Standalone Device Drivers in Linux

Theodore Ts'o

July 1, 1999

1 Introduction

Traditionally, Unix device drivers have been developed and distributed inside the kernel. There are a number of good reasons why this method is the predominant way most device drivers are distributed. First of all, it simplifies the packaging and distribution issues for the device driver. Secondly, it makes it easier to make changes to the interfaces between the kernel and the device drivers.

However, this traditional scheme has a number of major disadvantages. First of all, it means that each version of the device driver is linked to a specific version of the kernel. So if a device is only supported in a development release, a customer who might otherwise want to use a stable, production release might be forced to use a bleeding-edge system. Alternatively, there may be bugs present in the device driver as included in the stable release of the kernel, but which are fixed in the development kernel.

Moreover, including device drivers with the kernel is not scalable in the long term. If Linux-like [1] systems are ever to be able to support as many devices as various OS's from Redmond, Washington, hardware manufacturers will have to be able to support and distribute drivers separately from the main kernel distribution. Currently, the size of the total Linux kernel source base has been doubling (roughly) every 18 months. Almost all of this growth has been the result of new device drivers. If Linux is to be successful at Linus Torvald's goal of Total World Domination, it will be essential that we remove any limits to growth, and an exponential growth in the size of the Linux kernel is an obviously a long-term growth limitation.

2 Existing systems

Currently, there are a number of device drivers which are currently distributed outside of the Linux kernel: The Comtrol Rocketport and VS-1000 drivers, written by the author, and David Hind's PCMCIA drivers.

The Comtrol Rocketport driver is notable in that a single source base can support three stable Linux releases (1.2, 2.0, 2.2) as well as a number of the intervening development releases. It was necessary to support a wide range of Linux releases because the a number of customers of Comtrol Rocketport board were Internet Service Providers (ISP's) who were using the intelligent serial board to provide dialup access of various kinds (PPP, UUCP, login, etc.). For

[1] The term "Unix(tm)-like" is frowned upon by the Open Group copyright police :-)

these customers, stability is paramount, and often this causes them to defer upgrading their systems for a long time.

David Hind's PCMCIA drivers present a very good example of some of the advantages of stand-alone device drivers. Hardware manufacturers are constantly introducing new PCMCIA devices, and often these new require making minor tweaks to the PCMCIA package. These frequency of such changes often exceeded the release frequency of the stable kernel series. If the PCMCIA driver had been integrated into the mainline kernel code, then users of the stable kernel series might be forced to use the development kernels if they needed to use a newer PCMCIA device. The alternative, trying to merge changes back and forth between the stable and kernel series, and forcing frequent releases of the stable kernel, is almost equally unpalatable. The solution, then, of distributing the PCMCIA drivers separately from the main body of the Linux kernel sources, was almost certainly the best way of solving this conundrum.

3 Techniques for writing stand-alone drivers in Linux

Unfortunately, the structure of Linux kernel is such that stand-alone drivers is not easy. This is perhaps the reason why there are so relatively few drivers which are distributed in this way. Unfortunately, the various interfaces used by device drivers are not fixed, and change over time. (This is not unique to Linux; many Unix(tm) systems have similar issues.) Hence, the techniques used by existing standalone device drivers utilize a certain amount of brute-force. However, it is surprising that despite the a certain amount of elegance, it is easier than it first seems to make achive this goal.

3.1 Using Kernel Modules

The first way device drivers were distributed separately from the kernel were to simply distribute kernel patches which dropped the driver into the kernel sources. The system administrator would then configure the kernel to use the new driver, and recompile the kernel. While this provides the most easy and simplicity for the driver author, it is less than convenient for the users of the driver who have to compile, install, and use the driver.

Loadable kernel modules were originally introduced to solve a variety of problems. They allow a developer to try out their driver without having to reboot a new kernel, thus drastically shorting the edit, compile, and test cycle. They also allow a single common set of kernel and modules to be used across a wide variety of machines, while avoiding the expense of statically compiling a large number of drivers into a a single, large, generic kernel. This is very important for Linux distributions, since many of their customers may not be comfortable with configuring and compiling their own kernel.

Although most kernel modules today are distributed as part the Linux kernel, kernel modules turn out to be an extremely useful distribution mechanism for standalone device drivers as well. All of the advantages for modules distributed with the kernel sources apply doubly so for modules distributed separate from the kernel. Perhaps most importantly, in many cases the user doesn't even need to have the kernel sources installed on their system; a set of kernel include files which correspond to the currently installed kernel is sufficient.[2]

[2]Unfortunately, some more primitive distributions have distributed a set of kernel include files which do match with the kernel which they are distributing. Certain versions of Slackware distribution in particular have suffered

3.2 Building modules outside of the kernel tree

While writing a device driver which is to be included in the standard kernel source tree, the author can rely on the build infrastructure provided by the kernel. The author, however, can no longer depend on this infrastructure when distributing her device driver as in a standalone package.

Fortunately, the Linux kernel does not require an overly complex build system in most cases. (More on the exceptions in a moment). For the simplest modules, all that is required is that the following C preprocessor flags be defined by the Makefile: `__KERNEL__` and `LINUX`. If the kernel was configured to be built using SMP, the Makefile must also define preprocessor flag `__SMP__`. In addition, the C compiler needs to be told where to find the kernel include files, so a compiler directive such as `-I/usr/src/linux/include` is generally required.

In some cases, it may be desirable to write the kernel module so that it can used either inside the kernel source tree, or separately in a standalone distribution. In these cases, the driver Makefile may contain a definition some flag such as `-DLOCAL_HEADERS` which is not defined in the standard kernel build environment, so that the driver code can distinguish between the standalone compilation environment and the kernel compilation environment. In this example, the flag is used so that driver includes the local header files (which may have required changes or improvements), instead of the header files found in the standard kernel header files.[3]

3.2.1 Synchronizing kernel versions

A *very* common cause of user confusion and complaints is caused by mismatches between the version of the kernel header files, and the kernel actually installed and running on the system. This can occur due to a number of reasons. Some distributions erroneously include inconsistent kernel and kernel header files. Other users may have installed a set of kernel sources, but haven't yet installed the new kernel yet.

In most cases, the symptom reported by users suffering from this header file/kernel synchronization problem is simply that the `insmod` utility refuses to load the newly built device driver. However, sometimes the kernel module will load, but then fail in some mysterious way. Mercifully, these cases were fairly rare, because they are extremely difficult to debug.

In order to avoid these problems, it is a very good idea to have the makefile determine the kernel version of the header files, as well as the kernel version of the currently running kernel, and issue a warning if these two versions are different. Under Linux, the kernel version string should also include critical configuration information about whether the kernel is an SMP kernel (and thus requires the `-D__SMP__` flag when compiling the module object files) and whether CONFIG_MODVERSIONS option has been enabled.[4]

The makefile also should maintain a file which contains the kernel version of the header files. This file serves two purposes. It is used by the installation utility (see section 3.3). In addition, the makefile can use this file to determine when the version of the header files

from this problem.

[3]This can also be accomplished by adding a `-I.` directive ahead of the `-I/usr/src/linux/include` directive, so that local header files are used in preference to the standard kernel header files. This approach has the advantage of minimizing the number of #ifdef's in the code. However, it means that the header files must be laid out in the same include file hierarchy as was used inside the kernel.

[4]See section 3.2.2 for a discussion why this is necessary.

have changed, and force all of the object files to be rebuilt if necessary, thus avoiding another common user-reported problem.

3.2.2 Exporting symbols from a module

The job of a kernel module's Makefile becomes much more complicated if the module exports functional interfaces which are used by other modules. This is due to a requirement (if the `CONFIG_MODVERIONS` option is enabled) to include a 32-bit checksum in the symbol names of functions which are exported for use by modules, no matter whether those functions are defined by the kernel or by other modules. The 32-bit checksum is calculated over the function signature[5] including all of the definition of the types used by the function signature. This will prevent a previously compiled kernel module from linking with a new kernel if the function prototype changes, or any of the type definitions change, the changed checksum will cause the linker in the `insmod` utility to fail, since the checksum is part of the symbol name.

If the module is only using kernel-provided functions, the modversions scheme is very easy to use; a series of C preprocessor macros in the kernel header files automatically redefine references to functions such as `kmalloc` with a symbol name such as `kmalloc_R93d4cfe6`. However, if a module also needs to export functions for use by other modules, the makefile must a generate and modify the kernel header files, and this turns out to be rather tricky.

The method of specifying which symbols need to be exported changed between Linux 2.0 and Linux 2.2. Hence, modules that need to support both older and newer versions of Linux will need to have something like the following in their source:

```
#if (LINUX_VERSION_CODE > 0x20100)
EXPORT_SYMBOL(register_serial);
EXPORT_SYMBOL(unregister_serial);
#else
static struct symbol_table serial_syms = {
#include <linux/symtab_begin.h>
        X(register_serial),
        X(unregister_serial),
#include <linux/symtab_end.h>
};
#endif
```

Similarly, the method for invoking `genksyms`, the program which examines a source file and its header files to generate symbol checksum, has also changed over between Linux 2.0 and 2.2. In addition, if the module is being built for an SMP kernel, an appropriate argument to GENKSYMS must be passed by the Makefile to indicate this in the symbol checksums. All of this requires that the Makefile be cognizant of the kernel version of its compilation environment, which is another good reason why the Makefile should store this information in a conveniently accessible file:

```
serial.ver: serial.c $(DEP_HEADERS)
        @echo "Generating serial.ver..." ; \
        if test -s .smpflag ; then smp_prefix="-p smp_"; \
```

[5] That is, the types of all of its arguments as well as the type returned by the function.

```
        else smp_prefix=; fi; \
        kver=‘sed -e ’s/-SMP//’ -e ’s/-MOD//’ .kver‘ ; \
        $(CC) $(GENKSYM_FLAGS) -E serial.c > serial.tmp ; \
        echo $(CC) $(GENKSYM_FLAGS) -E serial.c \> serial.tmp; \
        case $$kver in \
        2.0.*)  echo $(GENKSYMS) . < serial.tmp ; \
                $(GENKSYMS) . < serial.tmp ;; \
        2.[12].*) echo $(GENKSYMS) $$smp_prefix -k $$kver \< serial.tmp ; \
                $(GENKSYMS) $$smp_prefix -k $$kver < serial.tmp \
                        > serial.ver ;; \
        *)      echo "Unsupported kernel version $$kver" ;; \
        esac ; rm -f serial.tmp ; echo rm -f serial.tmp ; \
        mkdir -p linux/modules ; cp serial.ver linux/modules ; \
```

One final subtlty which must be considered is if `.ver` file containing the symbol versions is not mentioned in the kernel header's `modversions.h` file, the Makefile must generate its own copy of the this file which has been modified to include this `.ver` file, and place it in an appropriate location inside the module build tree so that it is used instead of the system `modversions.h` header file.

3.3 Installing modularized device drivers

As many an software developer knows, it is not enough to write wonderful programs; in order to be used, the user has to be able to install it! Unfortunately, it is all-too-common in the Linux community (and in the Unix community in general) for developers to pay little attention to the installation process. This is perhaps partially responsible for the reputation which Linux and Linux-like systems have of being user-hostile.

What needs to be done during installation? Obviously, the object file containing the modularized device driver needs to be installed in its proper location in the `/lib/modules` hierarchy. The installation should install the module both in a kernel-version dependent location, such as `/lib/modules/2.2.1/misc/serial.o`, and in a version independent location, such as `/lib/modules/serial.o`. The kernel-version dependent location *must* be based on the kernel version for which the driver was compiled, and not the kernel version currently running on the machine at the time of the installation. (However, it is a good idea for the installation program to check to see if these two kernel versions are different, and print a warning message if they do not.

Secondly, if the driver is loaded at boot-time or if it requires some kind of special initialization, the installation program will need to install a /etc/rc.d script. There are several reasons why a device driver might need a boot-time script. For example, the driver might need to be loaded at all times, instead of being dynamically loaded by `kerneld` or the `kmod` thread. Or perhaps the drivers require special run-time configuration.[6]

Unfortunately, the exact location for installing the /etc/rc.d file varies from distribution to distribution. This means the shell script must detect the /etc/rc.d convention currently in use on the system, and adapt its installation accordingly.

Finally, if the module exports any functions (see section 3.2.2) the installation script will need

[6]In many circumstances, dynamically loaded drivers can be configured automatically by modprobe when they are loaded by making appropriate changes to the `/etc/conf.modules` file.

to modify the kernel header files so that other modules will be able to successfully use the driver's exported interfaces. (Note however that these kernel header files may be modified and revert back to the original contents if a `make depend` is executed in the kernel source tree.

Because of the potential complexity of the installation script, while it can be done as a Makefile target (i.e., what happens when you type `make install`), most of the time the steps required to install a device driver are best stored in a shell script.

3.4 Kernel version compatibility

The Linux kernel interfaces have changed over time. Although a device driver developer sometimes suspects the changes are made only to make his life more difficult, in reality the changes are needed to make the kernel code be more general or more performant.

A good example of this is the changes which were made to the interfaces used by kernel routines to read or write user memory which were made during the 2.1 development cycle for the i386 architecture. Instead of manually calling the verify_area() routine which would check the page tables to see whether or not the memory access was allowed (or whether a a page needed to be copied or faulted into memory), the new routines used the i386 memory management routines to notice if an access violation or page fault occured, and the fault handler looked up the faulting PC in a table to determine which exception handler should be invoked to recover from the memory fault. Unfortunately, the kernel API used in Linux 2.0 was not general enough to allow for this particular optimization —- hence, the interface was changed during the 2.1 development cycle.

When functional kernel interfaces changes, the best way for an author to adapt her code is to change the code to use the new interface, and then to provide backwards compatibility routines which implement the new interface in terms of the older interfaces. For example, the following routine can be found in the stand-alone serial driver:

```
#if (LINUX_VERSION_CODE < 131336)
static int copy_from_user(void *to, const void *from_user, unsigned long len)
{
        int     error;

        error = verify_area(VERIFY_READ, from_user, len);
        if (error)
                return len;
        memcpy_fromfs(to, from_user, len);
        return 0;
}
#else
#include <asm/uaccess.h>
#endif
```

In the above example, LINUX_VERSION_CODE macro is used to determine the version of the kernel for which the driver is being compiled. It is set by the kernel headers using the following forumla: $version * 65536 + patchlevel * 256 + sublevel$. Hence, the Linux kernel version 2.1.8 corresponds to a LINUX_VERSION_CODE of $2 * 65536 + 1 * 256 + 8$, or 131336.

In other cases, a the definition of a structure may change. For example, a new structure element might get added in the middle of the structure. In other cases, the structure elements might get reordered for performance reasons. A good way of insulating a program from such changes is to avoid the use of initializers, i.e,:

```
static struct console sercons = {
        "ttyS",
        serial_console_write,
        NULL,
        serial_console_device,
        .
        .
        .
};
```

Instead, it is safer to initialize the structure explicitly, like this:

```
static struct console sercons;

        memset(sercons, 0, sizeof(sercons));
        strcpy(sercons.name, "ttyS");
        sercons.write = serial_console_write;
        sercons.device = serial_console_device;
```

This approach is not without its dangers; for example, new structure elements may have to be properly initialized in or the kernel abstraction will not work correctly. So while this technique does not eliminate the need for testing the driver as kernel interfaces change, it does minimize the number of version specific changes which need to be made to the driver source.

4 Future work

Many of the techniques described in this paper will no doubt strike the reader as being extremely hackish and kludgy. There is unfortunately more than a grain of truth in such an observation. However, the techniques described in this do work, and have been successfully used in more than a few device drivers. Authors of device drivers are urged to try to try distributing works in standalone form despite the primitive nature of the support currently available for such distributions, because of the many advantages which we have discussed in this paper.

That being said, there are many ways in which Linux could be improved in order to make stand-alone device drivers distribution much cleaner, simpler, and elegant. Let us discuss some of the possibilities, with the hopes that future versions of Linux will incorporate at least some of these improvements.

4.1 Standards for installing rc.d scripts

It is extremely unfortunate that the various Linux distributions have not been able to standardize on a single standard for /etc/rc.d init files.[7] This was mainly due to an accident of history, in that the various distributions did not coordinate before adopting their /etc/rc.d conventions, and now do not feel that they can easily migrate to a single standard without compromising their backwards compatibility with older versions of their distribution.

The Linux Standard Base (LSB) effort has been working on this and other problems which have made life difficult for Independent Software Vendors (ISV's) who wish to distribute software which is compatible with all, or at least most, Linux distributions. The solution which has been proposed for solving this problem is a configurable tool which can be called by installation scripts who need to have their /etc/rc script installed at boot time. This tool would read structured comments placed at the beginning of the /etc/rc script to determine where in the logical boot and shutdown sequences the script needed to be called, and based on the conventions of that particular distribution, the tool would install the script and the necessary symbolic links in the proper places. While not as satisfying as simply being able to declare a standard for /etc/rc files, it does prove the adage that any computer science problem can be solved by adding an extra layer of indrection.

4.2 Binary Compatibility

All of the techniques described in this paper assumed that part of the installation process for the driver involved compiling the driver from source for a particular machine. Unfortunately, it is not at all clear whether or not this will suffice for the vast majority of the non-technically inclined user base which Linux will need to start attacking if it is to continue its current exponential growth rate. Even if the source to the drivers are available, the thought of needing to compile a kernel extension may scare away more naive users. Furthermore, their system might not even have the C compiler installed!

Currently, interfaces change often enough, particularly at the binary level, that it is impractical to ship binary modules. Even during the course of a stable relesae, the core Linux developers have refused to "tie their hands" by making commitments that the ABI exposed by the kernel module would remain unchanging. Their concern is that some critical bug fixes might require making changes to the structures which would affect the ABI.

One way of addressing these two concerns is to add an adoptor layer which provides this fixed ABI. This adaoptor layer might change over time to take advantage of new features in the kernel, but it would be very careful to maintain backwards comatibility at the ABI layer. The disadvantage of using such an adaptor layer is an inevitable performance loss. However, with sufficient tuning this performance loss might be made small enough to be insigificant, and in any case many might view a small performance loss to be an acceptable tradeoff in order to provide a standardized ABI for device drivers which are distributed in binary form.

[7]Even worse is the Slackware distribution, which is extremely primitive. It doesn't even support the System V-style init files at all. This means that it is extremely difficult to programmatically install a script to be run at boot-time. System administrators usually have to manually edit their /etc/rc files by hand.

4.3 Cross OS compatibility

Taking this the idea of compatibility one step further, there are those who believe it would be useful to provide an adaptor layer which would allow a single implementation of a device driver to serve multiple operating systems; further, that this be doable both on a binary and source level. There is currently a industry consortium of vendors who are working on just such a project, which has been named UDI, for “Uniform Device Interface.”

SCO Unix is a major backer of this initiative, and they are hosting the UDI web page, which can be found at `http://www.sco.com/UDI`. Furthermore, SCO has offerred to write a proof-of-concept adaptor layer which would allow Linux to use UDI device drivers. Most developers in the Linux community have viewed this initiative with ambivalence. On the one hand, if successful it would greatly increase the number of device drivers supporting exotic hardware devices. On the other hand, there are some genuine performance issues which are raised by such a scheme. Can a device driver which is written to be general across multiple different types of OS and OS programming styles really be efficient? Only time will tell; the results of the SCO UDI effort should be most interesting.

5 Conclusion

In this paper, we have discussed the motivations for distributing device drivers separetly from the kernel, and examined some techniques used by the few device drivers which are currently being distributed separately from the mainline Linux kernel. Finally, we have revewed some strategies for making it easier for device drivers to be packaged and distributed in this way. Although much work remains to be done in order to make standalone device driver development easier, I hope that this paper encourages developers to consider distributing device drivers outside of the kernel, either instead of or in addition to including a version of the driver in the latest development kernel.

In the future, if Linux continues to grow in size and importance, hardware manufacturers who today only feel obligated to supply Windows driver may decide that they too wish to write and distribute a Linux driver to increase the number customers willing to buy their product. It is my hope that at that point, Linux will have advanced sufficiently that a less technically inclined user can simply pull the new scanner, digital camera out of the box, plug it into her computer, insert a device driver diskette and be using the new hardware moments later. I believe that the use of stand-alone device drivers will make this type of scenarios much more likely to occur.

A A Sample Makefile

The following makefile is taken from the author’s stand-alone serial driver distribution:

```
LINUX_SRC=      /usr/src/linux

INCL_FLAGS=     -I. -I$(LINUX_SRC)/include
DEF_FLAGS=      -D__KERNEL__ -DLINUX -DLOCAL_HEADERS -DEXPORT_SYMTAB
BASE_FLAGS=     -m486 -O2 -Wall $(INCL_FLAGS) $(DEF_FLAGS) ‘cat .smpflag‘
```

```
CFLAGS=         $(BASE_FLAGS) ‘cat .modflag‘
GENKSYM_FLAGS=  $(BASE_FLAGS) -D__GENKSYMS__

CC=             gcc
GENKSYMS=       genksyms

VERFILE=        $(LINUX_SRC)/include/linux/modules/serial.ver
MODVERFILE=     $(LINUX_SRC)/include/linux/modversions.h

DEP_HEADERS=    serial.h serialP.h version.h \
                        $(LINUX_SRC)/include/linux/version.h

all: test_version maybe_modules serial.o

test_version:
        @ISMP='';KSMP='';cp /dev/null .smpflag; \
        if grep -q smp_ $(LINUX_SRC)/include/linux/modules/ksyms.ver ; then \
                ISMP='-SMP'; echo "-D__SMP__" > .smpflag ; \
        fi ; \
        if grep -q smp_ /proc/ksyms ; then \
                KSMP='-SMP'; \
        fi ; \
        IMOD=''; KMOD='';cp /dev/null .modflag; \
        if grep -q "#define CONFIG_MODVERSIONS" \
                        $(LINUX_SRC)/include/linux/autoconf.h ; then \
                IMOD='-MOD'; echo "-DMODVERSIONS" > .modflag; \
        fi ; \
        if grep -q "kfree" /proc/ksyms && \
           ! grep -q "kfree$$" /proc/ksyms ; then \
                KMOD='-MOD' ; \
        fi ; \
        IVER=‘grep UTS_RELEASE $(LINUX_SRC)/include/linux/version.h | \
                awk '{print $$3}' | tr -d \"‘$$IMOD$$ISMP; \
        KVER=‘uname -r‘$$KMOD$$KSMP; \
        if [ -f .kver ]; then \
            OVER=‘cat .kver‘; \
            if [ $$OVER != $$IVER ]; then \
                echo "Removing previously built driver for Linux $$OVER"; \
                make clean; \
                echo " "; \
            fi ; \
        else \
                make clean ; \
        fi ; \
        echo $$IVER > .kver ; \
        echo Building serial driver for Linux $$IVER ; \
        echo " " ; \
        if [ $$IVER != $$KVER ]; then \
            echo "WARNING: The current kernel is actually version $$KVER." ; \
            echo " " ; \
        fi

maybe_modules:
        @if test -s .modflag ; then \
                make modversions.h serial.ver ; \
        fi

install:
        @LINUX_SRC=$(LINUX_SRC) sh ./setup.sh

clean:
        rm -f serial.o *˜ .kver serial.ver modversions.h
        rm -rf linux

really-clean: clean
        rm -f *.o

serial.o: serial.c $(DEP_HEADERS)
        @echo $(CC) $(CFLAGS) -c serial.c -o serial.o ; \
        $(CC) $(CFLAGS) -c serial.c -o serial.o

serial.ver: serial.c $(DEP_HEADERS)
        @echo "Generating serial.ver..." ; \
        if test -s .smpflag ; then smp_prefix="-p smp_"; \
        else smp_prefix=; fi; \
        kver=‘sed -e ’s/-SMP//’ -e ’s/-MOD//’ .kver‘ ; \
        $(CC) $(GENKSYM_FLAGS) -E serial.c > serial.tmp ; \
        echo $(CC) $(GENKSYM_FLAGS) -E serial.c \> serial.tmp; \
        case $$kver in \
        2.0.*)  echo $(GENKSYMS) . < serial.tmp ; \
                $(GENKSYMS) . < serial.tmp ;; \
        2.[12].*) echo $(GENKSYMS) $$smp_prefix -k $$kver \< serial.tmp ; \
                $(GENKSYMS) $$smp_prefix -k $$kver < serial.tmp \
                        > serial.ver ;; \
        *)      echo "Unsupported kernel version $$kver" ;; \
        esac ; rm -f serial.tmp ; echo rm -f serial.tmp ; \
        mkdir -p linux/modules ; cp serial.ver linux/modules ; \

modversions.h: $(MODVERFILE)
        @if ! grep -q /serial.ver $(MODVERFILE) ; then \
                sed -f modversions.sed < $(MODVERFILE) > modversions.tmp ; \
        else \
                cp $(MODVERFILE) modversions.tmp ; \
        fi ; \
        if ! test -f modversions.h || \
           ! cmp -s modverisions.h modversions.tmp ; then \
                echo "Generating modversions.h" ; \
                mv modversions.tmp modversions.h ; \
                mkdir -p linux ; cp modversions.h linux ; \
        else rm -f modversions.tmp ; fi
```

B A sample installation script

The following installation script, `setup.sh` is taken from the author's standalone serial driver distribution:

```
#!/bin/sh
#
# This script is responsible for installing the Serial driver
#

if [ ‘whoami‘ != root ]; then
        echo "You must be root to install the device driver!"
        exit 1
fi

# Install a copy of the driver in the version independent location
if [ ! -d /lib/modules ]; then
        mkdir -p /lib/modules
fi

if test -d /lib/modules/misc ; then
    cp serial.o /lib/modules/misc
    rm -f /lib/modules/serial.o
else
    cp serial.o /lib/modules
fi

# Install a copy of the driver in a kernel version specific location
if [ -f .kver ]; then
    KVER=‘sed -e ’s/-SMP//’ -e ’s/-MOD//’ .kver‘
    KVERSION=‘cat .kver‘
else
    KVER=‘grep UTS_RELEASE /usr/include/linux/version.h | \
        awk ’{print $3}’ | tr -d \"‘
    KVERSION=$KVER
    if grep -q "#define CONFIG_MODVERSIONS" \
                        $LINUX_SRC/include/linux/autoconf.h ; then
        KVERSION=${KVERSION}-MOD
    fi
    if grep -q smp_ $LINUX_SRC/include/linux/modules/ksyms.ver ; then
        KVERSION=${KVERSION}-SMP
    fi
fi

if grep -q smp_ /proc/ksyms ; then
        RSMP=’-SMP’
```

```
fi
if grep -q "kfree" /proc/ksyms &&
   ! grep -q "kfree$$" /proc/ksyms ; then
        RMOD='-MOD'
fi
RVERSION=`uname -r`$RMOD$RSMP
echo " "
echo "Installing serial driver for Linux $KVERSION"
echo " "
if [ $KVERSION != $RVERSION ]; then
    echo "WARNING: The current kernel is actually version $RVERSION."
    echo " "
fi

#
# Actually install the kernel module
#
if [ ! -d /lib/modules/$KVER/misc ]; then
        mkdir -p /lib/modules/$KVER/misc
fi

cp serial.o /lib/modules/$KVER/misc

#
# Now that we're done, run depmod -a so that modprobe knows where to fund
# stuff
#
depmod -a

#
# Since the serial driver exports symbols, update the serial.ver and
# modversions file.
#
if test -z "$LINUX_SRC" ; then
    echo "Defaulting linux sources to /usr/src/linux"
    LINUX_SRC=/usr/src/linux
fi
VERFILE=$LINUX_SRC/include/linux/modules/serial.ver
MODVERFILE=$LINUX_SRC/include/linux/modversions.h

if ! test -f $VERFILE || ! cmp -s serial.ver $VERFILE ; then
        echo "Updating $VERFILE"
        cp serial.ver $VERFILE
fi
if ! grep -q /serial.ver $MODVERFILE ; then \
        echo "Updating $MODVERFILE to include serial.ver"
        sed -f modversions.sed < $MODVERFILE > ${MODVERFILE}.new
        mv $MODVERFILE ${MODVERFILE}.old
        mv ${MODVERFILE}.new $MODVERFILE
fi
```

```
#
# OK, now we install the installation script
#
if [ -d /etc/rc.d/init.d ]; then
        # This is a RedHat or other system using a System V init scheme
        RC_DEST=/etc/rc.d/init.d/serial
        RC_DIR=/etc/rc.d
        RC_START=S83serial
        RC_STOP=K18serial
elif [ -d /etc/init.d ]; then
        # This is a Debian system

        RC_DEST=/etc/init.d/serial
        RC_DIR=/etc
        RC_START=S83serial
        RC_STOP=K20serial
else
        # This must be a Slackware or other non-SysV init system
        if [ ! -f /etc/rc.d/rc.serial ]; then
                echo "The initialization script will be installed in "
                echo "/etc/rc.d/rc.serial.  You will need to edit your "
                echo "/etc/rc files to run /etc/rc.d/rc.serial"

                if [ ! -d /etc/rc.d ]; then
                        mkdir -p /etc/rc.d
                fi
        fi

        RC_DEST=/etc/rc.d/rc.serial
        RC_DIR=""
fi

#
# Determine the version numbers of the installed script (if any) and the
# rc.serial script which we have.
#
SRC_VERS=‘grep "ˆ# FILE_VERSION: " rc.serial | awk ’{print $3}’‘
DEST_VERS=0
if test -f $RC_DEST ; then
    DEST_VERS=‘grep "ˆ# FILE_VERSION: " $RC_DEST | awk ’{print $3}’‘
    if test -z "$DEST_VERS" ; then
        DEST_VERS=0
    fi
fi

#
# Only update the destination file if we have a newer version.
#
```

```
	if test $SRC_VERS -gt $DEST_VERS ; then
		if test -f $RC_DEST ; then
			echo "Updating $RC_DEST"
			NEW_INSTALL=no
		else
			echo "Installing $RC_DEST"
			NEW_INSTALL=yes
		fi
		cp rc.serial $RC_DEST
	fi

	if test -n "$RC_DIR" ; then
		rm -f $RC_DIR/rc?.d/[SK]??rc.serial
		if test "$NEW_INSTALL" = "yes" ; then
			echo " "
			echo "You are using a system which uses the System V init"
			echo "scheme.  The initialization script has been installed"
			echo "in $RC_DEST and it has been automatically "
			echo "set up to be run when you reboot"

			ln -sf ../init.d/serial $RC_DIR/rc2.d/${RC_START}
			ln -sf ../init.d/serial $RC_DIR/rc3.d/${RC_START}
			ln -sf ../init.d/serial $RC_DIR/rc5.d/${RC_START}
			ln -sf ../init.d/serial $RC_DIR/rc0.d/${RC_STOP}
			ln -sf ../init.d/serial $RC_DIR/rc1.d/${RC_STOP}
			ln -sf ../init.d/serial $RC_DIR/rc6.d/${RC_STOP}
		fi

		if [ -f /etc/rc.d/rc.serial ]; then
			echo "You have an old /etc/rc.d/rc.serial script"
			echo "After checking to make sure it is no longer being used"
			echo "you should delete it."
		fi
	fi
```