CHAPTER 4

Application Development

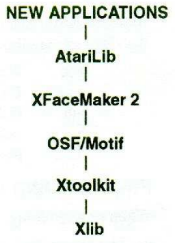
OVERVIEW

Use this chapter for designing and creating programs in the Atari System V environment. Some of the information you'll find here includes

- Brief descriptions of software development tools and libraries.
- Guidelines for creating window-based applications, and internationalized applications in particular.
- A summary of how to package an application.
- An overview of how to write a device driver program and the steps used to include it in the system.
- A section on rewriting existing TOS-GEM programs for the Atari System V windowing environment.

APPLICATION DEVELOPMENT LIBRARIES

User applications are developed on the Atari System V by writing programs in C programming language that use the functions and routines that are provided in several software libraries. Each library can be thought of as a layer.



Each layer is built upon the layers below it; that is, AtariLib makes direct calls to OSF/Motif routines, Xtoolkit routines, and Xlib routines, as well as XFaceMaker 2 routines.

Each layer, from the bottom up, contributes to the construction of a new application in the following ways:

- Xlib is library of basic windowing routines, such as mouse event, move window, size window.
- Xtoolkit is a library of windowing-associated widgets, including a scroll bar, a pop-up menu, and a toggle button.

- OSF/Motif is a library of windowing and widget routines that define the OSF/Motif style.
- XFacemaker 2 is an interactive application that creates the graphical interface of a new windowing application. It is also a library of interface routines.
- AtariLib is a library of graphical user interface routines, such as internationalization, alert boxes, and context-sensitive help.

Libraries in the Atari System V distribution include those shown in Table 4-1.

Table 4-1 Atari System V Libraries

Name	Description	Source
Atari	Atari library	Atari
Fm	XFaceMaker2	NSL
Xm	Motif	OSF
Xt	X Toolkit	MIT
X11	X Library	MIT
socket	Socket library	V.4
nsl	Socket hostname library	V.4
malloc	Network services library	V.4
gen	General purpose routiens	V.4
m	Math library	V.4

To link these libraries into your own program, enter the following lines near the top of your *makefile*.

```
LIBS=-lAtari -lFm -lXm -lXt -lX11 -lsocket \
-lsockhost -lnsl -lmalloc -lgen -lm
```

TOOLS

Included in the Atari System V set of software tools for applications development are the following GNU tools from the Free Software Foundation:

gcc	C compiler
g++	C++ compiler
gdb	C debugger
bison	compiler-generator
RCS	Revision Control System

PROGRAMMING NOTES

When programming, note the following:

- The object file format used in Atari System V is the Executable and Linking format (ELF), not the Common Object File Format (COFF) used in earlier releases of System V.
- When compiling X code, use **imake** to generate makefiles. The **imake** program automatically inserts the compiler flag **–DSYSV** to accommodate X11 header files in a System V environment.
- The debugger support format is called DWARF; it is supported by **gdb**.
- The gcc default compiler may issue calls to *gnulib*. Therefore, when porting an application to a platform that does not have it, include *gnulib* in the application package.

INTERNATIONALIZED APPLICATION DEVELOPMENT

Internationalization involves generalizing programs or systems so they can handle a variety of languages, character sets, and national customs.

All text visible to the user must be internationalized. That is, it must be displayed in the language of the environment variable LANG. This includes titles, label strings, icon names, product names, and format strings used to compose other strings. Even a format string as simple as %s:%s must be internationalized, since punctuation differs from language to language.

File names and log file output should not be internationalized. If you are developing an OSF/Motif-based application, routines in the Atari library simplify the internationalization process.

Within the source code, do the following:

1. **Initialize the program environment to be the language of the user's locale.**

   See the **environ(5)** manual page for a description of the environment variables that define a locale.

   The **setlocale** routine initializes the program environment for a particular language for one or more of the following variables:

   - LC_TYPE affects the behavior of character handling and multibyte character functions
   - LC_COLLATE affects string collation and transformation
   - LC_MESSAGES affects message catalog functions
   - LC_MONETARY affects monetary formats
   - LC_NUMERIC affects numeric formatting functions
   - LC_TIME affects date and time string conversions

   In addition, LC_ALL affects all of the above.

   Example:

   ```
   setlocale(LC_ALL,"")
   ```

   A value of "" for locale specifies that the locale should be taken from environment variables. Refer to the **setlocale(3C)** manual page for details.

2. **Use the following Atari System V macros and routines that automatically handle internationalization requirements.**

   See *Appendix E* to read how these routines correspond to the standard specified in the *X/Open Portability Guide*, Issue 3.

   Character routines that handle characters and strings according to the locale setting are

ctype(3C)	character handling
conv(3C)	character translation
mbchar(3C)	multibyte character handling
mbstring(3C)	multibyte string functions
strcoll(3C)	string collation
string(3C)	string operations
strxfrm(3C)	string transformation

The following local convention routines handle language-dependent representation of numbers, dates, and times. These functions are affected by the current locale setting.

nl_langinfo(3C)	retrieve local convention information from environment table
localeconv(3C)	get numeric formatting information
ctime(3C)	convert date and time to string
perror(3C)	print system error messages
printf(3C)	print formatted output
regexp(5)	regular expression matching routines
strftime(3C)	convert date and time to string
strtod(3C)	convert string to number
scanf(3C)	convert formatted input
vprintf(3C)	print formatted output

3. **Create and install a message catalog specific to the application.**

   gencat(1) produce a message catalog from a text file

   The output of the **gencat** command is placed in the directory

   /usr/lib/locale/$LANG/LC_ MESSAGES

   replacing *$LANG* with the appropriate language. For example, if the language is French from France, then *LANG=french_france* and the catalog would be placed in

   /usr/lib/locale/french_france/LC_MESSAGES

   In addition, the *gencat* file upon which all other translations are based should be placed in

   /usr/lib/local/C/LC_MESSAGES

   Translators may then **ungencat** the *message* file, translate it, **gencat** the translation, and move the file to the appropriate place for that language.

4. **Use the message catalog indicated by the locale setting.**

catopen(3C)	open message catalog specified by NLSPATH
catgets(3C)	retrieve messages from message catalog
catclose(3C)	close the message catalog

If possible, use the Atari Library routine **FmCatGetS** rather than these standard C library message catalog routines.

ADDING A LANGUAGE TO THE SYSTEM ENVIRONMENT

Table 4-2 shows locales supported by Atari System V environment files.

Use the following steps to add another locale:

1. **Decide on a language name.**

   `lang_territory.codeset`

   Since it is almost always ISO 8859-1, you can drop **.codeset**. **Territory** may also be optional. For instance, if the new language were Romansch, it need not be **romansch_swiss.8859-1**, but simply **romansch**.

2. **Create and install both a character translation table and a numeric representation table.**

Table 4-2 Environment File Locales

Locale	Description
english_usa*	American English
french_canada	Canadian French
danish	Danish
dutch	Dutch
english_uk	English
finnish	Finnish
french_france*	French
french_switzerland	Swiss French
german_germany*	German
italian_italy*	Italian
italian_switzerland	Swiss Italian
norwegian	Norwegian
portuguese	Portuguese
spanish	Spanish
swedish	Swedish
icelandic	Icelandic
C	Locale used when LANG not specified. Also used for translating to other locales, generally english.

*Default locale for this language

a. Construct an input file using the file supplied in */usr/lib/locale/C/chrtbl_C* as the starting point. (See manual page **chrtbl(1M).**)

b. Generate tables using the **chrtbl** command.

c. Install the two output tables in their respective directories, */usr/lib/locale/$LANG/LC_NUMERIC* and /usr/lib/locale/$LANG/LC_TYPE.

3. Create and install a character collation table.

a. Construct an input file that describes the collating sequence for the new language. You may use */usr/lib/locale/C/colltbl_C* as a starting point, but it may be more useful to use */usr/lib/locale/english_usa/colltbl_C*. (See manual page **colltbl(1M)** for format and content of this file.)

b. Use the **colltbl** command to generate a collation table.

c. Install the collation table in */usr/lib/locale/$LANG/LC_COLLATE*.

4. Create and install a table of monetary representations.

a. Construct an input file to describe formatting conventions for monetary quantities for the new language (see the **montbl(1M)** manual page for specifications). You can use the file */usr/lib/locale/C/montbl_C* as a starting point, but it may be more useful to use */usr/lib/locale/english_usa/colltbl_C*.

b. Create a monetary database table using the **montbl** command.

c. Install the monetary database table in the directory */usr/lib/locale/$LANG/LC_MONETARY*.

5. **Create and install a file of time representation.**
   a. See the format specified in the **strftime(4)** manual page. Use the file */usr/lib/locale/C/time_C* as a starting point.
   b. Install in directory */usr/lib/locale/$LANG/LC_TIME*
6. **Create and install message catalogs for applications that will be used in the new language.**
   a. Use the **cp** command to copy the original language version of the message file from */usr/lib/local/C/LC_MESSAGES;* using a text editor, change the text portions to the new language.
   b. Generate a formatted message catalog from the text source file.
   c. Install the formatted message catalog in the directory */usr/lib/locale/$LANG/LC_MESSAGES*

APPLICATION IMPLEMENTATION GUIDELINES

This section outlines recommendations for implementing XFM applications. It deals specifically with the interaction between the application and the */usr/lib/X11/Atari* library. All applications must conform to the Atari Style, as described in the *Atari Style Guide*.

Atari Library

The Atari library routines make it easier to conform to Atari Style. These routines are listed at the end of this section. On-line manual pages are listed in Appendix A.

Atari library routines facilitate the use of XFaceMaker 2 functions that

- integrate internationalization into an XFaceMaker application,
- implement both context-sensitive and general help, and
- implement pop-up alert boxes.

The Atari library provides functions that implement input callback for most application needs. For example, X library routines call back with an arbitrary amount of input data; use an Atari library function that will pack this data up in the amounts you want before calling your application back.

Window Construction

If there is only one primary window, it must be an XmApplicationShell. You must use XmTopLevelShell for the second and any subsequent primary windows. All primary windows must be fully decorated, must have XmMainWindow as their immediate child, and must have a menu bar.

You must use XmTransientShell for secondary windows, which may not have menu bars. They must have a row of buttons at the bottom and a separator above them. Secondary windows that are modal should always appear near the widget that caused them to appear. This is done automatically for Help and Alert boxes. Others may use **PositionNear()**. Modeless secondary windows may appear wherever the application designer thinks appropriate, but should defer to the window manager, if possible.

Secondary windows, even modal ones, must have a title bar so that they can be moved by the user, and they must not restrict the cursor to the window interior, which would prevent the user from consulting other applications.

All windows should be unmapped in their *.fm* file. After all *.fm* files are loaded, all active values attached, and all library initialization routines called, the application should use **FmShowWidget** to make the first primary window appear. The pointer to the shell widget may be obtained through an active value.

Internationalization

Within the source code, use the routine **FmCatGetS** to obtain an internationalized message. Mnemonics for the set number and message number should be declared with **#define** directives, instead of using plain integers. The message file (suffix *.msg)* should contain the mnemonic before the set or message declaration to aid in decoding.

For example, if you define

```
#define ERRORMSGS_SET 123
#define CANNOT_OPEN 32
```

the *.msg* file should contain something like

The dollar sign ($) is a comment sign for messages

```
$ ERRORMSGS_SET
set 123
$ CANNOT_OPEN
32 Cannot open that file.
```

- **FmCatGetS** uses **catgets(3I)** to obtain messages. These messages are read into a static space, so the string returned by FmCatGetS should be copied if it is to be used after the next call to FmCatGetS (or **catgets**).

Alert Popup Dialogs

Alert dialogs are used for all application errors. The only exception is when the **AlertInit()** function fails. In this case, the application writes a message to the log file and exits with a nonzero status. The alert displayed to the user will be of a nontechnical nature, with all technical and system call error information directed to the log file.

Fatal alert messages should be presented with the **InternalError()** function, which exits after displaying the message.

Nonfatal alert messages must be presented whenever a system call or library routine fails in a way that adversely affects the application, especially if the user provided the information, such as a file name, for the routine that failed. Some typical routines that may incur such errors are **fopen()** and **unlink()**.

Help Popup Dialogs

All applications must provide Help On Context and Help On Version. If an application uses a mnemonic not found in the OSF/Motif Style Guide, it must also provide Help On Keys. If it binds any custom mouse actions, it must also provide Help On Mouse. Help On Keys and Help On Mouse may be provided even though not required by the above conditions. Help On Help, Help On Window, Index help, and Tutorial help may be available, but are not required.

Running Subprocesses

The **system()** library routine should not be used to run processes, since it waits for the child to exit before returning and causes problems for the X11 Window

System's asynchronous event processing. Another drawback of the **system()** library routine is that it uses a shell process to execute the command, which should not be necessary. For this reason, the **popen()** routine should also be avoided. The **RunProcess()** function provides a way to run subprocesses with input/output redirection, including pipes.

Note that the asynchronous nature of X11 also requires that the application not block the waiting period for a subprocess to terminate. Thus, the SIGCLD signal must be caught by the application, and the wait for the terminated process must be done in the signal handler. To minimize the intrusion to X processing, the signal handler should limit its activity to obtaining the termination status of the process by means of the **wait(2)** system call and then registering a work process to do the actual signal handling. See the documentation on the X Toolkit routine, **XAppAddWorkProc**, for details on adding a work process.

Input/Output Handling

Any read or write operations that can potentially block may also interfere with the operation of the X11 Window System. When an application is programmed with all code in-line, effectively blocking whatever is waiting for file input or keyboard input—the window cannot be resized, iconified, moved, or otherwise manipulated during this wait time. In contrast, the window of an application programmed with input callbacks can be resized, iconified, moved, etc., regardless of where the data is.

For this reason, the X Toolkit routine **XtAppAddInput** allows input/output routines to be called only when the request may be done without blocking. Although the routine is called **XtAppAddInput**, it may in fact be used for detecting read, write, and exception conditions. Pipes, pseudo-TTYs, and network files are all especially susceptible to being blocked, on both reading and writing. Even ordinary files may be network files because of NFS and other transparent file systems, so it's a good idea to always use **XtAppAddInput** to handle input/output processing.

This input-callback programming technique is described in

- Nye, A., and O'Reilly, T.,.*X Toolkit Intrinsics Programming Manual, Volume Four*, Section 8, "Input Techniques" Subsection 8.3, "File,Pipe, and Socket Input"
- Nye, A. and O'Reilly, T., *X Toolkit Intrinsics Reference Manual, Volume Five,* "Xt Functions and Macros Subsection Event Handling, XtAddInput"
- Young, D. A., *The X Window System—Programming and Applications with Xt; OSF/Motif Edition*, Section 5.8.1, "Using Input Callbacks"

Log Files

Applications should log their progress using the logging routines provided in the Atari library. Every major state change, such as opening or closing a window, should be written to the log. More detailed information should use the **LogDebug** macro so it will not be compiled into the code for production.

Any abnormal condition, especially one resulting in a fatal alert message, should be noted in the log file.

The application must call **LogClose** upon normal termination in order to remove the log file.

Atari Library Routines

Help Subsystem:

HelpInit	Initialize help subsystem
PopupHelpAndWait	Present help box and wait for response
HelpOnContext	Provides help on context for application

Alert Subsystem:

AlertInit	Initialize alert box subsystem
InternalError	Present an internal error alert box, then exit
AlertSetButtons	Set buttons in an alert box
AlertHelp	Present help box for an alert box
PopupAlertAndWait	Present alert box and wait for a response

Convenience routines present a standard dialog box and wait for a response:

PopupErrorAndWait
PopupInformationAndWait
PopupMessageAndWait
PopupQuestionAndWait
PopupWarningAndWait
PopupWorkingAndWait

Interactive Command Execution Subsystem:

ExecuteList	Execute a command and a list of arguments
ExecuteListV	Execute a command and a vector of arguments
ExecuteString	Use the shell to execute a command in a string

Noninteractive Command Execution Subsystem:

RunProcess	Run a process with I/O redirection
RunProcessV	Run a process with command arguments as a vector
AddChildHandler	Add a child handler to a running process
GetCommandStatus	Run a process and call a function with exit status
GetCommandStatusV	Same, with command arguments as a vector

High-level Asynchronous Input/Output Routines:

ReadFileData	Read raw data from a file
ReadFileLines	Read lines from a file
ReadFileStrings	Read lines from a file and store in array of strings
ReadPipeData	Read raw data from a pipe
ReadPipeDataV	Like ReadPipeData, with command argument vector
ReadPipeLines	Read lines from a pipe
ReadPipeLinesV	Like ReadPipeLines, with command argument vector
ReadPipeStrings	Read lines from pipe and store in an array of strings
ReadPipeStringsV	Like ReadPipeStrings, with command argument vector

Low-level Asynchronous Input/Output Routines:

AddIoProc	Register an asynchronous I/O procedure.
RemoveIoProc	Remove an asynchronous I/O procedure
OpenReadPipe	Open a pipe for reading
OpenReadPipeV	Open a read pipe, with command arguments as a vector

OpenWritePipe	Open a pipe for writing
OpenWritePipeV	Open a write pipe, with command arguments as vector
OpenFilter	Run a filter process with pipes both in and out
OpenFilterV	Run a filter, with command arguments as a vector
GetPipePid	Get process ID of a pipe from its file descriptor
ClosePipe	Close pipe file descriptor

Application Logging Routines:

LogOpen	Open a log file
LogWrite	Write to a log file
LogDebug	Put debug message in log file
LogAssert	Check an assertion and report failure in log file
LogSystemError	Report a system error to the log file
LogClose	Close and remove a log file
LogReopen	Change a log file

Miscellaneous routines:

AddPath	Add an element to a path-like environment variable
AddPathV	Add a vector of elements to a path variable
AdjustWmPadding	Adjust padding values depending on resource values
FindWmPadding	Find out size of window manager decorations
FmCatGetSAlloc	Call FmCatGetS and copy result in a dynamic buffer
FmCatGetSRealloc	Call FmCatGetS and re-use given buffer
GetSimpleCharSet	Get default character set for OSF/Motif strings
PositionBeneath	Position a widget beneath another widget
PositionCenter	Position a widget centered over another widget
PositionNear	Position a widget near another widget
PositionOver	Position a widget over another widget
PositionRootCenter	Position a widget in the center of the root window

APPLICATION PACKAGING

You should bundle your application into an installable product so that it can be installed automatically using the system administration tool Product Installation, or the command **pkgadd**. There are no choices with regard to the directory in which the product will be installed, or to the parts of the package to install, unless incorporated into the installation portions of the package.

This section is an overview of how an application should be bundled into an installable product. For more detailed information, refer to the instructions, sample files, and scripts in the AT&T, *Unix System V Release 4 Programmer's Guide: System Services and Application Packaging Tools*, Chapter 8 "Packaging Application Software," Appendix B "Manual Pages," and Appendix C "Package Installation Case Studies."

1. **Create a file called *pkginfo*.**

   This ASCII file describes the application package name, release, and version numbers.

2. **Create a file called *prototype*.**

   This ASCII file has one entry per file that is a part of the application package, including the *pkginfo* and *request* files.

3. **Create an (optional) installation script.**

   This file may be a Bourne shell (sh) script, or may be an executable program. It can be a request, a class action, or a procedure script. Customize this script for the application package. Typical things an installation script can do are

   - Set up for selective installation (ask which parts of the package should be installed and where they should be placed).
   - Install a device driver (ask how many device nodes to create, run a postinstall script, and reboot the system upon installation).
   - Define extra disk space requirements required for this package (create a file called *space*).
   - Display a copyright message (create a file called *copyright*)
   - Define any software dependencies associated with this package (create a file called *depend*).
   - Modify a system file during installation.

   At the end of the request script, relevant parameters are made available to the installation environment for **pkgadd**.

4. **Run the pkgmk command, which will gather all components of a package, copy then onto the installation medium, and place them into a structure that pkgadd will recognize.**

DEVICE DRIVERS

Atari System V treats devices as special files that data is either read from or written to. These files are called device drivers. Files that provide interfaces to other system resources are called modules or "software drivers" (i.e., there is no physically removeable hardware device per se). Some examples of these device drivers and software modules are

- The block device driver that controls the hardware disk unit.
- The streams driver or module that controls the hardware terminal or the software terminal line-discipline module.
- The line discipline module.

Certain drivers are required to run the system—the keyboard driver, for example. Those shown in Table 4-3 and a few additional drivers are present in Atari System V.

Additonal drivers can be added to your system; write them as needed, or find someone who already has one that suits your needs.

A device driver must concern itself with three specific interfaces:

- the hardware device,
- the kernel, and
- the boot

The interface with the kernel is by means of data structures. See the *AT&T Device Driver Interface/Driver-Kernel Interface(DDI/DKI) Reference Manual*

Table 4-3 Atari System V Device Drivers

Driver	Function
IKDB	Intelligent keyboard
SCSI	Generic SCSI manager
HD	SCSI hard disk driver(interfaces with SCSI driver)
TP	SCSI tape (interfaces with SCSI driver) archive viper tape streamer
FFD	720KB floppy disk
CEN	Centronics parallel port
SCCIO	Two serial ports on Zilog 6530 SCC chip
USART	Two serial ports on MFP chips
LA	VME Ethernet board driver
VIDEO	Video subsystem
PSG	Programmable sound generator
CLOCK	Real-time clock
MFP	Multifunction peripheral/interrupt controller

for the System V.4 protocol approved by AT&T. There may be some name changes for writing device drivers in a Motorola 68000 environment, but the functionality should be the same.

If you are writing a device driver for a SCSI device, refer to the on-line "Guide to Writing Device Drivers for the Generic SCSI" located in the file */usr/local/src/samples/scsidriver/scsigen.doc*.

The source for a sample SCSI printer driver is available on-line in the file */usr/local/src/samples/scsidriver/pr.c*. Compile it as follows:

```
gcc -Xt -D_KERNEL -O -c pr.c
```

Adding a Device Driver

1. **Create an object file (in ELF format) with gcc, which will be added to the kernel. If a device uses several *a.out* files, link them together.**
2. **Create a master(4) file in */etc/master.d* directory.**
3. **If it is a software-only driver, add an entry to the */stand/system* file.**
4. **If the driver is for a hardware device, add entry to */stand/edt_data* file.**

   You cannot use **edittbl(1M)**, because the format of *edt_data* has been changed in the Atari System V release.

5. **Install the driver with drvinstall(1M). This creates a boot module from the master file and the driver object file and places it in /boot.**
6. **If the driver is for a hardware device, install a boot probe program in */stand*. When the system boots, it looks for these.**

   In */usr/src/uts/boot/probe* there is a probe program called **naiveprobe.c** that checks for a bus error when performing a byte read at the device base address. For straightforward cases this may be compiled unmodified for each device, with the define DEVNAME variable set to the appropriate device name by means of a **–DDEVNAME** flag in the makefile **probe.mk**. For more complex cases, dedicated probe programs may be written for each device. The *xedt* library contains the necessary structures.

7. **If the driver is a hardware device, make special files with *mknod(1M)*.**
8. **Reconfigure the kernel.** Refer to Chapter 3, "System Reconfiguration."
9. **Reboot and use the device.**

PORTING TOS/GEM APPLICATIONS

There is no simple rule for converting GEM programs to OSF/Motif. Both are graphical user interfaces with a windowing system and have several features in common.

Porting By Means of XFaceMaker 2

Atari System V uses the OSF/Motif widget set and the XFaceMaker 2 interface builder, which allow the programmer to concentrate on the functionality of the application. XFaceMaker also hides the complexity of the X Window System.

However, the XFaceMaker 2 library is more advanced, including such features as a built-in C-like language, callbacks, active values, and the availabiltity of powerful widgets.

As a GEM programmer, you are familiar with writing graphical user interface software. The software you want to port usually consists of two parts: the application and the user interface.

Application

If the application was written in C, it is easy to transfer it to Atari System V and recompile it. Calls to the TOS operating system must be converted to the equivalant Atari System V operating system calls. The most compatible way is to use the C standard library functions—**fopen**, **fwrite**, etc. (Refer to the section "Input/Output Handling" in this chapter.

User Interface

The user interface must be redesigned. OSF/Motif has a different look and feel than GEM and has new graphical objects you may want to use. Also, the application now has to run in a multi-tasking environment.

To start, redesign the user interface using XFaceMaker 2. Create a main window that contains the menu bar. The contents of the window should be the contents of your main window (form) in the GEM application.

Forms and Windows

Under the X Window system, windows are used, rather than forms; these windows may overlap. The windows controlled by the window manager have a title bar and are similar to GEM windows.

Thus, a form dialog box in GEM must be implemented as a window under Atari System V. XFaceMaker 2 helps the developer create windows that behave in the same way as dialog forms.

XFaceMaker 2 allows you to create a window and place interactive widgets and text edit fields inside it. You can specify callback functions that are the names of functions in the program you are writing. These functions will be called by the XFaceMaker 2 library when this object is selected.

After creating the window, save it and write the program. Usually the XFaceMaker 2 library opens the window automatically and handles all events. If an object such as a button is selected, the XFaceMaker 2 library calls the callback fucntion specified for the object and acts on the user's input. When using the XFaceMaker 2 library, the application developer has less to do than when using GEM.

To open a window without using XFaceMaker 2, use the Xt Toolkit. To draw circles, rectangles, etc., inside a window use the Xlib. It's similar to drawing with the VDI inside a GEM window. If you use Xlib drawings, redrawing the window is more complicated. It's possible to draw with Xlib functions inside a window created by XFaceMaker 2. See Appendix D for a table of GEM/Xlib equivalents.

Main Loop

The main loop in a GEM program is usually the **event_multi()** loop which waits for events to occur. The application then determines the type of event that occurred and calls the appropriate function.

With XFaceMaker 2, this is not necessary. Once the main event loop **(FmLoop)** has been called, your functions that react on events are called automatically.

During the design of the user interface of the application using XFaceMaker 2, you specify callback functions for interactive objects. The XFaceMaker 2 libary will call your function when an action takes place on that object.

For instance, you might create a window with one button. When the button is pressed, the window closes and the application exits. Design the window with XFaceMaker 2 and place the button inside it. Specify the name of the callback function for the button via the XFaceMaker 2 resource window using the activate callback resource. For example, use **button_pressed()**.

After saving the user interface you write your program. The core program in general has to contain only two functions:

- The **main()** function, calling **FmInitialize()**, **FmAttachFunction()** to connect the C-function with the user interface and the **FmLoop()**.
- Your callback function for the button in this example, **button_pressed()**. This function calls **exit(0)**, which causes the window to close and the application to exit.

The **FmLoop()** is like calling the **event_multi()** function in GEM and parsing the occured events, except that the parsing is done automatically by X and the XFaceMaker 2 library. Because the library knows the names and addresses of your functions, it is able to call them.

Porting Example

There is a small GEM address application available which was ported to OSF/Motif. There you are able to see how the GEM look and feel was converted to Motif/X Windows following the Atari Style Guide. This GEM application is on-line in */usr/local/src/samples/gem2motif*. The GEM example is in *subdirgem* and the Motif example in *subdirmotif*

REFERENCES

Atari Computer Corp., *Atari Style Guide*, Atari, 1991

AT&T, *UNIX System V Release 4* Documentation, Prentice Hall, 1990:

Device Driver Interface/Driver Kernel Interface(DDI/DKI) Reference Manual
Programmer's Guide: STREAMS
Programmer's Guide: Networking Interfaces
Programmer's Guide: BSD/XENIX Compatibility Guide
Programmer's Reference Manual
System Administrator's Guide
System Administrator's Reference Manual

environ(5)
langinfo(5)
nl_types(5)
chrtbl(1M)
colltbl(1M)
montbl(1M)

System Services and Application Packaging Tools,
Chapter 8, "Packaging Application Software"
Appendix B,
Appendix C,

Non Standard Logics, *XFaceMaker 2 Programmer's Guide*, Paris, France, 1991

Non Standard Logics, *XFaceMaker 2 User's Guide*, Paris, France, 1991

Nye, A. and O'Reilly, T., O'Reilly & Associates, 1990:

Volume One: Xlib Programming Manual
Volume Two: Xlib Reference Manual
Volume Four: X Toolkit Intrinsics Programming Manual
Volume Five: X Toolkit Intrinsics Reference Manual

Open Software Foundation, *OSF/Motif Programmer's Guide*, Prentice-Hall, 1990

Open Software Foundation, *OSF/Motif Style Guide, Revision 1.1*, Prentice-Hall Inc., 1991

Open Software Foundation, *OSF/Motif Programmer's Reference*, Prentice-Hall, 1991

Young, Douglas, *The X Window System—Programming and Applications with Xt—OSF/Motif Edition*, Prentice-Hall Inc., 1990

X/Open Company, Ltd., *X/Open Portability Guide, XSI Supplementary Definitions*, Prentice Hall, Inc., 1989