Improving the modularity of NetBSD’s compat code

Paul Goyette
pgoyette@netbsd.org
Improving the Modularity of NetBSD's COMPAT code

- Motivation
- Issues
- Approach/Solution
- Implementation and Status
- Recognition
Motivation

- NetBSD prides itself on maintaining backwards compatibility, all the way back to version 0.9
- NetBSD also provides for modular kernel components, loading functionality as needed
Motivation ("I got bitten, and have the scars as proof!")

- I personally run a stripped-down kernel, with as few as possible built-in modules
  - Some changes to sys/net/rtsock.c were made, and built-in compat_70 builds were accommodated via #ifdef, but
  - No provision was made for calling the compat_70 code loaded as a module

- So even if I loaded the compat module my system failed to run
Issues

- Building of compat module required careful selection of options, resulting in #ifdef hell
- The resulting compat module was monolithic, built with a single predefined set of options
- There was no reliable mechanism to prevent module code from being unloaded while executing
Kernel Options

- Lots of kernel configuration options available, controlling whether or not certain code is included, including calls to compat code.
- By default, we only include compat for NetBSD version 1.5 and above.
- Modules are built with their own set of options which might differ from those of the kernel.
Kernel Options (cont.)

- There’s no clear way to determine if optional code is included (e.g. a modular driver cannot tell if its compat_xx ioctl() routines exist and thus need to be called)

- Some code (notably, net/rtsock.c) assumes that compat functionality is always built-in to the kernel
Monolithic compat module

- Standard builds provide only a single module to contain all selected compat options
- Contents are pre-determined at build time
- No provision for incrementally loading additional compat code (for an earlier NetBSD version) if needed, without first unloading the current module
Preventing modunload() of active modules

- Device driver modules can check for existing units (or instances) of their device
- Buffer-queue strategy modules have a refcount
- Active syscalls “know” that they’re active, and refuse to be dis-established
Preventing modunload() of active modules (cont.)

- No equivalent mechanisms exist for a compat module to determine if it can be unloaded
Approach/Solution

- Define a “module hook” mechanism for callers to use when invoking optional code
  - Call through a function pointer in all cases
  - No #ifdef

- Split the monolithic compat module into many version-specific modules
The module_hook

• Optional module code “sets the hook” when it is loaded

• Caller defines a default action (or value) if the hook is not set
  – Frequently use ENOSYS
  – Hook ioctl code can return EPASSTHROUGH if it does not handle
The module_hook (cont.)

- Hooks are protected from being unloaded while executing
  - Use passive serialization to prevent new acquirers of the localcount
  - Use localcount to track active references (calls)
  - Drain the localcount before unsetting the hook
The module_hook (cont.)

#define MODULE_HOOK(hook, type, args)                           
extern struct hook##_t {                                      
  kmutex_t                mtx;                            
  kcondvar_t              cv;                             
  struct localcount       lc;                             
  pserialize_t            psz;                            
  bool                    hooked;                         
  type                    (*f)args;                       
} hook __cacheline_aligned;
The module_hook (cont.)

• Each hook’s prototype can be unique, so they are defined using macros.

```
#define MODULE_HOOK(hook, type, args) ...
#define MODULE_HOOK_SET(hook, waitchan, func) ...
#define MODULE_HOOK_UNSET(hook) ...
#define MODULE_HOOK_CALL(hook, args, default, retval) ...
#define MODULE_HOOK_CALL_VOID(hook, args, default) ...
```
The module_hook (cont.)

- Invoking the optional code – before

```c
... default:
    if (*((compat_ccd_ioctl_60)(0, cmd, NULL, 0, NULL, NULL) == 0)
        make = 1;
    else
        Make = 0;
...
The module_hook (cont.)

- Invoking the optional code - after

```c
default:
    MODULE_HOOK_CALL(ccd_ioctl_60_hook,
                      (0, cmd, NULL, 0, NULL, NULL), enosys(), hook);
    if (hook == 0)
        make = 1;
    else
        make = 0;
```
The module_hook (cont.)

- Setting and unsetting the hook

```c
void ccd_60_init(void)
{
    MODULE_HOOK_SET(ccd_ioctl_60_hook, "ccd_60",
                    compat_60_ccdioctl);
}

void ccd_60_fini(void)
{
    MODULE_HOOK_UNSET(ccd_ioctl_60_hook);
}
```
• The hooks are defined as globals

sys/sys/compat_stub.h:

...  
MODULE_HOOK(ccd_ioctl_60_hook, int, (dev_t, u_long, void *, int,  
          struct lwp *, int (*f)(dev_t, u_long, void *, int, struct lwp *))})  
...

sys/kern/compat_stub.c

...  
struct ccd_ioctl_60_hook_t ccd_ioctl_60_hook;  
...
Splitting the Monolithic Module

- The second major change was to separate the single monolithic compat module into many individual version-specific compat modules.
  - Each `compat_xx` module depends on `compat_xx_next`.
  - The `kern/syscalls.master` file was updated to indicate which specific module provides the functionality (used for auto-loading the `compat_xx` modules).
Splitting the Monolithic Module (cont.)

- The sheer number of versions involved caused us to exceed some compile-time limits
  - Maximum number of per-module dependencies
    - #define MAXMODDEPS 10
  - Maximum recursion depth for auto-loading module dependencies
    - #define MODULE_MAX_DEPTH 6
Status

- Merged to HEAD in mid-January, 2019
- Will be included in forthcoming NetBSD-9.0
Status (cont.)

- Mostly complete
  - Compile-time restrictions removed
    - Had to introduce some additional compat code for modstat(8)!
  - Smaller version-specific modules created, all the way back to NetBSD-0.9
  - Most compat-code calls converted to use the hooks
  - Similar changes made for compat_netbsd32
Still a few areas needing more work

- Various machine-dependent bits and pieces
- Build-system infrastructure needs work for properly building modules for XEN environment
- dev/gpio and dev/wscons/wsmux still have some old-style compat calls
- Need a full audit to ensure we got everything
Possible Improvement

• The hook definition mechanism may be excessively complex, with many “touch points”
  - Define and allocate hooks in kern_stub.[ch]
  - SET and UNSET the hook in implementation
  - CALL the hook in appropriate places
Possible Improvement (cont.)

• Perhaps some sort of non-precedural definition mechanism would help?

• Would an awk or sed script help for handling the details?
Possible Improvement (cont.)

- Something like this, perhaps?

```c
#include <compat/50_iflist_addr>

int compat_50_iflist_addr(struct rt_walkarg *, struct ifaddr *,
                           struct rt_addrinfo *);
```

```c
int compat_50_iflist_addr(struct rt_walkarg *w, struct ifaddr *ifa,
                           struct rt_addrinfo *info)
{
    /* ... */
}
```
Recognition

• I did most of the work, but would not have succeeded without some major assistance!
  – Taylor Campbell provided the basis for the module_hook mechanism, and
  – Christos Zoulas provided major encouragement as well as help with some especially tricky parts (like sys/net/rtsock.c)
Additionally, the entire NetBSD developer and user communities contributed by identifying and fixing various issues that arose post-merge.
Questions?