B
INSTALLING CONFIGURATION SOFTWARE

This appendix describes how to build and install *imake* and related configuration software if you do not already have it on your machine. The software (referred to here as ‘‘the distribution’’) is available as *imake.tar.Z* from the archive sites listed in Appendix A, *Obtaining Configuration Software*.

The essential programs you will need for working your way through this book are:

imake	*Makefile* generator
xmkmf	Bootstrapper that uses the X11 configuration files
imboot	General purpose bootstrapper
makedepend	Header file dependency generator
mkdirhier	Directory creation tool

Some of these may be new to you, particularly if you are looking at this appendix without reading the rest of the book first. *imake*, of course, is discussed throughout the book. *xmkmf*, *makedepend*, and *mkdirhier* are discussed in Chapter 2, *A Tour of imake*. *imboot* is discussed in Chapter 9, *Coordinating Sets of Configuration Files*, and Chapter 14, *Designing Extensible Configuration Files*.

The distribution also includes the X11 configuration files and some miscellaneous programs like *msub* and *imdent*.

Most of the distribution is based on the current release of X11 (X11R5, patchlevel 24 at the time of writing), but some programs are not part of the standard X distribution, e.g., *imboot*, *msub*, and *imdent*.

Distribution Layout

The software is distributed as a compressed *tar* file *imake.tar.Z*. Retrieve and unpack it according to the instructions in Appendix A. This will create the project tree shown in Figure B−1 (an *imake* directory and several subdirectories).

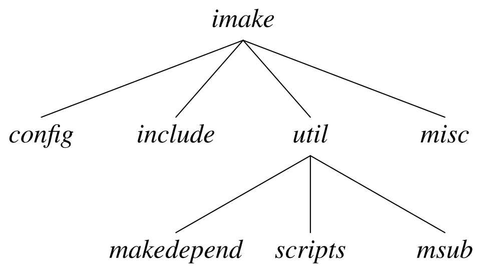

Figure 1. imake distribution project tree

You should then familiarize yourself with the distribution's structure and contents. Move into the distribution root so you can look around:

```
% cd imake
```

The distribution root directory contains these subdirectories:

config The X11 configuration files and source for *imake*.

include The header file *Xosdefs.h*, needed to compile *imake* and *makedepend*.

util Parent of utility directories.

misc Miscellaneous bits and pieces: portions of the X11 Release Notes that pertain to *imake*; any errors that have been found in these instructions since publication; troubleshooting information too detailed or specialized to be included in this appendix; reader-contributed notes about porting *imake* to systems not covered by the distribution, etc.

The *util* directory contains a number of subdirectories:

makedepend Source for the C version of *makedepend*. Several programs go by this name; the version included here is intended for use with *imake* and is written by Todd Brunhoff, *imake*'s author.

scripts Source for *xmkmf*, *mkdirhier*, *imboot*, and *bsdinst*, and some other miscellaneous scripts, such as *imdent*, *which*, and a script version of *makedepend* (if you can't get the C version to work).

msub Source for *msub*.

Find Out What You Already Have

Survey the landscape of your system, since some or all of the software may already be present on your machine. For instance, if you are using a workstation running X, there is a good chance the configuration files and some of the programs are already installed, because X itself is configured with *imake*.

To find programs, use the *which* command. It will tell you either where a program is located, or, if it couldn't find it, which directories it looked in (usually the directories named in your `PATH` variable).[1] Here, *which* tells you that *imake* is installed in */usr/local/bin*:

```
% which imake
```

[1] If you do not have *which*, you can use either of the scripts *which.sh* or *which.csh* supplied in the *scripts* directory. Alternatively, use *find* or poke around by hand looking for programs in system directories like */usr/local/bin*, */usr/bin*, */usr/bin/X11*, etc.

```
/usr/local/bin/imake
```

On the other hand, if *which* can't find *imake*, you will get a result like this:

```
% which imake
no imake in . /usr/local/bin /usr/bin/X11 /usr/ucb /bin /usr/bin
```

In most instances, if you discover that a program included in the distribution is already installed, you do not need to install it. However, you should make exceptions when you have out-of-date or broken versions of programs:

- Current versions of *makedepend* and *xmkmf* each understand an −*a* option. Older versions distributed prior to X11R5 do not. Install the versions in the distribution to update your system.
- If you are running Open Windows, be aware that the versions of *imake* and *xmkmf* distributed with it are difficult to work with. See the section "*imake* versus Open Windows" later in this appendix.

In addition to the programs, you need a copy of the X11 configuration files, because many examples throughout this book assume they are available to play with. The default location for the X files is */usr/lib/X11/config*. If they are not there, and you have *xmkmf* installed, look at it to see where it expects to find the X11 files.

Prepare to Install the Distribution

Before building the distribution you must decide where you will install the software to avoid misconfiguring the programs. Some installation locations are built in. The defaults are:

/usr/bin/X11	Location of programs
/usr/lib/X11/config	Location of the X11 configuration files
/usr/lib/config	Location of the *imboot* configuration root directory

You can change the default installation locations if you like. If you do, then whenever you see the defaults in examples elsewhere in this book, you must substitute the locations you have chosen.

Suppose you want to install programs in */usr/local/bin* and the X11 configuration files in */var/X11R5/lib/config*, and that you want to use */var/lib/config* for the *imboot* configuration root. Make the changes as follows:

To change the program installation directory, add the following to the second half of *config/site.def*:

```
#ifndef BinDir
#define BinDir /usr/local/bin
#endif
```

To change the X11 configuration file installation directory, add the following to the second half of *config/site.def*:

```
#ifndef ConfigDir
#define ConfigDir /var/X11R5/lib/config
#endif
```

To change the configuration root directory used by *imboot*, edit this line in *util/scripts/Imakefile*:

```
CONFIGROOTDIR = $(USRLIBDIR)/config
```

Change it to this:

```
CONFIGROOTDIR = /var/lib/config
```

Installing with Limited Privileges

If you have insufficient privileges to install software on your machine wherever you like, try to convince a sympathetic site administrator to install files that need to go in system directories.[2] If that is not possible, install everything under your own account (use the instructions just given for changing the default locations). If the path to your home directory is */u/you*, I suggest the following installation directories:

/u/you/bin	Location of programs
/u/you/lib/X11/config	Location of the X11 configuration files
/u/you/lib/config	Location of the *imboot* configuration root directory

Build the Distribution—Quick Instructions

This section contains quick instructions for building the distribution. If they do not work, read the section ''Detailed Instructions.''

Change into the *config* directory and look around; you will see the configuration files and the source code for *imake*:

```
% cd config
% ls
DGUX.cf       Server.tmpl   ibm.cf        noop.rules    sunLib.rules
Imake.rules   apollo.cf     ibmLib.rules  oldlib.rules  sunLib.tmpl
Imake.tmpl    att.cf        ibmLib.tmpl   pegasus.cf    sv4Lib.rules
Imakefile     bsd.cf        imake.c       rtcchack.bac  sv4Lib.tmpl
Library.tmpl  ccimake.c     imake.man     sgi.cf        ultrix.cf
Makefile.ini  convex.cf     imakemdep.h   sgiLib.rules  x386.cf
Mips.cf       cray.cf       luna.cf       site.def
Project.tmpl  generic.cf    macII.cf      sony.cf
README        hp.cf         moto.cf       sun.cf
```

Find a vendor-specific configuration file that applies to your machine. This will be one of the files named with a *.cf* suffix. For Sun machines, use *sun.cf*, for Silicon Graphics machines, use *sgi.cf*, etc. If you do not find a vendor file for your system, you may need to write one using the detailed instructions. (First look in the *misc* directory, though. It contains instructions for some systems that aren't supported in the standard X11 distribution.)

Find the definitions of `OSMajorVersion` and `OSMinorVersion` in the vendor file. They represent the major and minor release numbers for your operating system. Check that they are correct, and change them if they are not. Suppose *ultrix.cf* specifies Ultrix 4.2:

```
#define OSMajorVersion 4
#define OSMinorVersion 2
```

If you are running Ultrix 4.1, the major number is correct, but you must change the minor number to 1:

```
#define OSMajorVersion 4
#define OSMinorVersion 1
```

Look over *site.def* to see whether you want to change anything. In particular, to use *gcc* as your C compiler, define `HasGcc` as `YES` by uncommenting the `#define` that is already there.

Look in Table B−1 and find the bootstrap flags for your system. If the value is *`varies`*, several values are possible and you must look through the vendor file and figure out which value is appropriate for your machine. If your system is not listed, the distribution hasn't been ported to it and you need to use the

[2] Yes, I know that ''sympathetic site administrator'' is oxymoronic.

detailed instructions.

Table 1
Bootstrap flags for various systems

Vendor File	Bootstrap Flags	Remark
DGUX.cf	`-DDGUX`	
Mips.cf	`-DMips`	
apollo.cf	`-Dapollo`	
att.cf	`-Datt`	
bsd.cf	`-DNOSTDHDRS`	
convex.cf	`-D__convex__ -tm c1`	
cray.cf	`-DCRAY`	
hp.cf	`-Dhpux`	
ibm.cf	*varies*	Will include `-Dibm`
luna.cf	`-Dluna`	
macII.cf	`-DmacII`	
moto.cf	`-DMOTOROLA -DSVR4`	If SVR4 conformant
	`-DMOTOROLA -DSYSV`	If not SVR4 conformant
pegasus.cf	`-DM4310 -DUTEK`	
sgi.cf	`-Dsgi`	
sony.cf	`-Dsony`	
sun.cf	`-Dsun`	For SunOS 4.1 and up
	`-Dsun -DNOSTDHDRS`	For SunOS before 4.1
ultrix.cf	`-Dultrix`	
x386.cf	*varies*	Will include `-DSYSV386`

Look in the *misc* directory under the distribution root to see if there are any files pertaining to your system. Some patches to make the distribution build on older releases of OS'es are contained there, and you may need to apply one of them.

In the distribution root (the *imake* directory) execute the following command, replacing *`flags`* with the appropriate bootstrap flags for your system:[3]

```
% make World BOOTSTRAPCFLAGS="flags" >& make.world
```

Examples:

```
% make World BOOTSTRAPCFLAGS="-Dsun" >& make.world
% make World BOOTSTRAPCFLAGS="-Dhpux" >& make.world
% make World BOOTSTRAPCFLAGS="-DMOTOROLA -DSVR4" >& make.world
```

When the command terminates, take a look through *make.world* to see whether or not the `World` operation completed without error. If it failed, use the detailed instructions to build the distribution. Otherwise, go ahead and install the software.

[3] The command uses *csh* output redirection. If you are using *sh*, use this command instead:

```
$ make World BOOTSTRAPCFLAGS="flags" > make.world 2>&1
```

Install the Software

If you want to install everything, use the following command in the distribution root directory:

```
% make install
```

Otherwise, change into individual directories and install only those files in which you are interested. The instructions below show how to do this; leave out the parts for the files you do not want to install.

In *config*—install *imake* and the X11 configuration files:

```
% make install.imake
% make install.cffiles
```

Do not install *ccimake*; it is only needed for compiling *imake*.

In *util/makedepend*—if you built the compiled version of *makedepend* (the default on most systems) install it:

```
% make install.makedepend
```

If you did not build the compiled version, the script version will be installed from *util/scripts*.

In *util/scripts*—install Install gobs of stuff:

```
% make install.xmkmf
% make install.mkdirhier
% make install.imboot
% make install.imdent
```

If you have a System V version of *install* instead of a BSD-compatible one, the distribution contains a script *bsdinst* you can use instead (you need to install *bsdinst* if the installation commands you have just been executing used it):

```
% make install.bsdinst
```

If you built the script version of *makedepend*, install it:

```
% make install.makedepend
```

In *util/msub*—install *msub*:

```
% make install.msub
```

Detailed Instructions

This section contains detailed instructions for building *imake*. You're probably reading it either because *imake* hasn't been ported to your type of system, or because the quick instructions did not work on your machine. In the first case, you need to modify your copy of the distribution according to the following instructions. In the second case, you should look for discrepancies between what the distribution already says about your system and what these instructions indicate it should say. Make sure you have read through the quick instructions first because they contains information you will need here. You might also find it helpful to read Chapter 3, *Understanding Configuration Files*, to get some idea of how *imake* works. That will help you understand the result you are trying to produce.

When *imake* runs, it invokes *cpp* to process the configuration files. Default values for most configuration parameters are defined in *Imake.tmpl*, *Project.tmpl*, etc. Also included among the configuration files are many vendor files, each of which contains parameter values that override any defaults that are inappropriate for a particular vendor's systems. Since there are several different vendor files from which to choose, *cpp* has to figure out which is the right one for the machine you are actually using. *cpp* does this by checking various symbols known to identify different types of systems. The one that is defined determines the current system type.

The symbol that identifies your system can come from a couple of sources. Some vendors ship a *cpp* that predefines a symbol unique to that vendor's systems. For instance, on Ultrix, *cpp* predefines `ultrix`, which makes it easy to ascertain the system type:

```
#ifdef ultrix
    /* it's Ultrix, all right... */
#endif
```

Unfortunately, some vendor-supplied symbols aren't useful for system identification because they are ambiguous (or have become so). For instance, at one time (in X11R3) the `mips` symbol unambiguously indicated a true Mips Computers, Inc. machine, but the symbol later became ambiguous when several other OS's written to run on Mips processors also defined `mips` (Ultrix, NEWS OS, IRIX, etc.). In the absence of a unique predefined vendor symbol, it is necessary to make up an identifier symbol and to tell *imake* to define it when starting up *cpp*. Thus, to signify true Mips machines, the "artificial" symbol `Mips` is now used.

It might even be that your preprocessor predefines no useful system-indicating symbols at all. ANSI C takes a dim view of most predefined symbols, and the trend in preprocessors as the ANSI juggernaut rolls on is toward fewer predefinitions. Here, too, you invent an identifier symbol and make sure *imake* defines it so *cpp* can determine which vendor file to use.

Thus, to port *imake* to your machine, there are three requirements you must satisfy:

1. You need a vendor file that describes your system and contains any information needed to override default values of parameters specified in the other configuration files.

2. The configuration files must contain a block of code that "recognizes" your system type and selects the correct vendor file. The block is called the **vendor block** and the system-identifier symbol that activates it is called the **trigger** symbol. In the X11 configuration files, the vendor blocks are located near the beginning of *Imake.tmpl*.

3. *imake* must make sure the trigger symbol is defined when *cpp* starts up if *cpp* does not predefine it.

We'll discuss how to meet these requirements by porting *imake* to systems built by the hypothetical vendor Brand XYZ, as well as what to look for if *imake* has already been ported to your type of system but you can't get it to work on your machine.

Write the Vendor File

For a new port of *imake*, you have to write the vendor file from scratch. For Brand XYZ systems, we will write a file *brandxyz.cf*. If there is already a vendor file for your system type, you need to fix the vendor file you do have.

There are a few symbols you must define in the vendor file:

- Define the major and minor release numbers for your operating system. For instance, if your OS is at release 3.41, write this:

  ```
  #define OSMajorVersion 3
  #define OSMinorVersion 41
  ```

  The OS numbers are used when you need to select parameter values that vary for different releases of the system. Define them even if you do not use them anywhere else in the vendor file because *Imakefile* writers sometimes need to know the release numbers.

- If your system is based on System V Release 2 or 3, define `SystemV`:

```
#define SystemV YES
```

If your system is based on System V Release 4, define `SystemV4`:

```
#define SystemV4 YES
```

If your system is not based on System V, do not define either symbol.

The remaining contents of the vendor file are largely determined by whether the default parameter values provided in the template and project files are appropriate for your system. Provide override values in the vendor file for those that are not.

To some extent, you find out which defaults to override by trial and error: write the vendor file and try to build the distribution. If the build fails because a parameter value is incorrect, put the correct value in the vendor file and try again. However, you can minimize trial and error by taking advantage of the experience of those who've gone before you. Existing vendor files serve as a guide to help you figure out what should go in your own vendor file by giving you some idea of the parameters most likely to need special treatment (especially if any existing files are for OS's similar to yours).

Write the Vendor Block

The vendor block for your system goes in the vendor block section of *Imake.tmpl* and is activated by the trigger symbol. The SunOS and Ultrix blocks are shown below. Use them as a guide for writing your own block.

```
#ifdef sun
#define MacroIncludeFile <sun.cf>
#define MacroFile sun.cf
#undef sun
#define SunArchitecture
#endif /* sun */

#ifdef ultrix
#define MacroIncludeFile <ultrix.cf>
#define MacroFile ultrix.cf
#ifdef vax
#undef vax
#define VaxArchitecture
#endif
#ifdef mips
#undef mips
#define MipsArchitecture
#endif
#undef ultrix
#define UltrixArchitecture
#endif
```

The first line of each block tests the trigger symbol. If it is not defined, the block is skipped. Otherwise, the block does three things:

1. It defines the name of the vendor file. The name is defined two different ways because it is used in different contexts later.
2. It undefines the trigger symbol and any other OS- or processor-specific symbols that *cpp* might have predefined.
3. It defines a vendor-specific OS architecture symbol. If the OS runs on more than one type of processor, a processor-specific architecture symbol is usually defined, too (e.g., `VaxArchitecture`, `MipsArchitecture` in the Ultrix vendor block). Architecture symbols are useful in Imakefiles when it is necessary to know the OS or hardware type.

Following this model, you can write the vendor block for Brand XYZ systems using `brandxyz` as the trigger symbol and `BrandXYZArchitecture` as the architecture symbol:

```
#ifdef brandxyz
#define MacroIncludeFile <brandxyz.cf>
#define MacroFile brandxyz.cf
#undef brandxyz
#define BrandXYZArchitecture
#endif
```

After you write the vendor block, document the trigger symbol that identifies your system type by putting a definition for `BootstrapCFlags` in your vendor file. The following line goes in *brandxyz.cf*:

```
#define BootstrapCFlags -Dbrandxyz
```

You now have the vendor file and the vendor block written; all you need is *imake*.

Configure imake

imake is compiled using a minimal hand-written file *Makefile.ini*. You want the *Makefile.ini* that is in the *config* directory, not the one in the distribution root directory. (If you see a *Makefile* in the directory in which you are building *imake*, ignore it, as it was not configured on your machine.)

Makefile.ini builds *imake* in two steps because some systems require special flags to ensure proper compilation of all but the most trivial programs. *imake* is not especially complex, but it is not trivial, either, so a small helper program *ccimake* is compiled first. *ccimake* is designed to be simple enough to compile with no special treatment, and it figures out any extra flags needed to get *imake* to compile without error on your platform. Those flags are added to the *imake*-building command.

To build *ccimake* and *imake*, you run the following *make* command (do not do this yet; we won't be ready for a few pages):

```
% make -f Makefile.ini BOOTSTRAPCFLAGS="flags"
$(CC) -o ccimake $(CFLAGS) ccimake.c
$(CC) -c $(CFLAGS) `./ccimake` imake.c
$(CC) -o imake imake.o
```

The trigger is specified in the value of BOOTSTRAPCFLAGS.[4] For existing ports, determine *flags* from Table B−1. For new ports, use −D*trigger* (e.g., −*Dbrandxyz* for Brand XYZ systems).

Makefile.ini incorporates the value of BOOTSTRAPCFLAGS into CFLAGS and generates the three commands just shown. The first compiles *ccimake*. The second invokes *ccimake* and includes the result in the arguments passed to the command that compiles *imake.o*. The third produces the *imake* executable.

The trigger symbol passed in through BOOTSTRAPCFLAGS is used to determine the platform type, but in different ways for each program. The key to understanding trigger use is *imakemdep.h*. This header file is included by the source for *ccimake* and *imake* and has one part for each. (It is also used by *makedepend* and has a third part for that program; we will get to that shortly.)

The contents of *imakemdep.h* are organized like this:

```
#ifdef CCIMAKE
        /* part 1 (for ccimake) */
#else
```

[4] If your *cpp* predefines the trigger, BOOTSTRAPCFLAGS can be empty:

```
% make -f Makefile.ini BOOTSTRAPCFLAGS=""
```

However, it does not hurt to specify the trigger explicitly.

```
#   ifndef MAKEDEPEND
        /* part 2 (for imake) */
#   else
        /* part 3 (for makedepend) */
#   endif
#endif
```

The source for *ccimake* defines `CCIMAKE` before including *imakemdep.h* so that part 1 is processed. *makedepend* source defines `MAKEDEPEND` so that part 3 is processed. *imake* source does not define either symbol, so part 2 is processed.

For a new port of *imake*, you must add the proper information for your system to each part of *imakemdep.h*. For existing ports you need to verify that the information already there is correct, and fix it if it is not. The following discussion shows how to set up each part of *imakemdep.h*.

imakemdep.h—Part 1 (for ccimake)

The first part of *imakemdep.h* consists of a bunch of `#ifdef`/`#endif` blocks. Each of them defines `imake_ccflags` as a string containing the flags needed to get *imake* to compile on a particular platform. The proper definition is selected according to the trigger symbol that is defined when *ccimake* is compiled.

Some flags commonly specified in the definition of `imake_ccflags` are `−DSYSV` (System V, release 2 or 3), `−DSVR4` (System V, release 4), or `−DUSG` to indicate USG systems. For instance, if Brand XYZ systems are System V-based, specify the following:

```
#ifdef brandxyz
#define imake_ccflags "-DSYSV"
#endif
```

You might need more than one flag (see the `hpux` block for a particularly unpleasant example). Note that all flags are specified in a single string.

If no special flags are necessary to compile *imake*, there need not be any block for your system, and `imake_ccflags` is given a default definition (currently `−O`).

If you do not know whether *ccimake* needs to produce any special flags or not, experiment by trying to compile *imake* by hand. Once you figure out how to do it, make your knowledge explicit by defining `imake_ccflags` appropriately in *imakemdep.h*.

imakemdep.h—Part 2 (for imake)

The second part of *imakemdep.h* contains four sections. Each of them poses a question:

1. Do you have a working *dup2()* system call? If not, define a workaround for it.
2. Does your *cpp* collapse tabs in macro expansions? If not, define `FIXUP_CPP_WHITESPACE` or your Makefiles will not have tabs where necessary and the `@@\` sequences will not be stripped out properly.
3. Do you want to use the default preprocessor */lib/cpp*? If not, choose an alternate by defining `DEFAULT_CPP`. The value of `DEFAULT_CPP` must be a full pathname and only a pathname (no arguments). You can specify an alternate preprocessor at *imake* runtime by setting the `IMAKECPP` environment variable, but it is better to compile it in using `DEFAULT_CPP` so *imake* knows the correct value automatically. This may sound nonportable, and it is, because on your machine you want *imake* to know where *cpp* is so you do not have to tell it.
4. Are there arguments you want *imake* to pass to *cpp* automatically? This section initializes a string array `cpp_argv[]` with those arguments. The most important entry is a definition for the trigger symbol used to select the correct vendor block in *Imake.tmpl*, but other entries may be necessary, too.

Of the four sections of *imakemdep.h* that apply to *imake*, the one that sets up the string array `cpp_argv[]` is the most complex. It contains a set of `#ifdef`/`#endif` blocks that adds entries to the array. The most common construction looks like this:

```
#ifdef trigger
    "-Dtrigger",
#endif
```

If `trigger` is defined when *imake* is compiled, it causes a definition of itself to be passed to *cpp* when *imake* executes. (If your *cpp* predefines the trigger, you may not see an instance of this construction for your system, but there is no harm in adding one explicitly.)

For Brand XYZ systems, add the following entry to `cpp_argv[]`:

```
#ifdef brandxyz
    "-Dbrandxyz",
#endif
```

This causes *imake* to pass *−Dbrandxyz* to *cpp*, allowing *cpp* to select the Brand XYZ vendor block as it processes the configuration files.

If you need to pass more than one argument to *cpp*, add the definitions for each argument as separate strings:

```
#ifdef brandxyz
    "-Dbrandxyz",
    "-DSYSV",
#endif
```

imakemdep.h—Part 3 (for makedepend)

The third part of *imakemdep.h* applies to the compiled version of *makedepend*. (This has nothing to do with compiling *imake*, but since we're talking about *imakemdep.h*, we might as well cover it here.)

This part of *imakemdep.h* puts entries in the `predefs[]` string array based on which of various system- and compiler-related definitions are predefined by *cpp*. *makedepend* uses this information to be smart about which header files to pay attention to when it generates header file dependencies. Consider the following C program fragment:

```
#ifdef ultrix
#include "inca.h"
#else
#include "incb.h"
#endif
```

makedepend knows to generate a dependency for *inca.h* and not *incb.h* if `ultrix` is defined, and the other way around if it is undefined; a dumb dependency generator has to assume dependencies on both files in either case.

For existing ports, you can leave this part of *imakemdep.h* alone. For a new port, add symbols appropriate for your system if they are not already listed among the `predefs[]` initializers. First, find the lines that end the `predefs[]` initialization:

```
    /* add any additional symbols before this line */
    {NULL, NULL}
```

Then add new entries before those lines for any symbols predefined by your *cpp*:

```
#ifdef sym1
    {"sym1", "1"},
#endif
#ifdef sym2
```

```
    {"sym2", "1"},
#endif
    /* add any additional symbols before this line */
    {NULL, NULL}
```

Build imake

You're ready to build *imake*. This should be simple if you have set up the vendor file, vendor block, and *imakemdep.h* correctly. First remove the remains of any previous failed build attempts:

```
% make -f Makefile.ini clean
rm -f ccimake imake.o imake
rm -f *.CKP *.ln *.BAK *.bak *.o core errs ,* *~ *.a \
    tags TAGS make.log \#*
```

Then build *ccimake* and *imake*, substituting the bootstrap flags for your system into the following command:

```
% make -f Makefile.ini BOOTSTRAPCFLAGS="flags"
```

For Brand XYZ, the command looks like this:

```
% make -f Makefile.ini BOOTSTRAPCFLAGS="-Dbrandxyz"
making imake with BOOTSTRAPCFLAGS=-Dbrandxyz
cc -o ccimake -Dbrandxyz -O -I../include ccimake.c
cc -c -Dbrandxyz -O -I../include `./ccimake` imake.c
cc -o imake imake.o
```

After you build *imake*, test whether it is configured correctly, using *imake*'s *−v* option to tell it to echo the *cpp* command it executes. The normal input and output is of no interest here, so we also use *−T/dev/null* to provide an empty input template, and *−s/dev/null* to throw away the output:

```
% imake -v -T/dev/null -s/dev/null
cpp -I. -Uunix -Dbrandxyz
```

If you see a definition for your trigger symbol in the *cpp* command echoed by *imake* (*−Dbrandxyz* in the example above), *imake* is properly configured. If you do not see the definition, *imake* should still be okay if *cpp* predefines the trigger. Use the following command to check whether that is so (assuming the preprocessor *imake* uses is */lib/cpp*):

```
% echo "trigger" | /lib/cpp
```

If the trigger does not appear in the output or turns into "1", *cpp* predefines it. If you see the trigger string literally in the output, *cpp* does not predefine it, and you need to reconfigure *imake* to pass *-D*`trigger` to *cpp* explicitly.

Build the Rest of the Distribution

Now that you have the configuration files properly set up to build *imake*, you should be able to execute the `World` operation. In the distribution root directory, run this command:

```
% make World BOOTSTRAPCFLAGS="flags" >& make.world
```

This will rebuild *imake* and also build the rest of the distribution. After it finishes, see the section "Install the Software" for installation instructions.

If you can build *imake* but *make World* still fails, try building the distribution in stages to get an idea of where the problem lies. First, build the Makefiles:

```
% ./config/imake -I./config
% make Makefiles
```

If that works, try the following:

```
% make clean
% make depend
% make
```

If *make depend* fails while trying to compile *makedepend*, skip it and try the next step. You can go back later and try to figure out why *makedepend* does not build.

If you can't get any of this to work, read *misc/Porting* to see if it contains any helpful advice.

If you have to modify the distribution to build it on your system, please send your changes to me at *dubois@primate.wisc.edu*.

imake versus Open Windows

The versions of *imake* and *xmkmf* distributed with Open Windows (versions 2 and 3, at least) are nonstandard versions modified specially for use with Open Windows. If the pathnames for your versions of *imake* and *xmkmf* have *openwin* in them (e.g., */usr/openwin/bin/imake*), you are better off installing the standard versions because the versions distributed with Open Windows do not work very well to configure non-Open Windows software.

There are three difficulties:

- The Open Windows modification to *xmkmf* is not correct (as evidenced by the periodic requests on the net for the patch to the patch).
- The Open Windows version of *imake* is particularly obdurate: when the environment variable `OPENWINHOME` is set, *imake* assumes the configuration files are in `${OPENWINHOME}`*/lib/config* and looks for them there before looking in any directory specified on the command line. This means the Open Windows configuration files take precedence over any others you might be trying to use. You can defeat this behavior by setting the environment variable `IMAKEINCLUDE` to *`−Ipath`* (where *`path`* is the directory in which you keep your configuration files). But that is undesirable because using `IMAKEINCLUDE` makes successful configuration activities depend on something hidden in your environment, rather than something explicit in the configuration files. It is also difficult to switch between different sets of files when you use `IMAKEINCLUDE` because you must keep changing its value.
- The configuration files distributed with Open Windows contain no information about the location of Open Windows header files, libraries, etc.

It may be that these problems have been rectified by the time you read this. If not, you can work around them as follows:

- Throw away the Open Windows versions of *imake* and *xmkmf*.
- Follow the instructions in Chapter 9, *Coordinating Sets of Configuration Files*, for adding Open Windows-specific parameter values to the configuration files to create a set of configuration files that is Open Windows aware.
- Use the standard version of *imake* and the bootstrapper *imboot* to bootstrap Makefiles using the Open Windows configuration files.
- Use the standard version of *xmkmf* to bootstrap Makefiles using the X11 configuration files.