**The Kernel Symbol Table**

*insmod* resolves undefined symbols against the kernel's public symbols.

These include both functions and variables.

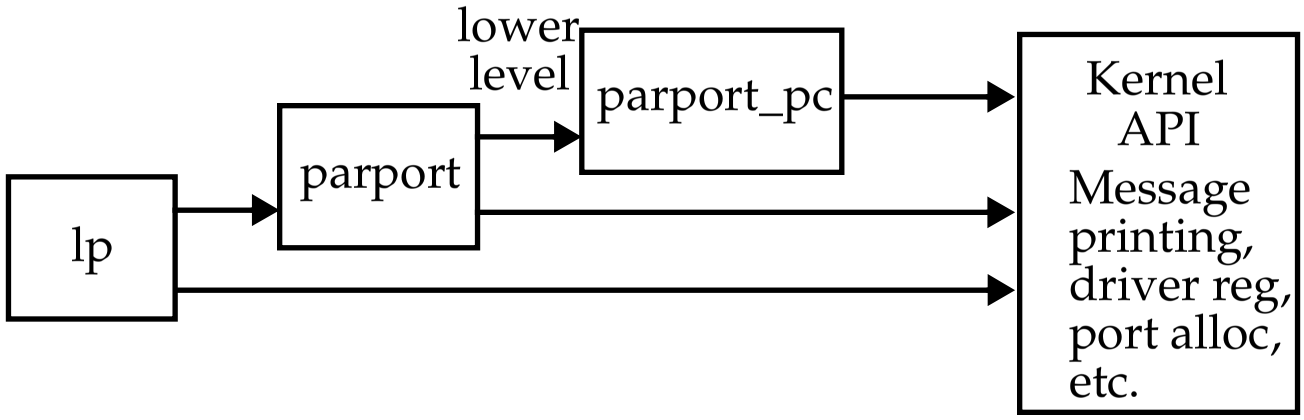
The symbol table lives in */proc/ksyms*.

Global symbols in your module are added here.

Use *ksyms* to output them directly.

Module loading order can be important, particularly if they are stacked (dependent on the symbols defined in other modules).

The parallel port subsystem is composed of a set of stacked modules:



**The Kernel Symbol Table**

Layered modularization can simplify development.

Lower layers export symbols for use in higher level modules.

Kernel header files provide a convenient way to manage the visibility of your symbols (you usually don't want all non-static symbols exported).

If no symbols are to be exported, add to *init_module* the line:

```
EXPORT_NO_SYMBOLS;
```

To export a **subset** of your symbols, add the following line *before* including *module.h*

```
#define EXPORT_SYMTAB
```

or define it using the *-D* compiler flag in the makefile.

If EXPORT_SYMTAB is defined, then outside any function add:

```
EXPORT_SYMBOL(name); /* or */
EXPORT_SYMBOL_NOVERS(name); /* no version info */
```

The symbol *name* is exported.

**Initialization and Shutdown**

*init_module* registers any **facility** offered by the module.

This is performed by passing several arguments to a kernel function:

- A pointer to a data structure, which embeds pointers to module functions.
- The name of the facility being added.

If any errors occur during registration, e.g., out of memory, you must undo any registration performed before the error.

Otherwise, the kernel is left in an unstable state.

```
int init_module(void)
   {
   int err;  /* Errors defined in <linux/errno.h> */
   err = register_this(ptr1, “skull”);
   if (err) goto fail_this;
   ...
   return 0;
   ...
   fail_this(unregister_this(ptr1), “skull”);
   return err;}
```

**Initialization and Shutdown**

The cleanup module also calls these unregister functions:

```
void cleanup_module(void)
   {
   ...
   unregister_this(ptr1, “skull”);
   return;
   }
```

Usage Counts:

The system keeps usage counts on modules to determine if a module can be safely removed.

A module cannot be removed if it is “busy”.

Macros, such as *MOD_INC_USE_COUNT*, are used to increment, decrement and check the status.

It is easy to lose track of the count during debug (e.g. a process gets destroyed because your driver references a NULL pointer).

Setting these macros to no-ops is useful in this case.

**Initialization and Shutdown**

Usage Counts:

The file */proc/modules* gives a list of the currently loaded modules.

Some of mine are given below:

```
ide-cd                  26848   0 (autoclean)
cdrom                   27232   0 (autoclean) [ide-cd]
autofs                  11264   1 (autoclean)
```

- The first field is the name of the module
- The second is the number of bytes in use.
- The third field gives the usage count.
- The forth field (kernels >2.1.18) lists optional flags
- The fifth field (kernels >2.1.18) lists modules that reference this module.

*rmmod* calls *cleanup_module* only if the usage count is zero.

As shown above, *cleanup_module* unregisters facilities **explicitly** while the symbol table is removed automatically.

Names other than *init_module* and *cleanup_module* can be used now via the functions *module_init(my_init)* and *module_exit(my_cleanup)* in <linux/init.h>

**Getting System Resources**

A module can't accomplish its task without using system resource, such as **memory**, **I/O ports**, **interrupt lines** and **DMA channels**.

For memory, use the analogues to *malloc* and *free*, *kmalloc* and *kfree*.

*kmalloc* takes a priority argument; use *GFP_KERNEL* in most cases.

**Ports**:

Ports are requested so the driver can be assured exclusive access to them.

However, the *request/free* mechanism cannot prevent unauthorized access since there is no hardware to enforce it.

The file */proc/ioports* contains information about registered ports.

(*/proc/iomem* contains info about I/O memory).

A sample of mine is shown below:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
```

**Using System Resources**

Ports:

The range given in hex enclose the ports locked by a driver.

Other drivers should not access these ports until they are released.

Collision avoidance:

The */proc/ioports* file is consulted to configure the jumpers on the hardware (if it is manual).

When the driver initializes, it can safely autodetect the hardware by **probing**.

**Probing** involves writing and then reading the unregistered ports.

If the "correct" device is connected to the probed port, it will reply to queries with sensible codes.

If a different device is present, the response is unpredictable !

A compliant driver calls *check_region* to determine the lock status of the ports, *request_region* to lock them and *release_region* to free them.

**Using System Resources**

Ports:

A typical registering sequence:

```
#include <linux/ioport.h>
#include <linux/errno.h>

static int skull_detect(unsigned int port,
   unsigned int range)
   {
   int err;

   if ((err = check_region(port, range)) < 0 )
      return err; /* Busy */

   if ( skull_probe_hw(port, range) != 0 )
      return -ENODEV;  /* Not found.*/

/* Always succeeds */
   request_region(port, range, "skull");
   return 0;
   }
```

**Using System Resources**

Ports:

Note that *skull_probe_hw* would contain driver specific code that writes ports and looks for specific responses.

Typical release code:

```
static void skull_release(unsigned int port,
   unsigned int range)
   {
   release_region(port, range);
   }
```

Memory:

Similarly, I/O memory information is available in */proc/iomem*

```
00000000-0009f7ff : System RAM
0009f800-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
```

**Using System Resources**

The same access mechanism is used to access the I/O memory registry as was used for ports.

The following routines are used to obtain and relinquish an I/O memory region:

*int check_mem_region(unsigned long start, unsigned long len);*
*int request_mem_region(unsigned long start, unsigned long len, char *name);*
*int release_mem_region(unisgned long start, unsigned long len);*

The typical driver will already know its I/O memory range and use:

```
if ( check_mem_region(mem_addr, mem_size)){
   printk("drivername: memory already in use\n");
   return -EBUSY;
   }

request_mem_region(mem_addr, mem_size, "drivername");
```

**Automatic and Manual Configuration**

Several parameters required by the driver can change from system to system.

For example:

- The actual I/O addresses.
- The memory range.
- Other driver specific parameters, such as the device model and release number.

The driver, of course, must be configured with the correct values at initialization time.

Again, most of these problems do not apply to PCI devices.

There are two ways to get these values:

- The user specifies them explicitly.
- The driver autodetects them.

**Autodetection** is the best method but more difficult to implement.

Best approach is to autoconfigure while allowing the user to override.

**Automatic and Manual Configuration**

Parameters can be assigned at load time by *insmod* and *modprobe* (modprobe can also read parameter assignments from a configuration file, */etc/modules.conf*).

These commands accept integer and string values as command line arguments.

The following command can be used to assign values:

```
insmod skull skull_ival=656 skull_sval="a string val"
```

Before *insmod* can change module parameters, the module must make them available.

Parameters are declared with MODULE_PARM macro as shown for the following global variables.

```
int skull_ival = 0;
char *skull_sval;

MODULE_PARAM(skull_ival, "i");
MODULE_PARAM(skull_sval, "s");
```

**Automatic and Manual Configuration**

The second argument is the type, other possibilities are *b* (one byte), *h* (two bytes), *i* (integer), *l* (long) and *s* (string).

All parameters should be given reasonable default values.

If this is done, an autoconfiguration strategy can be:

If the configuration variables have the default value, perform autodetection. Otherwise, keep the override value passed to *insmod*.

Of course, the default value should be some illegal value to trigger the autodetection.

The following code shows how *skull* autodetects the port address of a device.

This code looks for multiple devices.

Note that manual configuration is restricted to a single device.

**Automatic and Manual Configuration**

```
/* Look for the device in port range 0x280-0x300. */
#define SKULL_PORT_FLOOR 0x280
#define SKULL_PORT_CEIL 0x300
#define SKULL_PORT_RANGE 0x010

/* Autodetect unless the user overrides.*/
static int skull_port_base = 0; /* Force autodetect */
MODULE_PARM( skull_port_base, "i");

static int skull_find_hw(void) /* Return # of devices */
   {
   int base = skull_port_base ? skull_port_base:
      SKULL_PORT_FLOOR;
   int result = 0;
   do {
      if ( skull_detect(base, SKULL_PORT_RANGE) == 0)
         { skull_init_board(base);
         result++;}
      base += SKULL_PORT_RANGE;
      } while (skull_port_base == 0 &&
            base < SKULL_PORT_CEIL);
   return result;
   }
```