Revised Coding Practices for SUSI*

T. A. ten Brummelaar, A. J. Booth, & W. J. Tango

March 19, 1999

Software is a living thing. If you don't maintain it, it dies.
Patrick Wallace, Guru

If it ain't broke, don't fix it.
Anon., also a guru

Abstract

This document describes a standard approach for software development and layout. All code written for SUSI should adhere to the rules set out herein.

1 Introduction to the revised version

This document outlines a standard for coding practices to be followed when implementing code for the SUSI control system. The original reasons for introducing a coding standard are still applicable:

- 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 "hard copy documentation" and manual writing.

The original specification envisaged that the SUSI operating system would be implemented in a UNIX environment, using the standard UNIX **cc** compiler and other utilities. The actual situation is more complicated and it has evolved as a distributed, multi-platform system. The current configuration includes:

- A SPARC workstation running several "user interfaces" (UIs);
- Several PCs, each running a dedicated UI under MS-DOS or in an MS-DOS window under Win9x;

*Document SUSI/COM/030-1.TEX

- One PC running a dedicated UI under Win3.x;
- Approximately 20 AV68K single-board computers which serve as "embedded processors" that interface directly with the SUSI hardware;
- A VME-bus supporting several MVME333 computers and associated serial communications devices, that provide high speed intercommunication between the embedded processors;
- One PC running a stand-alone LabView application;
- Another PC running an MS-DOS UI which interfaces with the LSR-Astromod CCD camera.

2 The Development Environment

This coding document refers specifically to project development in a UNIX-like environment, including Solaris, Linux, etc. Although components of the SUSI system run under other environments, either for convenience or because of specific hardware requirements, code should conform as much as possible to the guidelines set out in this document.

2.1 Programming language

All code should be written using ANSI standard C. C++, or C++ "add-ons" to the basic C language should be avoided.

2.2 Version control

Version control is essential to managing the SUSI operating system. We have used SCCS in the past and users are directed either to the first edition of this document or to the SCCS documentation supplied with their operating system manuals for more information.

SCCS is part of the commercial UNIX system and is not available on all platforms. The RVS and more recent CVS version control systems are supplied under the GNU General License and are found on most UNIX/LINUX installations. Future development may well be done using CVS in particular, but this document assumes that SCCS is used.

2.3 Use of "make"

In a large project, how does one generate the final product? One way might be to compile and link everything, but since this process can be quite slow it is not a practical option during the development stage of a project. One alternative is to write a shell script which checks the creation dates of files and compiles and links only those components which have changed since the last update. The creation and maintenance of such a script is difficult and the **make** utility to simplify the process.

Make relies on special scripts called *makefiles*. A makefile entry supplies the *rule* for creating or updating a *target* from its *dependencies*. The **make** program determines whether or not a target is "up to date" (i. e., it's creation or modification time is *after* the corresponding times for all its dependencies). If a target is not up to date **make** uses the target's rule to generate a new version. **Make** will only rebuild modules which require rebuilding and therefore minimises compilation time.

A target is usually a file. It may depend on one or more other files. The object file `grommit.o` will normally generated from a file called `grommit.c`, and `grommit.o` is said to depend on `grommit.c`. The file `grommit.c` may depend on other files through `#include` statements, etc. **Make** is recursive so it is permissible to add an entry to the makefile which declares that `grommit.c` depends on `farnarkle.h`.

Makefiles are notorious for their obscurity. There are two reasons for this:

- Early implementations required concise and cryptic notation in order to conserve memory and increase speed. As with the C programming, "clever" constructs often inhibit productivity as they obscure the purpose of the code.
- Like all scripting languages, makefiles utilise variables, special syntax for control statements, and so on. Makefile syntax causes problems for novices and experts alike. The most notorious example is the distinction between the "space" and "tab" characters. The body of a "rule" **must** begin with a tab character; typing in the equivalent number of space characters will not work!

It should also be noted that there are several dialects of **make** and this compounds the problem.

The **make** utility has traditionally been used to manage the compilation, linking and installation of source code. It is now recognised that it can be a powerful tool in maintaining other aspects of a project. The makefile can be used to update documentation and place it in standard locations, generate startup scripts for different graphical user interfaces, compile code for different platforms, and so on.

SUSI makefiles should adhere to the coding practices mandated elsewhere in this document. In particular:

- Use the GNU **make** utility. On commercial platforms this is often called **gmake** to distinguish it from the one supplied by the vendor.
- Avoid clever, complex or obscure constructs unless essential for the task at hand.
- Do not rely on environment variables! This is a particular problem for older MS-DOS systems. Makefiles which use environment variables are obviously not very portable and in the worst case a change to the environment variable table can break all your makefiles!

Makefile targets are usually files, but **make** recognises more general targets. The classic example is the "*clean*" target. The command

```
>make clean
```

is typically used to remove all files generated by previous makes. The `install` target is frequently used to move the final product (executable code, documentation, etc.) to the appropriate system locations. The following targets are standard and should be incorporated into all SUSI makefiles. An asterisk denotes an operation that may require root permission.

`all`		Compile the entire program. This should be the default (i. e.) the first) target in a makefile.
`check`		You may wish to include a test program in your directory which exercises the newly made executable. This target can be used to run your test program.
`clean`		Remove all intermediate files from the source directory. The rule for `clean` might look like this: `rm -f $(PROGRAM) *.exe *.o *.dvi *.log *.toc core` One can write gentler rules that only remove certain intermediate files but `clean` is typically used to remove all files *that can be regenerated*. Core dumps should always be cleaned up.
`distclean`		Make a clean directory for distribution. This is more aggressive than `clean` and is used mainly for directories which are maintained using automatically generated makefiles.
`debug`		Compile using the `-g` option, which allows the use of a symbolic debugger to monitor the execution of the program. Allegedly code compiled with this option is larger and slower, and it should not be used for generating an installation.
`install`	*	Compile and copy all the executables, libraries, documentation, . . . , to their appropriate places. This target should not make any changes to the local (source) directories. If a destination directory does not exist it is polite to ask the user if she wants to create them. The **install** utility should be used in preference to **mv**ing or **cp**ing files.
`installdirs`	*	Create installation directories if they do not already exist.
`uninstall`	*	This removes a previous installation of the software. If you intend to implement an uninstall option you must save a record of the last installation, or even a full installation history, in a safe place. Your makefile must be able to access this location and correctly remove all the relevant files.

For further details you should refer to the GNU Make documentation.[1] See Appendix A for some standard SUSI templates.

2.3.1 Why "make" is essential

Beginners often avoid writing makefiles because of their perceived complexity. It is easier to simply type

```
>gcc -o crunch crunch.c -lm
```

instead of writing a makefile. There are two reasons why one should use makefiles *always*:

1. Simple programs have a tendency to grow. Without a makefile, programmers will put all new functions, etc., in the original source file. This is bad programming practice (see below).

2. Makefiles document the process for generating the program. Quite often one writes a program and perhaps six months later needs to modify it for another purpose. The makefile contains all the information needed for compiling the program, including library calls, header files, etc. If the compilation fails because certain files have been moved or are no longer present, **make**

[1] *GNU Make*, R. M. Stallman & R McGrath (version 3.77, 1998). A hard copy version is available; ask WJT.

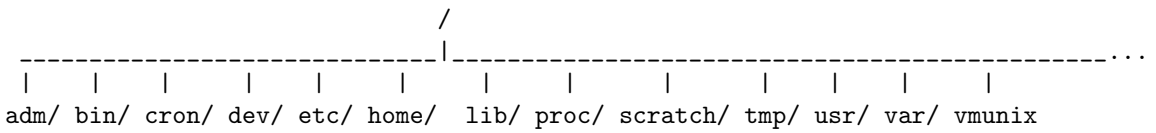

Figure 1: Directory tree for a typical UNIX system.

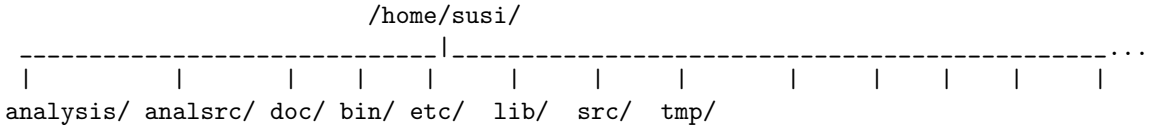

Figure 2: Structure of the SUSI home directory.

will generate error messages which will help to solve the problem. Without makefiles it can take hours or days to reconstruct the programming environment as it was when the program was first compiled.

2.4 Directory Structure

2.4.1 The UNIX directory tree

The directory structures for all UNIX systems are roughly similar.[2] The parent or *root* directory for the entire system is `/`, and all other files and subdirectories branch from the root directory. Figure 1 shows a typical directory file hierarchy.

Typically the only file in the top level of the tree is **vmunix** – the system kernel which is loaded when the computer is booted. All the other entries are directories (indicated by a trailing backstroke). The directory structure is recursive, so a directory at any level can have subdirectories.

Individual accounts are typically allocated home directories under `home`; e. g., `/home/tango`. Private users can access their own home directory and certain public access areas (such as a `/scratch`) directory for large, temporary storage). The other directories under `/` are used for "the system" and are maintained by the system administrator.

It is School IT policy to maintain a common system configuration on all the School SPARC workstations, and it is important to keep the SUSI system quite separate from this. The SUSI system resides in `/home/susi`. All the files in this directory are owned by the `susi` group, and anyone needing to work with the SUSI files should ask their system administrator to be added to the `susi` group.

2.4.2 The structure of the SUSI directory

The `susi` directory broadly follows the organisation of the system `usr` directory. Some of the principal sub-directories are shown in Fig. 2. The directory `src` contains all the source code for the operating system. With one exception, all SUSI-related files, including executables, should

[2]Readers familiar with the UNIX operating system can skip this section.

be placed in an appropriate directory under `/home/susi`. The `analsrc` directory, for example, contains the source files for programs used in analysing SUSI data. The `analysis` directory is a workspace for doing data analysis (this may be replaced with a SUSI scratch area).

The binary executables and scripts for running the SUSI system on the workstation have been placed in `/usr/local/bin/`. Because this is a standard system directory, in 1999 these files will be transferred to a new location, `/usr/susi/bin`.[3]

The software development consists of several projects, and each project is broken into logical modules. Each module should have its own directory containing several standard subdirectories. As detailed below, each directory will have its own *makefile* all source code and related materials will be accessed via a version control system.

A typical SUSI project will have code modules for the user interface (UI) and for an embedded controller (currently the embedded controllers are all AV68K computers). Separate subdirectories should be used for these modules. The UI code subdirectory is usually called `control` while the AV68K code is kept in a subdirectory called `av68k`.

Modules which use the RTP (Real Time Protocol) for interprocess communication should also have a subdirectory called `rtp`.

Each project should have a separate `include` subdirectory which will contain all the header files for the project.

The project should also have a documentation directory called `doc`. The documentation directory will typically contain non-release versions of the operating instructions for the particular module. Information relevant to programming (as opposed to operation) can also be kept here.

Some projects require data files. These should be kept in the `data` subdirectory. This data file is intended for "private" use. One might, for example, want to record information for debugging purposes during project development. Data should normally be placed in the appropriate place in the `~susi/output` directory.

A typical project directory is shown in Fig. 3, assuming that the SCCS version control system is being used.

3 General layout and style for coding

Although the detailed layout of a program is of course up to you a number of general rules apply to standardise 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.

- 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.
- Lines should not wrap beyond 80 characters. This can cause some problems especially with long function calls or algebraic statements. Consider the following example:

[3]These files need to be located in a directory on a user's path in order to run them. By tradition `/usr/local/bin` is always on the path, but putting SUSI specific files here creates problems for the system administrator. The setting of path variables at SUSI has been a problem; any difficulties should be reported to the system administrator. **Note to WJT: makefiles need to be modified!!**

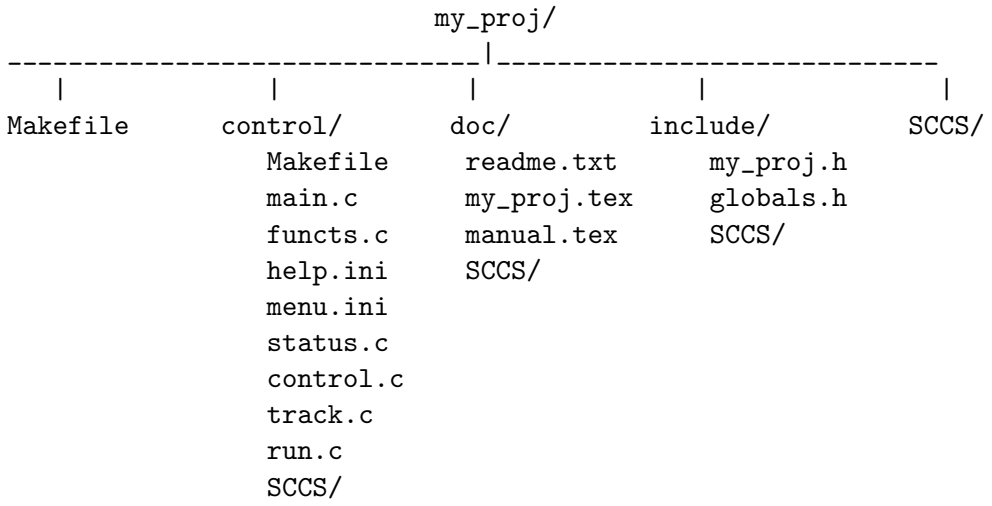

Figure 3: Structure of a typical SUSI project directory.

```
printf("The results are rather long and are as follows %d %d %d",argument1,ar
gument2,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 appear at the end of lines, not at the beginning. An analogous system can be used for long algebraic statements. Place operators at the end of lines and not at the beginning. Use indentation and brackets for clarity.

One reason for long lines is that string constants cannot be broken across lines. Code which uses a lot `printf` statements can become hard to read and harder to maintain if they contain a lot of string constants. Consider creating an an array of string pointers:

```
static char *data_header[] =
      {
      "                                                    \n",
      "----------------------------------------------------\n",
```

```
      "Result 1        Result 2      Result 3   Result 4  \n"
    };
```

This allows you to collect all the long strings together in one place, and makes it easier to visualise the final result.

If you have a long algebraic statement consider breaking it up into two or more statements. For example, use separate statements to calculate the numerator and denominator of a complicated fraction.

- Two forms are acceptable for function *prototypes:*

```
char *my_func( char * string1, char *string2, int * count);
```

or

```
char *my_func( char * , char *, int *);
```

The first form is to be preferred. The variable names are ignored by the compiler, but they make the prototype easier to read. Since the first form is nearly identical to the function *definition*[4] it is very easy to generate prototypes. There are utilities available which will do this automatically.

- 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 (modern compilers usually catch this kind of error).

- 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 */
```

[4] A declaration is followed immediately by ";". A definition is followed immediately by a compound statement "{ ... }" containing body of the function.

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. Note that each "case" calls a function. This improves the readability of the code and makes debugging much easier.

- Curly Braces should always be placed on lines by themselves and match the column of the statement they refer to, for example:[5]

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

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

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();
```

Code within a pair of braces should be indented by one tab, as shown in the examples. Braces can of course be nested:

[5]There are other conventions for positioning braces and parentheses which are quite commonly used. It is the authors' opinion that the recommended convention produces the most readable code. It does, however, result in a lot of white space which is why it is not commonly used in published books and manuals.

```
funny_sum = 0.0;
for ( i = 0 ; i < i_max; i++)
{
    if ( x[i] < 0.0 )
    {
        y = sqrt ( - x[i] );
        funny_sum -= y;
    }
    else
    {
        y = sqrt ( x[i] );
        funny_sum += y;
    }
}
```

- 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.
- Do not use the `goto` operator. This is a relic of the "spaghetti code" era of programming. The only sanctioned use of `goto` is as an emergency "escape hatch" from some deeply nested structure. Be aware, however, that if you are writing code for *control* processes you will need to handle exceptions very carefully.
- Avoid "clever" C constructs. C has a reputation for being difficult to read; this goes back to the early days of C programming when computers were slow and compilers were not very good at optimising code. If one or two machine cycles could be saved by rewriting the code one could sometimes get a significant improvement in performance, particularly within loop statements. With modern computers this is not necessary, and good optimising compilers will probably produce more efficient code than hand programming in assembler. In many cases, by writing "clever" code you may defeat the optimising process and actually produce less efficient code.[6] In particular:
  - Do not use the comma (`,`) operator to jam several unrelated operations together. One of the few legitimate uses might be in the following code:

    ```
    for ( sum = 0.0, i = 0; i < i_max; i++)
           sum += a[i]*b[i];
    ```

  - Try to avoid multiple indirection where possible. A construct like `***what_am_i` is inviting trouble.
  - C has subscripts; use them! Do not write `a+i` in place of `a[i]`.

[6] "Low level" programming is an exception. It may be necessary, for example, to split an input string into tokens and parse them recursively. Typically this kind of programming is only found in library functions, device drivers, etc.

– Do not rely on C's precedence rules to unambiguously resolve a complicated expression. Statements like

```
result = alpha*beta/gamma*delta/epsilon;
```

are dangerous. Write instead:

```
result = (((alpha*beta)/gamma)*delta)/epsilon;
```

or even better, result = (alpha*beta*delta)/(gamma*epsilon);

– Assuming `x` is a double precision number, do not write

```
if (!x)
. . .
```

Instead, use

```
if (x != 0.0)
. . .
```

As a general rule, do not use logical operators on variables which are not booleans. In C there is no separate boolean data type. It is a common programming practice to define a boolean type:

```
#define TRUE  1
#define FALSE 0
typedef BOOL unsigned int;
.
.
.
BOOL test;
.
.
.
if ( !test )
   /* Do something if 'test' not true  */
```

Booleans should only take on the values `TRUE` and `FALSE`. There is no way of enforcing this in the C language; it is the programmer's responsibility.

- Bullet-proof your code! Functions can fail for many reasons. if you build in some kind of error handling, you can catch many otherwise obscure bugs. One useful trick is to write your functions so they return an error code. Following the standard UNIX convention, a return value of zero indicates successful completion:

```
if ( ( errno = my_func(data) ) != 0)
{
    /* error processing */
}
    /* OK */
```

By adopting this procedure at all levels of your code you can trap errors at a fairly low level and handle them.

Beware of ending a function with a loop construct or if clause. Always provide a default return statement which returns an error code, even if the function can "never" reach this point.[7]

Finally, you should always follow the principles of structured or "object-oriented" programming. If you adhere to the guidelines set out in this section your code will already be reasonably well-structured. Structured programming, however, has more to do with the *logical* layout of your code rather than its syntax. Basic principles of structured programming include:

- Plan ahead! Think about what you want your program to do before writing any code.
- Collect related data items into structures or arrays of structures.
- The "top" level of your code (e. g., in `main()`) should only contain control structures (loops, etc.) and function calls.
- Try to separate the logical structure from algorithmic details. Functions should call other functions to do any "dirty work" like string manipulation, bit fiddling, etc.
- If you find that you have more than three or four nested compound statements (and consequently you are running off the right edge of the screen) you are probably not using enough functions.[8]

Object-oriented programming has been criticised for producing bloated, inefficient and slow programs. These criticisms, however, apply more to C++ programming. While structured or object-oriented programming in C can lead to some extra overhead in most cases this is unimportant. On the other hand, it makes the code much easier to write, debug and maintain.

4 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.

[7] If the code is truly unreachable the compiler will issue an "unreachable code" warning. We are more concerned with code which is *logically* as opposed to *syntactically* unreachable.

[8] In the early days of computing there was significant overhead involved in calling a function. The time required to call a function could be an important factor, and too many nested function calls could result in a fatal stack overflow. These are no longer considerations.

- 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.
- Precede variable definitions or declarations with a comment describing the use of the variable(s).
- 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 of the statement:

```
for ( ; ; ) /* main loop starts here */
{
     .
     .
     .
}  /* end of main loop */
```

- Multiline comments should be written as follows:

```
/*
 * 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.
 */
```

- The `gcc` compiler recognizes the Fortran and C++ style of comment

```
// This is a new style comment
```

  but this may produce warnings in a pure C compilation. Do not use.

- Add comments to individual lines if it clarifies the code:

```
for(sum = 0.0, i = 0; i < i_max; i++)
      sum += a[i]*b[i];
sum /= SCALE;  /* sum must be scaled by expt'lly determined SCALE factor */
. . .
```

- If the code implements a complicated or unusual algorithm, give suitable references:

```
/*
 *  Here we calculate the X-factor.
 *  The formula is taken from Blogg's
 *  PhD thesis (p. 345, 1997).
 */
...
```

This will save people (including yourself two years later) from wasting time trying to figure out what the hell the code is supposed to do.

The standardisation of file and function headers is important. The SUSI manual page generation utility is based on recognising the standard header layout. With a standard format it is relatively easy to change header information globally using **sed** scripts (changed telephone numbers, email addresses, etc.).

5 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:

- The first 8 characters of a name should be unique although don't shy away from long names if you think it would make your code clearer. Most compilers will recognise 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 */
```

- Capitalise all defines except function macros.
- Do not start/end defines with underscores (`_LIKE_THIS_`) except for "hidden" defines inside of header files. *Never* use double underscores; by convention this notation is reserved for constants predefined by your compiler. Examples include `__STDC__` which indicates ANSI C code, and `__TIME__` which some preprocessors expand to the time of compilation.
- Capitalise all type definitions.
- Since all mathematics in C is done in double precision always use doubles instead of floats unless you are concerned about memory space restrictions. Note that this will require the use of

  `"%lf"`

  in format statements. If you use "Numerical Recipes" be aware that their C code has been adapted from Fortran. In Fortran the default is single rather than double precision. This caused many obscure errors with the `cc` compiler, since strong function prototyping was not enforced. If you must use NR functions, first globally change `float` to `double` everywhere. If you use the `nrutils` module, you may also need to change things like `fmatrix` to `dmatrix`.
- Unless otherwise specified, physical quantities are to be expressed in SI units with angles in radians. Indicate units with comments!

- It is better to use a one dimensional array of pointers rather than a two dimensional array. The "Numerical Recipes" functions `dmatrix`, etc., can be used to dynamically allocate pointer arrays.
- 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 */
```

- As a general rule, do not use Hungarian notation.[9] In particular, never use it to indicate unsigned, short, or long data types as these often need to be changed during the course of program development. This is especially true for device drivers which need to interact with external hardware.

  On the other hand, it is useful to indicate the data type as part of the name. For example, `in_fp`, `out_fp`, `in_str` and `out_str` are perfectly acceptable names for file pointers and strings in a function that gets data from an input file, processes it, and outputs it to a second file. Global variables are another instance where Hungarian notation may be useful.

5.1 Global variables

Along with the `goto` statement, the use of global variables is frequently regarded as a mortal sin in programming. In control systems, however, globals are often essential, but should be used with care. The following guidelines should be adhered to:

- Consider using a distinctive notation for globals. Either capitalise the first letter (`File_name`) or use Hungarian notation (`g_file_name`).
- As a general rule, a global variable should be assigned a value in only one place. Many of the problems with global variables arise from multiple assignments.
- When *defining* a global variable, assign it a default value. The default should either be the "normal" value for the global, or a value which will be interpreted as "value not set" by your program.
- Globals must be *defined* in one location only. It is recommended that the global definitions be placed in a separate file (e. g., `globals.c`). Globals should be declared in a header file using the **extern** keyword.

[9] This is the convention that the first one or two letters of a variable or function name indicate its type. Fortran used Hungarian notation (variables beginning with I, J, ... N were integer, all others were floats) but this was abandoned many years ago. It is the standard naming convention used by a certain well-known software house.

- Do not use globals as a lazy way to pass arguments to a function! If you have a large number of arguments, consider putting them into a structure and pass a pointer to the structure. As the name suggests, a global variable is known to the entire program and is typically used to store characteristics of the operating environment, etc.

As an example, you may wish to record critical data in order to assist debugging a complicated control program. You want to be able to turn logging on and off, but you also want to monitor what is going on in many different locations in your program. A global variable can be used to control this. The standard header file for your project would include the line:

```
#include <stdio.h>
extern int g_log_on;
extern FILE * g_log_fp;
```

The logging functions will be in a file called `log.c` which would look a bit like the following:

```
#include <stdio.h>
#include "myproj.h"

int g_log_on = 0;   /* Definition: logging default is OFF */
FILE *g_log_fp;     /* Definition */
...

int logging_on(void)
{
    /* Try to open log file */
    ...
    if (g_log_fp != NULL)
    {
        g_log_on = 1;
        return 0;
    }
    else return 1;
}

int logging_off(void)
{
    /* Close log file */
    ...
    g_log_on = 0;
    return 0;
}

void log_data(char *format, double data)
{
    if (g_log_on)
    {
```

```
        fprintf(g_log_fp, format, data);
        fprintf(g_log_fp, "/n");
    }
    return;
```

Note that the file pointer has also been made a global. If logging is off the `log_data` function does nothing.

6 Header files

Any moderately large project will have many functions, data structures, etc. As well, it may also use "manifest constants" — symbols defined by the `#define` preprocessor command. Explicitly including this kind of information in each project file makes maintenance extremely difficult. Typically header files are used for this purpose. By convention C header files take the suffix `.h`, and are included in a project file by the `#include` preprocessor statement. This statement is replaced by the contents of the header file. Each project file will include some or all of the project headers, as well as standard header files. Headers are normally included at the beginning of a file:

```
#include <stdlib.h>
#include <stdio.h>
#include "../include/myproj.h"
#include "../include/protos.h"
```

Standard header files are enclosed in angle brackets, custom headers are enclosed in quotes. For custom headers a full or relative path must be given if the headers are not in the working directory.

Do not put executable code or variable *definitions* in header files. Since headers may get included several times during a compilation, this almost certainly will generate errors.

6.1 Header style: Small is beautiful

With large projects, header files have a tendency to grow and become difficult to read. There is no reason why large headers should be split into several smaller files:

```
#include "../include/my_protos.h"
#include "../include/my_defs.h"
#include "../include/my_macros.h"
#include "../include/my_typedefs.h"
#include "../include/my_err_codes.h"
```

6.2 Bullet-proofing a header file

Poorly constructed header files can generate obscure errors. If you write a particularly useful function, someone may try to use in a C++ program; this can lead to disaster! By following a few simple guidelines, these problems can be avoided.

1. Avoid multiple inclusions. In most cases, multiple inclusions of function prototypes, `typedef`s, etc., are harmless and at worst will generate a warning message. Sometimes, however, a second

inclusion of a header may produce an error. The reason is usually obscure,[10] and rather than waste time tracking down the problem it is easier to guarantee a single inclusion during a make. This can be done by "wrapping" your header file code with a conditional:

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

/* header code goes here ... */

...

/* end of header code */
#endif /* _MY_HEADER_H_ */
/* end of file */
```

The first time the header is `#include`d, the constant `_MY_HEADER_H_` will be defined. The next time it is included, the `#ifndef` test will be false and a second inclusion will not occur.

2. As a general rule, one shouldn't `#include` files within another header file. Apart from the possibility of errors, this hides dependencies from the programmer and can cause run-time problems.

3. An exception to rule (2): if you have a standard set of files which always get included, you might want to make a "package" file consisting of nothing but `#include` statements.

4. Although C++ is not included in this standard, this may change in the future. Also, others may wish to include your brilliant code in their C++ applications! Whilst C is a strict subset of C++, the two languages have different procedures for pushing and popping the stack. If a C++ program uses an object file compiled with the C compiler, run-time errors will almost certainly occur.

   Programmers moving to C++ will naturally want to recycle their "legacy" C programs with a minimum of rewriting. C++ recognises the following odd syntax: syntax: `extern "C"` *statement*
   where *statement* is a standard C compound statement. All declarations within the statement are assumed to refer to C-compiled code, and the correct linkage will be established. By "wrapping" your header files with the text below both C and C++ compilers will be happy:

   ```
   #ifndef MY_HEADER_H
   #define MY_HEADER_H

   #ifdef __cplusplus
   extern "C"
   {
   #endif
   ```

[10] A common error is to *define* a global variable in a header file. Headers can declare globals, but the definition occur only once.

```
/* header code including function declarations ... */

...

/* end of header code */

#ifdef __cplusplus
}
#endif

#endif /* end of MY_HEADER_H clause */
/* end of file */
```

The macro `__cplusplus` is defined if and only if the C++ compiler is being used.

7 Libraries

A *library* is a collection of frequently used, pre-compiled functions which are "linked" to your application when the final executable is built. If there are any "unresolved references" – i. e. functions, variable names and the like which are not defined in your source code – the compiler/linker searches for them in the standard libraries. You can specify additional libraries via compiler options. The standard math library, for example, is normally *not* searched and you need to add `-lm` as a compiler/linker option.

8 Documentation

All programs should be fully documented. Documentation includes:

- Programmer's guide. The programmers guide will normally take the form of a UNIX manual page. If the guidelines for commenting are followed, manual pages can be generated automatically from the source files.
- User manual. Detailed instructions for the user. User manuals should be written using LaTeX.
- Technical notes. SUSI control systems often interact with custom-built hardware units. Details of how the hardware is interfaced, special data handling issues, etc., should be covered in technical notes.

9 Pitfalls

In the development of the SUSI system some programming hazards occurred frequently. A few of these are given here.

9.1 Function prototyping

This should no longer be a problem, as `gcc` enforces strict function prototyping. At the very least you should get a warning message ("Possible redeclaration of function ... "). You should treat such warnings very seriously and track down the problem. With older, Kernigan & Richie C, it was possible to pass variables of type double to functions expecting a float (particularly a problem with functions adapted from "Numerical Recipes"), unsigned integers instead of signed integers, etc. This kind of coding error can cause all sorts of problems, frequently intermittent. In ANSI C strict function prototyping was included specifically to prevent these kind of problems.

9.2 The dreaded `scanf`

The input utility `scanf` (and its relations `fscanf`, `sscanf`, etc.) is a notorious source of programming errors. If you want to use `scanf` to store a value in the variable `x`, you must use *a pointer to* `x`:

```
scanf("%d", &x);
```

will attempt to get an integer from the standard input and store it in `x`. The common mistake

```
scanf("%d", x);
```

will give the wrong result and may have disastrous side-effects! Normally the compiler would report an error ("argument 2 is bad data type - expecting pointer" or something similar), but because `scanf` takes a variable number of arguments the compiler may not always catch this error.

Another common error is that the data type specification in the format string is incorrect. This can also escape the compiler's notice, and can again produce quite strange behaviour. A similar problem can occur with `printf`, but in this case it usually prints obvious garbage, and the error is quickly found.

9.2.1 More subtle problems

The `scanf` formatting options cover most possibilities, but sometimes they are not adequate, or simply don't work. A problem that took some months to find occured in an apparently simple algorithm which read in hexagesimal (base sixty) numbers and converted them to radians. It is standard astronomical practice to write such numbers with leading zeros: $02^{\mathrm{h}}\ 35^{\mathrm{m}}\ 08^{\mathrm{s}}$, for example. At least some versions of `scanf` assume that any number starting with a zero is octal. In the example, the "02" would scan correctly, but "08" is not a legal octal number and the program would crash.

A more common problem occurs when data is incorrectly typed. In this case `scanf` may get totally confused.

The safest thing is to always copy the next line of data (whether it is from a file or standard input) to a character buffer. The buffer can be read using the `sscanf` function, and you can use its return value to determine whether or not all the data has been read. An even safer method (which is used extensively in the SUSI control system) is to break the buffer contents into "tokens" using `strtok`. Each token can then be examined separately. Before converting the token to the appropriate internal representation we check it to ensure that it is indeed the correct type. If not, the user can be prompted to re-enter the data or, in the case of file input, a warning message can be printed.

The solution to these problems requires low-level C coding and can be quite tedious. There are, however, a number of safe utilities in `susistd` which can be used for this purpose. Functions like `angle2string` and `string2time` handle hexagesimal conversions properly. Functions like `getline` can be used to read the next data line from a file. This function, for example, automatically strips out comment lines.

10 Conclusion

In order to maintain standards within the department 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 WJT for the Data Dictionary, AJB for user documentation, RAM for coding style and commenting or TtB for Unix online Manual entries.

A Makefile Template

This is a template *makefile* for use in a module directory. Comments start with a "#" sign. The notation `#nn#` indicates a footnote to the script. If you use this example as the basis of a project makefile you should delete all instances of `>nn<`.

```
#
# $Id$                                  #1#
#
# Template makefile for compiling and linking a module to form
# an executable file called "prog" from sources in files
# main.c (containing the function "main()"), f1.c, f2.c, etc.
# using a project library "libp.a", a susi wide library
# "libsusi.a" and the standard C maths library, and a header
# file "header.h" in the project include file.
#

#
# variable definitions:
#

SHELL = /bin/sh                         #2#

# List source files here:
SOURCES = main.c f1.c f2.c ...          #3#
OBJECTS= $(SOURCES:.c=.o)

# The name of the executable we want to make:
PROGRAM = prog

# Things we need to know about:

# Directories, libraries, etc., specific to this project.
# By tradition these are called USERxxxx and it is usually
# safe to use relative paths here.
USERLIBS = ../lib/libp.a
USERINCL = ../include/header.h


# System libraries and locations. Note that directories
# and files are separate, and full path names are used for
# non-project objects.

LIBDIR = /usr/local/susi/lib  # where to find special libraries
BINDIR = /usr/local/susi/bin  # where to install binaries
```

```
SYSLIBS = -L /usr/local/susi/lib -lm -lsusi

# "make all" or "make debug" makes "prog"
all debug: $(PROGRAM)


debug := CFLAGS= -g
# when command is "make debug" run cc with "-g" flag for later
# use of "prog" with dbx or dbxtool

# here comes the important bit
$(PROGRAM): $(OBJECTS) $(USERLIBS) $(USERINCL)
        $(LINK.c) -o $@ $(OBJECTS) $(USERLIBS) $(SYSLIBS)
# compile and link (ie make an executable) in the ordinary way
# using "make all", "make prog" or simply "make".  $@ is the
# current target
$(USERLIBS): FORCE
        cd $(@D); $(MAKE) $(@F)
# run a make in the project library directory if necessary.
# you need a "makefile" in the ../lib directory
$(USERINCL) : FORCE
        cd $(@D); $(MAKE) $(@F)
# run a make in the project include directory if necessary
# you need a "makefile" in the ../include directory
FORCE:
# FORCE is a dummy target to make sure the last 2 makes are
# run

# this bit is really an option over the above bit, remove
# either if you like
$(LIBRARY): $(LIBRARY)($(OBJECTS))
        ar rvc $@ $?
        ranlib $@
$(LIBRARY)(%.o): %.o
# make a library "lib1.a" from all the objects that go to make
# "prog"

# only one install allowed! remove one or other
install: $(SOURCES)
        av68k $(SOURCES)
# this is a guess, to be refined later
install: $(PROGRAM)
        cp $(PROGRAM) $(DESTDIR)
# this install is for progs to run on the sun

clean:
```

```
        rm -f $(PROGRAM) $(OBJECTS)
# remove executable file and all .o files
```

Notes:

1. The string `$Id$` is a *version control identifier*. It expands into a text string. The details vary depending on the version control system being used, but typically it will expand to the filename, current release level, and time and date information. Makefiles should *always* be placed under version control since they are as valuable as your source files!

2. The `SHELL` variable specifies the command shell to be used. On POSIX systems it should be set to `/bin/sh`. This is the vanilla-flavoured C shell. This is essentially a paranoia setting, but it is useful to remind us that we should use `sh` syntax.

3. The variable `$(SOURCES)` will be a list of all your source files. The next line uses a "pattern rule" to define a list of object files. This list is obtained from the source list by replacing `.c` by `.o` in the file names.

Next is a template *makefile* for maintaining an include directory.

```
# Template makefile for an include directory containing .h
# files
#
# AJB Sept 89

FILES.h= header1.h header2.h ...
# fill in your own header file names

all: $(FILES.h)

clean:
        rm -f $(FILES.h)
```

Now a template *makefile* for an archive library.

```
# Template makefile for a lib directory containing archive
# .a libraries from C source in files f1.c, f2.c, etc
#
# AJB Sept 89

SOURCES= f1.c f2.c ...
LIBRARY= lib1.a
# you fill in your own information on the two lines above

OBJECTS= $(SOURCES:.c=.o)
# ie main.o, f1.o, etc

.KEEP_STATE
```

```
# this makes sure that .o files are updated even if you only
# change the compilation command line you are using (eg using
# debugging).  It also ensures that hidden dependencies (ie .h
# files not given in the dependencies list, but "#include"d in
# your .c file) are up to date
.PRECIOUS: $(LIBRARY)
# make sure that an interrupted make doesn't corrupt the library

all debug: $(LIBRARY)
# "make all" or "make debug" makes "lib1.a"

debug := CFLAGS= -g
# when command is "make debug" run cc with "-g" flag for later
# use with dbx or dbxtool

# here comes the important bit
$(LIBRARY): $(LIBRARY)($(OBJECTS))
        ar rvc $@ $?
        ranlib $@
        rm -f $?
$(LIBRARY)(%.o): %.o
# make a library "lib1.a" from all the objects then remove
# .o files

clean:
        rm -f $(LIBRARY) $(OBJECTS)
# remove library file and all .o files
```

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

```
# Template makefile for project management.  Runs "make"
# recursively in directories "module[1-2]", "include",
# and "lib"
#
# AJB Sept 89

TARGETS= all debug clean install
SUBDIRS= module1 module2 include lib
# fill in your bits for SUBDIRS above
# TARGETS should match "makefile" "targets" in all the SUBDIRS

.KEEP_STATE

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

```
$(SUBDIRS): FORCE
        cd $@ $(MAKE) $(TARGET)
FORCE:
# thus "make all" here runs "make all" in all the SUBDIR
# subdirectories.  Obviously this can be recursive, so
# running copies of itself in lower and lower subdirectories
```

B Standard headers

Each source code file (`.c` and `.h`) file should begin with a "header" like the following. Note the CVS keyword at the beginning. Text in UPPER CASE should be replaced by your own.

```
/*************************************************************/
/*  @(#) $Id$                                                */
/*                                                           */
/* BRIEF DESCRIPTION OF THIS FILE                            */
/*                                                           */
/*************************************************************/
/*                                                           */
/* PROJECT NAME                                              */
/*                                                           */
/* Sydney University Stellar Interferometer.                 */
/*                                                           */
/* Chatterton Astronomy Department                           */
/* School of Physics A28                                     */
/* University of Sydney, N.S.W. 2006, Australia.             */
/*                                                           */
/* Telephone: +61 2 9351 2544                                */
/* Facsimile: +61 2 9351 7726                                */
/*                                                           */
/* (C) This source code and its associated executable        */
/* program(s) are copyright.                                 */
/*                                                           */
/*************************************************************/
/*                                                           */
/* Author : YOUR NAME                                        */
/* Email  : YOUR_NAME@physics.usyd.edu.au                    */
/* Date   : DATE OF FILE CREATION                            */
/*                                                           */
/*************************************************************/
```

C Function Header Template

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

```
/*****************************************************************/
/* function_name()                                               */
/*                                                               */
/* description                                                   */
/*                                                               */
/*****************************************************************/
```

D Example Source File

```
/*****************************************************************/
/*  @(#) $Id$                                                    */
/*                                                               */
/*  funcs.c - this contains some useful functions needed by      */
/*            our project.                                       */
/*                                                               */
/*****************************************************************/
/*                                                               */
/* The SUSI Y2K monitor                                          */
/*                                                               */
/* SUSI                                                          */
/* Sydney University Stellar Interferometer                      */
/*                                                               */
/* Chatterton Astronomy Department                               */
/* School of Physics A28                                         */
/* University of Sydney, N.S.W. 2006, Australia.                 */
/*                                                               */
/* Telephone: +61 2 9351 2544                                    */
/* Facsimile: +61 2 9351 7726                                    */
/*                                                               */
/* (C) This source code and its associated executable            */
/* program(s) are copyright.                                     */
/*                                                               */
/*****************************************************************/
/*                                                               */
/* Author : W. J. Tango                                          */
/* Email  : tango@physics.usyd.edu.au                            */
/* Date   : 31 Dec 1999                                          */
/*                                                               */
/*****************************************************************/

/* include files */

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

```
#include "ui.h"

/* function prototypes not already included */

/* macros */

/* local (static) variable definitions */

/*****************************************************************/
/* initialise_menu_structure()                                   */
/*                                                               */
/* This is the controlling function of the menu structure        */
/* initialisation. 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 namesy 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. */
/*                                                               */
/*****************************************************************/

/*
 * Inputs:
 *    stream  - a pointer to a file stream
 *    n       - getline will read a maximum of n - 1 characters
 * Outputs:
 *    string  - a pointer to a buffer which holds the
 *              next line read
 * Returns:
 *    a pointer to the string, or EOF if the end of
 *    file has been reached.
 */

char *getline(char *string, int n, FILE *stream)

{
        /* 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() */
```