CHARA TECHNICAL REPORT

NO. 70 14 JULY 1998

Coding Practices for the CHARA Array

THEO TEN BRUMMELAAR (SUSI/CHARA) AND ANDREW BOOTH (SUSI) [2]

ABSTRACT: This document, slightly adapted from the manual *Coding Practices for SUSI* by Theo ten Brummelaar and Andrew Booth, describes a standard approach for software development and layout. All code written for the CHARA Array should adhere to the rules set out herein.

1. INTRODUCTION

This document outlines a standard for coding practices to be followed when writing software. We wish to adopt this standard in order to fulfill the following objectives:

- to assist in the process of structured analysis and design.
- to assist in the debugging and maintenance of programs.
- to assist in future updates of programs.
- to develop and maintain run-time libraries that all coders have access to.
- to facilitate the movement of code between different programmers.
- to assist in the development of automatic methods of "hardcopy documentation" and manual writing.

It is therefore expected that anyone writing code for the Array will accept the following standard as the definitive guide and supply code and associated documentation accordingly.

2. SOURCE CODE LANGUAGE

All programs in the CHARA Array software set are to be written in ANSI *C* or *Bourne shell scripting language* (*sh*, although *bash* will also be acceptable). Proper prototyping of all functions is to be maintained. Always compile code using the *gcc* command and it is strongly recommended that the -*Wall* flag always be used. Source code should produced no errors, and few warnings when using this flag.

All code will make use of the standard libraries:

[1]Center for High Angular Resolution Astronomy, Georgia State University, Atlanta GA 30303-3083
TEL: (404) 651-2932, FAX: (404) 651-1389, FTP: ftp.chara.gsu.edu, WWW: http://www.chara.gsu.edu

[2]with some SUSI/CHARA translation by Bill Hartkopf

- *charaui* : User interface.
- *nrc*: numerical recipes in C.
- *simpleX*: X interface.
- *rwfits*: FITS file read/write routines.

Other standard libraries will be added over time. Man pages for all these libraries are available online[3].

Assembly code is to be avoided at all costs, unless no alternative is available. *C++* is allowed for the object oriented fans, but it is not recommended for use in device controllers due to the expensive overhead. Furthermore, many of us are yet to become familiar with *C*, let alone *C++*, so it is best to stick with *C*.

3. DEVELOPMENT ENVIRONMENT

Development will be carried out under control of the **UNIX** software development tools **CVS(1)** and **make(1)**. For detailed and well written descriptions of CVS facilities refer to *Version Management with CVS* (Per Cederqvist et al.). Use of these facilities will ensure that different parts of the software development can proceed in a consistent way, and shared software can be easily controlled to avoid inconsistencies. Further, it should ease the problems of other programmers taking over management of some area of software development.

3.1. Directory Structure

The software development will consist of several projects, and each project should be broken into logical modules. Each module should have its own directory, and possibly subdirectories for large modules. As detailed below, each directory will have its own *Makefile* and *CVS* subdirectory.

Within each module directory there should be the *.c* files and *.h* files for that module. For the project there should possibly be a directory containing project wide header files (call this “include”) and a directory containing project wide archive libraries (call this “lib”), see **ar** below. There may well also be CHARA-wide *include* and *lib* directories. These *lib* and *include* directories should also be under the control of **CVS** and **make**. Finally there should be a project *Makefile* in the project directory to assemble the whole thing.

The project should also have a documentation directory (which can also be under the control of **CVS** if desired), though it should be remembered that release versions of the documentation will be stored in the central file as hard copy and be placed online.

The total directory structure should thus look something like:

[3]And will soon be available as a TR.

```
chara-array/   project1/         Makefile
                                 module1/     file1.c
                                              ⋮
                                              filen.c
                                              header.h
                                              Makefile
                                              CVS/
                                 ⋮
                                 modulen/     file1.c
                                              ⋮
                                              filen.c
                                              header.h
                                              Makefile
                                              CVS/
                                 include/     header1.h
                                              ⋮
                                              headern.h
                                              Makefile
                                              CVS/
                                 lib/         lib1.a
                                              ⋮
                                              libn.a
                                              Makefile
                                              CVS/
                                 docs/        document.tex
                                              CVS/
               ⋮
               other project
               directories
               ⋮
               include/          header1.h
                                 ⋮
                                 headern.h
                                 Makefile
                                 CVS/
               lib/              lib1.a
                                 ⋮
                                 libn.a
                                 CVS/
                                 Makefile
```

3.2. CVS

CVS stands for *Concurrent Versions System* and is a method of ensuring that modifications to programs are performed in a consistent way, and that conflicts do not arise when more than one person is working on the same piece of software. In addition, it provides an easy way of retrieving previous versions of software if subsequent modifications are found to be wrong.

CVS works by keeping a central repository of the entire software tree used in the system. This is kept in a single directory */chara/chara/cvsroot.* To reach this directory each programmer needs to have an environment variable pointing to it as follows (for *csh*

```
setenv CVSROOT /chara/chara/.cvsroot
```

which needs to be added to your *.login* file.

Software trees are entered into the repository using the command

```
cvs import Name CHARA Programmer
```

where name is the name of the module. Each programmer can then create his/her own personal copy of the source code they are working on by issuing the command

```
cvs checkout module_name
```

where *module_name* is the name of the module they wish to work on. Periodically this code can be sent back into the repository by using the

```
cvs update
```

command if you are inside the directory containing the code or the

```
cvs commit
```

command from the root directory of the code. Code should be committed at least once per day. Naturally, it is not a good idea to commit code that doesn't actually work.

When code is checked back into the repository it is given a new version number and compared to the existing code. It is possible for many people to be modifying the same program and *CVS* is smart enough to blend all of the changes together. It will, fortunately, warn you of any conflicting changes.

Many other *cvs* commands exist and it is highly recommended that you read the *CVS* man page.

3.3. make

make(1) is a utility that allows you to control more easily the compilation of complicated programs from several source files. The information required to do the compilation is stored in a file called *Makefile* in the working directory. **make** when invoked will do the minimum amount of compilation necessary to produce the program (thus saving time for large program suites), while ensuring that the most recent versions of files are used.

To make the most use of **make** you should divide your program into several source code files. This is good programming practice in any case, and will mean that when you are working on one part of the code you do not need to recompile all the other parts that already work. Try to divide your program up into logical groups of functions, and do not let any one file become more than about 10 pages long.

As a simple example of how **make** works, and the structure of a *Makefile*, consider a program called *prog* to be made from 3 source files *f1.c, f2.c, f3.c*, and compiled with the standard **C** maths library. Then **make** would call *prog* the *target*, *prog* would have *f1.o, f2.o* and *f3.o* (note the *.o* **not** *.c*) as *dependencies* and there would be a *command line* to compile the lot, of the form:

```
{\em gcc -o prog f1.c f2.c f3.c -lm}
```

A *Makefile* of the form:

```
prog: f1.o f2.o f3.o
        gcc -o prog f1.o f2.o f3.o -lm
```

when invoked with the command "`make prog`" will do everything required to produce *prog.* Note the space in front of **gcc** is a *tab*, this is a requirement to introduce a *command line.* Notice that you don't have to tell **make** to produce the .o files from the .c files, it is intelligent enough to do this itself.

A slightly more sophisticated use of **make** makes control of larger projects more convenient, and a template *Makefile* with comments is available (see Appendix A). This includes some useful extra *targets* such as *clean* and *all.* For example `make clean` removes all old `.o` files and executable files.

make can also control the production of archives (see below) of compiled functions, to be included in later compilations, including placing the archive library in the correct directory so that others can use it. It is important to realize that any **UNIX** command can be put on a *Makefile command line.*

3.4. ar

Once you have a set of working functions that either you or others will want to use as modules in other programs, you should compile them and store the compiled versions in a *library archive* using **ar(1)**. Library archives should be given names starting with "lib" and ending in the suffix ".a". For example, to create an archive called *fred* containing the compiled files *f1.o, f2.o* use

```
ar rcv libfred.a f1.o f2.o
ranlib libfred.a
```

To use that library, assumed stored in a directory "/usr/local/lib", at load time for a program compilation use, for example,

```
gcc -o prog prog1.c prog2.o -lfred
```

4. GENERAL LAYOUT AND STYLE

Although the detailed layout of the program is of course up to you a number of general rules apply to standardize code for the project. Appendix D contains an example of a source file that adheres to these rules. Please keep the following in mind when writing code for the project.

- No source file should be more than about 10 pages long.
- Functions should not be more than about 40 lines long unless you get carried away with your commenting. This is not always possible, but remeber that long functions are difficult for others to read and properly comprehend.
- The standard tab stop is 8 characters, although other tab stops will be allowed. The import thing is to be consistent.
- Lines must not wrap beyond 80 characters. This can cause some problems especially with long function calls or algebraic statements. Consider the following example:

```
printf("The results are rather long and are as follows %d %d %d",argum
ent1,argument2,argument3);  /* WRONG */

printf
(
        "The results are rather long and are as follows %d %d %d",
        argument1,
        argument2,
        argument3
);                  /* RIGHT */

printf("The results are rather long and are as follows %d %d %d",
        argument1,
        argument2,
        argument3); /* ALTERNATIVE */
```

  Note that the commas appeared at the end of the lines and not the beginning. An analogous system can be used for long algebraic statements. Place operators at the end of lines and not beginnings. Use indentation and brackets for clarity.

- Beware of the useful but dangerous `++` and `--` operators. For example the statement

```
a[i] = i++;
```

  will produce unpredictable results and will depend on the compiler used.

- Try and use the following format for switch statements:

```
switch(c = getch())
{
        case 'a' : do_this();   /* Action for case a */
                   break;

        case 'b' : do_that();   /* Action for case b */
                   break;

        case 'c' : do_such();   /* Action for case c */
                   break;

        default  : panic();     /* Default action */
                   break;
} /* end switch */
```

It is important that every switch contains a default case at the end if only for the sake of error trapping. You will also note that each case is ended with a break. It is dangerous to rely on fall through in switch statements.

- Closing curly braces should always be placed on lines by themselves and match the column of the statement they refer to, for example:

```
for (i = 0; i < 10; i++)
{
        do_this();

        if (the_pope_is_polish())
        {
                do_that();
                and_do_something_else();
        }
}  /* end loop */
```

It is also all right to place the opening curly brace on the same line as the opening statement, for example

```
for (i = 0; i < 10; i++) {
        do_this();

        if (the_pope_is_polish())
        {
                do_that();
                and_do_something_else();
        }
}  /* end loop */
```

although I have found that this is not as clear, even if it does take up less space.

Also consider

```
if (c == 1)
{
        do_this();
        do_that();
}
else if (c == 2)
{
        do_such();
        do_thus();
}
else
{
        do_so();

}  /* end if */
```

However for short statements it is allowable to put two statements on one line, for example;

```
if ( c != 2 ) do_something();
```

- The only braces to appear in column one are for beginning and ending functions or for structure definitions. Note that these structure definitions must be at the top of the block in which they appear.
- Where possible always leave spaces before and after operators and if it will make things clearer use more brackets than are strictly necessary.
- Try to avoid using globals. If you must it is a good idea to put them all in a single structure and declare it as a *static* so no conflicts can arise with other source files.

5. COMMENTING

All programs are to be fully commented. This means that anyone should be able to read and understand your code without having to go and ask you what it is all about. The following rules apply to commenting code.

- Each source file will begin with a standard module header which can be treated as a "fill in the blanks" system. This is to ensure that no particular sections of header code are overlooked. An example of a module header can be found in Appendix B.
- Every function will be preceded by a standard function header. An example of such a function header can be found in Appendix C.
- Comment openers are only to appear in column 1 to designate module/function headers or to divide sections such as include files or defines. All other comments should line up with the code that they apply to.
- Each variable declaration or definition will be immediately followed by a comment describing the use of the variable.

- The function name is to be repeated as a comment after the final closing brace of the function.
- For very long switch statements or loops it is also a good idea to place a comment at the end such as

  ```
  /* end switch */
  ```

  for clarity.
- Multiline comments will be in the following form:

  ```
  /*
   * this is the first line of a multiline comment.
   * this is the second line of a multiline comment.
   * this is the third line of a multiline comment.
   */
  ```

6. VARIABLE USE AND NAMING CONVENTIONS

It is important that function and variable names reflect the use and purpose of the variable or function. Short or cryptic names will only confuse and force you to use more comments. Please keep in mind the following rules:

- Don't shy away from long names if you think it will make your code clearer. Most compilers will recognize up to 30 characters as unique in variable names. For example:

  ```
  int geab();                    /* not suitable */
  int goeatabrick();             /* better */
  int go_eat_a_brick();          /* preferred */
  int EatBrick();                /* alternative for pascal lovers */
  ```

- Prototype parameters, formal parameters and actual parameters should have the same name whenever possible. For example:

  ```


  .
  .
  .

  char *my_function(char *string);        /* Prototype parameter=string */
  .
  .
  .
  my_function(string);      /* actual parameter = string */
  .
  .
  ```

```
.
.

char *my_function(char *string) /* formal parameter = string */
{
        .
        .                        /* Function body */
        .

} /* my_function() */
```

- Function and variable names should in general be in lower case.
- Capitalize all defines except function macros.
- Capitalize all type definitions.
- Unless otherwise specified, physical quantities are to be expressed in SI units with all angles in radians.
- All matrix and vector mathematics is to follow the *Numerical Recipes in C* format. The entire NRC library is available globally. To use it, use the include file *nrc.h* and the link command *-lnrc*.
- Use the following format for structure definitions:

```
typedef struct tagMENU
{
        int   number_of_wheels;
        char  brand[81];
        .
        . /* rest of structure definitions */
        .
} MENU;

MENU menus[10];            /* preferred method */
struct tagMENU menus[10];  /* another way of doing it */
```

7. DOCUMENTATION

All programs should be fully documented. The user manual for a program should be placed in a central file and allocated a document number. They should also start with an abstract that describes the program's purpose. Programmer manuals should conform to the standard UNIX manual page layout. This is done using the Troff(1) man set of macros. You will find an example of such a file in Appendix E.

The utility *mkman*, will read a source file and create basic man pages for all functions in that file. *mkman* will only work if you have followed the commenting practices outlined above but can save a lot of time when generating code documentation. The *Makefile* should normally invoke *mkman* automatically during compile time.

8. PORTABILITY

You should aim to write portable code wherever possible. It is also important not to rely on machine implementations of types. It is a good idea to use typedefs for any variable that requires a particular internal representation.

While we plan on using *Linux 2.0.33* as the basic development environment this may change. Avoid using any functions not more generally available.

9. CONCLUSION

In order to maintain standards within the project it is vital that all staff members are familiar with and adhere to these coding practices. You should, of course, use a bit of common sense and ignore any directives that will make your code hard to read. Any deviation from these rules, however, should be fully documented. Any further questions can be asked of TtB.

A. MAKEFILE TEMPLATE

This is a template *Makefile* for use in a module directory that may create archive libraries and test programs.

```
########################################################################
# Makefile                                                             #
#                                                                      #
# Example Makefile for library directories.                            #
#                                                                      #
########################################################################
#                                                                      #
# Center for High Angular Resolution Astronomy                         #
# Georgia State University, Atlanta GA 30303-3083, U.S.A.              #
#                                                                      #
#                                                                      #
# Telephone: 1-404-651-1882                                            #
# Fax      : 1-404-651-1389                                            #
# email    : theo@chara.gsu.edu                                        #
# WWW      : http:www.chara.gsu.edu/~theo.html                         #
#                                                                      #
# (C) This source code and its associated executable                  #
# program(s) are copyright.                                            #
#                                                                      #
########################################################################
#                                                                      #
# Author : Theo ten Brummelaar                                         #
# Date   : 25th July 1996                                              #
########################################################################

# If you are going to change something it'll probably be here:

CC= gcc
CFLAGS= -g -O -Wall
LFLAGS= -lc -lm
LIBDIR=/usr/local/lib
INCDIR=/usr/local/include
LIBNAME=example

# Add any new targets here:

OFILES= \
        file1.o\
        file2.o\
        file3.o\
        file4.o

#
# Rule for compiling and adding a unit to the library
#
```

```
.c.o:
		$(CC) $(CFLAGS) -c $<
		ar r lib$(LIBNAME).a $@
		mkman $(LIBNAME) $<

#
# Master target
#

all: lib$(LIBNAME).a testprogram

#
# All routines depend on the header file
#

$(OFILES): $(LIBNAME).h

#
# test routine
#

testprogram: lib$(LIBNAME).a testprogram.o
	$(CC) $(CFLAGS) -o testprogram testprogram.o ./lib$(LIBNAME).a $(LFLAGS)

testprogram: $(LIBNAME).h

#
# Master rule for creating the library
#

lib$(LIBNAME).a: $(OFILES)
		ranlib lib$(LIBNAME).a
		ranlib lib$(LIBNAME).a

#
# Rule for cleaning up the directory
#

clean:
	rm -f $(OFILES) core lib$(LIBNAME).a testprogram
	rm -rf man

#
# Rule for installation
#

install: lib$(LIBNAME).a $(LIBNAME).h
	cp lib$(LIBNAME).a $(LIBDIR)/lib$(LIBNAME).a
	cp $(LIBNAME).h $(INCDIR)/$(LIBNAME).h
```

```
        ranlib $(LIBDIR)/lib$(LIBNAME).a
        chmod 644 $(LIBDIR)/lib$(LIBNAME).a
        chmod 644 $(INCDIR)/$(LIBNAME).h

#
# Rule for uninstalling
#

uninstall: clean
        rm -f $(LIBDIR)/lib$(LIBNAME).a
        rm -f $(INCDIR)/$(LIBNAME).h
```

A *Makefile* for a project directory would be similar, but without the library build.

Below is a template *Makefile* to reside in the *project* directory and recursively **make** all the modules, libs, and includes under it.

```
########################################################################
# Makefile                                                             #
#                                                                      #
# Master makefile for libraries                                        #
#                                                                      #
# Based on a Makefile by AJB Sept 89 booth@physics.su.edu.au           #
########################################################################
#                                                                      #
# Center for High Angular Resolution Astronomy                         #
# Georgia State University, Atlanta GA 30303-3083, U.S.A.              #
#                                                                      #
#                                                                      #
# Telephone: 1-404-651-1882                                            #
# Fax      : 1-404-651-1389                                            #
# email    : theo@chara.gsu.edu                                        #
# WWW      : http:www.chara.gsu.edu/~theo.html                         #
#                                                                      #
# (C) This source code and its associated executable                   #
# program(s) are copyright.                                            #
#                                                                      #
########################################################################
#                                                                      #
# Author : Theo ten Brummelaar                                         #
# Date   : 6/25/96                                                     #
########################################################################

# fill in your bits for SUBDIRS

SUBDIRS= module1 module2 module3

# You should not have to change anything else.

TARGETS= all clean install uninstall
```

```
# TARGETS should match "makefile" "targets" in all the SUBDIRS.
# A missing target in a subdir will cause a "fatal" error which will be
# ignored

.KEEP_STATE:

$(TARGETS):
        $(MAKE) $(SUBDIRS) TARGET=$@

$(SUBDIRS): FORCE
        cd $@; $(MAKE) $(TARGET)

FORCE:
# thus "make all" here runs "make all" in all the SUBDIRS
# subdirectories.  Obviously this can be recursive, so
# running copies of itself in lower and lower subdirectories
```

B. MODULE HEADER TEMPLATE

Template header for a **C** source file. All modules are to contain this header.

```
/****************************************************************/
/* filename.c                                                   */
/*                                                              */
/* Description                                                  */
/* Try and make this a valuable description, including a list   */
/* of the functions within this file. Descriptions like         */
/*                                                              */
/* This file contains the code for the 'filename' program.      */
/*                                                              */
/* waists everyones time, including your own.                   */
/*                                                              */
/****************************************************************/
/*                                                              */
/* Project_name                                                 */
/*                                                              */
/* The CHARA Array                                              */
/*                                                              */
/* Center for High Angular Resolution Astronomy,                */
/* Georgia State University, Atlanta GA 30303-3083, USA         */
/*                                                              */
/* VOX: 404/651-2932                                            */
/* FAX: 404/651-1389                                            */
/* Email: theo@chara.gsu.edu                                    */
/* WWW: http://www.chara.gsu.edu                                */
/*                                                              */
/* (C) This source code and its associated executable           */
/* program(s) are copyright.                                    */
/*                                                              */
/****************************************************************/
/*                                                              */
/* Author : Fill this in                                        */
/* Date   : This too                                            */
/****************************************************************/

/* include files */

/* external function declarations */

/* defines */

/* globals */

/* Code */
```

C. FUNCTION HEADER TEMPLATE

The following is a template for function headers. All functions must be preceded by this header.

```
/****************************************************************/
/* function_name()                                              */
/*                                                              */
/* Place a description of the function here. This description   */
/* text is the text that the {\em mkman} utility will extract   */
/* and use as the body of the manual page created for this      */
/* function. Therefore, the better you make this description    */
/* the less work you will need to do later when you are doing   */
/* the documentation.                                           */
/*                                                              */
/****************************************************************/
```

D. EXAMPLE SOURCE FILE

```
/****************************************************************/
/* example.c                                                    */
/*                                                              */
/* This is an example resource file to demonstrate coding style.*/
/* It does not necessarily compile but it looks nice.           */
/*                                                              */
/* Functions included in this file are:                         */
/*                                                              */
/* struct smenu *initialise_menu_structure(void)                */
/* char *getline( char *string, int  n, FILE *stream)           */
/*                                                              */
```

```
/*************************************************************/
/*                                                           */
/* USER INTERFACE ROUTINES                                   */
/*                                                           */
/* The CHARA Array                                           */
/*                                                           */
/* Center for High Angular Resolution Astronomy              */
/* Georgia State University, Atlanta GA 30303-3083, USA      */
/*                                                           */
/* VOX: 404/651-2932                                         */
/* FAX: 404/651-1389                                         */
/* Email: theo@chara.gsu.edu                                 */
/* WWW: http://www.chara.gsu.edu                             */
/*                                                           */
/* (C) This source code and its associated executable        */
/* program(s) are copyright.                                 */
/*                                                           */
/*************************************************************/
/*                                                           */
/* Author : Tedy the Wonder Lizard                           */
/* Date   : 29 Feb 2000                                      */
/*************************************************************/

/* include files */

#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <charaui.h>

/*************************************************************/
/* initialise_menu_structure()                               */
/*                                                           */
/* This is the controlling function of the menu structure    */
/* initialization. See imbedded comments for a description of */
/* exactly what it does. It returns a pointer to the main menu. */
/*                                                           */
/*************************************************************/

struct smenu *initialise_menu_structure(void)
{
        /* Internal variables */

        FILE    *structure_file;   /* Pointer to structure text file */
        struct  smenu *mainmenu;   /* Pointer to main menu (returned) */
        char    errorstr[81];      /* Error message string */

        /* Initialise ptcome and ptgoto to zero */

        ptcome = 0;
```

```
ptgoto = 0;

/* Open menu structure file for reading */

if (( structure_file = fopen(STRUCTFILE,"r")) == NULL)
{
        sprintf
        (
                errorstr,
                "Can not open menu structure file %s",
                STRUCTFILE
        );
        fti_error(3,errorstr);
}

/*
 * First pass      - Find menu names and count them
 *                 - Check that nummenus <= MAXMENU
 *                 - Allocate memory space for their structure
 *                 - Place menu names into their structure
 *                 - Fill rest of structure with NULLs
 *                 - Set menus[nummenus] to NULL
 *                 - Rewind the menu structure file
 */

passone(structure_file);

/*
 * First check    - Ensure there is exactly one main menu
 *                - Ensure all menu names are unique
 *                - Ensure no menu names are the same as a function name
 *                - Return a pointer to the main menu
 */

mainmenu = checkone();

/*
 * Second pass   - Read in menu data and place it in menu structures
 *               - Ensure all menu references exist
 *               - Ensure no menus are self referential
 *               - Rewind the menu structure file
 */

passtwo(structure_file);

/* Third pass   - Read in auto list data and fill in array autolist[]
 *              - Check that the number of functions is <= MAXAUTO
 *              - Ensure all function references exist
 *              - Place NULL at end of list
 *              - Ensure that the following functions are not in
```

```
         *                the list ; auto, help, end, quit
         */

        passthree(structure_file);

        /* Close the menu structure file */

        fclose(structure_file);

        /* Return a pointer to the main menu */

        return mainmenu;

} /* initialise_menu_structure */

/*****************************************************************/
/* getline()                                                     */
/*                                                               */
/* Reads a line from a given stream as text. Leading white       */
/* space is ignored. A line that begins with the # character is */
/* considered a comment and is ignored. The new line character   */
/* at the end of the string (if present) is replaced by a NULL. */
/* Getline returns a pointer to the string or EOF if the end of */
/* the file has been reached. Getline will read up to n-1        */
/* characters and then terminate.                                */
/*                                                               */
/*****************************************************************/

char *getline(
        char *string,     /* Buffer for placing string */
        int  n,           /* Number of characters in buffer */
        FILE *stream      /* Stream to get data */
             )
{
        /* Internal variables */

        int    c;              /* First character of string */
        char   *retval;        /* Return value of function */

        /* Set return value to string beginning */

        retval = string;

        /* Loop until line does not start with '#' */

        do
        {
                /* Skip leading white space */

                while
                (
                        (c = getc(stream)) == ' ' ||
                        c == '\t' || c == '\r' || c == '\n'
                )
                {
                        if (c == EOF) return (char *) EOF;
                }

                /* Put the first character into the string */

                *string++ = c;

                /* Read in the rest of the string */

                if (fgets(string, n-1, stream) == NULL)
                        return (char *) EOF;

        } while (c == '#');

        /* Replace new line character with NULL */

        while( *string != NULL )
        {
                if ( *string == '\n') *string = NULL;
                string++;
        }

        /* Return pointer to the string */

        return retval;

} /* getline() */
```

E. EXAMPLE UNIX MANUAL PAGE FILE

The *mkman* utility will create basic man pages, although you will probably want to edit them before you release them. The following is an example of a manual page. It can be formatted using the command:

```
nroff -man filename
```

The next page contains the input file and the following page shows what *nroff -man* will produce. Note that the output of *nroff* is not suitable for printing. To produce a postscript file use the command

```
groff -man filename
```

which will send the postscript version of the manual page to standard output.

```
.TH ENBLE_INTRPT 3 AV68K "Programmer's Manual"
.SH NAME
enble_intrpt - Enable interrupts.
.SH SYNOPSIS
#include        <bios.h>

.br
void        enble_intprt(int level)
.SH DESCRIPTION
.I Enble_intrpt
sets the current interrupt level to the value level. Note that
this means the computer will respond to interrupts of levels
higher than that set. Thus setting the level to 7 will turn off
all interrupts. This could be dangerous as it may stop the
UART interrupt server at level 6. Thus the only values you
should use are 0,1,2,3,4 or 5. This routine is almost
always called after
.I set_up_intrpt.
.SH RETURN VALUE
Void.
.SH SOURCE FILE
cbios.as
.SH "SEE ALSO"
set_up_intrpt()
.SH AUTHOR
Theo ten Brummelaar
```

NAME

    enble_intrpt - Enable interrupts.

SYNOPSIS

```
#include        <bios.h>

void       enble_intprt(int level)
```

DESCRIPTION

    Enbleintrpt sets the current interrupt level to the value level. Note that this means the computer will respond to interrupts of levels higher than that set. Thus setting the level to 7 will turn off all interrupts. This could be dangerous as it may stop the UART interrupt server at level 6. Thus the only values you should use are 0,1,2,3,4 or 5. This routine is almost always called after setupintrpt.

RETURN VALUE

    Void.

SOURCE FILE

    cbios.as

SEE ALSO

    setupintrpt()

AUTHOR

    Theo ten Brummelaar