Linux Kernel Makefiles

2000-September-14
Michael Elizabeth Chastain, <mec@shout.net>

=== Table of Contents

This document describes the Linux kernel Makefiles.

=== 1 Overview

The Makefiles have five parts:

Makefile: the top Makefile.
.config: the kernel configuration file.
arch/*/Makefile: the arch Makefiles.
Subdirectory Makefiles: there are about 300 of these.
Rules.make: the common rules for all subdirectory Makefiles.

The top Makefile reads the .config file, which comes from the kernel configuration process.

The top Makefile is responsible for building two major products: vmlinux (the resident kernel image) and modules (any module files). It builds these goals by recursively descending into the subdirectories of the kernel source tree. The list of subdirectories which are visited depends upon the kernel configuration.

The top Makefile textually includes an arch Makefile with the name arch/$(ARCH)/Makefile. The arch Makefile supplies architecture-specific information to the top Makefile.

Each subdirectory has a Makefile which carries out the commands passed down from above. The subdirectory Makefile uses information from the .config file to construct various file lists, and then it textually includes the common rules in Rules.make.

Rules.make defines rules which are common to all the subdirectory Makefiles. It has a public interface in the form of certain variable lists. It then declares rules based on those lists.

=== 2 Who does what

People have four different relationships with the kernel Makefiles.

Users are people who build kernels. These people type commands such as "make menuconfig" or "make bzImage". They usually do not read or edit any kernel Makefiles (or any other source files).

Normal developers are people who work on features such as device drivers, file systems, and network protocols. These people need to maintain the subdirectory Makefiles for the subsystem that they are working on. In order to do this effectively, they need some overall

knowledge about the kernel Makefiles, plus detailed knowledge about the public interface for Rules.make.

Arch developers are people who work on an entire architecture, such as sparc or ia64. Arch developers need to know about the arch Makefiles as well as subdirectory Makefiles.

Kbuild developers are people who work on the kernel build system itself. These people need to know about all aspects of the kernel Makefiles.

This document is aimed towards normal developers and arch developers.

=== 3 Makefile language

The kernel Makefiles are designed to run with GNU Make. The Makefiles use only the documented features of GNU Make, but they do use many GNU extensions.

GNU Make supports elementary list-processing functions. The kernel Makefiles use a novel style of list building and manipulation with few "if" statements.

GNU Make has two assignment operators, ":=" and "=". ":=" performs immediate evaluation of the right-hand side and stores an actual string into the left-hand side. "=" is like a formula definition; it stores the right-hand side in an unevaluated form and then evaluates this form each time the left-hand side is used.

There are some cases where "=" is appropriate. Usually, though, ":=" is the right choice.

All of the examples in this document were drawn from actual kernel sources. The examples have been reformatted (white space changed, lines split), but are otherwise exactly the same.

=== 4 Variables passed down from the top

The top Makefile exports the following variables:

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

These variables define the current kernel version. A few arch Makefiles actually use these values directly; they should use $(KERNELRELEASE) instead.

$(VERSION), $(PATCHLEVEL), and $(SUBLEVEL) define the basic three-part version number, such as "2", "4", and "0". These three values are always numeric.

$(EXTRAVERSION) defines an even tinier sublevel for pre-patches or additional patches. It is usually some non-numeric string such as "-pre4", and is often blank.

KERNELRELEASE

$(KERNELRELEASE) is a single string such as "2.4.0-pre4", suitable for constructing installation directory names or showing in version strings. Some arch Makefiles use it for this purpose.

ARCH

This variable defines the target architecture, such as "i386", "arm", or "sparc". Many subdirectory Makefiles test $(ARCH) to determine which files to compile.

By default, the top Makefile sets $(ARCH) to be the same as the host system system architecture. For a cross build, a user may override the value of $(ARCH) on the command line:

```
    make ARCH=m68k ...
```

TOPDIR, HPATH

$(TOPDIR) is the path to the top of the kernel source tree. Subdirectory Makefiles need this so that they can include $(TOPDIR)/Rules.make.

$(HPATH) is equal to $(TOPDIR)/include. A few arch Makefiles need to use this to do special things using include files.

SUBDIRS

$(SUBDIRS) is a list of directories which the top Makefile
enters in order to build either vmlinux or modules. The actual
directories in $(SUBDIRS) depend on the kernel configuration.
The top Makefile defines this variable, and the arch Makefile
extends it.

HEAD, CORE_FILES, NETWORKS, DRIVERS, LIBS
LINKFLAGS

$(HEAD), $(CORE_FILES), $(NETWORKS), $(DRIVERS), and $(LIBS)
specify lists of object files and libraries to be linked into
vmlinux.

The files in $(HEAD) are linked first in vmlinux.

$(LINKFLAGS) specifies the flags to build vmlinux.

The top Makefile and the arch Makefile jointly define these
variables. The top Makefile defines $(CORE_FILES), $(NETWORKS),
$(DRIVERS), and $(LIBS). The arch Makefile defines $(HEAD)
and $(LINKFLAGS), and extends $(CORE_FILES) and $(LIBS).

Note: there are more variables here than necessary. $(NETWORKS),
$(DRIVERS), and even $(LIBS) could be subsumed into $(CORE_FILES).

CPP, CC, AS, LD, AR, NM, STRIP, OBJCOPY, OBJDUMP
CPPFLAGS, CFLAGS, CFLAGS_KERNEL, MODFLAGS, AFLAGS, LDFLAGS
PERL
GENKSYMS

These variables specify the commands and flags that Rules.make
uses to build goal files from source files.

$(CFLAGS_KERNEL) contains extra C compiler flags used to compile
resident kernel code.

$(MODFLAGS) contains extra C compiler flags used to compile code
for loadable kernel modules. In the future, this flag may be
renamed to the more regular name $(CFLAGS_MODULE).

$(AFLAGS) contains assembler flags.

$(GENKSYMS) contains the command used to generate kernel symbol
signatures when CONFIG_MODVERSIONS is enabled. The genksyms

command comes from the modutils package.

CROSS_COMPILE

This variable is a prefix path for other variables such as $(CC), $(AS), and $(LD). The arch Makefiles sometimes use and set this variable explicitly. Subdirectory Makefiles don't need to worry about it.

The user may override $(CROSS_COMPILE) on the command line if desired.

HOSTCC, HOSTCFLAGS

These variables define the C compiler and C compiler flags to be used for compiling host side programs. These are separate variables because the target architecture can be different from the host architecture.

If your Makefile compiles and runs a program that is executed during the course of building the kernel, then it should use $(HOSTCC) and $(HOSTCFLAGS).

For example, the subdirectory drivers/pci has a helper program named gen-devlist.c. This program reads a list of PCI ID's and generates C code in the output files classlist.h and devlist.h.

Suppose that a user has an i386 computer and wants to build a kernel for an ia64 machine. Then the user would use an ia64 cross-compiler for most of the compilation, but would use a native i386 host compiler to compile drivers/pci/gen-devlist.c.

For another example, kbuild helper programs such as scripts/mkdep.c and scripts/lxdialog/*.c are compiled with $(HOSTCC) rather than $(CC).

ROOT_DEV, SVGA_MODE, RAMDISK

End users edit these variables to specify certain information about the configuration of their kernel. These variables are ancient! They are also specific to the i386 architecture. They really should be replaced with CONFIG_* options.

MAKEBOOT

This variable is defined and used only inside the main arch
Makefiles. The top Makefile should not export it.

INSTALL_PATH

This variable defines a place for the arch Makefiles to install
the resident kernel image and System.map file.

INSTALL_MOD_PATH, MODLIB

$(INSTALL_MOD_PATH) specifies a prefix to $(MODLIB) for module
installation. This variable is not defined in the Makefile but
may be passed in by the user if desired.

$(MODLIB) specifies the directory for module installation.
The top Makefile defines $(MODLIB) to
$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE). The user may
override this value on the command line if desired.

CONFIG_SHELL

This variable is private between Makefile and Rules.make.
Arch makefiles and subdirectory Makefiles should never use this.

MODVERFILE

An internal variable. This doesn't need to be exported, as it
is never used outside of the top Makefile.

MAKE, MAKEFILES

Some variables internal to GNU Make.

$(MAKEFILES) in particular is used to force the arch Makefiles
and subdirectory Makefiles to read $(TOPDIR)/.config without
including it explicitly. (This was an implementational hack
and could be fixed).

=== 5 The structure of an arch Makefile

--- 5.1 Architecture-specific variables

The top Makefile includes one arch Makefile file, arch/$(ARCH)/Makefile.
This section describes the functions of the arch Makefile.

An arch Makefile extends some of the top Makefile's variables with
architecture-specific values.

SUBDIRS

The top Makefile defines $(SUBDIRS). The arch Makefile extends
$(SUBDIRS) with a list of architecture-specific directories.

Example:

```
# arch/alpha/Makefile

SUBDIRS := $(SUBDIRS) arch/alpha/kernel arch/alpha/mm \
           arch/alpha/lib arch/alpha/math-emu
```

This list may depend on the configuration:

```
# arch/arm/Makefile

ifeq ($(CONFIG_ARCH_ACORN),y)
SUBDIRS         += drivers/acorn
...
endif
```

CPP, CC, AS, LD, AR, NM, STRIP, OBJCOPY, OBJDUMP
CPPFLAGS, CFLAGS, CFLAGS_KERNEL, MODFLAGS, AFLAGS, LDFLAGS

The top Makefile defines these variables, and the arch Makefile
extends them.

Many arch Makefiles dynamically run the target C compiler to
probe supported options:

```
# arch/i386/Makefile

# prevent gcc from keeping the stack 16 byte aligned
CFLAGS += $(shell if $(CC) -mpreferred-stack-boundary=2 \
```

```
            -S -o /dev/null -xc /dev/null >/dev/null 2>&1; \
            then echo "-mpreferred-stack-boundary=2"; fi)
```

And, of course, $(CFLAGS) can depend on the configuration:

```
    # arch/i386/Makefile

    ifdef CONFIG_M386
    CFLAGS += -march=i386
    endif

    ifdef CONFIG_M486
    CFLAGS += -march=i486
    endif

    ifdef CONFIG_M586
    CFLAGS += -march=i586
    endif
```

Some arch Makefiles redefine the compilation commands in order to add architecture-specific flags:

```
    # arch/s390/Makefile

    LD=$(CROSS_COMPILE)ld -m elf_s390
    OBJCOPY=$(CROSS_COMPILE)objcopy -O binary -R .note -R .comment -S
```

--- 5.2 Vmlinux build variables

An arch Makefile cooperates with the top Makefile to define variables which specify how to build the vmlinux file. Note that there is no corresponding arch-specific section for modules; the module-building machinery is all architecture-independent.

HEAD, CORE_FILES, LIBS
LINKFLAGS

The top Makefile defines the architecture-independent core of thse variables, and the arch Makefile extends them. Note that the arch Makefile defines (not just extends) $(HEAD) and $(LINKFLAGS).

Example:

```
# arch/m68k/Makefile

ifndef CONFIG_SUN3
LINKFLAGS = -T $(TOPDIR)/arch/m68k/vmlinux.lds
else
LINKFLAGS = -T $(TOPDIR)/arch/m68k/vmlinux-sun3.lds -N
endif

...

ifndef CONFIG_SUN3
HEAD := arch/m68k/kernel/head.o
else
HEAD := arch/m68k/kernel/sun3-head.o
endif

SUBDIRS += arch/m68k/kernel arch/m68k/mm arch/m68k/lib
CORE_FILES := arch/m68k/kernel/kernel.o arch/m68k/mm/mm.o $(CORE_FILES)
LIBS += arch/m68k/lib/lib.a
```

--- 5.3 Post-vmlinux goals

An arch Makefile specifies goals that take the vmlinux file, compress it, wrap it in bootstrapping code, and copy the resulting files somewhere. This includes various kinds of installation commands.

These post-vmlinux goals are not standardized across different architectures. Here is a list of these goals and the architectures that support each of them (as of kernel version 2.4.0-test6-pre5):

```
balo        mips
bootimage       alpha
bootpfile       alpha, ia64
bzImage     i386, m68k
bzdisk      i386
bzlilo      i386
compressed      i386, m68k, mips, mips64, sh
dasdfmt     s390
Image       arm
image       s390
install     arm, i386
```

```
lilo            m68k
msb             alpha, ia64
my-special-boot alpha, ia64
orionboot       mips
rawboot         alpha
silo            s390
srmboot         alpha
tftpboot.img    sparc, sparc64
vmlinux.64      mips64
vmlinux.aout    sparc64
zImage          arm, i386, m68k, mips, mips64, ppc, sh
zImage.initrd   ppc
zdisk           i386, mips, mips64, sh
zinstall        arm
zlilo           i386
znetboot.initrd ppc
```

--- 5.4 Mandatory arch-specific goals

An arch Makefile must define the following arch-specific goals.
These goals provide arch-specific actions for the corresponding goals
in the top Makefile:

```
archclean       clean
archdep         dep
archmrproper    mrproper
```

=== 6 The structure of a subdirectory Makefile

A subdirectory Makefile has four sections.

--- 6.1 Comments

The first section is a comment header. Historically, many anonymous
people have edited kernel Makefiles without leaving any change
histories in the header; comments from them would have been valuable.

--- 6.2 Goal definitions

The second section is a bunch of definitions that are the heart of the subdirectory Makefile. These lines define the files to be built, any special compilation options, and any subdirectories to be recursively entered. The declarations in these lines depend heavily on the kernel configuration variables (CONFIG_* symbols).

The second section looks like this:

```
# drivers/block/Makefile
obj-$(CONFIG_MAC_FLOPPY)     += swim3.o
obj-$(CONFIG_BLK_DEV_FD)     += floppy.o
obj-$(CONFIG_AMIGA_FLOPPY)   += amiflop.o
obj-$(CONFIG_ATARI_FLOPPY)   += ataflop.o
```

--- 6.3 Rules.make section

The third section is the single line:

```
include $(TOPDIR)/Rules.make
```

--- 6.4 Special rules

The fourth section contains any special Makefile rules needed that are not available through the common rules in Rules.make.

=== 7 Rules.make variables

The public interface of Rules.make consists of the following variables:

--- 7.1 Subdirectories

A Makefile is only responsible for building objects in its own directory. Files in subdirectories should be taken care of by Makefiles in the these subdirs. The build system will automatically invoke make recursively in subdirectories, provided you let it know of them.

To do so, use the subdir-{y,m,n,} variables:

```
    subdir-$(CONFIG_ISDN)                   += i4l
    subdir-$(CONFIG_ISDN_CAPI)              += capi
```

When building the actual kernel, i.e. vmlinux ("make {vmlinux,bzImage,...}"), make will recursively descend into directories listed in $(subdir-y).

When building modules ("make modules"), make will recursively descend into directories listed in $(subdir-m).

When building the dependencies ("make dep") make needs to visit every subdir, so it'll descend into every directory listed in $(subdir-y), $(subdir-m), $(subdir-n), $(subdir-).

You may encounter the case where a config option may be set to "y", but you still want to possibly build modules in that subdirectory.

For example, drivers/isdn/capi/Makefile has

```
    obj-$(CONFIG_ISDN_CAPI)                 += kernelcapi.o capiutil.o
    obj-$(CONFIG_ISDN_CAPI_CAPI20)          += capi.o
```

where it's possible that CONFIG_ISDN_CAPI=y, but CONFIG_ISDN_CAPI_CAPI20=m.

This is expressed by the following construct in the parent Makefile drivers/isdn/Makefile:

```
    mod-subdirs                             := i4l hisax capi eicon
    subdir-$(CONFIG_ISDN_CAPI)              += capi
```

Having a subdir ("capi") listed in the variable $(mod-subdirs) will make the build system enter the specified subdirectory during "make modules" also, even though the subdir ("capi") is listed only in $(subdir-y), not $(subdir-m).

--- 7.2 Object file goals

O_TARGET, obj-y

The subdirectory Makefile specifies object files for vmlinux
in the lists $(obj-y). These lists depend on the kernel
configuration.

Rules.make compiles all the $(obj-y) files. It then calls
"$(LD) -r" to merge these files into one .o file with the name
$(O_TARGET). This $(O_TARGET) is later linked into vmlinux by
a parent Makefile.

The order of files in $(obj-y) is significant. Duplicates in
the lists are allowed: the first instance will be linked into
$(O_TARGET) and succeeding instances will be ignored.

Link order is significant, because certain functions
(module_init() / __initcall) will be called during boot in the
order they appear. So keep in mind that changing the link
order may e.g. change the order in which your SCSI
controllers are detected, and thus you disks are renumbered.

Example:

```
# Makefile for the kernel ISDN subsystem and device drivers.

# The target object and module list name.

O_TARGET        := vmlinux-obj.o

# Each configuration option enables a list of files.

obj-$(CONFIG_ISDN)                      += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)          += isdn_bsdcomp.o

# The global Rules.make.

include $(TOPDIR)/Rules.make
```

--- 7.3 Library file goals

L_TARGET

Instead of building an O_TARGET object file, you may also
build an archive which again contains objects listed in
$(obj-y). This is normally not necessary and only used in
the lib, arch/$(ARCH)/lib directories.

--- 7.4 Loadable module goals

obj-m

$(obj-m) specify object files which are built as loadable
kernel modules.

A module may be built from one source file or several source
files. In the case of one source file, the subdirectory
Makefile simply adds the file to $(obj-m)

Example:

```
obj-$(CONFIG_ISDN_PPP_BSDCOMP)          += isdn_bsdcomp.o
```

If a kernel module is built from several source files, you specify
that you want to build a module in the same way as above.

However, the build system of course needs to know which the parts
are that you want to build your module of, so you have to tell it
by setting an $(<module_name>-objs) variable.

Example:

```
obj-$(CONFIG_ISDN)                      += isdn.o

isdn-objs := isdn_net.o isdn_tty.o isdn_v110.o isdn_common.o
```

In this example, the module name will be isdn.o. Rules.make
will compile the objects listed in $(isdn-objs) and then run
"$(LD) -r" on the list of these files to generate isdn.o

Note: Of course, when you are building objects into the kernel,
the syntax above will also work. So, if you have CONFIG_ISDN=y,
the build system will build an isdn.o for you out of the individual
parts and then link this into the $(O_TARGET), as you'd expect.

--- 7.5 Objects which export symbols

export-objs

When using loadable modules, not every global symbol in the kernel / other modules is automatically available, only those explicitly exported are available for your module.

To make a symbol available for use in modules, to "export" it, use the EXPORT_SYMBOL(<symbol>) directive in your source. In addition, you need to list all object files which export symbols (i.e. their source contains an EXPORT_SYMBOL() directive) in the Makefile variable $(export-objs).

Example:

```
# Objects that export symbols.

export-objs     := isdn_common.o
```

since isdn_common.c contains

```
EXPORT_SYMBOL(register_isdn);
```

which makes the function register_isdn available to low-level ISDN drivers.

--- 7.6 Compilation flags

EXTRA_CFLAGS, EXTRA_AFLAGS, EXTRA_LDFLAGS, EXTRA_ARFLAGS

$(EXTRA_CFLAGS) specifies options for compiling C files with $(CC). The options in this variable apply to all $(CC) commands for files in the current directory.

Example:

```
# drivers/sound/emu10k1/Makefile
EXTRA_CFLAGS += -I.
ifdef DEBUG
    EXTRA_CFLAGS += -DEMU10K1_DEBUG
endif
```

$(EXTRA_CFLAGS) does not apply to subdirectories of the current directory. Also, it does not apply to files compiled with $(HOSTCC).

This variable is necessary because the top Makefile owns the variable $(CFLAGS) and uses it for compilation flags for the entire tree.

$(EXTRA_AFLAGS) is a similar string for per-directory options when compiling assembly language source.

Example: at the time of writing, there were no examples of $(EXTRA_AFLAGS) in the kernel corpus.

$(EXTRA_LDFLAGS) and $(EXTRA_ARFLAGS) are similar strings for per-directory options to $(LD) and $(AR).

Example: at the time of writing, there were no examples of $(EXTRA_LDFLAGS) or $(EXTRA_ARFLAGS) in the kernel corpus.

CFLAGS_$@, AFLAGS_$@

$(CFLAGS_$@) specifies per-file options for $(CC). The $@ part has a literal value which specifies the file that it's for.

Example:

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o    = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
             -DGDTH_STATISTICS
CFLAGS_seagate.o =   -DARBITRATE -DPARITY -DSEAGATE_USE_ASM
```

These three lines specify compilation flags for aha152x.o, gdth.o, and seagate.o

$(AFLAGS_$@) is a similar feature for source files in assembly languages.

Example:

```
# arch/arm/kernel/Makefile
AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
```

```
AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional
```

Rules.make has a feature where an object file depends on the value of $(CFLAGS_$@) that was used to compile it. (It also depends on the values of $(CFLAGS) and $(EXTRA_CFLAGS)). Thus, if you change the value of $(CFLAGS_$@) for a file, either by editing the Makefile or overriding the value some other way, Rules.make will do the right thing and re-compile your source file with the new options.

Note: because of a deficiency in Rules.make, assembly language files do not have flag dependencies. If you edit $(AFLAGS_$@) for such a file, you will have to remove the object file in order to re-build from source.

LD_RFLAG

This variable is used, but never defined. It appears to be a vestige of some abandoned experiment.

--- 7.7 Miscellaneous variables

IGNORE_FLAGS_OBJS

$(IGNORE_FLAGS_OBJS) is a list of object files which will not have their flag dependencies automatically tracked. This is a hackish feature, used to kludge around a problem in the implementation of flag dependencies. (The problem is that flag dependencies assume that a %.o file is built from a matching %.S or %.c file. This is sometimes not true).

USE_STANDARD_AS_RULE

This is a transition variable. If $(USE_STANDARD_AS_RULE) is defined, then Rules.make will provide standard rules for assembling %.S files into %.o files or %.s files (%.s files are useful only to developers).

If $(USE_STANDARD_AS_RULE) is not defined, then Rules.make will not provide these standard rules. In this case, the subdirectory Makefile must provide its own private rules for assembling %.S files.

In the past, all Makefiles provided private %.S rules. Newer Makefiles should define USE_STANDARD_AS_RULE and use the standard Rules.make rules. As soon as all the Makefiles across all architectures have been converted to USE_STANDARD_AS_RULE, then Rules.make can drop the conditional test on USE_STANDARD_AS_RULE. After that, all the other Makefiles can drop the definition of USE_STANDARD_AS_RULE.

=== 8 New-style variables

[This sections dates back from a time where the way to write Makefiles described above was "new-style". I'm leaving it in as it describes the same thing in other words, so it may be of some use]

The "new-style variables" are simpler and more powerful than the "old-style variables". As a result, many subdirectory Makefiles shrank more than 60%. This author hopes that, in time, all arch Makefiles and subdirectory Makefiles will convert to the new style.

Rules.make does not understand new-style variables. Thus, each new-style Makefile has a section of boilerplate code that converts the new-style variables into old-style variables. There is also some mixing, where people define most variables using "new style" but then fall back to "old style" for a few lines.

--- 8.1 New variables

obj-y obj-m obj-n obj-

These variables replace $(O_OBJS), $(OX_OBJS), $(M_OBJS), and $(MX_OBJS).

Example:

```
# drivers/block/Makefile
obj-$(CONFIG_MAC_FLOPPY)        += swim3.o
obj-$(CONFIG_BLK_DEV_FD)        += floppy.o
obj-$(CONFIG_AMIGA_FLOPPY)      += amiflop.o
obj-$(CONFIG_ATARI_FLOPPY)      += ataflop.o
```

Notice the use of $(CONFIG_...) substitutions on the left hand
side of an assignment operator. This gives GNU Make the power
of associative indexing! Each of these assignments replaces
eight lines of code in an old-style Makefile.

After executing all of the assignments, the subdirectory
Makefile has built up four lists: $(obj-y), $(obj-m), $(obj-n),
and $(obj-).

$(obj-y) is a list of files to include in vmlinux.
$(obj-m) is a list of files to build as single-file modules.
$(obj-n) and $(obj-) are ignored.

Each list may contain duplicates items; duplicates are
automatically removed later. Duplicates in both $(obj-y) and
$(obj-m) will automatically be removed from the $(obj-m) list.

Example:

```
    # drivers/net/Makefile

    ...
    obj-$(CONFIG_OAKNET) += oaknet.o 8390.o
    ...
    obj-$(CONFIG_NE2K_PCI) += ne2k-pci.o 8390.o
    ...
    obj-$(CONFIG_STNIC) += stnic.o 8390.o
    ...
    obj-$(CONFIG_MAC8390) += daynaport.o 8390.o
    ...
```

In this example, four different drivers require the code in
8390.o. If one or more of these four drivers are built into
vmlinux, then 8390.o will also be built into vmlinux, and will
not be built as a module -- even if another driver which needs
8390.o is built as a module. (The modular driver is able to
use services of the 8390.o code in the resident vmlinux image).

export-objs

$(export-objs) is a list of all the files in the subdirectory
which potentially export symbols. The canonical way to construct
this list is:

```
    grep -l EXPORT_SYMBOL *.c
```

(but watch out for sneaky files that call EXPORT_SYMBOL from an included header file!)

This is a potential list, independent of the kernel configuration. All files that export symbols go into $(export-objs). The boilerplate code then uses the $(export-objs) list to separate the real file lists into $(*_OBJS) and $(*X_OBJS).

Experience has shown that maintaining the proper X's in an old-style Makefile is difficult and error-prone. Maintaining the $(export-objs) list in a new-style Makefile is simpler and easier to audit.

$(foo)-objs

Some kernel modules are composed of multiple object files linked together.

For each multi-part kernel modul there is a list of all the object files which make up that module. For a kernel module named foo.o, its object file list is foo-objs.

Example:

```
    # drivers/scsi/Makefile
    list-multi  := scsi_mod.o sr_mod.o initio.o a100u2w.o

    ...

    scsi_mod-objs   := hosts.o scsi.o scsi_ioctl.o constants.o \
               scsicam.o scsi_proc.o scsi_error.o \
               scsi_obsolete.o scsi_queue.o scsi_lib.o \
               scsi_merge.o scsi_dma.o scsi_scan.o \
               scsi_syms.o
    sr_mod-objs := sr.o sr_ioctl.o sr_vendor.o
    initio-objs := ini9100u.o i91uscsi.o
    a100u2w-objs    := inia100.o i60uscsi.o
```

The subdirectory Makefile puts the modules onto obj-* lists in the usual configuration-dependent way:

```
    obj-$(CONFIG_SCSI)      += scsi_mod.o
```

```
    obj-$(CONFIG_BLK_DEV_SR)    += sr_mod.o
    obj-$(CONFIG_SCSI_INITIO)   += initio.o
    obj-$(CONFIG_SCSI_INIA100)  += a100u2w.o
```

Suppose that CONFIG_SCSI=y. Then vmlinux needs to link in all
14 components of scsi_mod.o.

Suppose that CONFIG_BLK_DEV_SR=m. Then the 3 components
of sr_mod.o will be linked together with "$(LD) -r" to make the
kernel module sr_mod.o.

Also suppose CONFIG_SCSI_INITIO=n. Then initio.o goes onto
the $(obj-n) list and that's the end of it. Its component
files are not compiled, and the composite file is not created.

subdir-y subdir-m subdir-n subdir-

These variables replace $(ALL_SUB_DIRS), $(SUB_DIRS) and
$(MOD_SUB_DIRS).

Example:

```
    # drivers/Makefile
    subdir-$(CONFIG_PCI)        += pci
    subdir-$(CONFIG_PCMCIA)     += pcmcia
    subdir-$(CONFIG_MTD)        += mtd
    subdir-$(CONFIG_SBUS)       += sbus
```

These variables work similar to obj-*, but are used for
subdirectories instead of object files.

After executing all assignments, the subdirectory Makefile has
built up four lists: $(subdir-y), $(subdir-m), $(subdir-n),
and $(subdir-).

$(subdir-y) is a list of directories that should be entered
    for making vmlinux.
$(subdir-m) is a list of directories that should be entered
    for making modules.
$(subdir-n) and $(subdir-) are only used for collecting a list
    of all subdirectories of this directory.

Each list besides subdir-y may contain duplicates items; duplicates

are automatically removed later.

mod-subdirs

$(mod-subdirs) is a list of all the subdirectories that should be added to $(subdir-m), too if they appear in $(subdir-y)

Example:

```
# fs/Makefile
mod-subdirs :=  nls
```

This means nls should be added to (subdir-y) and $(subdir-m) if CONFIG_NFS = y.

=== 9 Credits

Thanks to the members of the linux-kbuild mailing list for reviewing drafts of this document, with particular thanks to Peter Samuelson and Thomas Molina.