

android

Symbol Namespaces

Martijn Coenen <maco@android.com>



The problem

- There are >30000 EXPORT_SYMBOL(_GPL etc) symbols
 - All in a global namespace, visible to all modules
- Hard to manage the export surface
 - Should driver X even use these symbols?
 - Can be hard to catch in code review
- Hard to reason about the export surface
 - What subsystem does this symbol belong to?

How it affects Android

- We're moving to a model with a single generic arch image
 - We'll load many device-specific modules, from different parties
- No stable API means potential breakages
- We want to significantly reduce the chances of such breakages

Different categories of exported symbols

- Symbols actually meant for drivers (but only for some?)
- Symbols exported only because core functionality is split over multiple modules
- Symbols really meant only for internal (in-tree) use

Symbol namespaces

- Goal 1: Make the API surface more clear
 - Allow to differentiate different classes of exports
- Goal 2: Reduce the *global* API surface

Reducing the size of the exported API

What is the exported API?

```
// Regular C internal linkage (not visible to LKMs)  
static void usb_stor_scan_dwork(struct work_struct *work);
```

```
// Regular C global linkage (not visible to LKMs!)  
void usb_stor_disconnect( struct usb_interface *intf);
```

Exporting a symbol for LKM use



```
void usb_stor_disconnect( struct usb_interface *intf);  
  
// Export for LKM use as well  
  
EXPORT_SYMBOL(usb_stor_disconnect);
```

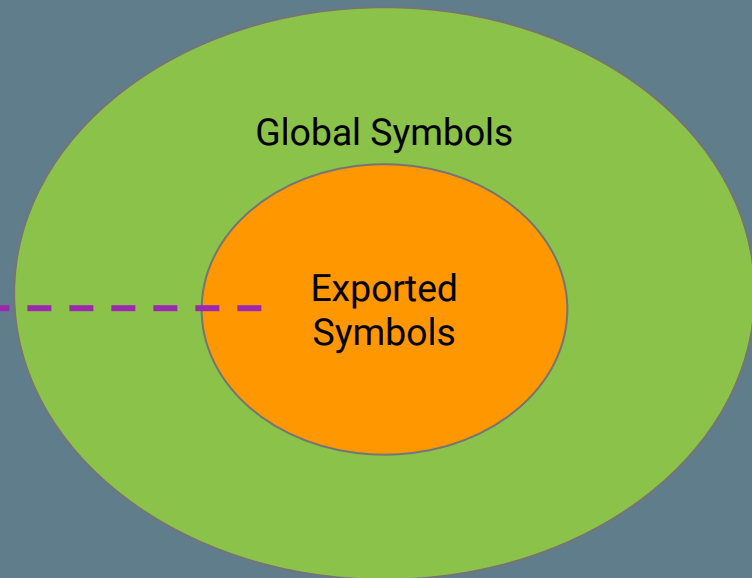
The exported API

Exported API



Visible by:

-  Built-in code only
-  Built-in + modules



Exporting a symbol to a namespace

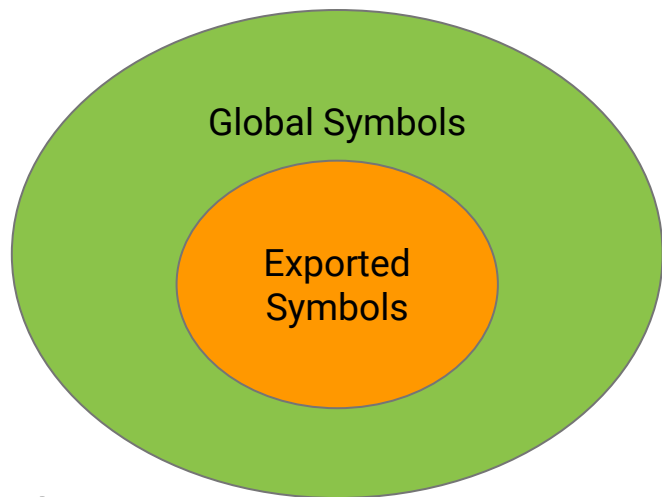
```
void usb_stor_disconnect( struct usb_interface *intf);  
  
// Only available for LKMs importing USB_STORAGE ns  
  
EXPORT_SYMBOL_NS(usb_stor_disconnect, USB_STORAGE);
```

Importing a namespace to a module




```
MODULE_IMPORT_NS (USB_STORAGE) ;
```

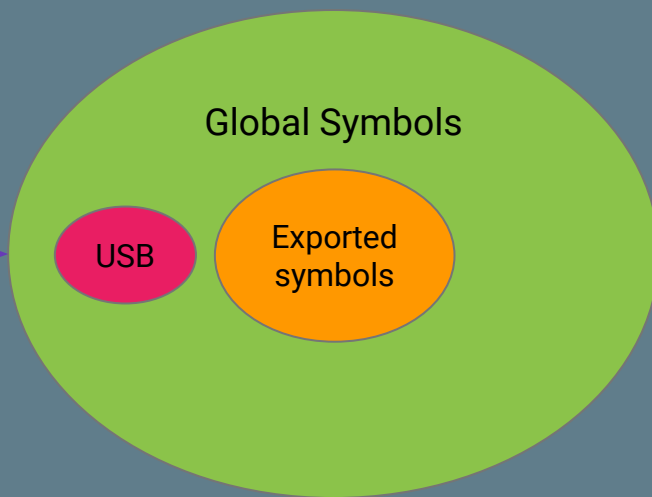
In the resulting API...

- # of default exported symbols is smaller
- APIs are more cleanly defined



Visible by:

-  Built-in code only
-  Built-in + modules
-  Built-in + modules importing USB_STORAGE



Automation

- Requires subsystem maintainer to think about where a symbol belongs
- Requires drivers using subsystems to explicitly import them
- Patchset contains a script that calculates dependencies and auto-adds import statements

Upstream status

- Patchset is really small (~300 LOC)
- v1 in series sent in July
- High-level feedback so far:
 - Auto-export to namespace based on `KBUILD_MODNAME`
 - Auto-import namespace through Makefile
- v2 next week :-)

Discussion

Symbol Namespace Implementation

Regular exported symbols

- Each symbol is represented by `struct kernel_symbol`
 - Placed in special `__ksymtab` sections
- Symbol name is `'__ksymtab_' + symbol name`
 - `__ksymtab_usb_stor_suspend`
- `modpost` and the kernel module loader use these sections
 - `modpost` verifies unresolved symbols are exported by others
 - Kernel loader resolves symbols at runtime and fixes up

Symbol namespaces implementation

- Only ~300 LOC
- Add namespace member to `struct kernel_symbol`
- Also encode namespace in symbol name with `'.'` separator
 - `__ksymtab_usb_stor_suspend.USB_STORAGE`
- Place imports in a `__knsimports` section
- Modpost warns for use of 'unimported symbols' at build time
- Kernel loader warns at runtime

Upstream feedback

- Feedback so far:
 - Use modinfo tag instead of section for imports
 - Auto-export to namespace based on KBUILD_MODNAME
 - Auto-import namespace through Makefile

Discussion

Points for discussion

- Warning vs errors
- Granularity of exports
- Memory usage

THANK YOU