X Window System, Version 11, Release 5

Release Notes

MIT X Consortium staff

MIT Laboratory for Computer Science

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:

    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.

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

2. Brief Overview of the Distribution

(If you want, you can skip to the next chapter first, and get your build started. While it is compiling you will have plenty of time to read the rest of the release notes.)

There are two parts to the Release 5 distribution: MIT software and documentation, and user-contributed software and documentation. The MIT part contains:

X Consortium Standards

The MIT X Consortium produces standards: documents which define network protocols, programming interfaces, and other aspects of the X environment. See the **XStandards** man page for a list of standards. See the **XConsortium** man page for information about the X Consortium.

Sample Implementations

For most of our standards, we provide *sample* implementations to demonstrate proof of concept. These are not *reference* implementations; the written specifications define the standards.

Fonts

A collection of bitmap and outline fonts are included in the distribution, contributed by various individuals and companies.

Utility Libraries

A number of libraries, such as the *Athena Widget Set,* are included. These are not standards, but are used in building MIT applications and may be useful in building other applications.

Sample Programs

We also provide a number of application programs. A few of these programs, such as **xdm**, should be considered essential in almost all environments. The rest of the applications carry no special status, they are simply programs that have been developed and/or maintained by MIT X Consortium staff. In some cases, you will find better substitutes for these programs in the user-contributed software.

The user-contributed part contains whatever people contribute. You find a variety of software and documentation here: application programs, demos, examples, libraries, Asian input methods, X server extensions, etc.

2.1. Structure of the MIT Sources

All of the MIT sources are under a single directory, **mit**. Sources are organized into the following subdirectories:

clients

This directory contains most of the sample applications. See the program man pages for details.

config

This directory contains configuration files and the **imake** program used to build the release. Details are covered in other sections below.

demos

This directory contains a small collection of graphics demonstration programs, a few utility/test programs, and some performance demonstration programs. These are by no means the ‘‘best’’ demo programs around, they just happen to be ones we try to maintain.

doc

This directory contains **troff** sources to X Consortium standards, server internals documentation, documentation for various utility libraries, some useful tutorial material.

extensions

This directory contains implementations of X server extensions, both the server internals and the application programming libraries, and some test programs. Of particular note here, new in Release 5, is PEX, the PHIGS Extension to X used for 3D graphics, and the PHIGS programming library which interfaces to the PEX protocol.

fonts

This directory contains bitmap fonts in source form, some outline fonts, a sample font server, a utility font library used by the X server and font server, a client font library for interacting with the font server, and programs for building fonts and querying the font server.

hardcopy

This directory contains pre-generated PostScript files for the client man pages and for most of the documentation found in the **doc** directory. The files are compressed with **compress** to save disk space. If you do not have **compress** on your system, you will find a version in the **mit/util/compress** directory.

include

This directory contains various library-independent C header files and a collection of bitmap files.

lib

This directory contains programming libraries and support files. Of note are Xlib (the lowest-level C programming interface to X), Xt (the X Toolkit Intrinsics), Xmu (an eclectic set of utility functions), Xaw (the Athena Widget Set), and CLX (a Common Lisp interface to X).

man

This directory contains a few top-level man pages about the release (general information, server access control mechanisms, the X Consortium, and X Consortium standards), and man pages for some of the programming libraries.

rgb

This directory contains a program to generate the color database used by the X server and sample databases.

server

This directory contains the X server sources, both device-independent (**dix**) and device-dependent (**ddx**). In this release, there is support for building the following servers:

DECstation 2100/3100 monochrome and color displays
DECstation 5000 CX and MX displays
IBM RS/6000 skyway adapter
Apple Macintosh monochrome and color displays
MIPS monochrome and color displays
OMRON LUNA monochrome displays (color displays operate in monochrome)
Tektronix 4319 color display
VAXstation QVSS and QDSS displays
Sun monochrome and 8-bit color displays (with GX support)
Various VGA displays under System V/386

If your favorite hardware is not listed above, please do not blame us at MIT, we ship what Consortium members provide. Only in a few cases do we try to maintain device-specific software for our own development needs.

util

This directory contains miscellaneous utility programs and shell scripts used to build, maintain, and install the release.

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

The distribution normally comes as multiple tar files, either on tape or across a network. Create a directory to hold the distribution, **cd** to it, and untar everything from that directory. For example:

mkdir *sourcedir*
cd *sourcedir*
tar xfp *tar-file-or-tape-device*

If you have obtained compressed and split tar files over the network, then the sequence for each part of the **mit** directory might be:

cd *ftp-dir*/mit-*N*
cat mit-*N*.?? | uncompress | (cd *sourcedir*; tar xfp -)

The sequence for each part of the **contrib** directory might be:

cd *ftp-dir*/contrib-*N*
cat contrib-*N*.?? | uncompress | (cd *sourcedir*; tar xfp -)

The *sourcedir* directory you choose can be anywhere in any of your filesystems that is convenient to you. After extracting the release, you should end up with an **mit** subdirectory, and a **contrib** subdirectory if you unpack user-contributed software. You will need about 100 megabytes of disk space to unpack the **mit** directory contents; building it will of course require more disk space.

3.2. Symbolic Link Trees

If you expect to build the distribution on more than one machine using a shared source tree, or you just want to keep the source tree pure, you may want to use the shell script **mit/util/scripts/lndir.sh** to create a symbolic link tree on each build machine. This is fairly expensive in disk space, however. To do this, create a directory for the build, **cd** to it, and type:

sourcedir/mit/util/scripts/lndir.sh *sourcedir*

where *sourcedir* is the pathname of the directory where you stored the sources. All of the build instructions given below should then be done in the build directory on each machine, rather than in the source directory.

The shell script is reasonably portable but quite slow to execute. If you want you can instead try compiling a similar C program, but it is slightly tricky to do before the distribution is built; **cd** to the directory **mit/util/progs** and try typing:

```
ln -s ../../include X11
cc -o lndir -I. lndir.c
```

If it compiles and links, it will probably work; otherwise you can try typing:

```
cc -o lndir -I. -DNOSTDHDRS lndir.c
```

If it still fails, use the shell script.

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. However, here are some notable parameters that you can set in the ‘‘after’’ section:

BuildXsi and **BuildXimp**

New in this release, Xlib contains support for internationalized input methods, using library- or network-based implementation methods. The implementation details internal to Xlib can vary considerably depending on the types of input methods supported. In this release, two different implementations are supported, named **Xsi** and **Ximp**. As distributed, the default on all systems except Sony is **Xsi**. If you want to use **Ximp** instead, add this:

```
#define BuildXimp YES
```

BuildServer

This controls whether or not an X server is built. If the variable is not set to **NO** in the **.cf** file, then the default is to build a server. If you want to disable the server, add this:

```
#define BuildServer NO
```

BuildFonts

Compiled fonts take up a lot of disk space. In this release, the compiled form (called ‘‘pcf’’) can be shared across all machines of all architectures, so you may only want to build the fonts on one machine. To disable font building, add this:

```
#define BuildFonts NO
```

BuildPex

PEX is an X extension supporting 3-D graphics and the PHIGS programming interface. The PEX sources are known to cause some compilers to exceed their static symbol table sizes. If this happens to you, you can disable PEX by adding this:

```
#define BuildPex NO
```

ManSuffix

User program man pages are installed by default in subdirectory ‘‘mann’’ with suffix ‘‘.n’’. You can change this to ‘‘man1’’ and ‘‘.1’’, for example, by adding this:

```
#define ManSuffix 1
```

InstallLibManPages

By default, the programming library man pages (Xlib, Xt, various extensions) are installed along with all of the other man pages. The library pages constitute a considerable number of files. If you do not expect to be programming with X, or prefer using other forms of documentation, you can disable installation of the library pages by adding this:

```
#define InstallLibManPages NO
```

InstallXdmConfig and **InstallXinitConfig**

The **xdm** and **xinit** programs are the normal ways to run X servers. By default, the configuration files for these programs are not installed, to avoid inadvertently destroying existing configuration files. If you are not yet using **xdm** or **xinit**, or will be installing into a new destination, or do not wish to retain your old configuration files, add these:

```
#define InstallXdmConfig YES
#define InstallXinitConfig YES
```

XdmServersType

Some of the **xdm** config files are generated based on configuration parameters. One of the files controls whether an X server is started by default. By default the choice is made based on whether an X server is built as part of this distribution (the **BuildServer** parameter). If you are not building a server, but you will be running a product server on the workstation under **xdm**, you should add this:

```
#define XdmServersType ws
```

HasXdmAuth

This release supports a DES-based form of authorization called XDM-AUTHORIZATION-1. The source file **mit/lib/Xdmcp/Wraphelp.c**, which is necessary for this to compile, might not be included in your distribution due to possible export regulations; if it is not included and you are a US citizen, you should be able to obtain the file over the network. To enable building of this mechanism, add this:

```
#define HasXdmAuth YES
```

InstallFSConfig

New to this release is a network font server, **fs**. By default, the configuration files for the font server are not installed. To have them installed, add this:

```
#define InstallFSConfig YES
```

MotifBC

If you want to use the Release 5 Xlib and Xt with Motif 1.1, you will need to enable a backward compatibility flag, by adding this:

```
#define MotifBC YES
```

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

This release has support for internationalization, based on the ANSI C and POSIX locale model. On some systems, you may discover that while the locale interface is supported, only the ‘‘C’’ locale is actually provided in the base operating system. If you have such a system, and would like to experiment with a broader set of locales, the Xlib implementation contains support you can use, although use of this override has not really be tested. You need to add the following defines to the **StandardDefines** parameter:

```
-DX_WCHAR -DX_LOCALE
```

In most cases you will have to directly edit the **.cf** file to do this, or else you will have to know what the rest of the values are supposed to be, and add this to **site.def**:

```
#undef StandardDefines
#define StandardDefines previous-values -DX_WCHAR -DX_LOCALE
```

It is also possible to directly edit the file **mit/include/Xosdefs.h**, but this is not recommended.

With this setup, you will have to be careful that the system’s declaration of **wchar_t** (in **<stddef.h>**) never gets used; this might be tricky.

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]

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.

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

If your **/etc/termcap** and **/usr/lib/terminfo** databases do not have correct entries for **xterm**, sample entries are provided in the directory **mit/clients/xterm/**. System V users may need to compile and install the **terminfo** entry with the **tic** utility.

Since each **xterm** will need a separate pseudoterminal, you need a reasonable number of them for normal execution. You probably will want at least 32 on a small, multiuser system. On most systems, each pty has two devices, a master and a slave, which are usually named /dev/tty[pqrstu][0-f] and /dev/pty[pqrstu][0-f]. If you don't have at least the ''p'' and ''q'' sets configured (try typing ''ls /dev/?ty??''), you should have your system administrator add them. This is commonly done by running the **MAKEDEV** script in the **/dev** directory with appropriate arguments.

4.2. Starting Servers at System Boot

The **xdm** program is designed to be run automatically at system startup. Please read the **xdm** man page for details on setting up configuration files; reasonable sample files are in **mit/clients/xdm/config**. If your system uses an **/etc/rc** file at boot time, you can usually enable **xdm** by placing the following at or near the end of the file:

```
if [ -f /usr/bin/X11/xdm ]; then
    /usr/bin/X11/xdm; echo -n ' xdm'
fi
```

The example here uses **/usr/bin/X11**, but if you have installed into a different directory (for example by setting **ProjectRoot**) then you need to substitute the correct directory.

If you are going to use the font server, you can also start it at boot time by adding this:

```
if [ -f /usr/bin/X11/fs ]; then
    /usr/bin/X11/fs &; echo -n ' fs'
fi
```

If you are unsure about how system boot works, or if your system does not use **/etc/rc**, consult your system administrator for help.

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

The software in **contrib** is not set up to have everything built automatically. It is assumed that you will build individual pieces as you find the desire, time, and/or disk space. You need to have the MIT software built and installed before building the contributed software. To build a program or library in **contrib**, look in its directory for any special build instructions (for example, a **README** file). If there are none, and there is an **Imakefile**, **cd** to the directory and type:

```
xmkmf -a
make >& make.log
```

This will build a **Makefile** in the directory and all subdirectories, and then build the software. If the build is successful, you should be able to install it using the same commands used for the **mit** software:

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

7. Filing Bug Reports

If you find a reproducible bug in software in the **mit** directory, or find bugs in the **mit** documentation, please send a bug report to MIT using the form in the file **mit/bug-report** and the destination address:

xbugs@expo.lcs.mit.edu

Please try to provide all of the information requested on the form if it is applicable; the little extra time you spend on the report will make it much easier for us to reproduce, find, and fix the bug. Receipt of bug reports is generally acknowledged, but sometimes it can be delayed by a few weeks.

Bugs in **contrib** software should not be reported to MIT. Consult the documentation for the individual software to see where (if anywhere) to report the bug.

8. Public Fixes

We occasionally put out patches to the MIT software, to fix any serious problems that are discovered. Such fixes (if any) can be found on **export.lcs.mit.edu**, in the directory **pub/R5/fixes**, using anonymous **ftp**. Fixes are applied using the **patch** program; a copy of it is included in the directory **mit/util/patch**.

For those without **ftp** access, individual fixes can be obtained by electronic mail by sending a message to

xstuff@expo.lcs.mit.edu

(Note that the host here is ‘‘expo’’, not ‘‘export’’.) In the usual case, the message should have a subject line and no body, or a single-line body and no subject, in either case the line looking like:

send fixes *number*

where *number* is a decimal number, starting from one. To get a summary of available fixes, make the line:

index fixes

If you need help, make the line:

help

Some mailers produce mail headers that are unusable for extracting return addresses. If you use such a mailer, you won't get any response. If you happen to know an explicit return path, you can include include one in the body of your message, and the daemon will use it. For example:

path *user%host*.bitnet@mitvma.mit.edu
or
path *host1*!*host2*!*user*@uunet.uu.net

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

Unfortunately, for a new system there are a potentially large number of files that you may have to modify. Only the most prominent ones are listed here.

```
mit/include/Xfuncs.h
mit/include/Xmd.h
mit/include/Xos.h
mit/include/Xosdefs.h
mit/clients/xload/get_load.c
mit/clients/xman/vendor.c
mit/clients/xman/vendor.h
mit/clients/xterm/main.c
mit/lib/X/Xlibnet.h
mit/server/include/servermd.h
```

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;
    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

---

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’’.

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 */,
```

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

```
                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
);
```

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.

11. What's New, What's Changed

In this section we briefly describe some of the more significant new features of Release 5.

11.1. New standards

The following standards are new in Release 5:

X Font Service Protocol

Instead of forcing each X server to read all fonts from the filesystem, the X Font Server Protocol makes it possible to manage fonts separately from the X server, directing the X server to request fonts via this new Consortium standard network protocol from a font server. In addition, for fonts which take a long time to open, this allows the X server to continue with other clients while the font server services the font requests.

XLFD changes for scalable fonts

The X Logical Font Description standard has been compatibly enhanced to allow clients to specify and use scalable fonts.

X Input Device Extension

This extension has been promoted from Draft Standard to full Consortium Standard with this release.

Inter-Client Communications Conventions

This standard has been updated to cover the new X Device Color Characterization Conventions for device-independent color support in Xlib.

11.2. General

We have tried hard with this release to make our code use standard features from POSIX.1 and ANSI C when possible. A new include file **<X11/Xosdefs.h>** describes which systems comply or do not comply with these standards.

Two new X authorization schemes are included, a DES based private-key system which was described in the R4 XDMCP document - XDM-AUTHORIZATION-1 (along with the associated XDMCP authentication system XDM-AUTHENTICATION-1) and the Sun Secure RPC based SUN-DES-1 system, which uses the SunOS supplied security system.

11.3. Clients

Most clients participate in the WM_DELETE_WINDOW protocol.

New clients: **editres**, **viewres**, **xconsole**, **xcmsdb**. New demos: **beach_ball**, **auto_box**, **gpc**, **xcmstest**, **xgas**, **x11perf**. **Xlswins** has been removed; it is replaced by **xwininfo -tree**. Moved to contrib: **muncher**, **plaid**. Completely new implementation: **bitmap** and **xmag**. Other changes of note:

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

editres

Editres is a tool that allows users and application developers to view the full widget hierarchy of any X Toolkit client that speaks the Editres protocol. In addition **editres** will help the user construct resource specifications, allow the user to apply the resource to the application and view the results dynamically. Once the user is happy with a resource specification, editres will append the resource string to the user's resources file.

xdm

Xdm can now display a menu of hosts for XDMCP-capable terminals using the new *chooser* client. This is useful for X terminals that do not themselves offer such a menu. XDMCP works with STREAMS. A new setup program is invoked by **xdm** prior to putting up the login window; this program can be used to run **xsetroot**, **xcmsdb**, and do any other custom initialization required.

xterm

Cuts of wrapped lines are now treated as a single line. Cuts of multi-page regions now work and highlight correctly. Pasting large amounts of data into **xterm** now works (on systems with properly-working pty implementations). New arguments have been added to the send-signal action: quit, alarm. The **titeInibit** resource has been modified to also inhibit the escape sequence which switches to the alternate screen. Two new items have been added to the VT Fonts menu: 5x7 (Tiny) and 10x20 (Huge). The following resources have been added: **resizeGravity**, **bellSuppressTime**, **appcursorDefault**, **appkeypadDefault**, **ginTerminator**, **autoWrap**. The *Xterm Control Sequences* document is up to date. **Xterm** is installed securely when made setuid on SunOS 4.1.1 with shared libraries.

xmh

Xmh now uses the **MH** environment variable, if set. **Xmh** now supports checking for mail in multiple maildrops. Enhanced participation in WM_PROTOCOLS has been added. New resources have been added, including: **checkpointInterval**, **checkpointNameFormat**, **mailInterval**, **rescanInterval**, **showOnInc**, **noMailBitmap**, **newMailBitmap**, **newMailIconBitmap**, and **noMailIconBitmap**. New actions have been added: XmhWMProtocols, XmhShellCommand, XmhCheckForNewMail, XmhViewMarkDelete. Better recovery from unexpected inconsistencies with the filesystem has been implemented. Better POP support has been added. See the file **mit/clients/xmh/CHANGES** for more details.

oclock

Oclock has a new **–transparent** option.

xload

Xload is secure on SunOS 4.1.1 with shared libraries.

xditview

Xditview now supports **pic**, scalable fonts, settable device resolution, and has a better user interface.

11.4. Libraries in General

All of the useful libraries now use function prototypes by default for systems which support them. SunOS shared libraries now use much less swap space than in R4. In addition, System V Release 4 and AIX 3.1 shared libraries are also supported now. Configuring new shared library systems should be much easier than before.

11.5. Xlib

Two new major pieces of functionality have been added to Xlib: device independent color, and internationalization (i18n). In addition, a few other additions and improvements have been made.

11.5.1. Xlib Manual

The Xlib manual has been reorganized for Release 5. Unfortunately, this may cause considerable confusion for a while when people quote section numbers without reference to the release. However, we feel that the new organization is a considerable improvement.

11.5.2. Device-independent Color

The **Xcms** (X Color Management System) functions in Xlib support device-independent color spaces derivable from the CIE XYZ color space. This includes the CIE XYZ, xyY, L*u*v*, and L*a*b* color spaces as well as the TekHVC color space. In addition, linear RGB intensity value space has been added, as well as gamma correction for device RGB values, and a uniform syntax has been adopted for specifying colors in strings. Xlib now supports client-side color name databases, and the existing Xlib functions that use color names (e.g., **XLookupColor** and **XAllocNamedColor**) now handle all color spaces, so that the contrivance of using **XParseColor** followed by **XAllocColor** is no longer necessary. Xlib provides direct programming interfaces for dealing with color values in different spaces, and for converting between spaces. New device-independent color spaces can also be added. For details on the new color functionality, read Chapter 6 of the new Xlib manual.

Monitors are characterized by data stored on root window properties; the new **xcmsdb** program can be used to set these properties. Unfortunately, you need a color analyzer instrument to generate characterizations; choosing a random one will almost certainly produce inaccurate colors. However, you will find some sample database files in **mit/clients/xcmsdb/datafiles** and in **contrib/clients/ca100/monitors**.

11.5.3. Internationalization

An internationalized application is one which is adaptable to the requirements of different native languages, local customs, and character string encodings. The process of adapting the operation to a particular native language, local custom, or string encoding is called localizaton. A goal of internationalization is to permit localization without program source modifications or recompilation.

Internationalization in Xlib is based on the concept of a *locale*. A locale defines the ‘‘localized’’ behavior of a program at run-time. Locales affect Xlib in its:

- Encoding and processing of input method text
- Encoding of resource files and values
- Encoding and imaging of text strings
- Encoding and decoding for inter-client text communication

Characters from various languages are represented in a computer using an encoding. Different languages have different encodings, and there are even different encodings for the same characters in the same language.

Xlib provides support for localized text imaging and text input. Sets of functions are provided for multibyte (char*) text as well as wide character (**wchar_t**) text in the form supported by the host C language environment. For details on the new internationalization functionality, read Chapter 13 of the new Xlib manual.

Two sample implementations of the internationalization mechanisms exist in this release, one called **Xsi** and one called **Ximp**. You will find documentation for them in **mit/doc/I18N** and **contrib/im**, and locale definition files in **mit/lib/nls**. In **contrib/im** you will also find network-based Input Method servers.

Unfortunately, none of the programs in the MIT software use the new internationalization facilities. However, you will find some sample clients in **contrib/im**, and internationalized versions of some of the MIT clients in **contrib/clients**.

11.5.4. Keysyms

By default a database of all registered vendor-private keysyms gets installed, so that Xlib can map between keysym values and names.

11.5.5. Resource Databases

A new SCREEN_RESOURCES property has been defined, permitting screen-specific resources to be set, so that (for example) colors can be specified for a color screen and not effect a monochrome screen on the same server. The **xrdb** program has been enhanced to ‘‘do the right thing’’ automatically in most cases.

New functions have been defined to merge a resource database directly from a file, and to combine two databases with either ‘‘augment’’ or ‘‘override’’ semantics.

A ‘‘#include’’ syntax is now supported in resource files. A specific example of using this is to have a customized app-defaults file (in an Xt application) include the base app-defaults file.

A new reserved component name, ‘‘?’’, has been defined that matches a single level in the resource hierarchy. This makes it easier to override resources specified in app-defaults files.

A new function, **XrmEnumerateDatabase**, has been defined to search for matching entries in a resource database. The **appres** program has been enhanced in this release to become a convenient interface to this function.

A new function, **XrmPermStringToQuark**, has been introduced to avoid having Xlib allocate needless storage for constant strings.

11.5.6. Extensions

A new function has been added to permit an extension to convert errors with additional data into Xlib format, and one has been added to permit an extension to print out the values in an error handler.

11.5.7. Miscellaneous

A new type **XPointer** has been introduced, replacing uses of the non-standard type **caddr_t**. Old programs using **caddr_t** will still work, of course.

11.5.8. Performance

The new color and internationalization facilities have the unfortunate effect of making executables quite a bit larger on systems that do not have shared libraries.

The resource database functions have been completely rewritten for this release. Databases should be significantly smaller in memory, and loading and parsing resources should be faster.

11.6. Xt Intrinsics

At the data structure level, Release 5 retains complete binary compatibility with Release 4. The specification of the **ObjectPart**, **RectObjPart**, **CorePart**, **CompositePart**, **ShellPart**, **WMShellPart**, **TopLevelShellPart**, and **ApplicationShellPart** instance records was made less strict to permit implementations to add internal fields to these structures. Any implementation that chooses to do so would, of course, force a recompilation. The Xlib specification for **XrmValue** and **XrmOptionDescRec** was updated to use a new type, **XPointer**, for the *addr* and *value* fields respectively, to avoid ANSI C conformance problems. The definition of **XPointer** is binary compatible with the previous implementation.

11.6.1. baseTranslations Resource

A new pseudo-resource, **XtNbaseTranslations**, was defined to permit application developers to specify translation tables in application defaults files while still giving end users the ability to augment or override individual event sequences. This change will affect only those applications that wish to take advantage of the new functionality, or those widgets that may have previously defined a resource named ‘‘baseTranslations’’.

Applications wishing to take advantage of the new functionality would change their application defaults file, e.g., from

app.widget.translations: *value*

to

app.widget.baseTranslations: *value*

If it is important to the application to preserve complete compatibility of the defaults file between different versions of the application running under Release 4 and Release 5, the full translations can be replicated in both the ‘‘translations’’ and the ‘‘baseTranslations’’ resource.

11.6.2. Resource File Search Path

The current specification allows implementations greater flexibility in defining the directory structure used to hold the application class and per-user application defaults files. Previous specifications required the substitution strings to appear in the default path in a certain order, preventing sites from collecting all the files for a specific application together in one directory. The Release 5 specification allows the default path to specify the substitution strings in any order within a single path entry. Users will need to pay close attention to the documentation for the specific implementation to know where to find these files and how to specify their own **XFILESEARCHPATH** and **XUSERFILESEARCHPATH** values when overriding the system defaults.

11.6.3. Customization Resource

XtResolvePathname supports a new substitution string, %C, for specifying separate application class resource files according to arbitrary user-specified categories. The primary motivation for this addition was separate monochrome and color application class defaults files. The substitution value is obtained by querying the current resource database for the application resource name ‘‘customization’’, class ‘‘Customization’’. Any application that previously used this resource name and class will need to be aware of the possibly conflicting semantics.

11.6.4. Per-Screen Resource Database

To allow a user to specify separate preferences for each screen of a display, a per-screen resource specification string has been added and multiple resource databases are created; one for each screen. This will affect any application that modified the (formerly unique) resource database associated with the display subsequent to the Intrinsics database initialization. Such applications will need to be aware of the particular screen on which each shell widget is to be created.

Although the wording of the specification changed substantially in the description of the process by which the resource database(s) is initialized, the net effect is the same as in prior releases with the exception of the added per-screen resource specification and the new customization substitution string in **XtResolvePathname**.

11.6.5. Internationalization of Applications

Internationalization as defined by ANSI is a technology that allows support of an application in a single locale. In adding support for internationalization to the Intrinsics the restrictions of this model have been followed. In particular, the new Intrinsics interfaces are designed to not preclude an application from using other alternatives. For this reason, no Intrinsics routine makes a call to establish the locale. However, a convenience routine to establish the locale at initialize time has been provided, in the form of a default procedure that must be explicitly installed if the application desires ANSI C locale behavior.

As many objects in X, particularly resource databases, now inherit the global locale when they are created, applications wishing to use the ANSI C locale model should use the new function **XtSetLanguageProc** to do so.

The internationalization additions also define event filters as a part of the Xlib Input Method specifications. The Intrinsics enable the use of event filters through additions to **XtDispatchEvent**. Applications that may not be dispatching all events through **XtDispatchEvent** should be reviewed in the context of this new input method mechanism.

In order to permit internationalization of error messages the name and path of the error database file is now allowed to be implementation dependent. No adequate standard mechanism has yet been suggested to allow the Intrinsics to locate the database from localization information supplied by the client.

The previous specification for the syntax of the language string specified by **xnlLanguage** has been dropped to avoid potential conflicts with other standards. The language string syntax is now implementation-defined. The example syntax cited is consistent with the previous specification.

11.6.6. Permanently Allocated Strings

In order to permit additional memory savings, an Xlib interface was added to allow the resource manager to avoid copying certain string constants. The Intrinsics specification was updated to explicitly require the Object *class_name*, *resource_name*, *resource_class*, *resource_type*, *default_type* in resource tables, Core *actions string* field, and Constraint *resource_name*, *resource_class*, *resource_type*, and *default_type* resource fields to be permanently allocated. This explicit requirement is expected to affect only applications that may create and destroy classes on the fly.

11.6.7. Arguments to Existing Functions

The *args* argument to **XtAppInitialize**, **XtVaAppInitialize**, **XtOpenDisplay**, **XtDisplayInitialize**, and **XtInitialize** were changed from **Cardinal*** to int* to conform to pre-existing convention and avoid otherwise annoying typecasting in ANSI C environments.

11.6.8. Implementation

Function prototypes are now fully supported in the header files.

<X11/Intrinsic.h> no longer includes **<X11/Xos.h>** by default. Inclusion of this file was a bug in earlier releases. If you have old code that depends on this bug, you can define -DXT_BC when you compile to get back the old behaviour.

String constants are now defined in a single array, saving memory and external symbols. Note that because the new implementation uses #defines, string constants in widget header files which duplicate a constant defined by Xt should either be removed or protected from a collision.

The translation manager facilities have been completely reimplemented in this release, resulting in substantially less memory consumed by some applications. A number of other memory-saving changes have been implemented, and in a few cases execution time should be faster.

The default keycode to keysym translator deals with all Latin keysyms.

11.6.9. Extension Events

Unfortunately, the Xt standard as of R5 still does not address the issues of integrating events from protocol extensions into the normal Xt dispatch mechanism. The adventurous will find a set of patches to Xt in **contrib/lib/Xt** that attempt to address this problem. These patches are non-standard, experimental, subject to change, not guaranteed, may adversely affect your ability to apply public patches from MIT, and have not reviewed by the X Consortium.

11.7. PEX

The PEX Sample Implementation (SI) is composed of several parts. The major components are the extension to the X Server, which implements the PEX protocol, and the client side Application Protocol Interface (API), which provides a mechanism by which clients can generate PEX protocol. The API provided with the PEX-SI is the ISO IS PHIGS Binding and the yet to be standardized PHIGS PLUS Binding.

In addition to these major components, several other minor components are provided. These include documentation, 3D fonts for PEX, demos, and a verification suite called InsPEX. Also provided in **contrib** are additional example programs and demos.

These elements are located in the following area:

The PEX server extension is located in the directories under **mit/extensions/server/PEX**. Device independent portions are located in **mit/extensions/server/PEX/dipex**. Device dependent functionality appears in **mit/extensions/server/PEX/ddpex**. Operating system font dependent code appears in **mit/extensions/server/PEX/ospex**. General purpose server include files are in **mit/extensions/server/PEX/include**.

The API code is located under the directory **mit/extensions/lib/PEX**. The PHIGS/PHIGS PLUS Binding routines are in the **c_binding** subdirectory. The PHIGS Monitor (PM), a separate process started at client runtime to handle PHIGS Input functionality, is in the **cp** subdirectory. Other code located in the various subdirectories handles PHIGS archival, error handling, and comprises the internal library level that PHIGS calls to generate the PEX Protocol.

All PEX documentation is located in the directory **mit/doc/extensions/PEX**, with pregenerated PostScript files in **mit/hardcopy/extensions/PEX**. The PEX Protocol Specification itself is in the **Proto** subdirectory. All SI documentation is in the **SI** subdirectory. Three subdirectories there contain an Architecture Specification, a Porting Guide (with implementation details), and a User's Guide. The sources and programs used to generate these files are located in the **mit/doc/extensions/PEX/SI** directory. Also located there is the **PHIGS** subdirectory which contains PHIGS man pages and macros for printing these pages.

Font source for PEX and utilities to build them are located in the directory **mit/fonts/PEX**. Two stroke fonts are supplied.

The PEX verification tool InsPEX can be found in the **mit/extensions/test/InsPEX** directory. Shell scripts are provided there to run InsPEX. More information on InsPEX is available in the User's Guide.

Demos for PEX can be found in the **mit/demos** directory. Two demos and the NCGA Graphics Performance Characterization (GPC) Suite can be found there. The demos are in the **mit/demos/auto_box** and **mit/demos/beach_ball** directories, and are named **auto_box** and **beach_ball** respectively. The GPC suite is found in **mit/demos/gpc**. This suite consists of demos (in the **objects** subdirectory), benchmarks (various directories below **benchmarks**) and tests (in **tests**). For more information on how to run these demos see the User's Guide.

There are also several unsupported demos and examples available in **contrib**. In **contrib/demos/beach_ball2** a newer version of the **beach_ball** demo with enhanced functionality can be found. In **contrib/examples/PEX** various PHIGS based clients that demonstrate how to use PEX via the PHIGS API are available.

11.7.1. Standards and Functionality

This release conforms to the PEX Protocol Specification V5.0P. The release comes with 2 fonts, Roman and Roman_M (see the User's Guide for more details). It implements the minimum required HLHSR (Hidden Line/ Hidden Surface Removal) for PHIGS compliance (i.e., NONE). The release only supports 8-bit color.

The API binding has been updated to the ISO IS PHIGS binding. The directory **mit/util/PEX** contains **sed** scripts for converting programs from the previous binding to the new binding. These scripts do most of the work, but some manual editing is still needed. There is a **README** file in this directory with notes and information.

The PHIGS Binding provides most PHIGS/PHIGS PLUS functionality. The full PHIGS Input Model (Valuator, Locator, Stroke, Choice, String, Pick) is implemented in a device independent manner using the Athena Widget Set. PHIGS/PHIGS PLUS functionality includes, but is not limited to the following graphical primitives: Polylines, Polymarkers, Fill Areas, Triangle Strips, NURBS Curves and Surfaces, 2D and 3D Text. Other operations include Depth Cueing, Modelling Clip, Backface removal, Lighting Models and Surface Reflection.

Functionality not completed in this release is as follows:

In the API:

Mapping of PHIGS font ids to PEX fonts

In the Server:

Backface Attributes and Distinguish Flag
Font sharing between clients
Patterns, Hatches and associated attributes
Color Interpolation
Transparency
Depth Cueing for Markers
Z-buffering
Double Buffering

In InsPEX:

Completion of port to ISO IS PHIGS Binding

11.7.2. PEX and PHIGS Documents

The following documents are provided with this release:

PEX-SI User Guide
PEX-SI Graphics Library Manual Pages
PEX-SI Architecture Specification
PEX-SI Porting Guide

They are located in subdirectories of **mit/doc/extensions/PEX**. Please read the PEX-SI User's Guide for descriptions of the documents and how to use them. Instructions for printing the documents are provided in a README file in each of the document directories.

The User's Guide is provided as a starting point in the documentation. It describes the various documents provided with the release, and includes instructions on using the clients, the API and the server code. It also includes specifications for the server functionality and archive format.

The Graphics Library Manual Pages are for the client-side library, written to the ISO IS binding.

The Architecture Specification describes the PEX-SI architecture at a high level.

The Porting Guide is intended as an extension to the Architecture Specification. There is a lot of good information in this document, and it is organized fairly well, but it lacks some polish. It is not a complete document.

11.7.3. InsPEX

This release of InsPEX includes coverage of all the PHIGS PLUS graphics primitives, such as fill area sets with data, quadrilateral meshes, triangle strips, and NURBS. PHIGS PLUS attributes such as direct color specification, depth cuing, and lighting are also exercised.

The testing of input is somewhat limited by the problem of simulating mouse and keyboard input in a reliable and portable fashion. For the pick, locator, and stroke devices, simulating the mouse events is straightforward, but since the string, valuator, and choice devices are built upon a toolkit (Athena Widgets in the PEX-SI's case), getting window id's for the appropriate windows and sending mouse clicks to the right place on those windows is more difficult, and probably impossible to do in a way that could be quickly ported to another toolkit along with these input devices. The technology for automatic testing of software using a graphical user interface under X has not progressed to the point where this functionality could be tested in a way that would be useful to all the potential users of InsPEX.

For nearly all of the tests that use image comparison to verify graphical output, reference images have been supplied. Due to outstanding bugs in the code, however, some tests are shipped without reference images. Also, since back-facing attributes are not implemented, the images saved for these tests are actually incorrect. These have been shipped, however, because they still can be helpful to someone porting the PEX-SI. It should be expected that when backfacing attributes are implemented, these tests will fail and image comparison will be required.

Along with the **README** in the main InsPEX directory, there is a sample log file, sample.log, and an automatically-generated summary of all the current tests, **test_descrip**. See the **README** for a description of how that file was generated.

11.8. Athena Widget Set

Many minor bugs have been fixed. The Xaw examples have been moved to **contrib**. However, please note that the Athena Widgets have been and continue to be low on our priority list, so many bugs remain (particularly in the Text widget) and many requests for enhancements have not been implemented. Because some incompatible changes have been made, the shared library major version number on Suns has been incremented.

Header Files

Function prototypes have been added to the public interfaces.

AsciiSrc

No warning message is printed when the file cannot be written to; the return value should be enough info. **GetValues** on the string resource was failing when "useStringInPlace" was set to true; fixed. A memory leak when freeing pieces in a source of type "ascii String" has been plugged. The buffer is now updated whenever the "string" resource is set using **XtSetValues**. If the type is file then the file is read in again.

Box

Box.h now includes **<X11/Xmu/Converters.h>** for the orientation resources.

Clock

Changed to be a subclass of **Simple** instead of **Core**.

Command

A bug in changing the shape style back to Rectangular has been fixed.

Dialog

The Icon resource type has changed from **Pixmap** to **Bitmap**.

Form

The geometry handler now will now disallow geometry management requests that will force the child outside the **Form**'s window. EdgeType names have been changed to have prefix ‘‘Xaw’’ instead of ‘‘Xt’’, but the old definitions are still around with a #define. The string-to-widget converter no longer caches resources.

Logo

Changed to be a subclass of **Simple** instead of **Core**. Reverse video now works correctly.

Mailbox

Changed to be a subclass of **Simple** instead of **Core**. Reverse video now works correctly.

MenuButton

The **MenuButton** widget no longer places a server grab on itself. Instead, **PopupMenu** is registered as a grab action. As a result of this, clients which popped up menus without using **XtMenuPopup** or **MenuPopup** or **PopupMenu** in the menu button translations will fail to have a grab active. They should make a call to **XtRegisterGrabAction** on the appropriate action in the application initialization routine, or use a different translation.

Paned

Paned.h now includes **<X11/Xmu/Converters.h>** for the orientation resources.

Panner

This widget is new in R5, see the Xaw manual for details.

Porthole

This widget is new in R5, see the Xaw manual for details.

Repeater

This widget is new in R5, see the Xaw manual for details.

Scrollbar

Changed to be a subclass of **Simple** instead of **Core**. The type of thumb resource has changes from **Pixmap** to **Bitmap**. However, if applications provide the resource conversion, the **SetValues** method can still handle pixmaps of correct depth.

Simple

A color cursor converter has been added, as well as the new new resource types: **XtNpointerColor**, **XtNpointerColorBackground**, and **XtNcursorName**.

SmeBSB

The Right bitmaps are now painted in the correct location. Right and Left Bitmaps can be specified in resource files, and at startup time.

Text

If there is no current selection the the selection extends from the insert point, rather than some random location. Forward (Backward) Paragraph works at the paragraph boundaries now. Selecting a word now transitions correctly at both end points. An infinite loop when using fill paragraph in a read only text widget has been found and fixed. When the "resize" resource is set the text will start up with exactly enough space to contain the text in the widget. A bug that could cause an infinite loop when Meta-Q was used to invoke the form-paragraph function on a read-only text widget has been fixed. Problems dealing with exposure events have been fixed. In **TextP.h**, the names of the following symbolic constants have each had the prefix Xaw added to them: **XawLF**, **XawCR**, **XawTAB**, **XawBS**, **XawSP**, **XawDEL**, and **XawBSLASH**.

Toggle

The widget state is preserved across changes in sensitivity. A string-to-widget converter is registered for radio groups.

Tree

This widget is new in R5, see the Xaw manual for details.

Paned

A bug that caused **XtGeometryYes** returns to have bogus values, and caused panes to get random sizes, has been fixed.

Vendor

Support has been added for the **editres** protocol. All applications using the Athena Widgets are automatically editable with **editres**. A bug that cause **Shell** to ignore all but first child has been fixed.

Viewport

XawPannerReport support has been added.

11.9. X Server

Considerably more work speeding up the server has been done, particularly in the **cfb** and **mfb** code. The font interfaces are completely new. Compressed fonts are not supported in the release. Other changes are documented in **mit/doc/Server/r5.tbl.ms**.

11.10. Fonts

Font Server

A sample implementation of the server side of the X Font Service Protocol is provided in a new program, **fs**. On the Macintosh, a special version of the server called **MacFS** can be used to serve TrueType fonts.

New Font Format

Both the X server and the font server use a new font format, *pcf* (portable compiled font). Pcf is readable across different machines and contains more information about a font than the old snf format. Fonts in snf format can still be used.

Font Applications

The following new utilities talk to the font server: **fsinfo**, **fslsfonts**, **fstobdf**, and **showfont**. To build pcf fonts, **bdftosnf** has been replaced by **bdftopcf**.

Scalable Fonts

Bitstream, Inc. has donated an outline scaling engine along with a set of sample fonts, matching the donated bitmap fonts included with Release 4. In addition, a usable (but not very pretty) bitmap scaling engine has been implemented which allows the use of all other fonts at arbitrary point sizes.

Font Changes

Many **misc** fonts now have ISO Latin-1 upper half characters and some tuning. The following sets of fonts are new: Latin/Hebrew in ISO8859-8 encoding, Kanji in JISX0208.1983-0 encoding,[6] Hangul in KSC5601.1987-0 encoding.

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)

6. The JIS Kanji fonts were purchased on behalf of the MIT X Consortium from the Japanese Standards Association, 1-24, Akasaka 4, Minato-ku, Tokyo 107, Japan. They were converted to BDF format, keeping within the JIS rules. In keeping with JIS rules, these fonts should not be transformed into other encodings, they should only be used in the JISX0208.1983-0 encoding. It is also strongly recommended that companies wishing to use these fonts in commercial products should purchase the original JIS font standards directly from JSA. The purchase price is nominal.