Sprite Engineering Manual

John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This document provides general information for people who will write or maintain Sprite commands or kernel code. It describes the layout of the Sprite file system and documents our conventions for program structure, coding style, documentation, and bug reporting.

1. Introduction

As its name implies, this is a manual for people who will provide engineering support for Sprite. It is intended for people who work on system programs like **csh** or **cc**, people who work on the Sprite kernel, and anyone else who wishes to use a standard structure for code and documentation.

This manual describes the conventions that we have developed for organizing programs, libraries, kernel code, and their associated documentation. The purpose of the conventions is to provide a standard framework, so that all our code will have a uniform style, and so that it will be easy to use, read, and maintain programs written by other people. In addition, the conventions guarantee that certain things get done: for example, that all procedures have a minimum level of documentation.

The rest of this manual is organized as follows. Section 2 describes the directory structure of Sprite's file system. Section 3 discusses how we manage different machine types in the same Sprite system. Section 4 shows how files are organized in directories containing source code, and Section 5 describes how to structure an individual code file. Section 6 discusses the naming conventions we use in Sprite, and Section 7 presents additional syntax conventions for comments, curly brace placement, and other low-level details. Section 8 describes the organization of include files. Section 9 presents the notion of callbacks and ClientData, which we use to make Sprite code more modular and extensible. Sections 10 and 11 present our conventions for documentation (in-line comments in Section 10, manual entries in Section 11). Section 12 gives additional information about the kernel source directories, and Section 13 describes how to report bugs and query the database of bug reports.

2. File System Directory Structure

The Sprite file system has a different directory structure than traditional UNIX file systems. We did this partly to eliminate idiosyncracies that have built up in UNIX over the years, and partly to make it easier to support different machine types and independently-distributed subsystems. In order to support UNIX programs, most of the standard UNIX directories and files also exist under Sprite as symbolic links to the corresponding Sprite directories.

2.1. The root directory

The root directory in a Sprite system always contains at least the following subdirectories:

/hosts

Contains one subdirectory for each machine. Each subdirectory contains files relating specifically to that host. Most of these files are I/O devices and pseudo-devices for things on that host, such as pseudo-devices for the host's X server and for rlogin connections to that host.

/sprite

Contains all of the information related to the Sprite kernel and system programs, including source code, binaries, libraries, documentation, and administrative information. See Section 2.2 below for more details.

/swap

This directory contains one subdirectory or symbolic link for each machine in the Sprite system; its name is the machine's integer Sprite identifier. The subdirectory for each machine is used by that machine's kernel to create backing files for virtual memory.

/tmp This directory is used by many programs to create temporary working files. No long-lived information should be stored here. Any files in this directory that are older than a few days may be deleted without notice.

The entries in the root directory are often symbolic links to other mounted file systems, in order to make better use of the disks mounted in the Sprite system. However, the official names for all of these directories are the ones given above.

The root directory will typically contain many other entries in addition to the ones given above. The additional entries are mostly top-level directories for other disk structures containing user files and separate subsystems.

Major additional subsystems should appear as additional subdirectories of the root directory. For example, the X window system and its associated programs are all stored in the subtree **/X**. Site-specific system programs should be stored in a subtree named after the site, **/ucb** for us. The information kept in each subsystem should generally correspond to a collection of programs and libraries that are distributed together. Thus the core Sprite system code is all in one subsystem, **/sprite**, and the X-related information disributed by MIT is under **/X**, and so on. This arrangement allows new versions of each subsystem to be installed without impacting other subsystems. Ideally, each of subsystem should have the same basic structure as **/sprite**, which is described below; in some cases, however, it makes more sense to retain the organization in which the subsystem is distributed, rather than the standard Sprite organization.

2.2. /sprite

The subtree under **/sprite** contains information related to the Sprite kernel and standard system programs. This includes source code, binaries, libraries, and documentation. We expect this information to be present in every Sprite system.

For most things in Sprite (include files, library archives, program executables, etc.) there are two versions: an installed stable version and an uninstalled version in a source directory. You should never edit an installed version directly, except in the few cases where there isn't a separate uninstalled version. Normally, you make changes in the uninstalled source directory and test them there before installing. When everything appears to be stable, you use the **make install** command to copy things to the installed area.

Figure 1 shows the basic structure of **/sprite**. Some of the more important subdirectories contained in **/sprite** are:

/sprite/admin
Contains administrative information about the system, plus programs used only by system administrators.

/sprite/cmds
A symbolic link to the directory **/sprite/cmds.***machine* containing binaries suitable for the machine on which you are currently executing. See Section 3 below for more details on machine dependencies like this.

/sprite/cmds.*machine*
These directories contain the installed executable versions of all the programs used by normal users. Each of these directories is equivalent to the combined contents of the UNIX directories **/bin**, **/usr/bin**, **/usr/ucb**, and so on. There is one of these directories for each

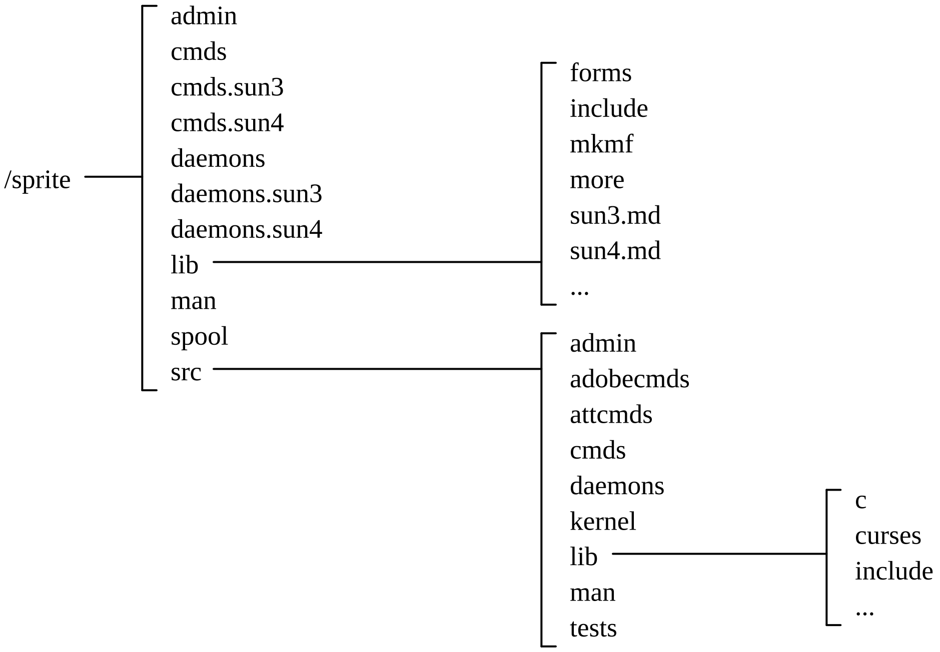

Figure 1. The directory structure under **/sprite**.

machine type supported by Sprite (such as **/sprite/cmds.sun3** or **/sprite/cmds.sun4**), which holds executables compiled for that particular machine type. See Section 3 below for more information on machine dependencies.

/sprite/daemons

A symbolic link to the directory **/sprite/daemons.***machine* corresponding to the machine on which you are currently executing.

/sprite/daemons.*machine*

These directories contain the installed executable versions of daemon programs that execute automatically in background, such as the network servers. There are several of these directories, each containing executables compiled for a particular *machine*.

/sprite/lib

Contains installed versions of library files used by system programs. See Section 2.3 for more information.

/sprite/man

Contains installed versions of all the entries in the Sprite reference manual.

/sprite/spool

Used by a few major subsystems like **mail** and **lpr** to store information temporarily while it is waiting to be processed.

/sprite/src

Contains the complete source code for the Sprite kernel and all the programs in **/sprite/admin**, **/sprite/cmds**, and **/sprite/daemons**. See below for more information.

2.3. /sprite/lib

The directory tree under **/sprite/lib** contains the installed versions of files that are referenced by system programs while they run. Typically, information in the library changes only slowly: it is generally used in a read-only fashion. Information in the library includes system **.h** files, object files for the C library, font information for **ditroff**, and many other things.

Our convention for the library area is that the library files for a particular program are stored in a subdirectory of **/sprite/lib** named after the program. For example, **/sprite/lib/mkmf** contains shell scripts and Makefile templates used by the **mkmf** Makefile generator, and **/sprite/lib/more** contains help files displayed by the **more** program.

There are a few subdirectories of **/sprite/lib** that are not named after programs. Some of the more important ones are:

/sprite/lib/forms
: The files in this directory contain standard templates used when writing Sprite code. For example, **proto.c** is a prototype C file with a standard Sprite file header, **prochead** is a standard procedure header, and **proto.csh** is a template for a C shell script. Please use the files in this directory when programming, so that your code looks like all the other Sprite code.

/sprite/lib/include
: This directory and its subdirectories contain the installed versions of system-supplied C include files, like **stdio.h** and **sys/types.h**. The C compiler automatically looks here when processing **#include** statements.

/sprite/lib/*machine***.md**
: These directories contain archive files holding the system libraries, like **libc.a**. For each library *x*, there is a file **lib***x***.a** that holds the normal version of the library, a file **lib***x***_g.a** that holds a version of the library with debugging information, and a file **lib***x***_p.a** that holds a profiled version of the library. The debuggable and profiled versions, if they exist, are usually symbolic links to uninstalled versions in the library source directories. There is one **.md** directory for each *machine* supported for Sprite, containing library archives compiled for that machine. If you specify the **–l***x* (or **–l***x***_g** or **–l***x***_p**) switch to **cc** or **ld**, the linker will look for the corresponding archive in the appropriate **.md** subdirectory.

As with all the other information in **/sprite**, the libraries in **/sprite/lib** pertain to the programs in **/sprite**. Libraries for programs in other areas, like **/X** or **/ucb**, are stored in directories like **/X/lib** and **/ucb/lib**.

2.4. /sprite/src

The subtree rooted at **/sprite/src** contains the source code for the Sprite kernel and all the user-level programs, include files, libraries, and manual pages whose installed executables lie under **/sprite**. In most cases the subdirectories of **/sprite/src** correspond to the subdirectories of **/sprite**: each directory **/sprite/src/***x* contains the sources for information in **/sprite/***x*. However, there are a few exceptions to this rule, the most notable of which is the source for **/sprite/cmds**. First of all, there are not separate source areas for different machine types (see Section 3 below for more on this). Second, we split up the command sources according to licensing restrictions. **/sprite/src/cmds** contains source code that is not subject to any licensing restrictions, **/sprite/src/attcmds** contains source code that is licensed by AT&T, and so on.

Some of the subdirectories of **/sprite/src** are:

/sprite/src/admin
Contains source code for all the programs whose binaries are installed under **/sprite/admin**. Each subdirectory corresponds to a single program.

/sprite/src/adobecmds
Contains source code for some of the programs in **/sprite/cmds**. Programs in this subdirectory are subject to licensing restrictions imposed by Adobe, and mostly relate to Postscript and printing. Each subdirectory corresponds to a single program.

/sprite/src/attcmds
Contains source code for the programs in **/sprite/cmds** that are subject to AT&T licensing restrictions. All of the programs here were taken from the BSD distribution, and thus have the same licensing restrictions as BSD code. Each subdirectory corresponds to a single program.

/sprite/src/cmds
Contains source code for the programs in **/sprite/cmds** that may be distributed freely, without any licensing restrictions. Each subdirectory corresponds to a single program.

/sprite/src/daemons
Contains source code for the programs that are installed in **/sprite/daemons**. Each subdirectory corresponds to a single program.

/sprite/src/kernel
Contains source code for the Sprite kernel. See Section 12 for more information.

/sprite/src/lib
Contains sources for some of the information in **/sprite/lib**. See Section 2.5 below for more details.

/sprite/src/man
Contains sources for some of the entries in the Sprite reference manual. The sources for manual entries describing programs and library procedures are kept in the same directories as the source code for the programs and libraries. The directory **/sprite/src/man** contains sources for manual entries that don't correspond to a single piece of code, such as those describing file formats and I/O devices. The subdirectories under **/sprite/src/man** correspond to (some of) the subdirectories under **/sprite/man**.

/sprite/src/tests
Contains source code for a number of programs used to test the Sprite system. Information here doesn't get installed anywhere else; it's used directly from this directory when needed.

2.5. /sprite/src/lib

The directory **/sprite/src/lib** holds installed versions of some of the information installed under **/sprite/lib**. Most of the subdirectories under **/sprite/src/lib** correspond to the library archives in **/sprite/lib/*.md**, with each subdirectory containing all of the source files for a single **.a** file in **/sprite/lib/*.md**. In other cases, the directories under **/sprite/src/lib** hold uninstalled versions of directories by the same name under **/sprite/lib**.

Some of the directories under **/sprite/lib** have no corresponding directories under **/sprite/src/lib**. In most of these cases the uninstalled library files are in the source directories for particular programs (e.g. the sources for **/sprite/lib/more** are in **/sprite/src/cmds/more**). In a few cases, the directory **/sprite/lib** serves as its own uninstalled directory. This is probably a bad idea, but still occurs for a few directories like **/sprite/lib/pmake** and **/sprite/lib/mkmf**; these directories are marked by the presence of an **RCS** subdirectory.

Some of the subdirectories of **/sprite/src/lib** are:

/sprite/src/lib/c
Holds all the sources for the standard C library, which is installed in **/sprite/lib/*.md/libc.a**, **/sprite/lib/*.md/libc_g.a**, and **/sprite/lib/*.md/libc_p.a**.

/sprite/src/lib/curses
Holds all of the sources for the **curses** library, which is installed in **/sprite/lib/*.md/libcurses.a**, **/sprite/lib/*.md/libcurses_g.a**, and **/sprite/lib/*.md/libcurses_p.a**.

/sprite/src/lib/include
Holds the uninstalled source versions of the **.h** files corresponding to the C library (e.g. **stdio.h**) and the operating system (e.g. **sys/types.h**). For the other libraries (anything for which you have to specify a **–l** switch when compiling, like **curses**), the include files for that library have their sources with the library's source code.

3. Machine Dependencies

One of our goals for Sprite is to allow machines of different types (e.g., workstations made by Sun, H-P, and DEC) to coexist in a single Sprite system, sharing a single file system and having uniform access to all the resources of the system. Unfortunately, the differences between the machines introduce at least three problems. First of all, the different instruction sets require different versions of executable binaries. Second, some of the Sprite source code must be different for different machines. Most user-level programs will not have any machine dependencies, but there will be some machine dependencies in the library routines and even more in the kernel (e.g., different machines have different virtual memory architectures and different I/O systems). Third, whenever information is stored in the file system as binary files (rather than ASCII text) it may be difficult for one machine to read a file written another machines, due to machine dependencies such as floating-point format or byte order within integers.

3.1. Use ASCII whenever possible

The third problem is the easiest to solve. Information stored in files should always be in ASCII except when binary format is required by the most dire performance problems. If you feel that you must read and write files in binary format (for example, by dumping data structures directly to files), you should first take enough performance measurements to be certain that binary format is necessary. Second, when you write a file in binary format, the file should include the type of machine in whose format the file is written. Your code for reading the file back in again should check the format type and reformat the data (e.g., by byte-swapping) so that it can be processed correctly on any machine that reads it. The **Fmt_Convert** library procedure provides a convenient mechanism for performing byte-swapping and other kinds of reformatting. However, when the overhead of reformatting binary data is included, your code is unlikely to be much faster than if you had stored the data in ASCII to begin with.

The first two forms of machine dependencies (executables and source code) cannot be legislated away, so we have developed a framework for dealing with them in Sprite. One alternative, which we decided *not* to use for Sprite, is to place all the machine-dependent files in one area of the file system. For example, **/md/sun3** and its subdirectories might contain all the Sun3-dependent code, **/md/sun4** all the Sun4-dependent code, and so on. The problem with this approach is that the machine-dependent parts of programs would be stored a long way from the machine-independent parts, which would make it difficult to manage individual programs.

3.2. Machine dependencies go in .md subdirectories

For Sprite, we decided to put the machine dependencies at the leaves of the file system, within individual modules, programs, and libraries. In other words, the top levels of the file system branch out based on the different functions provided by the system. All the code for each function is together in the same subtree of the file system, including both machine-independent and machine-dependent things. Only the bottom-most directories of the file system contain machine dependencies.

Whenever a situation arises where some of the files in a directory are machine-independent and some are machine-dependent, the directory must be restructured into several directories as shown in Figure 2. The original directory should contain the machine-independent files plus one subdirectory for each type of machine. The subdirectories should have names of the form *machine***.md**, where *machine* is the name of a machine type. Each subdirectory should hold all of the machine-dependent files for a single machine type. A **.md** directory should never have subdirectories of its own except for an **RCS** subdirectory. In other words, machine dependencies should be the last thing you see when exploring the Sprite file system.

There is only one exception to this rule that I know of, and that is for the include files in **/sprite/lib/include** and **/sprite/src/lib/include**. The machine-dependent subdirectories of these directories contain further subdirectories; this structure is needed in order to allow the C compiler to locate machine-dependent versions of include files like **<sys/param.h>**.

3.3. Special case for /sprite/cmds

For directories holding the installed executable versions of programs, we took a shortcut to the above approach. Instead of having subdirectories **/sprite/cmds/sun3.md**, **/sprite/cmds/sun4.md**, and so on, we simply replaced the **/sprite/cmds** directory with several directories, one for each machine type: **/sprite/cmds.sun3**, **/sprite/cmds.sun4**, and so on. This eliminated a level of directory lookup for commands, but the non-uniformity it introduces is a

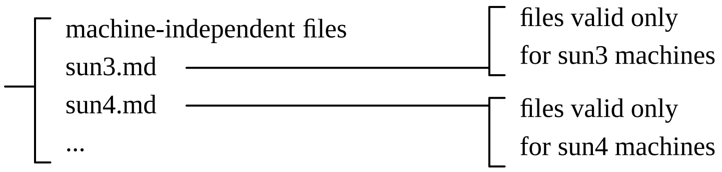

Figure 2. When a collection of files contains some that are machine-dependent and some that are not, the machine-independent files go in the base directory, and the machine-dependent files (like object files) go in *machine***.md** subdirectories, one for each machine type supported.

wart. Also, this shortcut makes it hard to add machine-independent files later on: all the programs that refer to the directory would have to be changed. In retrospect, this probably wasn't a good idea, so it should never be used except in the special case of installed binary executables.

3.4. $MACHINE in path names

The Sprite filesystem provides a special feature to help hide machine differences. If a file name or symbolic link contains the string **$MACHINE**, then whenever the name is looked up the Sprite filesystem will replace **$MACHINE** with the type of the machine from which the lookup was invoked. For example, the file **/sprite/cmds** is a symbolic link to **cmds.$MACHINE**. If a process running on a Sun-4 executes the file **/sprite/cmds/csh**, then the actual file executed will be **/sprite/cmds.sun4/csh**. This makes it possible to execute a program without having to worry about what machine type you're running on. Judicious placement of symbolic links containing **$MACHINE** strings simplifies the porting of old applications that were written without knowing about Sprite's handling of machine dependencies.

3.5. Ifdefs are a bad way to handle machine dependencies

It is common practice in other systems to use **#ifdef** statements in C code to handle machine dependencies. We strongly discourage this practice in Sprite. **#ifdef**s result in contorted code that quickly becomes impossible to read, even with a small number of machine types. Given our goal to support a large number of machine types, **#ifdef**s aren't workable. Instead, modules should be defined with clean interfaces between machine-dependent code and machine-independent code. Collect all the machine dependencies into a few small procedures, and move those procedures to a machine-dependent file in a **.md** subdirectory.

The use of **#ifdef**s to handle machine dependencies is strictly forbidden in ''machine-independent'' code: this would require the code to modified to support new machine types, and thereby makes it machine-dependent.

However, we occasionally use **#ifdef**s in machine-dependent files when there exist machines that are nearly identical. For example, the Sun-2 and Sun-3 architectures are so similar that much of the machine-dependent code for these two machines is shared in Sprite, with **#ifdef**s for the few differences. The **.c** files in **sun2.md** subdirectories are symbolic links to the same files in the associated **sun3.md** subdireocties. This is a rare exception, though; it's usually better to segregate the code for different machines.

3.6. Support for cross-compilation

We extended our C compiler and many other program management tools (e.g., **ld**) to provide special support for cross-compilation. For example, both **cc** and **ld** support a **−m** switch. If the switch **−m***target* is given, then *target* specifies the machine type for which the program is being compiled. The target defaults to the machine type being used to do the compilation. Thus, for example, **ld** will look for library binaries in **/sprite/lib/***target***.md**.

We also changed the C compiler and **make**-related programs is to support machine-dependent include files. For example, machine-dependent system include files for different machines are stored in **/sprite/lib/include/*.md**. The C compiler automatically looks for include files in both **/sprite/lib/include** and **/sprite/lib/include/***machine***.md**, where *machine* is the machine being compiled for. The **mkmf** program, which automatically generates most of the Sprite Makefiles, also arranges for the C compiler to check for include files in **.md** subdirectories associated with the code being compiled; this allows individual programs and libraries to have their own machine dependencies. See Section 4 below and the **mkmf** manual entry for more details.

4. How to Organize A Source Directory

For each program or library in Sprite there is a separate source directory whose name is the same as the program or library. Different programs do not cohabitate the same directory. The only exception to this rule is when one program is a symbolic link to another (e.g. **vi** is a symbolic link to **ex**); in this case there need not be a separate directory corresponding to the symbolic links.

4.1. Little programs

Source directories take one of two forms. Small programs and libraries use the simple form shown in Figure 3, consisting of a top-level directory containing machine-independent sources plus a collection of **.md** subdirectories. Each **.md** subdirectory corresponds to one machine type, and contains things specific to that machine, such as object files compiled for the machine, the executable binary or library archive for the machine, and machine-dependent source files if any are required. Each of these directories has an **RCS** subdirectory to hold the RCS master files for the sources in the directory. The files **Makefile**, ***.md/md.mk**, and ***.md/dependencies.mk**

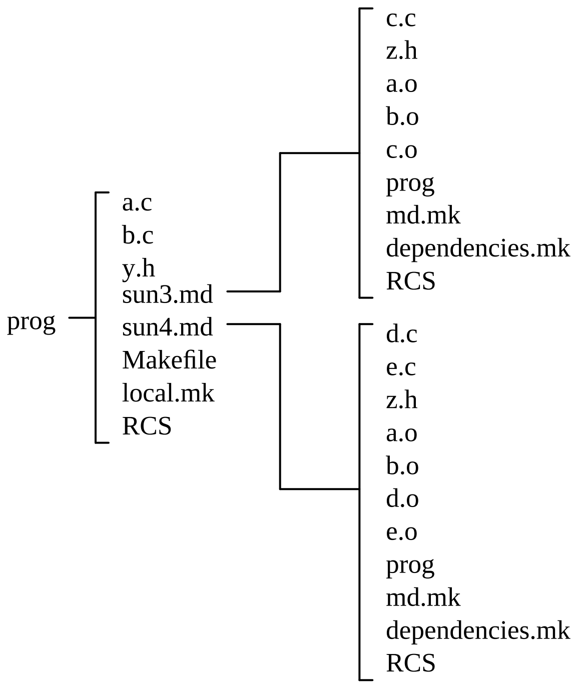

Figure 3. The source code for a small program is stored in a single machine-independent directory and its **.md** children. In this example there are different versions of the include file **z.h** for different machine types (the right version will be included when compiling for each machine), as well as different **.c** files. The contents of the **RCS** subdirectories are not shown.

are all generated automatically by **mkmf** and should never be modified by hand. Instead, the file **local.mk** should be used to modify flags or otherwise customize **Makefile**. See the **mkmf** manual entry for more information on how **make**-related information is organized.

4.2. Big programs

For some large programs and libraries there are too many files to be managed in a single directory. These things are called ‘‘big commands’’ and ‘‘big libraries’’. A big command’s directory structure is illustrated in Figure 4: the sources are not contained in the top-level directory, but rather in a collection of subdirectories, each of which in turn has **.md** subdirectories. The top-level directory also has **.md** subdirectories, which hold the complete version of the program or library, assembled from the lower-level **.md** subdirectories.

4.3. RCS and mkmf

We use RCS for version control on *all* our source files. Every file that is edited by a human being should be managed under RCS, using an **RCS** subdirectory to hold the master files. The

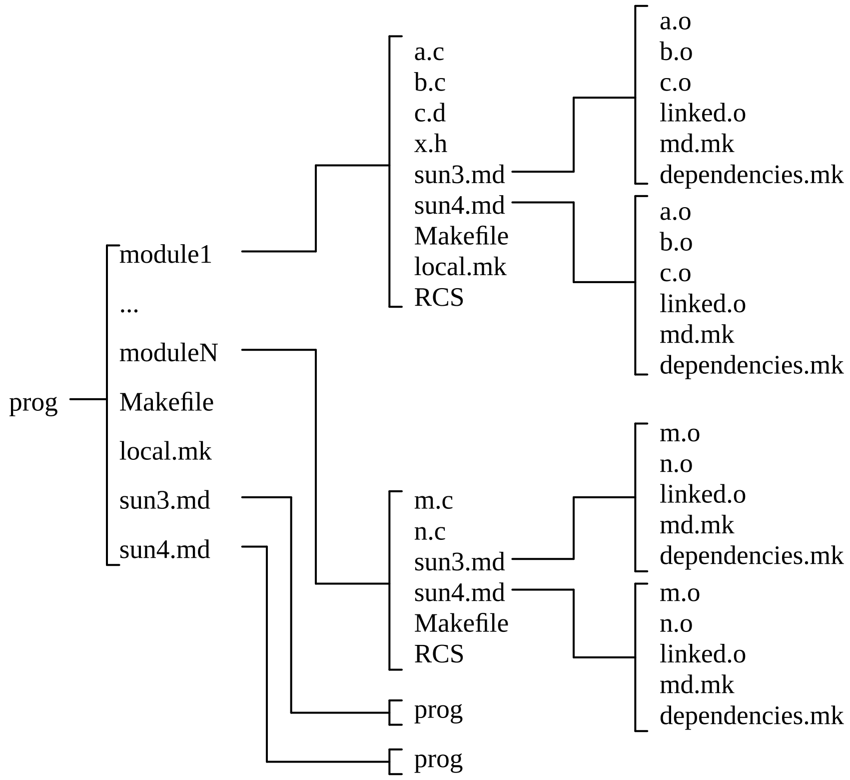

Figure 4. A large program has its sources spread over several directories. The top-level directory contains child directories and **.md** directories in which the final result is stored. Each second-level directory contains one or more machine-independent files, plus **.md** subdirectories to hold **.o** files and machine-dependent sources. In the case of a program, all of the **.o** files from each second-level directory are combined into **linked.o** files in the lowest-level **.md** subdirectories. The **linked.o** files are linked again into a final result (**prog**) placed in the top-level **.md** subdirectories.

only exception to this rule is for files that have been copied from some other distribution source (e.g. BSD) and have not been modified on Sprite. In this case, the file should be left read-only and no RCS master file should be created. The lack of an RCS master file indicates that no changes have been made for Sprite, which makes it easier to bring in new releases of the program.

RCS-ed files should be normally be left checked out but unlocked. You should lock files only long enough to make changes and test them.

We use **mkmf** to generate all of the Makefiles for Sprite source directories, plus the files **md.mk** and **dependencies.mk** in machine-dependent subdirectories. You should almost never generate a Makefile by hand. In general, all you should have to do is create source files in the right places and then type **mkmf** in the source directory; everything else should happen automatically, leaving you with a Makefile that provides a number of standard targets such as **make**, **make install**, **make clean**, and so on. If you need to do things a little differently than the **mkmf** defaults, create a **local.mk** file in the directory to augment or modify the default actions. Read the **mkmf** manual entry for details on how **mkmf** works.

4.4. Imported programs

If a program is imported from somewhere else it may be painful to recast the program from the structure in the program's distribution to the Sprite form described above. If the program is small, or if new versions of the program are imported infrequently, then the best thing is just to re-organize the program to look like other Sprite programs. However, we occasionally use an alternate approach for large programs that are being updated frequently, particularly if the program doesn't require many code changes to run on Sprite.

For such a program, the source directory should contain two additional subdirectories, **dist** and **sprite**. The **dist** subtree should contain all of the program's sources in exactly the form in which the program is distributed, complete with subdirectories etc. No changes should ever be made to files in the **dist** subtree. If any of the source files need to be modified to run on Sprite, the modified files should be stored in the **sprite** subdirectory (with a subdirectory **sprite/RCS** for the RCS master files).

The source directory should also contain a Sprite-like source code structure, except that all of the source ‘‘files’’ should be symbolic links to the corresponding files in the **dist** or **sprite** subdirectories. This allows **mkmf** and **make** to be used in the same way as for other Sprite programs.

The structure described above is designed to make it easy to import new releases of the program. Since the **dist** subdirectory is never modified, it can simply be replaced with the program's newest release. If no code changes are needed for Sprite, then the program need only be recompiled. If there are Sprite modifications in the **sprite** subdirectory, these will have to be updated to correspond to the new release before compiling.

5. How to Organize A Code File

Each source code file should contain a related set of procedures, such as the code for a program, or a set of procedures to implement hash tables, or a collection of procedures to implement pull-down menus. Before writing any code, you should think carefully about what functions are to be provided and divide them up into files in a logical way. We've found that the most manageable size for files is usually in the range of 500-1000 lines. If files get much larger than this then it's hard to remember everything that the file does; if files get much smaller than this, then there may end up being too many files in a directory, which is also hard to manage.

One case where very small files are permitted is for libraries. It's often convenient to have one source file for each distinct library procedure. This allows users to selectively override individual library procedures.

In order to make it easy for you to follow the conventions described in this section, the directory **/sprite/lib/forms** provides templates for various pieces of source files, such as a file header (**proto.c**) and a procedure header (**prochead**). When writing code, read the templates into your code files and modify them as needed; this will save you time typing and ensure consistency across all of our code. The **mx** editor provides a **Forms** menu for easy access to the templates in **/sprite/lib/forms**.

5.1. The file header page

Files containing source code are divided into pages separated by control-L characters. The first page of the file is a header page containing information global to the file. See Figure 5 for an example. Templates for header pages are in **/sprite/lib/forms/proto.c** for C files and **/sprite/lib/forms/*.md/proto.s** for assembler files. The header page starts off with a block of comments containing the name of the file and a short description of the overall function provided by the file. Following this are a copyright notice, RCS version string, **#include** statements, and declarations for variables and structures that are shared by the procedures in the file.

Source files should never contain **extern** statements for things defined in other files. Instead, create **.h** files to hold the **extern** statements and **#include** the **.h** files. This makes code files easier to read and makes it easier to manage the **extern** statements, since they're centralized in **.h** files instead of spread around dozens of code files. See Section 8 for more information on include files.

5.2. One procedure per page

Each page after the first one in a code file should contain exactly one procedure (unless the compiler or assembler cannot deal with the control-L characters separating the procedures). A procedure consists of a procedure header, a declaration, and a body, as illustrated in Figure 6.

5.3. Procedure headers

A procedure header is a block of comments that gives information to would-be callers of the procedure. Thus it only contains information on how to use the procedure, not information on how the procedure works internally. The header comments have three parts:

[1] The first part of the header gives the name of the procedure and a short description of what the procedure does. This should *not* be a detailed description of how the procedure is implemented, but rather an abstract summary of its overall function and how it relates to the rest of the program.

[2] The **Results** part of the header describes how the procedure affects those things immediately visible to its caller. This includes the return value of the procedure and any modifications made via pointer arguments.

[3] The **Side Effects** part of the header describes changes the procedure makes to its internal state or to global variables, which may not be visible to the caller but which will affect later

```
/*
 * mxFile.c --
 *
 *      This file implements low-level operations to manipulate
 *      files for the Mx editor, such as reading files from disk
 *      and inserting and deleting bytes.
 *
 * Copyright (C) 1986, 1988 Regents of the University of California
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for any purpose and without
 * fee is hereby granted, provided that the above copyright
 * notice appear in all copies.  The University of California
 * makes no representations about the suitability of this
 * software for any purpose.  It is provided "as is" without
 * express or implied warranty.
 */

#ifndef lint
static char rcsid[] = "$Header: mxFile.c,v 1.10 88/08/25 15:04:08 ...";
#endif not lint

#include <errno.h>
#include <list.h>
#include <stdio.h>
 ...

/*
 * The data structure below stores information about a single
 * spy procedure, which is to be called at particular times when
 * a file is modified.
 */

typedef struct Spy {
    int type;                   /* When to call proc:  an OR'ed combination of
                                 * MX_BEFORE_INSERT, MX_AFTER_INSERT,
                                 * MX_BEFORE_DELETE, and MX_AFTER_DELETE. */
    void (*proc)();             /* Procedure to call. */
    ClientData clientData;      /* Parameter to pass to proc. */
    struct File *filePtr;       /* File spy is associated with. */
    struct Spy *nextPtr;        /* Next spy in list for file, or NULL for
                                 * end of list. */
} Spy;

 ...
```

Figure 5. A sample header page for a C file.

calls to this or other procedures. Once again, this section should not describe every internal variable modified by the procedure. It should simply provide the sort of information that users of the procedure need to know in order to use the procedure correctly.

A template for this comment block is in **/sprite/lib/forms/prochead** for C procedures and **/sprite/lib/forms/***machine***.md/asmhead** for assembler procedures for different *machine*s. Follow the exact syntax of Figure 6 (same indentation, etc.).

5.4. Procedure declarations

The procedure declaration should also follow exactly the syntax shown in Figure 6. The first line should give the type of the procedure's result. All procedures should be typed: use **void** if the procedure returns no result. The second line gives the procedure's name and its argument list. If there are many arguments, they may spill onto additional lines. After this come the declarations of the argument types, one argument per line, indented, with a comment after each

```
/*
 *----------------------------------------------------------------------
 *
 * InsertBytes --
 *
 *      Add a range of bytes to a file.
 *
 * Results:
 *      The return value is the position of the byte just after the
 *      last new one added.
 *
 * Side effects:
 *      The file is modified to insert all the characters in the
 *      given string just before the character at the given position
 *      in the file.
 *
 *----------------------------------------------------------------------
 */

static Mx_Position
InsertBytes(filePtr, position, string)
    register File *filePtr;     /* Token for file to be modified. */
    Mx_Position position;       /* Position of character before which string
                                 * is to be inserted.  This character must
                                 * exist in the file. */
    char *string;               /* New characters to be added.  May contain
                                 * newlines.  Terminating NULL character is
                                 * not inserted. */
{
 ...
}
```

Figure 6. The header comments and declaration for a procedure.

argument giving a brief description of the argument.

5.5. Parameter order

Parameters are classified into three categories. ‘‘In’’ parameters only pass information into the procedure (either directly or by pointing to information that the procedure reads). ‘‘Out’’ parameters point to things in the caller’s memory that the procedure modifies. ‘‘In-out’’ parameters do both. Below is a set of rules for deciding on the order of parameters to a procedure:

[1] Parameters should normally appear in the order in, in/out, out, except where overridden by the rules below.

[2] If a set of procedures all operate on a structure of a particular type, such as a hash table, the token for the structure should always be the first argument to all the procedures.

[3] When two parameters are a buffer pointer and the size of the buffer, the size should immediately precede the pointer.

[4] When two parameters are the address of a procedure to call and a **clientData** value to pass to that procedure (see Section 9), the procedure address should immediately precede the **clientData**.

The procedure body follows the declaration. Coding conventions for the body of a procedure are described in Section 7. The curly braces enclosing the body should be on separate lines, just as shown in Figure 6.

6. Naming conventions

I think that choosing names is one of the most important aspects of programming. Good names clarify the function of a program and reduce the need for other documentation. Poor names result in ambiguity, confusion, and error. This section gives some general principles to follow when choosing names, then lists specific rules for name syntax, such as capitalization, and finally describes how we use prefixes to highlight the module structure of the system.

6.1. General considerations

When choosing names, play devil's advocate with yourself to see if there are ways that a name might be misinterpreted or confused. Here are some things to consider:

[1] Is someone who sees the name out of context likely to realize what it stands for, or could they easily confuse it for something else? Our procedure for doing byte-swapping and other reformatting was originally called **Swap_Buffer**: when I first saw that name I assumed it had something to do with I/O buffer management, not reformatting. We subsequently changed the name to **Fmt_Convert**.

[2] Does the name look a lot like another name that is used for a different purpose in the same context? For example, it's probably a mistake to have two variables named **proc** and **process**, both referring to processes in the same piece of code: it will be very difficult for readers to remember which is which. Instead, add a bit more information to the names to distinguish them, for example **masterProc** and **slaveProc**.

[3] Is the name so generic that it doesn't convey any information? For example, a Sprite performance-monitoring tool was originally called **syswidget**. Unfortunately, every window-based tool is referred to as a ''widget'', and almost everything has something to do with some sort of system, so this name didn't say much about what the program did. The name was eventually changed to **spritemon**, which is more descriptive.

[4] Use the same name to refer to the same thing everywhere. For example, you might use the name **filePtr** each time you have a pointer to an open file.

6.2. Basic syntax rules

The second set of naming conventions consists of specific rules governing the syntax of names. It is very important that you follow these rules, since they allow us to determine certain properties about an object just from its name.

[1] Variable names always start with a lower-case letter. Procedure and type names always start with an upper-case letter.

```
int counter;
extern char *FindElement();
typedef int Boolean;
```

[2] In multi-word names, the first letter of each trailing word is capitalized. Do not use underscores as separators between words of a name, except as described in rule 6 below and in Section 6.3.

```
int loopCounter;
```

[3] Any name referring to a pointer should end in **Ptr**. If the name refers to a pointer to a pointer, then it should end in **PtrPtr**, and so on. The only exception to this rule is for strings: since they are used frequently and are always referred to with pointers, we decided

to drop one level of **Ptr** from string pointers.

```
File    *filePtr;
Proc    **procPtrPtr;
char    *name;
char    **namePtr;
```

[4] Variables that point to heads of lists should end in **List** rather than **Ptr**. This distinguishes the list header pointer from list element pointers.

```
List *fileList;
```

[5] Variables that hold the addresses of procedures should have names ending in **Proc**.

```
void (*callBackProc)();
```

[6] **#define**-d constants and macros should have names that are all capitals, except for macros used as procedures. In that case follow the naming rules for procedures. If names in all caps consist of multiple words, then it is permissible, and encouraged, to use underscores to separate the different words.

```
#define NULL            0
#define BUFFER_SIZE     1024
#define Min(a,b)        (((a) < (b)) ? (a) : (b))
```

[7] Names of programs should always be all lower-case, in spite of all the rules listed above.

6.3. Names reflect module structure

The final set of rules for choosing names is designed to clarify the module structure of programs. All programs except the smallest ones are divided up into modules, where a module is one or more files that implement a related set of functions. In our coding style, each module is assigned a short prefix that distinguishes it from other modules. For example, the library routines that implement hash tables use the prefix **hash**, the Sprite file system uses the prefix **fs**, the terminal driver uses the prefix **td**, and so on. Prefixes should be short: not more than four or five characters. The following rules apply to prefixes:

[1] If a variable or procedure is used outside the module in which it is defined, the first letters of its name must consist of the prefix of the defining module followed by underscore. Only the first letter of the prefix is ever capitalized, and it is subject to the capitalization rules defined above. The first letter after the prefix is always capitalized.

```
extern Tcl_Interp *Tcl_CreateInterp();  /* Exported procedure. */
extern int vm_FreeMem;                  /* Exported variable. */
```

[2] If a module contains several files, and if a name is used in several of those files but isn't used outside the module, then the name must have the module prefix but no underscore. The prefix guarantees that the name won't conflict with a similar name from a different module; the missing underscore indicates that the name is used only within this module.

```
extern File *FsFileAlloc();             /* Internal global procedure. */
```

[3] If a name is only used within a single procedure or file, then it need not have the module prefix. To avoid conflicts with similar names in other modules, variables and procedures with global scope should always be declared **static** if they have no module prefix. It's fine to use the module prefix even for procedures and variables that are local to a single file.

```
static int initialized;                 /* Private variable. */
```

[4] For each module there should be an include file that declares all of the variables, types, macros, and procedures exported by that module to the rest of the program or system. This include file should have a name equal to the module's prefix, e.g. **hash.h** for the module that uses the **hash** prefix. See Section 8 below for more details on include files.

7. Low-Level Coding Conventions

This section describes the syntactic rules we use in writing C code. The reason for having these rules is not because they're substantially better than other ways of structuring code (although we picked the best rules we could think of at the start of the project), but because we want all of our code to look the same. Although the conventions are intendend primarily for C code, they should also be used as much as possible in assembler and other languages, subject to limitations of the language processors.

7.1. Indents are 4 spaces

Each level of indentation should be four spaces deeper than the previous one. There are ways to set 4-space indents in all the editors we know of.

7.2. Code comments occupy full lines, except for assembler

Comments that are used to document code (as opposed to declarations) should occupy full lines, rather than being tacked onto the ends of lines containing code. Tacked-on comments are hard to see. Comments should have exactly the structure shown in Figure 7, and should be indented to the same level as the surrounding code. Use proper English in your comments (e.g., capitalize the first word of the comment and structure your comments in sentences).

Assembler code is an exception to this rule: side-by-side comments are easier to read than in C (because the assembler statements are all short), and are used traditionally, so we permit them.

```
    firstPtr = GetLinePtr(filePtr, first.lineIndex);

    /*
     * Move the hint out of the way so it can’t get invalidated
     * by the deletion.
     */

    if (first.lineIndex > 0) {
        filePtr->curLineNumber = first.lineIndex - 1;
        filePtr->curLinePtr = (Line *) List_Prev(&firstPtr->links);
    } else {
        filePtr->curLineNumber = -1;
        filePtr->curLinePtr = NULL;
    }
    ...
```

Figure 7. Comments should have the form shown above, using full lines, with lined-up stars, the open and close comment symbols on separate lines from the text, and blank separator lines around the comment.

7.3. Declaration comments are side-by-side

When you are documenting declarations for procedure arguments and structure members, place the comments on the same lines as the declarations. Figure 6 shows comments for procedure arguments and Figure 8 shows a simple structure declaration. The format for comments is the same in both cases. Place the comments to the right of the declarations, with all the left edges of all the comments lined up. When a comment requires more than one line, indent the additional lines to the same level as the first line, with the close-comment characters on the same line as the end of the text. For structure declarations it is usually useful to have a block of comments preceding the declaration, as shown in Figure 8. This comment block should have the format given in Section 7.2 above.

7.4. Curly braces: { doesn't normally stand alone

Open-curly-braces should not appear on lines by themselves. Instead, they should appear at the end of the preceding line. See Figure 8 for an illustration of how curly braces should be used in a structure declaration, and Figure 7 for an example of their use in an **if** statement. Note that **else** appears on the same line with the preceding close-curly-brace and the following open-curly-brace. Close-curly-braces are indented to the same level as the outer code, i.e. four spaces to the left of the statements they enclose.

The only case when an open curly brace appears on a line by itself is the initial curly brace for the body of a procedure (see Figure 6).

Always use curly braces around compound statements, even if there is only one statement in the block. Thus you shouldn't write code like

```
if (filePtr->numLines == 0) return -1;
```

Instead, you should write

```
if (filePtr->numLines == 0 {
    return -1;
}
```

This practice makes Sprite code less dense, but avoids mistakes when adding additional lines to an existing single-statement block. It also makes it easier to debug, since it guarantees that each

```
/*
 * The data structure below is for a "floater", which marks a range of
 * bytes within a file in a way that is adjusted automatically as
 * the file is modified.
 */

typedef struct Floater {
    File *filePtr;                /* File to which floater belongs. */
    Mx_Position *firstPtr;        /* Position to be updated:  gives beginning
                                   * of range. */
    Mx_Position *lastPtr;         /* Position to be updated:  gives end of
                                   * of range. */
    struct Floater *nextPtr;      /* Next floater in linked list of all those
                                   * for this file (NULL means end of list). */
} Floater;
```

Figure 8. Side-by-side comments are a good idea in the declarations of structure members and procedure arguments.

statement is on a separate line (and hence can be named individually to the debugger).

7.5. Continuation lines have exaggerated indents

With the longer names that our coding style encourages, it is easy for lines to get long. You should use continuation lines in order to keep any single line from exceeding 80 characters in length. Continuation lines should be indented more than four spaces, so that they won't be confused with an immediately-following nested block. There are two possible ways to do this, both of which are illustrated in Figure 9. One way is to indent continuation lines two levels (8 spaces) more than the initial part of the line, as shown in Figure 9(a). In this scheme, all the continuation lines for a single parent should be indented the same amount. The second approach is to indent continuation lines in a way that illustrates the nesting of the statement, as shown in Figure 9(b). This is done by lining up the continuation line with something on the previous line that is at the same nesting level (e.g. line up left edges of procedure parameters or line up parentheses). You can choose whichever of these approaches you prefer, but be consistent in your code.

Try to pick ''clean'' places to break your lines for continuation, so that the continuation doesn't obscure the structure of the statement.

```
if ((linePtr->position.lineIndex > position.lineIndex)
        || ((linePtr->position.lineIndex == position.lineIndex)
        && ((linePtr->position.charIndex + linePtr->length)
        > position.charIndex))) {
    return;
}
line = Mx_GetLine(mxwPtr->fileInfoPtr->file,
        linePtr->position.lineIndex, (int *) NULL);
XDrawImageString(mxwPtr->display, mxwPtr->fileWindow,
        mxwPtr->textGc, x, y + mxwPtr->fontPtr->ascent,
        control, 2);
```

(a)

```
if ((linePtr->position.lineIndex > position.lineIndex) ||
    ((linePtr->position.lineIndex == position.lineIndex) &&
     ((linePtr->position.charIndex + linePtr->length) >
      position.charIndex))) {
    return;
}
line = Mx_GetLine(mxwPtr->fileInfoPtr->file,
                  linePtr->position.lineIndex,
                  (int *) NULL);
XDrawImageString(mxwPtr->display, mxwPtr->fileWindow,
                 mxwPtr->textGc, x, y + mxwPtr->fontPtr->ascent,
                 control, 2);
```

(b)

Figure 9. Two acceptable styles for handling continuation lines. In (a) all continuation lines are indented two levels from the initial line. In (b) continuation lines are indented so that the first character of each continuation line lies underneath the first character of an item at the same nesting level.

7.6. Avoid macros except for simple things

#define statements are a wonderful tool for parameterizing constants, and you should always use them instead of embedding specific numbers in your programs. However, it is generally a bad idea to use macros for complex operations; procedures are almost always better. The only time it's OK to use **#define**'s for complex operations is if the operations are critical to performance and there's no other way to get the performance (do you have numbers to prove this?).

When defining macros, remember always to enclose arguments in parentheses:

```
#define Min(a,b) (((a) < (b)) ? (a) : (b))
```

Otherwise, if the macro is invoked with a complex argument it may result in a parse error or, worse, an unintended result that is difficult to debug.

8. Include Files

Include files (those with names ending in **.h**) are the glue that holds together the modules of a system. This section describes the Sprite conventions for writing them.

8.1. One exported include file per module

Each module of a program should provide one include file to the rest of the program, which declares everything needed by the rest of the program to use the module. The exported include file should have the module's prefix as its name (all lower-case). Thus, if a module uses **Hash_** as the prefix for its exported procedures, its include file should be named **hash.h**. In order to use the facilities of a module, it should be necessary only to include the module's **.h** file. It should never be necessary to place **extern** declarations in the module's clients.

8.2. Private include files have special names

The exported include file for a module should define only the minimum amount of information needed to use the module. It should not expose the internal structure of the module any more than is absolutely necessary. Definitions and procedures that are only used inside a module should be declared in one or more separate include files. These private include files should be named in one of two ways. One possibility is to end their names in **Int.h**. The normal and simplest case is for a module to have one exported include file and one private include file. In this case, the private include file should use the module's prefix followed by **Int.h**: **hashInt.h**, for example. The other way to name private include files is to use any name that does not start with the module prefix. This is preferable when a module has many private include files. For example, in the **fs** there might be private include files named **block.h** and **disk.h**. (In Sprite's current **fs** module, the names are **fsBlockInt.h** and **fsDiskInt.h**, which are also OK but more verbose)

8.3. How to organize an include file

Figure 10 shows part of a sample include file. Each include file should start out with a block of header comments that is similar to the header for a code file: it gives a terse summary of what the include file does, followed by the standard copyright notice and an RCS header line. Structure declarations should appear at the beginning of the include file, and procedure declarations should appear at the end of the include file.

```
/*
 * option.h --
 *      This defines the Option type and the interface to the
 *      Opt_Parse library call that parses command lines.
 *
 * Copyright 1988 Regents of the University of California
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for any purpose and without
 * fee is hereby granted, provided that the above copyright
 * notice appear in all copies.  The University of California
 * makes no representations about the suitability of this
 * software for any purpose.  It is provided "as is" without
 * express or implied warranty.
 *
 * $Header: /sprite/src/lib/include/RCS/option.h,v ...
 */

#ifndef _OPTION
#define _OPTION

/*
 * An array of option descriptions (type Option) is passed into the
 * routine which interprets the command line.
 */

typedef struct Option {
    char        *key;           /* Command-line string that flags option */
    char        *address;       /* Address of variable to modify */
    char        *docMsg;        /* Documentation message */
} Option;

/*
 * Macro to determine size of option array:
 */

#define Opt_Number(optionArray) (sizeof(optionArray)/sizeof((optionArray)[0]))

/*
 * Exported procedures:
 */

exterin int     Opt_Parse();
extern void     Opt_PrintUsage();

#endif /* _OPTION */
```

Figure 10. A sample include file. Structure declarations go first, procedure declarations last. The **#infdef** guarantees that the file will only be processed once, even if it is **#include**-d several times.

8.4. Nested include files

It is legal and desirable for one include file to **#include** another. For example, the exported include file for a module should **#include** any additional files needed to use the module. With this approach, clients need only **#include** the exported include files of modules they use.

However, this introduces a potential problem with include files being **#include**-d multiple times. This might happen, for example, if a source file **#include**’s two different **.h** files, each of which **#include**’s a third **.h** file. We use **#ifdef**s to make sure that an include file is processed only once, even if it is **#include**d many times. The body of each include file should have the following form:

```
#ifndef _NAME
#define _NAME

 ... body of include file ...

#endif /* _NAME */
```

In actual use, **NAME** should be replaced with the include file's name (except for the **.h**) all in caps. See the example in Figure 10.

9. Callbacks and ClientData (Procedure Variables)

We use the term ‘‘callbacks and ClientData’’, or alternatively, ‘‘procedure variables’’ to refer to a particular style of programming that is used frequently in the Sprite kernel code and many user-level applications. If used correctly, callbacks make code much more modular and extensible.

Callbacks are most useful in situations where one piece of code needs to invoke one or more other pieces of code, often in different modules, and the specific code to be invoked is hard to predict in advance or changes over time. For example, the timer module in the Sprite kernel invokes particular procedures at particular times to perform kernel housekeeping chores. One way to implement this would have been to ‘‘hard-code’’ the calls by placing particular calls to particular procedures in the timer code. As the system evolved and new procedures needed to be called, the timer module would have to be modified to reflect this.

A better way to handle this sort of situation is with callbacks. If a module in Sprite needs one of its procedures (**X**, for example) to be invoked at a particular time, it calls the procedure **Timer_ScheduleRoutine**, giving it the address of **X** and the time when **X** is to be invoked. The timer module stores this information in an internal data structure and calls **X** at the desired time. The procedure **Timer_ScheduleRoutine** is called a *registration procedure* because it registers a callback.

The callback approach is more flexible than the hard-coded approach because the module making the callbacks (the timer module in this case) no longer has the names of specific callback procedures embedded in its code. This makes the module less dependent on the rest of the system's structure and allows the module to be used in other situations with totally different callbacks. New callbacks can be added without changing the the timer; all that is needed is to insert additional calls to **Timer_ScheduleRoutine**.

Whenever a callback procedure is registered, e.g. by calling **Timer_ScheduleRoutine**, an extra argument is provided. Its type is **ClientData** and its name is **clientData** (except for a few special situations discussed below). This argument is stored along with the callback procedure's address. When the callback is invoked, **clientData** is passed to it as its first argument. **ClientData** indicates to the callback what it should do, such as a data structure to manipulate or a process to check. **ClientData** allows the same callback procedure to be used in many different situations by passing it different **clientData** arguments.

The **ClientData** type is defined in **sprite.h**. It is a dummy type suitable for a pointer or single-word value. Normally, **clientData** arguments are pointers to structures understood by the callback procedures; the value is cast to type **ClientData** when calling the registration procedure and cast back to its original type in the callback. Modules that invoke callback procedures (such as the timer module) should never do anything with **clientData** arguments except store them and pass them to callback procedures. They should never assume any particular meaning for the argument or use it to access data structures.

The normal way to use callbacks and clientData is to pass the procedure address and **clientData** to the registration procedure as separate arguments with the **clientData** immediately following the procedure address. When the callback is invoked, the **clientData** should normally be its first argument.

If a procedure takes more than one callback procedure as arguments, with a separate **ClientData** argument for each callback, then the **ClientData** arguments should be named after their callbacks. For example, if the callback procedure arguments are named **rawProc** and **cookedProc**, then the corresponding **ClientData** arguments should be **rawData** and **cookedData**. Each **ClientData** argument should immediately follow its corresponding procedure in the argument list.

10. Documenting Code

The most important thing to remember in documenting your code is ‘‘quality, not quantity.’’ A few carefully-chosen words in the right place may be more helpful than a page of drivel. This section lists a few things to consider in order to improve the quality of your in-line documentation.

10.1. Document things with wide impact

The most important things to document are those that affect many different pieces of a program. Thus it is essential that every procedure interface, every structure declaration, and every global variable be documented clearly. If you haven’t documented one of these things it will be necessary to look at all the uses of the thing in order to figure out how it’s supposed to work; this will be tedious and error-prone.

On the other hand, things with only local impact may not need much documentation. For example, in short procedures I don’t usually have comments explaining the local variables. If the overall function of the procedure has been explained, and if there isn’t much code in the procedure, and if the variables have meaningful names, then it will be easy to figure out how they are used. On the other hand, for long procedures with many variables I usually document the key variables. Similarly, when I write short procedures I don’t usually have any comments in the code at all; the procedure header should provide enough information to figure out what is going on. For long procedures I place a comment block before each major piece of the procedure to clarify the overall flow through the procedure.

10.2. Don’t just repeat what’s in the code

The biggest mistake made in documentation is simply to repeat what’s already obvious from the code, such as this trivial (but exasperatingly common) example:

```
/*
 * Increment i.
 */

i += 1;
```

Documentation should provide higher-level information about the overall function of the code, what a complex collection of statements really means. For example, the comment

```
/*
 * Probe into the hash table to see if the symbol exists.
 */
```

is likely to be much more helpful than

```
/*
 * Mask off all but the lower 8 bits of x, then index into table t,
 * then traverse the list looking for a character string
 * identical to s.
 */
```

Everything in this second comment is probably obvious from the code that follows it.

Another thing to consider in your comments is word choice. Try to to use different words in the comments than the words that appear in variable or procedure names. For example, the comment

```
 *
 * VmMapPage --
 *
 *      Map a page.
 *
```

appearing in the header for the procedure **VmMapPage** doesn't provide any new information: everything in the comment is already obvious from the procedure's name. A much more useful comment is

```
 *
 * VmMapPage --
 *
 *      Make the given physical page addressable in the kernel's virtual
 *      address space.  This routine is used when the kernel needs to
 *      access a user's page.
 *
```

This comment also tells *why* you might want to use the procedure, in addition to *what* it does; this makes the comment much more useful.

10.3. Document each thing in exactly one place

Systems evolve over time. If something is documented in several places, it will be hard to keep the documentation up-to-date as the system changes. Instead, try to document each major design decision in exactly one place, preferably as near as possible to the code that implements the thing. For example, I put the documentation for each structure right next to the declaration for the structure, including the general rules for how the structure is used. I don't explain the fields of the structure again in the code that accesses the structure; people can always refer back to the structure declaration for this. The principal documentation for each procedure goes in the procedure header. There's no need to repeat this information again in the body of the procedure (but you might have additional comments in the procedure body to fill in details not described in the procedure header). If a library procedure is documented thoroughly in a manual entry, then I make the comment header for the procedure very terse, simply referring to the manual entry.

The other side of this coin is that every major decision needs to be documented *at least* once. If a design decision is used in many places, it may be hard to pick a central place to document it. Try to find a data structure or key procedure where you can place the main body of comments; then reference this body in the other places where the decision is used. If all else fails, add a block of comments to the header page of one of the files implementing the decision.

11. Manual Entries

This section describes how to write entries for the Sprite reference manual. Most manual entries either describe commands or library procedures. There are also other kinds of manual entries, but I won't talk about them here; they tend to be similar in overall format to entries for commands or libraries, but with a few sections missing.

Manual entries in Sprite are formatted with the **ditroff** program using the **−man** macros. Sprite manual entries have a slightly different format than traditional UNIX manual entries. For detailed descriptions of the macros provided by the **−man** package, see the **man** entry in the **files** section of the manual (type **man −s files man** to the shell). The Sprite **−man** macros will still work with UNIX manual entries, and we've retained the UNIX manual entries for programs that were ported unchanged from other UNIX systems. However, any new things written for Sprite should use the format described here.

Each manual entry describes a single command or one or more library procedures. In the case of procedures, use your judgment in deciding how many procedures to describe in a single manual entry. For a small collection of related procedures such as those in the **str***xxx* family it's better to put them all in a single manual entry: there will be less repeated verbiage and all the information will be available in one place. If a library contains a large number of procedures, then it may make more sense to split the library up into a number of smaller manual entries, possibly with a summary manual entry to explain the overall function of the library. The **Tcl** library is an example of this approach.

Figures 11 and 13 show examples of manual entries for a command and a set of library procedures, respectively. There are three essential parts to each entry: a summary box, a textual description, and a keyword list. These are described in the subsections below.

11.1. The summary box

The summary box appears at the top of the manual entry and gives a complete but very terse description of how to use the command or procedure(s), including arguments, results, options, etc. If a user is already familiar with a command or procedure, the summary box should provide everything needed to use the thing.

For commands, the summary box contains three sections: **NAME**, **SYNOPSIS**, and **OPTIONS**. The **NAME** section gives the name of the command, followed by a dash, followed by a one-line summary of what the command does. The format of this line is important, since it is used to generate the manual index. The **SYNOPSIS** section contains a template of what you type to the shell to invoke the command, and the **OPTIONS** section describes the command-line switches for this command. Figure 12 shows the input text that produced the summary box in Figure 11.

The summary box for a library manual entry contains **NAME**, **SYNOPSIS**, and **ARGUMENTS** sections. The **NAME** section for library procedures is similar to that for a command, except that there may be many procedures listed before the dash; these must be separated by commas in order for the manual index to be generated correctly. The **SYNOPSIS** section gives

Print login.man, reduce it to 77%, and paste it inside this box.

Figure 11. An example of a manual page for a command.

```
.BS
.SH NAME
login \- Allow a user to login
.SH SYNOPSIS
\fBlogin\fR [\fIoptions\fR] \fIdevice\fR
.SH OPTIONS
.IP "\fB\-help\fR" 15
Print a summary of the command-line options and exit without
performing any logins.
.IP "\fB\-l\fR"
Don’t record information about the user in a file of logins.
The default is to record the login and logout in a file
of logins used by programs like \fBfinger\fR.
.IP "\fB\-P \fIportNum\fR"
Use \fIportNum\fR as the port number associated with this login in
the file of logins (ignored if the \fB\-l\fR option is given).
.IP "\fB\-r\f"
Repeat:  when a login shell exits, prompt for another account and
password, and start another login shell.  The default is for \fBlogin\fR
to exit as soon as its login shell exits.
.IP "\fB\-u \fIuser\fR"
Instead of prompting for a login user, use \fIuser\fR as the account
for the login.
.BE
```

Figure 12. Part of the source file for the **login** manual entry; the text in the figure produced the summary box and its contents in Figure 11.

templates for calls to the procedures, and the **ARGUMENTS** section gives a brief description of each argument to each procedure. The **.AP** and (if needed) **.AS** macros should be used in formatting the **ARGUMENTS** section. Figure 14 shows the input text that produced the summary box in Figure 13.

11.2. The textual description

The main body of the manual entry is a textual description of how to use the command or procedure(s). For simple entries it will consist of a single section, **DESCRIPTION**. For more complex entries it will contain several sections. The textual description should provide more background and detail than is in the summary box, but it should still be fairly terse. Manual entries are intended as reference documents, not tutorials or internal documents. This means that an entry should contain everything a would-be-user needs to know in order to use the program or procedures, but it should be organized for easy reference lookup rather than for gentle introduction. This also means that entries should not describe the internal structure of the program or procedure.

11.4. Formatting details

Here are some low-level formatting conventions to follow when writing manual entries:

Boldface Use boldface for anything that a user would type verbatim, such as a procedure name, command name, file name, or specific argument or option.

Italics Use italics for anything that serves as a meta-symbol, i.e. a term that refers to a general class of things that a user might type, as opposed to a specific thing the user would type. For example, argument and option names are italicized (as opposed to specific values, which are in boldface).

Long Dashes Use the long form for dashes (‘‘\–’’), particularly in command-line switches. This makes them much more visible.

Print mem.man, reduce it to 77%, and paste it inside this box.

Figure 13. An example of a manual page for a set of library procedures.

11.3. Keywords

The last section in each manual entry should be titled **KEYWORDS**. It should contain a list of words or short phrases, separated by commas, that give attributes of the thing described in the manual entry. This information is compiled into the manual index and allows users to locate an entry even if its name isn't known, merely by naming one of its keywords in a **man –k** command.

```
.BS
.SH NAME
memchr, memcmp, memcpy, memset \- Operations on byte arrays
.SH SYNOPSIS
.nf
\fB#include <string.h>

\fBchar *
\fBmemchr(\fIs\fB, \fIc\fB, \fIn\fB)

\fBint
\fBmemcmp(\fIs\fB, \fIs2\fB, \fIn\fB)

\fBchar *
\fBmemcpy(\fIdest\fB, \fIsource\fB, \fIn\fB)

\fBchar *
\fBmemset(\fIdest\fB, \fIc\fB, \fIn\fB)
.SH ARGUMENTS
.AS char source
.AP char *s in
Pointer to array of characters.
.AP char *source in
Pointer to array of characters to copy.
.AP char *dest out
Pointer to array of characters to modify.
.AP int c in
Value to search for (\fBmemchr\fR), or value to set to (\fBmemset\fR).
.AP int n in
Count of number of characters to test, copy, or set.
.BE
```

Figure 14. Part of the source file for the **mem** manual entry; the text in the figure produced the summary box and its contents in Figure 13.

11.5. Templates

The file **/sprite/lib/forms/cmd.man** contains a template of a manual entry for a command, and **/sprite/lib/forms/lib.man** contains a template for a manual entry for a library procedure.

12. The Sprite Kernel

The Sprite kernel sources are stored in the directory subtree rooted at **/sprite/src/kernel**. The kernel is organized something like a ‘‘big command’’, in that it consists of several subdirectories, each holding the sources for one of the kernel's modules. However, there are several differences between the kernel's organization and that of a big command. The main reason for the differences is that we keep separate ‘‘installed’’ and ‘‘uninstalled’’ versions of each kernel module.

12.1. Uninstalled modules

The contents of each kernel subdirectory, such as **/sprite/src/kernel/rpc**, represent the uninstalled version of the module. Each of these directories is organized like one of the modules of a big command, except that the linked version of the module is stored in ***.md/***module***.o (where** *module* is the name of the module), rather than in ***.md/linked.o**. Changes to a module are made in the uninstalled directories and tested from there.

12.2. Installed modules

When a set of changes to a module have been tested thoroughly, the module should then be installed using **make install**, which makes the changes an official part of the kernel. During installation, files are copied from the module's uninstalled subtree to the following directories:

/sprite/src/kernel/*.md
: The file *module***.o**, which contains all the files in the module linked together, is copied here. When linking together the modules to form a complete kernel, the files in this directory are normally used.

/sprite/src/kernel/Include
: All of the module's public include files are copied here. See Section 6.3 for the naming conventions that determine whether a include file is public or private (for unfortunate historical reasons, some private include files still have the module prefix, making them appear to be public). When one module of the kernel includes **.h** files defined by other modules, it always includes them from this directory and its **.md** subdirectories. In other words, modules only use the installed versions of other modules' include files.

/sprite/src/kernel/Installed/*module*
: All of the module's source files are copied to this directory and its **.md** subdirectories. When debugging a version of the kernel, source files are normally read from this area.

/sprite/src/kernel/Lint/*.md
: The lint library file for the module is stored in these directories (a different version for each machine type). The lint libraries are used to lint modules against each other.

To generate a kernel using all of the installed modules, the normal approach is to compile and install each of the modules, then run **make** in the directory **/sprite/src/kernel/sprite**. This directory contains a Makefile that will link a complete kernel from the installed versions of all of the modules.

12.3. Testing new kernels

When testing a new kernel with changes to some of the modules, you should test the changes before you install them. To do this, you should link new kernels in the directory **/sprite/src/kernel/***user*, where *user* is your login name. Use a Makefile that links with the installed versions of all the modules that you haven't modified, plus the uninstalled versions of the modules that you have modified. This approach allows several people to work simultaneously on different modules in the kernel without getting in each other's way. When you've thoroughly tested your changes you should install the modules you changed, which will make the changes available to the rest of the kernel.

12.4. Installed kernels

The directory **/sprite/kernels** holds the kernel binaries that are ready to be booted on machines. This directory is searched automatically when diskless workstations boot. The names of kernels indicate both the machine type they are for and also other information. For example **/sprite/kernels/sun3.sprite** is the standard stable kernel for Sun-3's. Sun-3 test kernels are kept in files named **/sprite/kernels/sun3.***person* where *person* is the login name of the person testing the kernel.

For machines that will boot off disk rather than over the network, the default kernel for that machine must be in the file **/vmsprite**, and there must also be a hard link to this file from **/vmunix**. One of the files in **/sprite/kernels** is usually a symbolic link to this file too.

We also keep copies of kernels on other Suns under SunOS for emergency purposes. The directories used for these kernels are totally haphazard right now. The default Sun-3 kernel is in **/sprite3/sun3**, and Sun-3 test kernels are in files like **/sprite3/tmp/sun3.***person*. The default Sun-2 kernel is in **/bnf2/sprite/sun2**, and test kernels are in files like **/bnf2/sprite/sun2.***person*.

13. Filing and Reviewing Bug Reports

When you encounter bugs or other unexpected behavior in Sprite, you should report them by sending mail to **sprite@sprite.berkeley.edu**. Mail sent to this address will be forwarded to each of the Sprite developers. In addition, a copy of the message will be recorded in a permanent log, so that we won't forget about it until the problem has been fixed. If you wish for a message to be entered into the log without sending it also to the Sprite developers, send the message to **sprite-log** instead of **sprite**.

The Sprite bug log is indexed. When a message is received by the bug log, one or more keywords are associated with it, and the message is indexed by each of those keywords. The keywords are chosen in one of two ways. If the message contains a **Keywords** line in the mail header, or if the first line of the message body starts with the string **Keywords:**, then the words on that line are the keywords for the message. If there is no **Keywords** line in the message header or body, then the words of the **Subject** header line are used as keywords (however, noise words like ''a'' and ''the'' are not used as keywords).

Messages sent to the bug log are retained on-line and can be retrieved using the indexes described above. For details on how to do this, type **man maillog**. This manual entry also gives complete information on message indexing, including additional indexes by sender name and date sent. The bug log is currently stored under UNIX, but it will soon move to Sprite.

Index

Table of Contents