Software Standards

Paul Heckbert
ph@miro.berkeley.edu

University of California, Berkeley

27 March 1991†

ABSTRACT

This document proposes standards for organization, naming, style, maintenance, and documentation of software written in the C programming language on the UNIX operating system. The standards are intended to be useful in both industry and academia.

1. WHY BOTHER?

Software standards aren't necessary when you're working alone, but when a software development team grows and collaborative efforts and shared source become common, standards are helpful in avoiding chaos. Without standards you have as many organizational schemes, naming conventions, and coding styles as you have programmers, and heavily modified programs can turn into unreadable, unmaintainable mush in very short order.

Software standards are also important because the source code that's sold or distributed should obviously reflect well on the company, lab, or author that developed it. Although much R&D software is originally written for experimental or internal use, software has a tendency to migrate into products over the years (especially if it is good!). Therefore it would help to make it clean and maintainable from the outset. Also, if the standards required for the development and maintenance of software products are simple, painless modifications to existing habits then why not follow these standards throughout a company or research group where possible? Conversely, experimental software should not be sold hastily without checking its reliability, maintainability, and documentation.

The goal of standardization is to facilitate software development by reducing organizational, naming, modularity, and style questions to a routine procedure. We do not propose that all existing software be modified to conform to these standards but only that future software follow them. These standards will be most successful if people follow them voluntarily.

2. ORGANIZATION

Experimental and private work belongs in user's directories and public work belongs in public directories. If a personal program becomes generally useful then it should be moved to a public place.

2.1. Standard Directory Structure

Public software directories should follow the UNIX conventions:

bin:	executables
lib:	subroutine libraries (.a) and miscellaneous data files

† Revised version of Pixar Tech Memo #156, 15 April 1986. Reprinted with permission.

etc:	system maintenance only
include:	public header files (.h)
src:	source
src/bin:	source to commands which go into bin
src/lib:	source to subroutines which go into lib
man:	manual entries
doc:	miscellaneous documentation

The bin directory contains nothing but executables. The lib directory contains primarily subroutine libraries but also can contain subdirectories for miscellaneous collections of data files.

When both a bin and lib directory exist the src directory is split into two subdirectories called src/bin and src/lib. The src/bin directory contains the source to commands. Single file sources are placed directly in the src/bin directory, but when multiple source files are required they are placed in a subdirectory having the same name as the program. These subdirectories often contain several related programs which share a common function or share header files. When the program is extremely large another hierarchy is often started at this point, containing subdirectories src, include, and so forth. The src/lib directory contains one subdirectory for each of the libraries in lib. Often, when there is no lib directory, the source files which would have gone in src/bin are simply put in src to eliminate the extra level.

The man directory is organized with the usual convention, its most important subdirectories being man1 for commands, man3 for subroutines, and man5 for file formats. The doc directory contains miscellaneous documentation which does not fit into the ‘‘man’’ format, such as user’s manuals and tutorials.

All source files (including header files and Makefiles) should be kept under RCS (Revision Control System) to maintain old versions. RCS can be useful for maintaining old versions of documentation and other ascii file types as well. In its normal state source should be checked out in read-only mode using ‘‘co’’. The use of RCS is discussed further in the section on maintenance.

Public directories should be clean and complete. The bin, lib, and include directories should be completely reconstructible from the contents of src. This modularity of directories, together with good Makefiles, is frequently used when porting software to other machines.

If there are test programs and model files for verifying a program, these should be stored in a subdirectory called ‘‘test’’ to avoid cluttering up the source directory. Huge picture or data files used only for occasional testing do not belong in the public hierarchy.

2.2. Master Source Directories

Suppose we work for a hypothetical company called ‘‘Blazing Pixels’’ that sells two software systems called ‘‘PixelTeco’’ and ‘‘PhotoRealistic Blur’’, and it has an R&D group, and a few systems people. A reasonable organizational scheme for the directory hierarchy at Blazing Pixels would be:

DIRECTORY	CONTENTS
/u/teco	PixelTeco software
/u/blur	PhotoRealistic Blur software
/u/sys	systems software source
/u/rnd	public R&D software
/u/user	user directories

This structure reflects the organization of the company. The purpose is to facilitate releases by modularizing into independent hierarchies where possible, but is not meant to discourage collaboration between groups. The names of these software hierarchies will be set by each group, however they should all go in

/u for consistency. Since projects, products, and people come and go, it is helpful to create an on-line table listing the master source machine and directory for each project.

2.3. Minimize Redundancy

Avoid redundant source. One of a programmer's worst nightmares is integrating two versions of a program which evolved from a common ancestor. An anecdote: Frank Crow wrote some subroutines for antialiased line drawing at NYIT in 1979. They were very popular but no one was maintaining them and no one bothered to put them in a public library, so many people made bootleg copies of the source and modified it. One day in 1981 I scoured the disk and found 27 copies, some of which had bug fixes, and others not. Most of the versions were unique. It was years before we got that one cleaned up. We could have avoided this by keeping a single source file and adding in all modifications as run-time or compile-time (#ifdef) options.

Links can be a great convenience, for instance to alias a directory, but they often become broken, for instance when files are copied. It can also be confusing having two names for a single object. Therefore we recommend that the use of links be avoided. If they are needed, however, symbolic links are preferred over standard links.

3. NAMING

Good naming is a difficult balance between uniqueness, mnemonics, and brevity. Uniqueness has top priority.

3.1. Directories

Directories should have lower case names (‘‘RCS’’ being one of the few exceptions).

3.2. Commands

Commands should have lower case names which should not conflict with existing programs in /bin, /usr/bin, etc. unless they are new versions of those programs. Names should be mnemonic and preferably pronounceable but not excessively cute. Some examples of bad names: in the Cedar operating system the equivalent of diff is called ‘‘waterlily’’; at NYIT the program for recording a cycle of animation was called ‘‘tdsph’’. In general, short, abbreviated names should be reserved for those commands which are most commonly used, e.g. ‘‘cd’’ or ‘‘ls’’ and long, mnemonic names used for less common or less benign commands, e.g. ‘‘reformatdisk’’.

3.3. Subroutines

One of the weakest features of C is the lack of control over name scoping. All external (as opposed to static) variables and subroutines are susceptible to name clashes with other programs, as anyone who has ever created an external variable called ‘‘y1’’ in the presence of math.h can attest. Therefore it is essential that **prefixes be added to all external symbols in public subroutine packages and in programs which use packages.**

It is useful to make a distinction between subroutine packages and subroutine libraries. A package is a collection of subroutines which manipulate a certain data structure, say, and form a complete module. A library is an archive (.a) file which might contain several related packages.

When a new package is created some thought should be devoted to the choice of a unique, mnemonic name two to five characters in length. This prefix, in various forms of capitalization, will be used extensively in all source and header files.

It is easiest to explain the naming standards with an example. Say we're creating a package of subroutines called ‘‘quat’’ for manipulating quaternions and a similar package called ‘‘v3f’’ for working with vectors of three floats, both of which would go into ‘‘libmath.a’’.

package name	quat	v3f
external psymbol	quat_rot	v3f_cross
external type	Quat, Quat_matrix	V3f_point
internal psymbol	foo	bar
internal type	This	That
defined constants, macros, and enums	QUAT_LEN	V3F_XAXIS
source files	libmath/quat/quat.c	libmath/v3f/arith.c, ...
header file	quat.h	v3f.h
man file for library	libmath.3	libmath.3
man file for package	quat.3, quat.5	v3f.3
man file for subroutine	quat_rot.3	v3f_cross.3

By psymbol we mean a physical symbol: a subroutine or variable occupying physical space; and by type we mean a structure name or typedef. We recognize two classes of symbols: external, and internal (static).

In summary, all names are lower case, except for types, which are capitalized, and defined constants, macros, and enums, which are upper case. An exception to the macro naming rule is macros which are designed to look like subroutines, e.g. ‘‘#define v3f_copy(a,b) bcopy(a,b,3*sizeof(float))’’. Internal (static and local) subroutines and variables are much less prone to name clashes, so no prefix is necessary, and the programmer has complete freedom. **Global variables should be static if possible.**

3.4. Naming Versions

If it is necessary to keep old versions of a file then RCS is the machinism of choice for ascii files, but for binary files we recommend the convention *name.version.suffix* as illustrated below:

foo.pic	current version
foo.1.pic	1st version
foo.2.pic	2nd version
...	
foo.n.pic	nth version

3.5. File Formats

Some conventions for file suffixes in addition to the usual .c, .h, .s, .y, .l, .o, .a, .f:

Graphics:	
.mod	ascii 3-D model file
.pic	raster image
Miscellaneous:	
.tar	output of tar program
.mail	mailbox on a specific topic
.awk	awk script file
.nr	nroff/troff/eqn/tbl/pic file
.ps	postscript file
.sh	shell script (unless it's in a bin)
.csh	C shell script (unless it's in a bin)

The use of file suffixes is voluntary but strongly recommended; programs should not enforce their use.

When a new file format is created:

(1) A unique suffix should be chosen.

(2) A unique magic number should be chosen and ‘‘registered’’ by placing it in a common header file. This 4 byte number should head all files of this type.

(3) A manual page describing the format should be written for section 5 of the manual.

(4) A subroutine library for reading and writing such files should be created. When machine-independence of binary files is important, shorts and longs should be written in ‘‘network order’’ (high byte first) using the *byteorder(3N)* subroutines. Portability of floating point formats is easy between IEEE floating point machines, but more difficult when other machines (such as DECs and Crays) are involved.

4. USAGE

Command and subroutine usage should be standardized in order to eliminate redundant code and reduce user confusion. If done on an ad hoc basis by each program, user interface programming and argument parsing can be some of the most unpleasant tasks in preparing a program for public use. We do not attempt to standardize user interfaces here.

In non-interactive programs, much of the tedium of argument parsing can be eliminated by using a general argument parser such as the *arg_parse* subroutine†. Arg_parse supports multiple-character options with a variable number of arguments, arbitrary option order, automatic argument conversion and type checking, interactive commands, and built-in usage messages. Arg_parse also allows parsing instructions to be nested, so that the parsing code for popular packages (for example frame buffer, window, and camera parameter selection) can be modularized and placed in public libraries. All programs (including shell scripts) should observe the convention that a usage message is printed when the lone argument ‘‘–’’ is given. This usage message should list the command line options and their default values.

In order to standardize subroutine calling sequences it is helpful to identify commonly used or potentially useful structures and put their definitions in a public header file. For graphics, examples of such structures are points, colors, 4x4 matrices, polygons, and windows.

5. STYLE

A basic rule of programmer etiquette states that you should **follow another person’s style when modifying their programs.** Doing this often means using their naming style, indentation style, spacing style, and commenting style. Most experienced programmers follow this ‘‘ego-less programming’’ rule.

Comments should be included in a program to clarify its meaning. At the bare minimum, **there should be a comment at the top of each file summarizing its contents, e.g.**

```
/* gt: get a picture file into a frame buffer */
```

The author’s name and a modification log is kept in the RCS files (and extracted with the ‘‘rlog’’ command), so there is no need to keep it in the actual source as well.

Unfortunately, self-documenting code is rather rare in a low level language such as C. Understanding each statement individually is no problem, but grasping the purpose of an entire subroutine, even if it is given a descriptive name, can be greatly assisted by a brief **comment preceding each subroutine summarizing its purpose and side-effects.** Comments explaining obscure statements are also extremely helpful.

† Available from the author.

Some people use elaborate header comments which summarize the purpose, algorithm, inputs, outputs, and side-effects of a program. For unusually intricate software this is probably a good idea, but beware that overly verbose comments can bloat and obfuscate rather than clarify a simple program.

6. UTILITY

Utility is a fairly nebulous thing; we want our software to be as useful as possible but there are no formulas for achieving this. It takes a good deal of artistry to design a subroutine package which is at once elegant, general, efficient, and modular. We can only give a few pointers here.

6.1. Modularity

Subroutine packages should be well-defined modules. Clear definition of the external access to the package, the *interface*, is one of the most important phases in software design. **Ideally a document listing the arguments and effects of each external subroutine should be roughed out before implementation begins.** In order to maximize consistency this design work should be done by one or two people (not by committee).

In cases of complex databases it is often helpful to shield the user from internal details which he need not know (i.e. *information hiding*). That way the internal details can be changed without notifying him. Also, if every external access to the database leaves it in a self-consistent state then the user is less likely to accidentally trash it. These ideas are the foundation of object-oriented programming. A simple example is the macros in stdio.h which shield us from detailed knowledge of the _iobuf structure.

6.2. Error Handling

On error a subroutine should either return an error code if this is a fairly regular occurrence or print a message on stderr (never on stdout) if it is a rare occurrence. **It is poor form for a low-level subroutine to print a message, or worse yet to exit, when a common, recoverable error occurs.**

Error codes should be clearly documented in the man entry and defined for users in a header file. Needless to say, a program should check returned values for errors which could conceivably occur. This can be a nuisance when the error is extremely improbable. In such cases the assert macro is handy, for example:

```
#include <assert.h>
...
assert(p = malloc(sizeof(thing)));
```

If its argument is zero, the assert macro prints the file name and line number where the error occurred and exits.

Packages with numerous error codes should include an error diagnosis routine similar in function to *perror*, the diagnosis routine for system calls.

6.3. Header Files

The header file corresponding to a package should define everything one needs in order to use that package: the structures and typedefs used in its subroutine arguments, the types of all exported subroutines (even if int), error codes and other defines. Header files should be heavily commented. For example, the header file for our ‘‘v3f’’ package might look like:

```
#ifndef V3F_HDR
#define V3F_HDR

#include <math.h>

#define v3f_copy(a, b) bcopy(a, b, sizeof(V3f))
#define v3f_len(a) sqrt(v3f_dot(a, a))
#define V3F_FORMAT "(%g,%g,%g)"

typedef float V3f[3];      /* vector of three floats */

extern V3f *v3f_stk;       /* stack of vectors */

extern void v3f_add();
extern void v3f_cross();
extern float v3f_dot();

#endif
```

This include file is self-contained in the sense that it *#include*s all other header files which must precede it. The first two lines and last line are a trick which ensures that the definitions appear only once even if the file is included several times.

In ANSI C, one can declare the types of a subroutine's arguments using *prototypes*. Use of prototypes can help catch errors early and reduce the need for *lint*, but unfortunately, prototypes are not understood by older C compilers, so they are not very portable at the moment.

6.4. Portability

Write software to be device independent where possible. Sometimes this principle must be abandoned to achieve maximum performance, of course.

Ideally, software should be portable between a variety of machines. These might be based on microprocessors from Motorola, MIPS, DEC, and Intel, for example. These machines have different byte orders within their words, and different floating point formats, so care must be taken when moving binary data files from machine to machine. There are also different versions of UNIX, unfortunately, such as Berkeley UNIX, System V, and hybrids. When the source must be modified for different machine types this should be done in a single source file rather than multiple versions of the source, using #ifdef to switch between code sections at compile time, e.g.:

```
getshort()
{
        unsigned char a,b;

        a = getchar();
        b = getchar();
#       ifdef vax       /* little-endian */
                return b<<8 | a;
#       else            /* big-endian */
                return a<<8 | b;
#       endif
}
```

The symbols ‘‘sgi’’, ‘‘sun’’, ‘‘sparc’’, ‘‘mips’’, ‘‘unix’’, ‘‘hpux’’, ‘‘SVR3’’, ‘‘SYSTYPE_BSD43’’, and ‘‘SYSTYPE_SYSV’’ are predefined appropriately by some compilers. The command ‘‘man cc’’ will tell you which symbols are predefined. Device-dependent sections of the software should be isolated as much as possible. Mark device dependent sections of a program with a bold comment.

Unless absolutely necessary, do not let two versions of the source evolve independently on different machines. Whenever changes are made while porting a program to a new machine take the source back to the master location and see that it still works there.

Commonly used ‘‘magic numbers’’ and expressions should be placed in defines in order to reduce redundancy, improve readability, and simplify modifications. Below are some examples which illustrate the wide range of uses for defined constants and macros:

```
#define PI 3.14159265358979323846264
#define XMAX 1024               /* max picture width */
#define TOLERANCE 1e-8          /* tolerance to roundoff error */
#define MODEL_BADPRIM 23        /* error code for bad primitive */
#define SURF_DIR "/u/gfx/surfaces"
#define IK_YMASK 0xffff0000

#define MAX(a, b)  ((a)>(b) ? (a) : (b))
#define STREQ(a, b)  ((a)==(b) || strcmp((a), (b))==0)
#define SWAP(a, b, t)  {t = a; a = b; b = t;} /* swap a and b with t as temp */

#define PVBITS 8          /* number of bits in a pixel value */
#define PVMAX ((1<<PVBITS)-1)   /* maximum pixel value */
```

The last two illustrate the use of cascading macro definitions. The use of PVMAX as shown does not result in less efficient code since most C compilers perform evaluation of constant expressions at compile time.

Arguments to macros which could possibly be expressions should *always* be parenthesized in the macro definition.

Configuration-dependent values should be set at run time, not compile time. A convenient mechanism for this is the use of environment variables, which can be accessed with the *getenv* subroutine. An alternative is the use of configuration files. One must be careful about overuse of environment variables, however, since they act as hidden arguments to a program and can hinder debugging.

We are not particularly concerned with 16 bit machines, so it is common practice to assume that int=long. Use sizeof(), however, rather than assuming short=2 bytes, int=4 bytes, etc.

6.5. Temporary Files and Memory

Any temporary files created should go in /tmp and have a name chosen to avoid clashes. The *mktemp* subroutine is helpful for this task. On some systems, files anywhere in the file system whose name begins with ’#’ will be deleted automatically after three days. Check your local system guru.

When dealing with data structures of variable size it is better to use dynamic memory allocation (malloc) rather than declare a fixed length array, since the latter is always too big until it’s too small. If possible, free any memory you’ve allocated when you’re done with it.

7. DOCUMENTATION

There are several levels of documentation:

- comments in the source
- README files
- usage messages
- manual entries
- user's manuals
- technical reports

We've already discussed comments. README files are commonly used to summarize the contents of a directory for those unfamiliar with it. They are especially valuable when software is being passed from person to person. Usage messages are summaries of a program's purpose and options, typically printed when the single command line argument ''–'' is given. The arg_parse subroutine enforces this convention.

7.1. Manual Entries

If a program is worth writing then it's worth documenting. **Documentation in the format of the UNIX manual should be written for every public command and subroutine** so that help is available both on-line and in nicely printed hardcopy form. The man program locates documentation by checking all of the directory hierarchies listed in the MANPATH environment variable. Hardcopy can be created using the ''man -t'' commands. Documentation should also be written summarizing any major data structures, languages, and file formats which are defined. These go in the man5 directory.

In writing manual pages it is important to follow several simple rules in formatting style. These rules are necessary if the programs used to automatically generate table of contents, keyword indices, etc. are to work properly.

Manual pages should follow the structure shown below:

> .TH *entry section date*
> .SH NAME
> entry-name \– entry synopsis and/or summary
> .SH SYNOPSIS
> library entry points and definitions or program usage information
> .SH DESCRIPTION
> description of a library's contents or a program function;
> if interactive, all commands should be described
> .SH FILES
> any unusual files utilized by the program or library (i.e. not /etc/passwd)
> .SH SEE ALSO
> other manual pages and/or documents
> .SH BUGS
> a synopsis of known bugs

The .TH definition specifies the title and footer lines for the manual entry, e.g. '.TH PRINTF 3S "January 14, 1984"'. The *entry* definition should be completely upper-case, no matter whether the program's name or subroutine entry points are mixed case. If the *section* specification contains a character, that character should be in upper-case. The date should be specified with a string of the form shown; remember to enclose the date in quotes as it will contain spaces.

For the NAME section, the entries should exactly correspond to the program name or library entry points.

Each entry in this section must, however, separate the entry point from its description with the string ‘‘\–’’. If multiple entry points are to receive the same description, they should be specified on one line, separated by commas; e.g.

v3f_dot, v3f_cross \– dot and cross product of 3-D vectors

Manual pages for libraries should specify the include file required to use the package, if any. Manual pages should be terse and factual; tutorial information and explanatory material belongs in supplementary documents. As a general rule of thumb, if a manual page is longer than two pages, most of the information in the manual page probably belongs in an auxiliary document.

7.2. User’s Manuals

Documentation for more complex programs often requires tables, diagrams or pictures, which unfortunately cannot be integrated into manual entries. Such documents are usually written for the nroff and troff formatters using the ‘‘me’’ or ‘‘ms’’ macros, often using preprocessors such as *eqn* for mathematics, *tbl* for tables, and *pic* for diagrams. The instructions for printing the document should be collected in a Makefile.

The source to user’s manuals should be stored in the ‘‘doc’’ directory and archived using RCS.

7.3. Technical Reports

Technical reports describe ideas and algorithms (which are usually unimplemented). They should be collected in a designated, common place, with commands for printing them encapsulated in a Makefile.

8. MAINTENANCE

8.1. File Protection

Public source directories should have mode 775.

8.2. Understanding Software

UNIX has a few tools for helping a programmer understand a complex piece of software. The *ctags* program makes a list of the locations of all subroutines and typedefs in a list of source files which can be used as an aid to source searching in vi. The *xref* program (available on some systems) will produce a cross-reference table for variables and subroutines in several source files.

8.3. Debugging

For ‘‘interactive’’ debugging compile with ‘‘cc -g’’ and run *dbx* or your local debugger.

The *lint* program should be run over the source just prior to release. It is particularly helpful for spotting uninitialized variables and incorrect subroutine arguments. To perform checking against subroutines from other packages it is necessary to create a *lint library file* for each public library. This can easily be created by running ‘‘lint -C*name*’’ over all of the source files in a library. The resulting lint library file, called ‘‘llib-l*name*.ln’’ should be placed in the lib directory alongside the library itself. When linting software which accesses this library one simply adds ‘‘-l*name*’’ to the command line, as with *ld*. Lint uses the LINTPATH environment variable to locate the lint library file.

8.4. Version Control: RCS

For version control we use RCS. This is especially important when several people are working on a program. As discussed in the RCS manual, after each round of modifications but prior to compilation, the source to the current version of a program should be checked in and back out (with ‘‘ci’’), so that the RCS files contain the latest version. **Except during editing, all source files outside of RCS are considered reconstructible and hence expendable.**

One feature of RCS is that it can mark a source file with one of several keywords containing the version number, the date of modification, or similar information. If this string is included in a global array then the version numbers will be compiled into object files and executables. RCS keywords in binary or ascii files can be extracted with the ‘‘ident’’ command. To support this put the line

```
static char rcsid[] = "$Header$";
```

at the top of all source files. For header files put the keyword ‘‘$Header$’’ in a comment.

8.5. Compilation: Make

Ideally, anyone should be able to recompile public software without intimate knowledge of it. In particular, public source should have a corresponding Makefile or Makefile entry for rebuilding it. If a program is created from multiple files, those files and a Makefile should reside in a directory of their own and the command ‘‘make all’’ or ‘‘make’’ should regenerate the program. Otherwise, ‘‘make *command*’’ should rebuild the specified *command*. A Makefile should also include comments listing compile-time configuration flags (‘‘cc -D...’’) and their meanings. The following minimal set of entry points should be supported by a Makefile:

COMMAND	EFFECT
make *cmd*	recompile just the program *cmd* contained in the current directory
make	recompile all the programs and/or libraries contained in the current directory
make depend	create a new version of Makefile which includes up-to-date source file interdependencies
make install	install the executable programs or library archive files for use
make clean	clean the directory of all reconstructible and junk files
make tags	create a *tags* file containing definitions found in the current directory’s files

If there are public header files associated with a library then ‘‘make install’’ should place copies of the files in the appropriate include directory. Similarly for shell scripts, the master source should be kept in *src* and installed into bin after stripping the .sh or .csh suffix. The master source for manual entries should not be in src but in man.

Makefiles for libraries often create a second version with profiling enabled plus a lint library file. These are both installed in lib. When software is spread across a multi-level directory structure, master Makefiles should be created to glue together those Makefiles located in subdirectories. Example Makefiles following these conventions are listed in the appendix.

The default in C is to coerce floats into doubles for arithmetic operations or argument-passing, but many compilers have options to disable this coercion, allowing the programmer to get faster speed. For example, on a Sun, to get float arithmetic on a 68881 floating point processor, one compiles with ‘‘cc -fsingle -f68881’’, and to get fast floats on an SGI machine, one compiles with ‘‘cc -float’’. Parameterization of Makefiles to facilitate porting is illustrated in the appendix.

8.6. Installation

When a formerly private program is being released to the public the programmer should make a pass over it checking for all of the qualities we have been discussing: naming, style, comments, portability, error handling, and documentation. For complex software the program should be run through a series of tests prior to installation.

When a new version with major modifications is installed the man page should be updated and interested users should be notified via mail. This will encourage them to take advantage of new features and be on their toes for new bugs. When the programmer is not confident of the new version of a program the old version should be saved with the suffix ‘‘.old’’ in the bin directory.

8.7. Synopsis of RCS and Make Usage

The routines for creating, releasing, and recompiling are summarized below:

Creating:

write programs and Makefile
mkdir profiled   *(for a library)*
make, *test, and debug until working*
put in RCS keywords
mkdir RCS
ci -u Makefile *.c *.h </dev/null
make
make install

Making a New Version:

co -l RCS
edit, make, *and test until working*
ci -u -m’*description_of_changes*’ *modified_files* </dev/null
make
make install

Releasing (a substantial version change):

make depend   *(to ensure that Makefiles are up-to-date)*
make clean   *(to delete cruft)*
check that all directories are clean
co RCS/*   *(synchronize all version nos.)*
ci -r*version* -u -m’*description_of_version*’ RCS/*
make   *(reconstruct everything)*
make DESTDIR=*release_dir* install

8.8. Network Distribution

In a production environment **it is essential that all machines have the same version of every program.** Because we have several different kinds of computers, we must sometimes copy the source to other machines and recompile. If possible, all redundant copies of the source should be deleted.

The bin, lib, include, and man directories (everything but src) should be updated on all other machines whenever a change is made. This can be done automatically every night using the rdist program. Recompilation for machines of another type is often a manual procedure, however.

8.9. Responsibility

In order to ensure that the guidelines discussed herein are followed it is advisable to designate a few people who are ultimately responsible for maintaining the source directories and for looking after the documentation and technical reports.

9. ACKNOWLEDGEMENTS

Sam Leffler contributed the manual entry and makefile templates and helped me refine these standards along with Pat Hanrahan, Bill Reeves, Eben Ostby and Rob Cook.

10. SEE ALSO

Make − A Program for Maintaining Computer Programs, by S. I. Feldman.
Design, Implementation, and Evaluation of a Revision Control System, by Walter F. Tichy, Proc. 5th Intl. Conf. on Software Engineering, IEEE, Sept. 1982.
An Introduction to the Source Code Control System, by Eric Allman.
Programming Languages: Concepts and Constructs, by Ravi Sethi, Addison-Wesley, Reading MA, 1989.

11. APPENDIX – MAKEFILE DETAILS

This appendix describes Makefile conventions which should be observed and provides templates for the major types of Makefiles. These Makefiles demonstrate the conventions that might be used in a hypothetical organization with two main projects: R&D work in /u/rnd, and systems work in /u/sys.

As described in §8.5., a Makefile must support certain standard entry points and allow them to be cascaded when they are part of a hierarchy of software. The commands ‘‘make’’, ‘‘make depend’’, ‘‘make install’’, ‘‘make clean’’, and ‘‘make tags’’ should recurse on all subdirectories. In addition, it is highly desirable that Makefiles be portable to other machines without modification. On other machines the graphics and systems software might not be in the standard places (/u/rnd and /u/sys) so their locations should be parameters to all Makefiles. To facilitate software releases it is helpful to keep the destination directory variable as well. To these ends several standard Makefile macros have been defined:

DESTDIR	installation hierarchy
RND	research & development hierarchy [/u/rnd]
SYS	systems hierarchy [/u/sys]
COPTS	options to *cc* such as ‘‘-O’’ and ‘‘-g’’
FLOATOPT	options to *cc* to control floats & doubles

Makefiles should set these macros to the default values listed above in brackets. The FLOATOPT variable is used to set compiler options to control coercion of floats into doubles. The following templates have FLOATOPT configured for SGI machines. Such assignments may be overridden on the command line, for example:

```
make  RND=/usr/ph/rnd  SYS=/usr/sam/sys  DESTDIR=/usr/sam/rnd  install
make  DESTDIR=/rel1.1/rnd  install
make  COPTS=-g  FLOATOPT="-fsingle -f68881"  foo
```

the first illustrating compilation on a strange machine, the second illustrating creation of a binary release, and the third illustrating compilation for debugging on a Sun.

Since they are variable these macro definitions should be passed on to all Makefiles in subdirectories. Aside from the above macros, Makefiles should always reference *cc* (1) with $(CC) and *as* (1) with $(AS); and pass on command line options to *make* using $(MFLAGS). These macros have special meaning in *make*.

11.1. Makefile Templates

We present three types of Makefiles: one for src/bin, one for a library, and one for a package within a library. All variable portions of the Makefiles are shown in italics. A Makefile for a ‘‘bin’’ directory is shown below:

```
# Makefile for src/bin
# $Header$
#
RND=        /u/rnd
SYS=/u/sys
DESTDIR=$(RND)
MAKE=       make $(MFLAGS) RND=$(RND) SYS=$(SYS) DESTDIR=$(DESTDIR)

FLOATOPT=       –float
COPTS=      –O
IPATH=      –I$(RND)/include –I$(SYS)/include
LINTPATH=$(RND)/lib:$(SYS)/lib:/usr/lib/lint
LIBS=       –lpic –lm
CFLAGS=     $(COPTS) $(FLOATOPT) $(IPATH)

# Programs that live in subdirectories, and have makefiles of their own.
#
SUBDIR=     dir1 dir2

# Shell scripts that need only be installed and are never removed.
#
SHSCRIPT=       uno dos
CSHSCRIPT=      tres

# C programs that live in the current directory and do not need
# explicit make lines.
#
STD=one two

# C programs that live in the current directory and need explicit make lines.
#
NSTD=       three four

PROGS=      $(STD) $(NSTD)


all: $(SUBDIR) $(PROGS)

$(SUBDIR): GHOST
	cd $@; $(MAKE)

GHOST:

$(STD):
	$(CC) $(CFLAGS) –o $@ $@.c $(LIBS)

install: all
	for i in $(SUBDIR); do \
		(cd $$i; $(MAKE) install); done
```

```
	for i in $(SHSCRIPT); do \
		install −c $$i.sh $(DESTDIR)/bin/$$i; done
	for i in $(CSHSCRIPT); do \
		install −c $$i.csh $(DESTDIR)/bin/$$i; done
	for i in $(PROGS); do \
		install −s $$i $(DESTDIR)/bin; done
#	break and recreate links here, if any

clean:
	rm −f a.out core *.s *.o #* *.lint $(PROGS)
	for i in $(SUBDIR); do \
		(cd $$i; $(MAKE) clean); done

depend:
	depend −e $(IPATH) −c $(PROGS)


# Files listed in $(NSTD) have explicit make lines below.

three:
	$(CC) $(CFLAGS) −o three three.c −larg $(LIBS)

four:
	$(CC) $(CFLAGS) −o four four.c −lm

four.lint:
	LINTPATH=$(LINTPATH); export LINTPATH; \
		lint $(IPATH) four.c −lm >four.lint
```

This may look excessively complex, but normally the bulk of this Makefile will remain unchanged. Usually the macro assignments at the top and the explicit make commands at the bottom would be the only portions modified.

Here we have used a trick used to ensure that subdirectories are always checked: all subdirectories are made to depend on a non-existent file called ‘‘GHOST’’. Each subdirectory has its own Makefile. ‘‘Make depend’’ invokes the *depend* program, which finds all header files accessed by a list of source files and appends a list of dependencies to *Makefile*. This Makefile also demonstrates setting the shell variable LINTPATH and the use of the ‘‘-lm’’ lint library under the rule for ‘‘four.lint’’.

Note that when setting IPATH we explicitly list all special include directories needed for compilation. Some versions of ‘‘ld’’ support a ‘‘-L’’ option analogous to ‘‘-I’’ for specification of a library path. Yet other compilers employ environment variables called INCPATH and LIBPATH for these purposes. For maximum portability, only the least common denominator of these options should be used, as shown here.

The following Makefile is suitable for a library directory:

```
# Makefile for the libmath library
# $Header$
#
RND=      /u/rnd
SYS=/u/sys
DESTDIR=$(RND)
MAKE=     make $(MFLAGS) RND=$(RND) SYS=$(SYS) DESTDIR=$(DESTDIR)

NAME=     math
LIB= lib$(NAME)
LLIB=     llib−l$(NAME).ln
ARCS=     $(LIB).a $(LIB)_p.a

FLOATOPT=      −float
COPTS=    −O
IPATH=    −I$(RND)/include −I$(SYS)/include
CFLAGS=   $(COPTS) $(FLOATOPT) $(IPATH)
PKG=      math
INCS=     root.h
SRCS=     root.c
OBJS=     root.o

SUBDIR=   quat v3f


.c.o:
        $(CC) −c −pg $(CFLAGS) $*.c
        −ld −X −r $*.o
        mv a.out profiled/$*.o
        $(CC) −c $(CFLAGS) $*.c
        −ld −X −r $*.o
        mv a.out $*.o

all: $(ARCS) $(LLIB)

$(ARCS): $(OBJS) $(SUBDIR)
        @echo updating profiled and normal libraries
        @cd profiled; ar ru ../$(LIB)_p.a $(OBJS)
        @ar ru $(LIB).a $(OBJS)
        ranlib $(ARCS)

$(SUBDIR): GHOST
        cd $@; $(MAKE)

GHOST:

$(LLIB): $(SRCS) $(INCS)
        lint $(IPATH) −C$(NAME) $(SRCS)

install: all
        install −m 644 $(LIB).a $(DESTDIR)/lib
        install −m 644 $(LIB)_p.a $(DESTDIR)/lib
```

```
		cat */$(LLIB) >>$(LLIB)
		install −m 644 $(LLIB) $(DESTDIR)/lib
		for i in $(INCS); do \
			install −c −m 444 $$i $(DESTDIR)/include; done
		for i in $(SUBDIR); do \
			(cd $$i; $(MAKE) install); done

	tags: $(SRCS) $(INCS)
		rm −f tags
		cwd=‘pwd‘; for i in $(SRCS) $(INCS); do \
			ctags −a $$cwd/$$i; done
		for i in $(SUBDIR); do \
			(cd $$i; $(MAKE) tags); done
		cat tags */tags | sort −o tags

	lint: $(SRCS) $(INCS)
		lint $(IPATH) $(SRCS) >$(PKG).lint

	depend:
		depend $(IPATH) $(SRCS)
		for i in $(SUBDIR); do \
			(cd $$i; $(MAKE) depend); done

	clean:
		rm −f profiled/*.o *.o core a.out errs #* *.lint $(LLIB) $(ARCS)
		for i in $(SUBDIR); do \
			(cd $$i; $(MAKE) clean); done
```

This one is even more complex, but aren’t you glad we figured this out so you wouldn’t have to?

This is the Makefile for our hypothetical libmath, which contains two packages in subdirectories plus source files in the current directory. ‘‘Make all’’ will create three files: the normal library, a profiled version for use with the *gprof* program, and a lint library file. This Makefile assumes that all subdirectories with Makefiles have rules for ‘‘install’’, ‘‘tags’’, ‘‘depend’’, and ‘‘clean’’. It is assumed that the subdirectories add their object files to the archive files in the current directory, and that each creates its own lint library file and a tags file with full pathnames which are concatenated to form the lint library file and tags file for the entire library. The tags file uses absolute pathnames. This Makefile can be simplified substantially when there are no subdirectories.

The following Makefile is suitable for a package within a library:

```
# Makefile for the v3f package, which is part of libmath
# $Header$
#
RND=       /u/rnd
SYS=/u/sys

NAME=      math
LIB= ../lib$(NAME)
LLIB=      llib–l$(NAME).ln
ARCS=      $(LIB).a $(LIB)_p.a

FLOATOPT=      –float
COPTS=     –O
IPATH=     –I$(RND)/include –I$(SYS)/include
CFLAGS=    $(COPTS) $(FLOATOPT) $(IPATH)
PKG=       v3f
INCS=      v3f.h
SRCS=      dot.c unit.c print.c
OBJS=      dot.o unit.o print.o

.c.o:
	$(CC) –c –pg $(CFLAGS) $*.c
	–ld –X –r $*.o
	mv a.out profiled/$*.o
	$(CC) –c $(CFLAGS) $*.c
	–ld –X –r $*.o
	mv a.out $*.o

all: arcs $(LLIB)

arcs:
	@echo updating profiled and normal libraries
	@cd profiled; ar ru ../$(LIB)_p.a $(OBJS)
	@ar ru $(LIB).a $(OBJS)
	ranlib $(ARCS)

$(LLIB): $(SRCS) $(INCS)
	lint $(IPATH) –C$(NAME) $(SRCS)

install:
	for i in $(INCS); do \
		install –c –m 444 $$i $(DESTDIR)/include; done

tags: $(SRCS) $(INCS)
	rm –f tags
	cwd=‘pwd‘; for i in $(SRCS) $(INCS); do \
		ctags –a $$cwd/$$i; done
	sort tags –o tags

lint: $(SRCS) $(INCS)
	lint $(IPATH) $(SRCS) >$(PKG).lint
```

```
depend:
        depend $(IPATH) $(SRCS)

clean:
        rm −f profiled/*.o *.o core a.out errs #* *.lint $(LLIB)
```

This is essentially a simplified version of the library Makefile. Note that ‘‘make all’’ adds to an archive file in the parent directory.

11.2. Depend

The *depend* program takes a list of C source files and creates from it a list of header file dependencies that is appended to the Makefile in the current directory. For example, the command ‘‘depend -I/u/rnd/include *.c’’ in our hypothetical ‘‘v3f’’ directory might append something like the following to the Makefile:

```
arith.o: /usr/include/math.h
arith.o: /u/rnd/include/v3f.h
geom.o: /usr/include/math.h
geom.o: /u/rnd/include/v3f.h
...
```

The *depend* program facilitates the maintenance of such dependency lists. The dependencies generated by *depend* default to those needed in the Makefile for a library; when compiling executables, one often wants the ‘‘-e’’ or ‘‘-o’’ options. For example, the dependencies of a program in foo.c can be generated with ‘‘depend -o -I*include_dir* -c foo’’.

A shell script implementation of *depend* is listed below. Note that ‘‘/lib/cpp -M’’ does the hard work.

```
#! /bin/sh
makeexec=false
otoe=false
suff=
for i in $*
do
	case $i in
	–e)	makeexec=true;;
	–o)	otoe=true;;
	–c)	if /bin/test x$suff = x; then suff=.c; else suff=; fi;;
	–I*)	INCS="$INCS $i";;
	–*)	cat <<EOF
depend: bad flag: $i
Usage:  depend [–e] [–o] [–Idir] [–c] files
–e	make executable directly from source
–o	make object on the way from source to executable
(default)	make object from source, but no executable
–Idir	include directory
–c	add .c suffix to supplied names
EOF
		exit;;
	*)	SRCS="$SRCS $i$suff";;
	esac
done
temp=/tmp/depend$$
cat <<EOF >$temp
# DO NOT DELETE THIS LINE
# depend REWRITES EVERYTHING FROM HERE ON
#
EOF
for i in ${SRCS}; do
	if /bin/test $otoe = true; then
		echo ‘basename $i .c‘: ‘basename $i .c‘.o >>$temp
	fi
	/lib/cpp –M ${INCS} $i | sort –u | sed ’s| \./| |’ >>$temp
done
if /bin/test $makeexec = true; then
```

```
        sed 's/\.o:/:/' <$temp >${temp}x
        mv -f ${temp}x $temp
fi
echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >>$temp
mv -f Makefile \#Makefile
sed '/^# DO NOT DELETE THIS LINE/,$d' <\#Makefile >Makefile
cat $temp >>Makefile
rm $temp
```

Table of Contents