[The document below is an abridged version of the original: those parts with no relevance to *imake* have been excised. Deletions are marked by ‘‘[deleted]’’. The original may be obtained as the file *mit/RELNOTES.ms* in the X11R5 distribution. – Paul DuBois, *dubois@primate.wisc.edu*]

X Window System, Version 11, Release 5

Release Notes

MIT X Consortium staff

MIT Laboratory for Computer Science

Copyright © 1991 by the Massachusetts Institute of Technology.

Permission to use, copy, modify, and distribute this document for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of MIT not be used in advertising or publicity pertaining to this document without specific, written prior permission. MIT makes no representations about the suitability of this document for any purpose. It is provided ‘‘as is’’ without express or implied warranty.

X Window System is a trademark of MIT.

This document describes how to build, install, and get started with Release 5 of the X Window System from MIT and gives a brief overview of the contents of the release.

1. For the Impatient Explorer

For those of you who will try to build the distribution without reading the entire release notes first, here is a quick summary of what to do.

If you want to build with **gcc**, edit **mit/config/site.def** by uncommenting the **HasGcc** line.

If you want to install into somewhere other than **/usr/bin/X11**, **/usr/include/X11**, etc., edit **mit/config/site.def** by uncommenting the **ProjectRoot** lines and changing "/usr/X11R5" to whatever directory you want to install into. (Do *not* use **DESTDIR**.)

Check the appropriate **mit/config/***vendor***.cf** file to make sure that **OSMajorVersion** and **OSMinorVersion** are set correctly (change them if necessary).

Find the **BootstrapCFlags** line, if any, in the *vendor***.cf** file. If there isn't one, **cd** to the **mit** directory and type:

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

If there is a **BootstrapCFlags**, take its value[1] and type:

make World BOOTSTRAPCFLAGS="*value*" >& world.log

Do not call the output file ‘‘make.log’’, or it will be deleted. If the build is successful, you can install most of it with:

```
make install >& install.log
```

You can install man pages with:

1. If you are using the **x386.cf** file, you will have to compute the correct value.

    make install.man >& man.log

You can install lint libraries (if desired) with:

    make install.ln >& lintlib.log

If things fail, read the rest of the release notes.

2. Brief Overview of the Distribution

[deleted]

2.1. Structure of the MIT Sources

[deleted]

3. Building the Release

The core distribution (code under the **mit** directory) has been built and tested at MIT on the following systems:

AIX 3.1.5, on IBM RS/6000
Apollo SR10.3 (very minimal testing, bsd4.3 only)
AT&T Unix System V Release 4 V2, on AT&T WGS6386
A/UX 2.0.1
HP-UX 7.0, on HP9000/s300
IRIX 4.0
Mach 2.5 Version 1.13, on OMRON Luna 88k
NEWS-OS 4.1, on Sony NWS-1850
NEWS-OS 5.0U, on Sony NWS-3710
SunOS 4.1.1, on Sun 3, Sparc 1, and Sparc 2
Ultrix-32 4.2, VAX and RISC
UNICOS 5.1
UTek 4.0
VAX 4.3bsd (with unknown local changes)

In somes cases, we have not used the most recent version of the operating system (sorry). Support for earlier versions of the operating systems listed is not claimed, and not guaranteed.

In addition to the systems above, support has been provided by vendors for:

AIX 2.2 and AOS 4.3, on IBM RT
AIX 1.2.1, on IBM PS/2
ConvexOS V9.0
DG/UX 4.32
INTERACTIVE UNIX Version 2.2.1
Mach 2.5 Version 1.40, on OMRON Luna 68k
Motorola R32V2/R3V6.2 and R40V1
RISCOS 4.50
UNIOS-B 4.3BSD UNIX: 2.00
Unix System V/386 Release 3.2, on ESIX, SCO, and AT&T (“work in progress”)
Unix System V/386 Release 4.0, on DELL

3.1. Unpacking the Distribution

[deleted]

3.2. Symbolic Link Trees

[deleted]

3.3. Setting Configuration Parameters

You will notice that few if any of the subdirectories under **mit** contain a **Makefile**, but they do contain an **Imakefile**. The **Imakefile** is a template file used to create a **Makefile** containing build rules and variables appropriate for the target machine. The **Makefile** is generated by the program **imake**. Most of the configuration work prior to building the release is to set parameters so that **imake** will generate correct files.

The directory **mit/config** contains configuration files that control how the distribution is built. On systems directly supported by this distribution, only minimal editing of these files should be necessary. If your system is not supported by the distribution but conforms to ANSI C and POSIX.1 and has socket-style networking, then you should be able to build a new configuration file relatively easily. Otherwise, edits to many files throughout the system may be necessary. We only deal with minor editing for supported systems here.

The main files to be concerned with in the **mit/config** directory are **site.def** and one of the *vendor***.cf** files. The **site.def** file should be used for most site-specific configuration customizations. The **.cf** file should normally only need to be edited if you are using a different release of the operating system.

3.3.1. The vendor.cf File

Find the appropriate **.cf** file from this table:

AIX	ibm.cf
AOS	ibm.cf
Apollo	apollo.cf
AT&T Unix SVR4	att.cf
A/UX	macII.cf
BSD	bsd.cf
ConvexOS	convex.cf
DG/UX	DGUX.cf
HP-UX	hp.cf
INTERACTIVE	x386.cf
IRIX	sgi.cf
Mach (Luna)	luna.cf
Motorola	moto.cf
NEWS-OS	sony.cf
RISCOS	Mips.cf
SunOS	sun.cf
Ultrix	ultrix.cf
UNICOS	cray.cf
UTek	pegasus.cf
UNIOS-B	luna.cf
Unix System V/386	x386.cf

Look through this file, and check the **OSMajorVersion** and **OSMinorVersion** values. The numbers have been preset to what was tested at MIT or what was supplied by the vendor. If the version numbers match the operating system you are currently running, all is well. If they do not, you will need to edit to file to make them correct. In a few cases (specifically changing UNICOS from 5.1 to 6.0) there should not be a problem in moving the version numbers forward to a newer release. However, if you are moving the version numbers backwards, or moving forward to a version that hasn't been pre-tested, you may have problems, and you have have to edit other parts of the file (and possibly other files) to get things to work.

You can browse through the rest of the items in the **.cf** file, but most of them you should not need to edit.

3.3.2. The site.def File

There are two main variables to set in the **site.def** file: **HasGcc** and **ProjectRoot**. If you are going to compile the distribution with **gcc**, find the line that looks like

```
/* #define HasGcc YES */
```

and remove the comment markers, turning it into

```
#define HasGcc YES
```

If you are sharing a single **site.def** across multiple systems, you can do something more complicated. For example, if you only want to use **gcc** on a Sun 3 (but not on Sparcs) you might use this:

```
#ifdef SunArchitecture
#define HasGcc mc68000
#endif
```

The most common error when using **gcc** is to fail to run the **fixincludes** script (from the **gcc** distribution) when installing **gcc**. Make sure you have done this before compiling the release. Another common error is likely to be using **gcc** ANSI C include files when the vendor operating system supplies correct ones. The **gcc** include files **assert.h**, **limits.h**, and **stddef.h** are prime candidates for not installing.

The **ProjectRoot** variable controls where the software will eventually be installed. The default as distributed for most systems is to install into ‘‘system’’ directories: **/usr/bin/X11**, **/usr/include/X11**, **/usr/lib**, and **/usr/man** (this is the behaviour when **ProjectRoot** is not defined). If you prefer to install into alternate directories, the simplest thing to do is to set **ProjectRoot**. Find the four **ProjectRoot** lines in the **site.def** file, and again remove the ‘‘/*’’ and ‘‘*/’’ comment markers that surround them. You will see a default choice for **ProjectRoot** of **/usr/X11R5**; if you don’t like that one, replace it with another. Assuming you have set the variable to some value */path*, files will be installed into */path*/bin, */path*/include/X11, */path*/lib, and */path*/man.

Note that in a few cases (**ibm.cf** and **x386.cf**) the vendor-supplied **.cf** file supplies a **ProjectRoot** by default. If you want to accept this one, do not uncomment the one in **site.def**; otherwise the one you place in **site.def** will override the default setting.

The directories where the software will be installed are compiled in to various programs and files during the build process, so it is important that you get the configuration correct at the outset. If you change your mind later, you will want to do a ‘‘make Everything’’ to rebuild correctly.

Notice that the **site.def** file was two parts, one protected with ‘‘#ifdef BeforeVendorCF’’ and one with ‘‘#ifdef AfterVendorCF’’. The file is actually processed twice, once before the **.cf** file and once after. About the only thing you need to set in the ‘‘before’’ section is **HasGcc**; just about everything else can be set in the ‘‘after’’ section.

There are a large number of parameters that you can modify to change what gets built and how it gets built. An exhaustive list and explanation will not be given here; you can browse through **mit/config/README** to see a list of parameters.

[rest of section deleted]

3.4. System Pitfalls

On a few systems, you are likely to have build problems unless you make some minor changes to the system. Naturally, you should exercise caution before making changes to system files, but these are our recommendations based on our experience.

On VAX Ultrix systems, you may find that **<stdlib.h>** contains declarations of **malloc**, **calloc**, and **realloc** with a return value of ‘‘void *’’. You may find this causes problems when compiling with a non-ANSI-C compiler, in which case a workaround is to change the return values to ‘‘char*’’ in the ‘‘#else’’ section.

Ultrix may not provide **<locale.h>** unless you load the Internationalization subset. You will need this file to compile the distribution (or else you will need to reset a configuration parameter, see below).

On SunOS systems, you may find that statically linking (when debugging) against both Xlib and the **libc** will result in unresolved symbols to dynamic linker functions, because Xlib contains calls to **wcstombs**. Either link dynamically against **libc**, or compile and link the stub routines in **mit/util/misc/dlsym.c**.

On Sun 3s, the default is to compile library files with no special floating point assumptions. If all of your Sun 3s have floating point hardware, you may want to change this, for better performance of Xlib color functions. For example, in the ‘‘after’’ section of your **site.def** file, you might add:

```
#if defined(SunArchitecture) && defined(mc68000)
#undef LibraryCCOptions
#define SharedLibraryCCOptions -f68881 -pipe
#endif
```

On AOS, you may find that **<stdarg.h>** is missing. In that case, you should be able to copy **mit/util/misc/rt.stdarg.h** to create the file.

On some System V/386 systems, you may find when using **gcc** in ANSI mode that there are inconsistent declarations between **<memory.h>** and **<string.h>**. In that case, you may find it convenient to remove **<memory.h>** and make it a link to **<string.h>**.

On some System V/386 systems, you may need to build and install a **dbm** library before building the X server and RGB database. One can be found in **contrib/util/sdbm**.

3.4.1. Internationalization

[deleted]

3.5. Typing “make World”

One more piece of information is required before building, at least on some systems: bootstrap flags. Look in your **.cf** file for a line of the form

#define BootstrapCFlags *value*

If there isn’t one things are simple, otherwise things are only slightly more complicated. If there is more than one (for example, in **ibm.cf**, **moto.cf**, and **sony.cf**), then you need to select the right one; it should be pretty obvious by the grouping according to operating system type. Note that on A/UX you only need this value if you are using **gcc**, and that on a Sun you only need this value if you are using an earlier version of the operating system.

If you are using **x386.cf**, you will have to ‘‘compute’’ the value from the information given in the file. You may also need to do other preparatory work; please read **mit/server/ddx/x386/README**.

If no value is required on your system, you can **cd** to the **mit** directory and start the build with:

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

If a value is required, start the build with:

make World BOOTSTRAPCFLAGS="*value*" >& world.log

You can call the output file something other than ‘‘world.log’’, but do not call it ‘‘make.log’’ because files with this name are automatically deleted during the ‘‘cleaning’’ stage of the build.

Because the build can take several hours to complete, you will probably want to run it in the background, and keep a watch on the output. For example:

```
make World >& world.log &
tail -f world.log
```

If something goes wrong, the easiest thing is to just start over (typing ‘‘make World’’ again) once you have corrected the problem. It is possible that a failure will corrupt the top-level **Makefile**.

If that happens, simply delete the file and recreate a workable substitute with:

```
cp Makefile.ini Makefile
```

When the build completes, examine the **world.log** file for errors. If you search for ':' (colon) characters, and skip the obvious compile lines, it is usually pretty easy to spot any errors.[2]

4. Installing the Release

Although it is possible to test the release before installing it, it is a lot easier to test after it has been installed. If everything is built successfully, you can install the software by typing the following as root, from the **mit** directory:

```
make install >& install.log
```

Again, you might want to run this in the background and use **tail** to watch the progress.

You can install the man pages by typing the following as root, from the **mit** directory:

```
make install.man >& man.log
```

You can install lint libraries (useful if your systems does does not have an ANSI C compiler) by typing the following as root, from the **mit** directory:

```
make install.ln >& lintlib.log
```

4.1. Setting Up xterm

[deleted]

4.2. Starting Servers at System Boot

[deleted]

5. Rebuilding the Release

You shouldn't need this right away, but eventually you are probably going to make changes to the sources, for example by applying public patches distributed by MIT. If only C source files are changed, you should be able to rebuild just by going to the **mit** directory and typing:

```
make >& make.log
```

If configuration files are changed, the safest thing to do is type:

```
make Everything >& every.log
```

"Everything" is similar to "World" in that it rebuilds every **Makefile**, but unlike "World" it does not delete the existing objects, libraries, and executables, and only rebuilds what is out of date.

Note that in both kinds of rebuilds you do not need to supply the **BootstrapCFlags** value any more, the information is already recorded.

6. Building Contributed Software

[deleted]

7. Filing Bug Reports

[deleted]

---

2. Searching for colon does not work particularly well on the RS/6000 because it appears in command lines when building shared libraries. Try searching for colon followed by space.

8. Public Fixes

[deleted]

9. Configuring for a New Architecture

Here is a very brief overview of the files that **imake** reads. All the files are in the **mit/config** directory, except for the **Imakefile** in the directory for which the **Makefile** is being created. The processing order is:

Imake.tmpl	variables not related specifically to X
site.def	site-specific BeforeVendorCF part
*.cf	machine-specific
*Lib.rules	shared library rules
site.def	site-specific AfterVendorCF part
Project.tmpl	X-specific variables
*Lib.tmpl	shared library variables
Imake.rules	rules
Imakefile	specific to the program or library
Library.tmpl	library rules
Server.tmpl	server rules

The indentation levels indicate what files include other files.

9.1. Imake.tmpl

The first part of **Imake.tmpl** determines which **.cf** file to include. If your **cpp** defines a unique symbol, that should be used to select the file. Otherwise, you should place a -D symbol definition in **BootstrapCFlags** in your **.cf** file and use that. The canonical code to add to **Imake.tmpl** is:

#ifdef *symbol*
#define MacroIncludeFile <*symbol*.cf>
#define MacroFile *symbol*.cf
#undef *symbol*
#define *Symbol*Architecture
#endif /* *symbol* */

9.2. imakemdep.h

You also need to edit the file **imakemdep.h**. There are three parts to this file. The first contains defines (beyond **BootstrapCFlags**) or compiler options that are required to get **imake** itself built the first time.

The next section is for **imake** itself. There is a hook in case your **cpp** collapses tabs down to single spaces. There is also a way to override the **cpp** to use. Finally, add specific defines to pass to **cpp** when processing configuration files.

The last section is for **makedepend**, to tell it about predefined symbols that will be used to control inclusion of header files.

9.3. vendor.cf

Most of the rest of your vendor-specific configuration information goes here. We won't try to tell you everything you need; study the other **.cf** files and copy from systems that are similar. One good rule to follow is to not define anything that will get the correct default value from somewhere else; this will make it easier to see what is special, and will make it easier for sites to customize in their **site.def**.

If you have shared libraries, the convention is to place the configuration rules and standard parameters in a file named *os***Lib.rules**, and to place version number parameters and **make** variables in a file named *os***Lib.tmpl**. Look at the existing files and mimic them.

9.4. Other Files

[deleted]

10. Writing Portable Code

In this section we give a brief introduction to using various header files to aid in writing portable code.

10.1. <X11/Xosdefs.h>

The file **<X11/Xosdefs.h>** defines symbols that describe the system environment for ANSI C and POSIX. We likely will extend it to other standards in the future. We have found these symbols useful in writing portable code, and hope that other writers of X software will use them as well. This file is not part of any X Consortium standard, it is simply part of our software distribution.

<X11/Xosdefs.h> can be included directly by a file, or it will be automatically included when you include **<X11/Xos.h>**.

The symbols in **<X11/Xosdefs.h>** tell when you can, for example, do

```
#include <stdlib.h>
```

without getting a ‘‘no such header file’’ error from the compiler. If the system provides a declaration for a function or value for a constant, it is important to use the system's definition rather than providing your own, particularly because you might not use function prototypes and the system might, or vice versa.

<X11/Xosdefs.h> currently controls two symbols: **X_NOT_STDC_ENV** and **X_NOT_POSIX**.

X_NOT_STDC_ENV means the system does not have ANSI C header files. Thus, for example, if **X_NOT_STDC_ENV** is not defined, it is safe to include **<stdlib.h>**. Do not confuse this symbol with **__STDC__**, which says whether the compiler itself supports ANSI C semantics. **X_NOT_STDC_ENV** is independent, and tells what header files it is safe to include.

Lack of the symbol **X_NOT_STDC_ENV** does *not* mean that the system has **<stdarg.h>**. This header file is part of ANSI C, but we have found it more useful to check for it separately because many systems have all the ANSI C files we need except this one. **__STDC__** is used to control inclusion of this file.

An example of using **X_NOT_STDC_ENV** might be to know when the system declares **getenv**:

```
#ifndef X_NOT_STDC_ENV
#include <stdlib.h>
#else
extern char *getenv();
#endif
```

We usually put the standard case first in our code, using ‘‘#ifndef’’.

X_NOT_POSIX means the system does not have POSIX.1 header files. Lack of this symbol does *not* mean that the POSIX environment is the default. You may still have to define **_POSIX_SOURCE** before including the header file to get POSIX definitions.[3]

An example of using **X_NOT_POSIX** might be to determine the type that **getuid** would be declared by in **<pwd.h>**:

```
#include <pwd.h>
#ifndef X_NOT_POSIX
   uid_t uid;
#else
   int uid;
```

3. We have found it very unfortunate that POSIX did not define a standard symbol that means ‘‘give me POSIX, plus any non-conflicting vendor-specific definitions’’.

```
    extern int getuid();
#endif
    uid = getuid();
```

Note that both of these symbols, when declared, state a non-compliance. This was chosen so that porting to a new, standard platform would be easier. Only non-standard platforms need to add themselves to **<X11/Xosdefs.h>** to turn on the appropriate symbols.

Not all systems for which we leave these symbols undefined strictly adhere to the relevant standards. Thus you will sometimes see checks for a specific O/S near a check for one of the Xosdefs.h symbols. However, we have found it most useful to label systems as conforming even if they have some holes in their compliance. Presumably these holes will become fewer as time goes on.

10.2. <X11/Xos.h>

In general, **<X11/Xos.h>** should be used instead of the following header files:

```
<string.h>
<strings.h>
<sys/types.h>
<sys/file.h>
<fcntl.h>
<sys/time.h>
<unistd.h>
```

This file is not part of any X Consortium standard, it is simply part of our software distribution.

Some common routines for which you need to include **<X11/Xos.h>** before using are:

```
index
rindex
strchr
strrchr
(all the other standard string routines)
gettimeofday
time
```

Data types and constants that should be obtained with **<X11/Xos.h>** are:

caddr_t
O_RDONLY
O_RDWR
(and other **open** constants)
R_OK
W_OK
X_OK
(and other **fcntl** constants)

Unfortunately, we did not create a header file for declaring **malloc** correctly, and it can be a bit tricky. You can use what we currently have by copying, for example, from **mit/lib/Xt/Alloc.c**:

```
#ifndef X_NOT_STDC_ENV
#include <stdlib.h>
#else
char *malloc(), *realloc(), *calloc();
#endif
#if defined(macII) && !defined(__STDC__)  /* stdlib.h fails to define these */
char *malloc(), *realloc(), *calloc();
#endif /* macII */
```

10.3. <X11/Xfuncs.h>

This file contains definitions of **bcopy**, **bzero**, and **bcmp**.[4] You should include this header in any file that uses these functions. This file is not part of any X Consortium standard, it is simply part of our software distribution.

10.4. <X11/Xfuncproto.h>

This file contains definitions for writing function declarations to get function prototypes to work right. It deals with ANSI C compilers as well as pre-ANSI C compilers that have parts of function prototypes implemented. This file is not part of any X Consortium standard, it is simply part of our software distribution.

For external header files that might get used from C++, you should wrap all of your function declarations like this:

```
_XFUNCPROTOBEGIN
function declarations
_XFUNCPROTOEND
```

When in doubt, assume that the header file might get used from C++.

A typical function declaration uses **NeedFunctionPrototypes**, like this:

```
extern Atom XInternAtom(
#if NeedFunctionPrototypes
    Display*        /* display */,
    _Xconst char*   /* atom_name */,
    Bool            /* only_if_exists */
#endif
);
```

If there are **const** parameters, use the symbol **_Xconst** instead, as above. If it is plausible to pass a string literal to a char* parameter, then it is a good idea to declare the parameter with **_Xconst**, so that literals can be passed in C++.

If there are nested function prototypes, use **NeedNestedPrototypes**:

```
extern Bool XCheckIfEvent(
#if NeedFunctionPrototypes
    Display*        /* display */,
    XEvent*         /* event_return */,
    Bool (*) (
#if NeedNestedPrototypes
             Display*              /* display */,
            XEvent*          /* event */,
            XPointer         /* arg */
#endif
          )         /* predicate */,
    XPointer        /* arg */
#endif
);
```

If there is a variable argument list, use **NeedVarargsPrototypes**:

```
extern char *XGetIMValues(
#if NeedVarargsPrototypes
    XIM /* im */, ...
#endif
```

4. Yes, we should have used the ANSI C function names, but we thought we had too much existing code using the BSD names.

```
);
```

If you have parameter types that will widen in K&R C, then you should use **NeedWidePrototypes**:

```
extern XModifierKeymap *XDeleteModifiermapEntry(
#if NeedFunctionPrototypes
    XModifierKeymap*    /* modmap */,
#if NeedWidePrototypes
    unsigned int        /* keycode_entry */,
#else
    KeyCode             /* keycode_entry */,
#endif
    int                 /* modifier */
#endif
);
```

If you use **_Xconst**, **NeedNestedPrototypes**, **NeedVarargsPrototypes**, or **NeedWidePrototypes**, then your function implementation also has to have a function prototype. For example:

```
#if NeedFunctionPrototypes
Atom XInternAtom (
    Display *dpy,
    _Xconst char *name,
    Bool onlyIfExists)
#else
Atom XInternAtom (dpy, name, onlyIfExists)
    Display *dpy;
    char *name;
    Bool onlyIfExists;
#endif
{
    ...
}
```

Actually, anytime you use a function prototype in a header file, you should use a function prototype in the implementation, as required by ANSI C. The MIT X sources do not follow this (we've never had time to make all the changes), and there are almost certainly compilers that will complain if the implementation does not match the declaration.

10.5. Other Symbols

Do not use the names **class**, **new**, or **index** as variables or struct members. The names **class** and **new** are reserved words in C++, and you may find your header files used by a C++ program someday. Depending on your system, **index** can be defined as **strchr** or a macro in **<X11/Xos.h>**; this may cause problems if you include this header file.

The following system-specific symbols are commonly used in X sources where OS dependencies intrude:[5]

USG	based on System V Release 2
SYSV	based on System V Release 3
SVR4	System V Release 4

For other system-specific symbols, look at the **StandardDefines** parameters in the **mit/config/*.cf** files.

5. At most *one* of these symbols should be defined on a given system!

11. What's New, What's Changed

[deleted]

12. Acknowledgements

The MIT Release 5 distribution is brought to you by the MIT X Consortium. A cast of thousands, literally, have made this release possible. We cannot possibly acknowledge them all here. The names of all people who made it a reality will be found in the individual documents and source files. We greatly appreciate the work that everyone has put into this release.

Hoping you enjoy Release 5,

Donna Converse
Stephen Gildea
Susan Hardy
Jay Hersh
Keith Packard
David Sternlicht
Bob Scheifler
Ralph Swick

(R5 Survival Club)