commands that follow modern conventions: `_IO`, `_IOR`, `_IOW`, and `_IOWR`. These should be used for all new commands, with the correct parameters:

_IO/_IOR/_IOW/_IOWR
: The macro name specifies how the argument will be used. It may be a pointer to data to be passed into the kernel (_IOW), out of the kernel (_IOR), or both (_IOWR). _IO can indicate either commands with no argument or those passing an integer value instead of a pointer. It is recommended to only use _IO for commands without arguments, and use pointers for passing data.

type
: An 8-bit number, often a character literal, specific to a subsystem or driver, and listed in Documentation/userspace-api/ioctl/ioctl-number.rst

nr
: An 8-bit number identifying the specific command, unique for a give value of 'type'

data_type
: The name of the data type pointed to by the argument, the command number encodes the `sizeof(data_type)` value in a 13-bit or 14-bit integer, leading to a limit of 8191 bytes for the maximum size of the argument. Note: do not pass sizeof(data_type) type into _IOR/_IOW/IOWR, as that will lead to encoding sizeof(sizeof(data_type)), i.e. sizeof(size_t). _IO does not have a data_type parameter.

Interface versions

Some subsystems use version numbers in data structures to overload commands with different interpretations of the argument.

This is generally a bad idea, since changes to existing commands tend to break existing applications.

A better approach is to add a new ioctl command with a new number. The old command still needs to be implemented in the kernel for compatibility, but this can be a wrapper around the new implementation.

Return code

ioctl commands can return negative error codes as documented in errno(3); these get turned into errno values in user space. On success, the return code should be zero. It is also possible but not recommended to return a positive 'long' value.

When the ioctl callback is called with an unknown command number, the handler returns either -ENOTTY or -ENOIOCTLCMD, which also results in -ENOTTY being returned from the system call. Some subsystems return -ENOSYS or -EINVAL here for historic reasons, but this is wrong.

Prior to Linux 5.5, compat_ioctl handlers were required to return -ENOIOCTLCMD in order to use the fallback conversion into native commands. As all subsystems are now responsible for handling compat mode themselves, this is no longer needed, but it may be important to consider when backporting bug fixes to older kernels.

Timestamps

Traditionally, timestamps and timeout values are passed as `struct timespec` or `struct timeval`, but these are problematic because of incompatible definitions of these structures in user space after the move to 64-bit time_t.

The `struct __kernel_timespec` type can be used instead to be embedded in other data structures when separate second/nanosecond values are desired, or passed to user space directly. This is still not ideal though, as the structure matches neither the kernel's timespec64 nor the user space timespec exactly. The get_timespec64() and put_timespec64() helper functions can be used to ensure that the layout remains compatible with user space and the padding is treated correctly.

As it is cheap to convert seconds to nanoseconds, but the opposite requires an expensive 64-bit division, a simple __u64 nanosecond value can be simpler and more efficient.

Timeout values and timestamps should ideally use CLOCK_MONOTONIC time, as returned by ktime_get_ns() or ktime_get_ts64(). Unlike CLOCK_REALTIME, this makes the timestamps immune from jumping backwards or forwards due to

leap second adjustments and clock_settime() calls.

ktime_get_real_ns() can be used for CLOCK_REALTIME timestamps that need to be persistent across a reboot or between multiple machines.

32-bit compat mode

In order to support 32-bit user space running on a 64-bit machine, each subsystem or driver that implements an ioctl callback handler must also implement the corresponding compat_ioctl handler.

As long as all the rules for data structures are followed, this is as easy as setting the .compat_ioctl pointer to a helper function such as compat_ptr_ioctl() or blkdev_compat_ptr_ioctl().

compat_ptr()

On the s390 architecture, 31-bit user space has ambiguous representations for data pointers, with the upper bit being ignored. When running such a process in compat mode, the compat_ptr() helper must be used to clear the upper bit of a compat_uptr_t and turn it into a valid 64-bit pointer. On other architectures, this macro only performs a cast to a `void __user *` pointer.

In an compat_ioctl() callback, the last argument is an unsigned long, which can be interpreted as either a pointer or a scalar depending on the command. If it is a scalar, then compat_ptr() must not be used, to ensure that the 64-bit kernel behaves the same way as a 32-bit kernel for arguments with the upper bit set.

The compat_ptr_ioctl() helper can be used in place of a custom compat_ioctl file operation for drivers that only take arguments that are pointers to compatible data structures.

Structure layout

Compatible data structures have the same layout on all architectures, avoiding all problematic members:

- `long` and `unsigned long` are the size of a register, so they can be either 32-bit or 64-bit wide and cannot be used in portable data structures. Fixed-length replacements are `__s32`, `__u32`, `__s64` and `__u64`.
- Pointers have the same problem, in addition to requiring the use of compat_ptr(). The best workaround is to use `__u64` in place of pointers, which requires a cast to `uintptr_t` in user space, and the use of u64_to_user_ptr() in the kernel to convert it back into a user pointer.
- On the x86-32 (i386) architecture, the alignment of 64-bit variables is only 32-bit, but they are naturally aligned on most other architectures including x86-64. This means a structure like:

  ```
  struct foo {
      __u32 a;
      __u64 b;
      __u32 c;
  };
  ```

  has four bytes of padding between a and b on x86-64, plus another four bytes of padding at the end, but no padding on i386, and it needs a compat_ioctl conversion handler to translate between the two formats.

  To avoid this problem, all structures should have their members naturally aligned, or explicit reserved fields added in place of the implicit padding. The `pahole` tool can be used for checking the alignment.
- On ARM OABI user space, structures are padded to multiples of 32-bit, making some structs incompatible with modern EABI kernels if they do not end on a 32-bit boundary.
- On the m68k architecture, struct members are not guaranteed to have an alignment greater than 16-bit, which is a problem when relying on implicit padding.
- Bitfields and enums generally work as one would expect them to, but some properties of them are implementation-defined, so it is better to avoid them completely in ioctl interfaces.
- `char` members can be either signed or unsigned, depending on the architecture, so the __u8 and __s8 types should be used for 8-bit integer values, though char arrays are clearer for fixed-length strings.

Information leaks

Uninitialized data must not be copied back to user space, as this can cause an information leak, which can be used to defeat kernel address space layout randomization (KASLR), helping in an attack.

For this reason (and for compat support) it is best to avoid any implicit padding in data structures. Where there is implicit padding in an existing structure, kernel drivers must be careful to fully initialize an instance of the structure before copying it to user space. This is usually done by calling memset() before assigning to individual members.

Subsystem abstractions

While some device drivers implement their own ioctl function, most subsystems implement the same command for multiple drivers. Ideally the subsystem has an .ioctl() handler that copies the arguments from and to user space, passing them into subsystem specific callback functions through normal kernel pointers.

This helps in various ways:

- Applications written for one driver are more likely to work for another one in the same subsystem if there are no subtle differences in the user space ABI.
- The complexity of user space access and data structure layout is done in one place, reducing the potential for implementation bugs.
- It is more likely to be reviewed by experienced developers that can spot problems in the interface when the ioctl is shared between multiple drivers than when it is only used in a single driver.

Alternatives to ioctl

There are many cases in which ioctl is not the best solution for a problem. Alternatives include:

- System calls are a better choice for a system-wide feature that is not tied to a physical device or constrained by the file system permissions of a character device node
- netlink is the preferred way of configuring any network related objects through sockets.
- debugfs is used for ad-hoc interfaces for debugging functionality that does not need to be exposed as a stable interface to applications.
- sysfs is a good way to expose the state of an in-kernel object that is not tied to a file descriptor.
- configfs can be used for more complex configuration than sysfs
- A custom file system can provide extra flexibility with a simple user interface but adds a lot of complexity to the implementation.