Symbol Namespaces

The following document describes how to use Symbol Namespaces to structure the export surface of in-kernel symbols exported through the family of EXPORT_SYMBOL() macros.

1. Introduction

Symbol Namespaces have been introduced as a means to structure the export surface of the in-kernel API. It allows subsystem maintainers to partition their exported symbols into separate namespaces. That is useful for documentation purposes (think of the SUBSYSTEM_DEBUG namespace) as well as for limiting the availability of a set of symbols for use in other parts of the kernel. As of today, modules that make use of symbols exported into namespaces, are required to import the namespace. Otherwise the kernel will, depending on its configuration, reject loading the module or warn about a missing import.

2. How to define Symbol Namespaces

Symbols can be exported into namespace using different methods. All of them are changing the way EXPORT_SYMBOL and friends are instrumented to create ksymtab entries.

2.1 Using the EXPORT_SYMBOL macros

In addition to the macros EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL(), that allow exporting of kernel symbols to the kernel symbol table, variants of these are available to export symbols into a certain namespace: EXPORT_SYMBOL_NS() and EXPORT_SYMBOL_NS_GPL(). They take one additional argument: the namespace. Please note that due to macro expansion that argument needs to be a preprocessor symbol. E.g. to export the symbol `usb_stor_suspend` into the namespace `USB_STORAGE`, use:

```
EXPORT_SYMBOL_NS(usb_stor_suspend, USB_STORAGE);
```

The corresponding ksymtab entry struct `kernel_symbol` will have the member `namespace` set accordingly. A symbol that is exported without a namespace will refer to `NULL`. There is no default namespace if none is defined. `modpost` and kernel/module.c make use the namespace at build time or module load time, respectively.

2.2 Using the DEFAULT_SYMBOL_NAMESPACE define

Defining namespaces for all symbols of a subsystem can be very verbose and may become hard to maintain. Therefore a default define (DEFAULT_SYMBOL_NAMESPACE) is been provided, that, if set, will become the default for all EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL() macro expansions that do not specify a namespace.

There are multiple ways of specifying this define and it depends on the subsystem and the maintainer's preference, which one to use. The first option is to define the default namespace in the `Makefile` of the subsystem. E.g. to export all symbols defined in usb-common into the namespace USB_COMMON, add a line like this to drivers/usb/common/Makefile:

```
ccflags-y += -DDEFAULT_SYMBOL_NAMESPACE=USB_COMMON
```

That will affect all EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL() statements. A symbol exported with EXPORT_SYMBOL_NS() while this definition is present, will still be exported into the namespace that is passed as the namespace argument as this argument has preference over a default symbol namespace.

A second option to define the default namespace is directly in the compilation unit as preprocessor statement. The above example would then read:

```
#undef  DEFAULT_SYMBOL_NAMESPACE
#define DEFAULT_SYMBOL_NAMESPACE USB_COMMON
```

within the corresponding compilation unit before any EXPORT_SYMBOL macro is used.

3. How to use Symbols exported in Namespaces

In order to use symbols that are exported into namespaces, kernel modules need to explicitly import these namespaces. Otherwise the kernel might reject to load the module. The module code is required to use the macro MODULE_IMPORT_NS for the namespaces it uses symbols from. E.g. a module using the usb_stor_suspend symbol from above, needs to import the namespace USB_STORAGE using a statement like:

```
MODULE_IMPORT_NS(USB_STORAGE);
```

This will create a `modinfo` tag in the module for each imported namespace. This has the side effect, that the imported namespaces of a module can be inspected with modinfo:

```
$ modinfo drivers/usb/storage/ums-karma.ko
[...]
import_ns:      USB_STORAGE
[...]
```

It is advisable to add the MODULE_IMPORT_NS() statement close to other module metadata definitions like MODULE_AUTHOR() or MODULE_LICENSE(). Refer to section 5. for a way to create missing import statements automatically.

4. Loading Modules that use namespaced Symbols

At module loading time (e.g. `insmod`), the kernel will check each symbol referenced from the module for its availability and whether the namespace it might be exported to has been imported by the module. The default behaviour of the kernel is to reject loading modules that don't specify sufficient imports. An error will be logged and loading will be failed with EINVAL. In order to allow loading of modules that don't satisfy this precondition, a configuration option is available: Setting MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS=y will enable loading regardless, but will emit a warning.

5. Automatically creating MODULE_IMPORT_NS statements

Missing namespaces imports can easily be detected at build time. In fact, modpost will emit a warning if a module uses a symbol from a namespace without importing it. MODULE_IMPORT_NS() statements will usually be added at a definite location (along with other module meta data). To make the life of module authors (and subsystem maintainers) easier, a script and make target is available to fixup missing imports. Fixing missing imports can be done with:

```
$ make nsdeps
```

A typical scenario for module authors would be:

```
- write code that depends on a symbol from a not imported namespace
- ``make``
- notice the warning of modpost telling about a missing import
- run ``make nsdeps`` to add the import to the correct code location
```

For subsystem maintainers introducing a namespace, the steps are very similar. Again, `make nsdeps` will eventually add the missing namespace imports for in-tree modules:

```
- move or add symbols to a namespace (e.g. with EXPORT_SYMBOL_NS())
- ``make`` (preferably with an allmodconfig to cover all in-kernel
  modules)
- notice the warning of modpost telling about a missing import
- run ``make nsdeps`` to add the import to the correct code location
```

You can also run nsdeps for external module builds. A typical usage is:

```
$ make -C <path_to_kernel_src> M=$PWD nsdeps
```