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 *itools.tar.gz* or *itools.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 10, *Coordinating Sets of Configuration Files*, and Chapter 15, *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 (X11R6.1, public patch 1 at the time of writing), but some programs are not part of the standard X11 distribution, e.g., *imboot*, *msub*, and *imdent*.

Distribution Layout

The software is distributed as a *gzip*ed or *compress*ed *tar* file *itools.tar.gz* or *itools.tar.Z*. Retrieve the distribution and unpack it according to the instructions in Appendix A. This will create an *itools* directory and several subdirectories. You should then familiarize yourself with the distribution's structure and contents. Move into the distribution root so you can look around:

```
% cd itools
```

The distribution root directory contains an *include* subdirectory in which some header files needed to compile *imake* and *makedepend* are located. The *itools* directory also contains a *config* subdirectory under which the rest of the configuration software is located. *config* is divided into the following subdirectories:

cf	The X11 configuration files.
imake	The source for *imake*.
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.
util	Source for *xmkmf*, *mkdirhier*, *bsdinst*, *install.sh*, *lndir*, *mkshadow*, and some other miscellaneous scripts, such as *which* and a script version of *makedepend* (if you can't get the C version to work).
extras	Source for *imboot*, *msub*, and *imdent*.
misc	Miscellaneous bits and pieces: portions of the X11 Release Notes that pertain to *imake*; any errors that have been found in this appendix 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.

Finding out What You Already Have

Survey the landscape of your system, since some or all of the software you need may already be present on your machine. For instance, if you are using a workstation running X11, there is a good chance the configuration files and some of the programs are already installed, because X11 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). In the following example, *which* tells you that *imake* is installed in */usr/local/bin*:

```
% which imake
/usr/local/bin/imake
```

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

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

If you don't have *which*, you can use either of the scripts *which.sh* or *which.csh* supplied in the *util* directory. If *which* doesn't find *imake*, try looking in */usr/local/bin*, */usr/bin*, */usr/bin/X11*, or in any */usr/X11Rn/bin* directory you may have. You can also look in */var/X11Rn/bin*, */opt/X11Rn/bin*, or */local/X11Rn/bin*. If you have OpenWindows, look for */usr/ openwin/bin*.

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

- Use the X11R6.1 version of *imake* if you have an earlier version.
- The current versions of *makedepend* and *xmkmf* each understand a −*a* option. Older versions (distributed prior to X11R5) do not. The current version of *makedepend* also has improved parsing of *cpp* preprocessor conditional directives, which results in more accurate results than with earlier versions. Install the versions in the distribution to update your system.
- The versions of *imake* and *xmkmf* distributed with Sun systems are nonstandard and modified specially for use with OpenWindows. If the pathnames for your versions of *imake* and *xmkmf* have *openwin* in them (e.g., */usr/openwin/bin/imake*), you may be better off installing the standard X11 versions because the OpenWindows versions do not work very well to configure non-OpenWindows software. See Appendix J, *Using imake with OpenWindows*, for further discussion of this issue.

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 X11 files is usually */usr/X11R6.1/lib/X11/config*, */usr/X11R6/lib/X11/config*, or */usr/lib/X11/config* for X11R6.1, X11R6, or X11R5. If you can't find the X11 files but you have *xmkmf* installed, look at it to see where it expects to find the files.

Preparing To Install the Distribution

The first thing you should do is read through the *README* file in the *itools* directory. Then, 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/X11R6.1/bin	Location of programs
/usr/X11R6.1/lib/X11/config	Location of the X11 configuration files
/usr/local/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/X11R6.1/lib/X11/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/cf/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/cf/site.def*:

```
#ifndef ConfigDir
#define ConfigDir /var/X11R6.1/lib/X11/config
#endif
```

- To change the configuration root directory used by *imboot*, edit this line in *config/extras/Imakefile*:

```
CONFIGROOTDIR = /usr/local/lib/config
```

  Change it to this:

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

The *imdent* script is built assuming that *perl* is installed as */usr/local/bin/perl* on your system. If that is incorrect, edit the *Imakefile* in *config/extras* to have the correct value for `PERLPATH`.

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.* 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/config/X11R6.1	Location of the X11 configuration files
/u/you/lib/config	Location of the *imboot* configuration root directory

* Yes, I know that "sympathetic site administrator" is oxymoronic.

Building the Distribution—Quick Instructions

This section contains quick instructions for building the distribution. If they don't work, read the section "Building the Distribution—Detailed Instructions."

Change into the *config/cf* directory and look around; you'll see the configuration files listed below:

```
% cd config/cf
% ls
Amoeba.cf     Server.tmpl   hp.cf         necLib.rules  site.def
DGUX.cf       Threads.tmpl  hpLib.rules   necLib.tmpl   site.sample
FreeBSD.cf    Win32.cf      hpLib.tmpl    noop.rules    sony.cf
Imake.cf      Win32.rules   ibm.cf        oldlib.rules  sun.cf
Imake.rules   WinLib.tmpl   ibmLib.rules  osf1.cf       sunLib.rules
Imake.tmpl    apollo.cf     ibmLib.tmpl   osfLib.rules  sunLib.tmpl
Imakefile     bsd.cf        linux.cf      osfLib.tmpl   sv4Lib.rules
Library.tmpl  bsdLib.rules  lnxLib.rules  pegasus.cf    sv4Lib.tmpl
Makefile      bsdLib.tmpl   lnxLib.tmpl   sco.cf        svr4.cf
Mips.cf       bsdi.cf       luna.cf       scoLib.rules  ultrix.cf
NetBSD.cf     convex.cf     macII.cf      sequent.cf    usl.cf
Oki.cf        cray.cf       moto.cf       sgi.cf        x386.cf
Project.tmpl  fujitsu.cf    ncr.cf        sgiLib.rules  xf86.rules
README        generic.cf    nec.cf        sgiLib.tmpl   xfree86.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 find no vendor file for your system, you may need to write one using the detailed instructions. (First look in the *config/misc* directory, though. It contains instructions for some systems that aren't supported in the standard X11 distribution.)

Find the definitions of `OSMajorVersion`, `OSMinorVersion`, and `OSTeenyVersion` in your vendor file. They represent the major, minor, and subminor release numbers for your operating system. Check that they are correct, and change them if they are not. Suppose *linux.cf* specifies Linux 1.2.11:

```
#ifndef OSMajorVersion
#define OSMajorVersion      1
#endif
#ifndef OSMinorVersion
#define OSMinorVersion      2
#endif
#ifndef OSTeenyVersion
#define OSTeenyVersion      11
#endif
```

If you're running Linux 1.2.8, the major and minor numbers are correct, but you must change the subminor number to 8:

```
#ifndef OSTeenyVersion
#define OSTeenyVersion      8
#endif
```

If your OS release is defined only by major and minor numbers, your vendor file may not contain any definition for `OSTeenyVersion`.

Look over *site.def* to see whether you want to change anything. In particular, to use *gcc* (version 2.x.x) as your C compiler, define `HasGcc2` as `YES` by uncommenting the `#define` that is already there.

The distribution should build on many systems at this point. In the distribution root (the *itools* directory) execute the following command:

```
% make World >& world.log
```

The command just shown uses *csh* redirection. If you are using a shell from the Bourne shell family (*sh*, *ksh*, *bash*), use this command instead:

```
$ make World > world.log 2>&1
```

For Windows NT, use this command:

```
nmake World.Win32 > world.log
```

When the command terminates, look through *world.log* to see whether or not the `World` operation completed without error. If everything looks okay, go to the next section, "Installing the Software." Otherwise you may need to specify `BOOTSTRAPCFLAGS`. 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 examine 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 detailed instructions.

Table B−1: Bootstrap Flags for Various Systems

Vendor File	Bootstrap Flags	Remark
Amoeba.cf	`-DAMOEBA -DCROSS_i80386 -DCROSS_COMPILE`	For i80386 architecture
	`-DAMOEBA -DCROSS_mc68000 -DCROSS_COMPILE`	For Sun 3 architecture
	`-DAMOEBA -DCROSS_sparc -DCROSS_COMPILE`	For SPARC architecture
DGUX.cf	`-DDGUX`	
FreeBSD.cf	`-D__FreeBSD__`	
Mips.cf	`-DMips`	
NetBSD.cf	`-D__NetBSD__`	
Oki.cf	`-DOki`	
Win32.cf	`-DWIN32`	
apollo.cf	`-Dapollo`	
bsd.cf	`-DNOSTDHDRS`	
bsdi.cf	`-D__bsdi__`	
convex.cf	`-D__convex__ -tm c1`	
cray.cf	`-DCRAY`	possibly `-D_CRAY`?
fujitsu.cf	`-D__uxp__`	For SPARC architecture
	`-D__sxg__`	For mc68000 architecture

Table B−1: Bootstrap Flags for Various Systems (continued)

Vendor File	Bootstrap Flags	Remark
hp.cf	`−Dhpux`	
ibm.cf	*varies*	Will include `−Dibm`
linux.cf	`−Dlinux`	
luna.cf	`−Dluna`	
macII.cf	`−DmacII`	
moto.cf	`−DMOTOROLA −DSVR4`	If SVR4 conformant
	`−DMOTOROLA −DSYSV`	If not SVR4 conformant
ncr.cf	`−DNCR`	
nec.cf	`−DNEC`	
osf1.cf	`−D__osf__`	
pegasus.cf	`−DM4310 −DUTEK`	
sco.cf	*none*	
sequent.cf	`−Dsequent`	For Dynix 3
	`−D_SEQUENT_`	For Dynix Ptx
sgi.cf	`−DSYSV`	For IRIX 3.x, 4.x
	`−DSVR4`	For IRIX 5.x and up
sony.cf	`−Dsony`	
sun.cf	`−Dsun −DNOSTDHDRS`	For SunOS before 4.1
	`−Dsun`	For SunOS 4.1 up to 5.0
	`−Dsun −DSVR4`	For SunOS 5.0 and up
svr4.cf	`−DSVR4 −Di386`	For i386 architecture
	`−DSVR4`	For non-i386 architecture
ultrix.cf	`−Dultrix`	
usl.cf	`−DUSL`	
x386.cf	*varies*	Will include `−DSYSV386`

You should also look in the *config/misc* directory to see if there are any files pertaining to your system. Some patches to make the distribution build on older releases of OSes are contained there, and you may need to apply one of them.

In the distribution root, execute the following command, replacing `flags` with the appropriate bootstrap flags for your system:

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

Examples:

```
% make World BOOTSTRAPCFLAGS="−Dsun −DSVR4" >& world.log
% make World BOOTSTRAPCFLAGS="−Dhpux" >& world.log
```

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

Installing the Software

If you want to install everything, use the following command in the distribution root directory (you may need to be *root* to do this):

```
% make install
```

Otherwise, change into individual directories under the *config* directory 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 don't want to install.

- In *cf*, install the X11 configuration files:

  ```
  % make install.cffiles
  ```

- In *imake*, install *imake*:

  ```
  % make install.imake
  ```

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

- In *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 the *util* directory.

- In *util*, install various scripts:

  ```
  % make install.xmkmf
  % make install.mkdirhier
  ```

  If you have a System V version of *install* instead of a BSD-compatible one, the distribution contains two scripts *bsdinst* and *install.sh* that you can use instead. Some systems use one, others use the other. You can tell which one to install by watching what commands *make* has been executing as you've been installing the software above:

  ```
  % make install.bsdinst      (or)
  % make install.install.sh
  ```

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

  ```
  % make install.makedepend
  ```

- In *extras*, install *imboot*, *msub*, and *imdent*:

  ```
  % make install.imboot
  % make install.msub
  % make install.imdent
  ```

Building the Distribution—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 you just tried the quick instructions and they didn't 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 how the distribution expects your system to be set up and how it actually is set up. Make sure you've read through the quick instructions first because they contain 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're trying to produce.

If you're porting *imake* to a system on which it's not currently supported, use one of the following locators to see if there's any information for your type of system:

```
http://www.primate.wisc.edu/software/imake-stuff/OS
ftp://ftp.primate.wisc.edu/software/imake-stuff/OS
```

If you're not running UNIX, you should also read Chapter 18, *Using imake on Non-UNIX Systems*.

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 the days of 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, so ANSI *cpp*s tend to provide few symbols that can be used to determine the system type. 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, you must satisfy three requirements:

1. 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 in the *Imake.cf* file.
2. 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.
3. *imake* must make sure the trigger symbol is defined when *cpp* starts up if *cpp* doesn't 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.

Writing the Vendor Block

The vendor block for your system goes in the *Imake.cf* file and is activated by the trigger symbol. You should examine that file for examples of how to write a vendor block. The Linux block is relatively simple:

```
#ifdef linux
# define MacroIncludeFile <linux.cf>
# define MacroFile linux.cf
# undef linux
# define LinuxArchitecture
# define i386Architecture
# undef i386
#endif /* linux */
```

The SunOS block is more complex:

```
#ifdef sun
# define MacroIncludeFile <sun.cf>
# define MacroFile sun.cf
# ifdef SVR4
#  undef SVR4
#  define SVR4Architecture
# endif
# ifdef sparc
#  undef sparc
# define SparcArchitecture
# endif
# ifdef mc68000
#  undef mc68000
# define Sun3Architecture
# endif
# ifdef i386
#  undef i386
```

```
#  define i386Architecture
# endif
# undef sun
# define SunArchitecture
#endif /* sun */
```

The first line of each block tests the trigger symbol. If it is undefined, 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., `Sparc–Architecture`, `Sun3Architecture`, and `i386Architecture` in the SunOS 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
```

Writing the Vendor File

For a new port of *imake*, you have to write the vendor file from scratch. For Brand XYZ systems, we'll 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, and, if necessary, the subminor number. For instance, if your OS is at release 3.4.1, write this:

```
#ifndef OSMajorVersion
#define OSMajorVersion 3
#endif
#ifndef OSMinorVersion
#define OSMinorVersion 4
```

```
#endif
#ifndef OSTeenyVersion
#define OSTeenyVersion 1
#endif
```

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 don't 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 isn't based on System V, don't define either symbol.

The remaining contents of the vendor file are largely determined according to whether or not 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 a minimal vendor file containing only the symbols described above, then 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. If any existing files are for operating systems that are similar to yours, your vendor file is likely to be similar to those files.

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

Configuring imake

The source for *imake* is in the *imake* directory. *imake* is compiled using a minimal hand-written file *Makefile.ini*. (You want the *Makefile.ini* that is in the *imake* directory, not the one in the distribution root directory.) If you see a *Makefile* in the *imake* directory, 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* isn't especially complex, but it isn't 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'll run the following *make* command (but don't run it yet; we won't be ready for a few pages):

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

The trigger is specified in the value of BOOTSTRAPCFLAGS. For existing ports, determine *flags* from Table B–1. For new ports, use *–Dtrigger* (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* in the *config/imake* directory. 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'll get to that shortly.)

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

```
#ifdef CCIMAKE
        /* part 1 (for ccimake) */
#else
# 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 doesn't 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 isn't. The following discussion shows how to set up each part of *imakemdep.h*.

---

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

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

However, it doesn't hurt to specify the trigger explicitly.

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 based on System V Release 4, specify the following:

```
#ifdef brandxyz
#define imake_ccflags "-DSVR4"
#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 don't know whether or not *ccimake* needs to produce any special flags, 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:

- Do you have a working *dup2()* system call? If not, define a work-around for it.
- Does your *cpp* collapse tabs in macro expansions? If not, define FIXUP_CPP_WHITE-SPACE or your Makefiles will not have tabs where necessary and the @@\ sequences will not be stripped out properly.
- 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 non-portable, and it is. But that's okay, because on the particular machine where you're building *imake*, you want it to know exactly where *cpp* is so you don't have to tell it.
- 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",
    "-DSVR4",
#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 "headera.h"
#else
#include "headerb.h"
#endif
```

makedepend knows to generate a dependency for *headera.h* and not *headerb.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}
```

Building imake

You're ready to build *imake*. This should be simple if you have set up the vendor block, vendor file, 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 files are of no interest here, so we use −*T/dev/null* and −*f/dev/null* to provide an empty input template and target description file, and −*s/dev/null* to throw away the output:

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

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 don't 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 doesn't appear in the output or turns into "1", *cpp* predefines it. If you see the trigger string literally in the output, *cpp* doesn't predefine it, and you need to reconfigure *imake* to pass −*Dtrigger* to *cpp* explicitly.

Building 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" >& world.log
```

This will rebuild *imake* and also build the rest of the distribution. After it finishes, see the section "Installing 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 from the distribution root:

```
% ./config/imake/imake -I./config/cf
% 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 step following it. You can go back later and try to figure out why *makedepend* doesn't build.

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