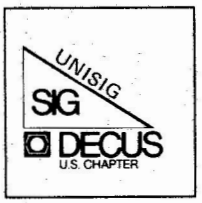
volume 3 number 1 april 1984

This is your first Subscription Service Issue

TOOLKIT

the UNISIG newsletter



The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	PDT
DECnet	Digital Logo	RSTS
DECsystem-10	EduSystem	RSX
DECSYSTEM-20	IAS	UNIBUS
DECUS	MASSBUS	VAX
DECwriter	PDP	VMS
		VT

UNIX is a trademark of Bell Laboratories.

TOOLKIT is the newsletter of UNISIG, the DECUS UNIX-oriented special interest group. **TOOLKIT** articles range widely over UNIX topics: DEC's UNIX/v7m and ULTRIX, BSD 2.9 and BSD 4.2, device drivers, operating systems comparisons, varieties of user interfaces, performance analysis, multi-operating system applications, UNIX standards, local area networking, long haul networking, uucp, USENET, UNIX communication with DEC and IBM operating systems, UNIX hints and kinks, C, f77, ratfor, lex, yacc, nroff/troff, developments in the Software Tools movement, site surveys, DECUS Symposia, and more.

The editors are:

Don Crabb
University of Chicago
Computation Center
5737 South University Avenue
Chicago, IL 60637

312-962-7173

William Toth
Harvard-Smithsonian Center for Astrophysics
60 Garden Street, P-353
Cambridge, MA 02138

617-495-7181
{ihnp4|allegra|genrad|amd70|ima}!wjh12!hscfa!toth

We invite readers to submit articles, book reviews, software reviews, and thoughtful letters for publication in **TOOLKIT**. Text should be submitted in machine-readable form, if possible. We can read RX02 floppy disks and 9-track magtape (800 bpi or 1600 bpi); computer-to-computer text transfers via telephone can be accomplished by using **uucp** or by arranging ordinary dialups with either of the editors.

In this issue...

we reproduce a classic of the DECUS literature, Martin Minow's **A C Style Sheet** [page 3]. Minow probably is best known to most of us as the man who wrote the DECUS C compiler. His notes on style are filled with programming wisdom.

From the St. Louis Spring 1983 DECUS Symposium we have three articles. **UNIX Hints and Kinks** [page 35] summarizes a general panel discussion led by Armando Stettner. David Moore's and John Livingston's **Writing C Code for VAX/VMS and UNIX Systems** [page 37] details portability issues of special interest to C programmers who work in both UNIX and VAX/VMS environments. **Networking with UNIX in Tokyo** [page 39] by Ishida, Adachi, Kimura, and Takano, records how technical problems (some of them amusing) were handled in a rather difficult project.

This issue's back matter includes a transcript of the **operating principles** of UNISIG, a listing of the UNISIG **steering committee membership**, and, of course, a subscription form.

In the coming issues we will feature an article on uucp and USENET, an article on UNIX in local area networks, and a review of the new book, The UNIX Programming Environment, by Brian Kernighan and Rob Pike.

A C Style Sheet

Martin Minow
Digital Equipment Corp.
146 Main St. MLO 3-3/U8
Maynard MA 01754

1.0 Introduction

This document presents a common set of coding standards, as well as a series of hints to aid in producing maintainable C software. It is abstracted from a number of sources:

Brian Kernighan and Dennis Ritchie. The C Programming Language.

Indian Hill C Style and Coding Standards (Bell Telephone Labs unpublished Technical Memorandum 78-5221, March 29, 1978). (with annotations by Henry Spencer (utzoo!henry), University of Toronto.)

Joe Kalish. "Ingres Coding Conventions for C/Unix Programming" (INGVAX.kalish @ Berkeley)

Dan Franklin. BBN Programming Standards for C. (Dan @ BBN-UNIX)

Andrew Shore, et al. Network Graphics C Style Sheet. Stanford University, (from CSL.LANTZ @ SU-SCORE).

Ray Van Tassle. C Language Programming Standards for Motorola DCS. Motorola, Inc. 1301 E. Algonquin Road, Room 4135, Schaumburg, IL 60196.

It is also based heavily on my own experience in developing a number of large, transportable applications in C, using Decus C, Vax-11 C, and several varieties of Unix C. (Unix is a trademark of Bell Laboratories).

As could be expected, the suggested style does not totally agree with any of the referenced documents.

The reason a you should maintain a consistent coding style is that good programs will evolve. When writing a new program, you will often take routines and data structures from old -- working -- software. This is much easier to do if the old software is

understandable. Unreadable software is unusable, no matter how well it works.

The single most important thing about a typographical style is sticking to it consistently. There are many good styles, but the differences among them are totally drowned out by the difficulty of reading a program with pieces in different styles. If you modify a program, stick to its original style. If you must change things -- you really cannot live with the old style -- change the whole program, or at least all the parts logically related to what you are changing.

In recommending against automatic beautifiers (prettyprinters), the Indian Hill standards committee noted:

> "First, the main person who benefits from good program style is the programmer himself. This is especially true in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers ... are not available when the need for attention to white space and indentation is greatest. It is also felt that programmers can do a better job of making clear the complete visual layout of a function or a file with the normal attention to detail of a careful programmer."

These comments are relevant to any rigid application of a programming or typographical style. There will always be cases where the automatic rule is unsatisfactory and you, as the person responsible, must be able to understand that your primary goal is to achieve clarity and understandability.

2.0 File Organization

A file consists of several sections separated by blank lines or a form-feed (<FF>). If you use a form-feed, it should be the only character on the line.

In general, source files should not be much longer than 1000 lines. Larger files are often difficult to edit and -- if too large -- cannot be processed by the diff (differences) program. 1000 lines translates to about 12-15 pages of text. Source lines should not be longer than 78 characters long.

A source file should be organized as follows:

1. A prologue comment gives the file name and one or two sentences telling what is in the file. This is followed, if necessary, by copyright and license "boilerplate." The prologue tells the reader the purpose of the text of the file, whether it contains functions, data definitions, tables, or support code. It should not generally be a list of function names.

In some programs, C source files may be created by program generators. For example, a dictionary may be compiled into a keyword vector (one file) and a definition vector (one file). In this case, the program generating the files should write the date of generation (as a comment) into the C file source. If the generated program file may require editing, consider including the source of the generated information, either as comments or as text bracketed by #ifdef's, as an aid to the debugger.

2. Usage and operating instructions follow next. Decus C programs should use the format accepted by the "getrno" utility program. This allows the program source file to contain the source of its documentation and lessens the burden of keeping the documentation in synchronization with the program itself.

   Program compilation instructions are included in this section. Decus C programs should use the build utility to maintain compilation instructions. Unix programs should include a source copy of the program's makefile. Again, this centralizes everything relevant to maintenance of a program in one place.

3. Header files are specified (using the #include preprocessor directive). The suggested header file order is

```
#include <stdio.h>
#include <other system headers>
#include "user header files"
```

   Note that header files should be given the ".h" filetype, while all C source files should be given the ".c" filetype.

   In allocating header files for large packages of programs, you should avoid absolute pathnames for header files. Use the <name> construction for system files, relative directories for Unix and VMS systems, and externally defined logical devices for RSTS/E and RT11. If the subsystem is reasonably small, put all source files in one directory.

4. Typedefs, #defines and structure definitions that apply to the file as a whole are next. Structures should always be defined as "typedef struct { ... } NAME;". Definitions should be grouped functionally.

5. Global data definitions are next. The suggested order is

   1. Global variables defined in this file.

   2. Static (file global) variables.

   3. External variables and functions.

   If a program is large enough to require multiple source

files, all global data should be defined by a "data.c" file which only contains data definitions, while all other source files contain extern references. Alternatively, the file containing the main program, documentation, and build instructions should contain data definitions.

A very large program would contain global references in an "extern.h" include file in addition to a "data.c" definition file.

6. Functions come last. If the file is a main program, the main() function is first.

2.1 Header Files

Header files are included in other files during compilation. Some, such as "stdio.h" are defined system-wide, and must be included by any C programs that use the standard I/O library. Others are used within a single program or suite of programs.

Header files should be functionally organized. Declarations for separate subsystems should be in separate header files.

Header files should not be nested. This is not permitted by Decus C and some objects, such as typedefs and initialized data definitions must not be seen twice by the compiler in one compilation.

Header files should contain all #defines, typedefs, and extern declarations necessary for a given program and shared among two or more of its files.

Header files should not declare variables. This is frequently a symptom of poor partitioning of code between files. One header organization that has worked well for several medium-sized projects is:

- Definitions, including common data structures go in one header file. Things defined in this file are common to the entire package.
- All external (global) data is defined in a second header file.
- This file is paralleled by a data.c file containing all global data allocations.
- Definitions required by bounded subsystems go in separate files.

3.0 Declarations and Definitions

The use of the #define preprocessor command is especially recommended. In general, numerical constants and array boundaries should never be coded directly. They should be assigned a meaningful name and assigned their permanent value by the #define. This will make it much easier to administer large and evolving programs as the constant value can be changed uniformly by changing the #define and recompiling.

The enumeration data type (not in Decus C) offers an improved manner of managing constant definitions as additional type checking is then available.

In general, all constant values which are not strictly numeric should be specified by #defines. Exceptions to this rule are the values 0 and 1 when used as the lower boundary of an array, relative indices (if p is a pointer to an array element, p[1] is the next element, while p[-1] is the previous element). and strictly numeric quantities. #defines may even be useful in the latter situation as well:

```
#define SPEED_LIMIT 55
```

Note that defined quantities should always be in upper-case.

Directly-coded numerical constants must have a comment explaining the derivation of the value.

It is generally poor practice to use #defines to modify C syntax. for example, the following definitions are not recommended:

```
#define reg         register
#define begin       {
#define end         }
```

In certain circumstances, however, this may be necessary for proper compilation or fastest possible execution:

```
#define DIV_2       >> 1
#define DIV_4       >> 2
```

Replacing divides by right-shifts cannot be done by the compiler as it would yield incorrect results if the divisor were negative. If the programmer knows that the divisor must be positive (which fact being duly documented), this optimization becomes possible.

Also, the programmer may need to conceal non-portable quirks by means of centrally-placed definitions:

```
#ifdef decus
#define UNSIGNED_LONG       long
#else
```

```
#define UNSIGNED_LONG        unsigned long
#endif
```

As will be noted under portability, most C compilers predefine a small number of variables that may be used to conditionally compile machine or operating-system specific code. This allows one program run on multiple systems without hand-editing.

It is highly recommended that you use the following definitions freely and consistently:

```
#define     NULL    0
#define     EOS     '\0'
#define     FALSE   0
#define     TRUE    1
```

NULL is defined by <stdio.h> and generally need not be explicitly specified by your program. EOS marks the end of a C string, while FALSE and TRUE are used for Boolean testing. You will probably get in the habit of only referring to FALSE in your if statements:

```
if (test != FALSE) {
```

This generates the best possible code. TRUE is usually used to return a "success" value from a function. Don't use both TRUE and YES in the same program to mean the same thing.

If a structure contains a data element that can take on one of several values, it is a good practice to put the #define's for that element within the structure definition. For example, here is a fragment (slightly reorganized to fit on the documentation page) from a Vax-11 C header files that defines a VMS system structure:

```
/*
 * XABSUM -- Summary Extended Attribute Block
 */

struct XABSUM {
    char    xab$b_cod;
#define       XAB$C_SUM     22      /* type code    */
    char    xab$b_bln;
#define       XAB$C_SUMLEN  0x0C    /* block length */
#define       XAB$K_SUMLEN  0x0C
....
```

Note that the information that would be placed in each field is #define'd following that field. The definitions and structure fields follow standard VMS syntax conventions.

The empty initializer "{}" should never be used. Initialized structures should be fully delimited with braces. Constants used to initialize longs should be explicitly long.

In any file which is part of a larger context, all local information should be identified by use of the static keyword. Variables, in particular, should not be accessible outside the file unless there is an overriding need for global access. If these variables are shared by only one or two other files, you should name these files in a comment.

4.0 Comments

The importance of comments cannot be overemphasized. In any professional environment, many people will have to read your code, trying to understand what you have done. Sometimes, they wish to modify it to do other things; sometimes they need to modify it to do what you originally intended to do. "Try to make life easy for them and maybe they will be nice to you someday."

The purpose of a comment is to describe your intention. If properly written, the code itself will adequately tell what you actually did. There are two general types of comments:

Block comments are narratives describing the purpose of a portion of the program text. They are written in the following format:

```
/*
 * The comment text is written
 * here in complete sentences.
 */
```

The comment text should be at the same level of indentation as the source code it discusses. You should never write a comment that could be interpreted as a C statement (unless the comment is blocking out temporary debugging code). A block comment should always be included at the beginning of a major segment of the program.

Very short comments may appear on the same line as the code they describe. They should be tabbed over far enough to separate them from the statements. if more than one short comment appears in a block of code, they should all start at the same tab position:

```
while (!finish()) {          /* Main sequence:       */
    inquire();               /* Get user request     */
    process();               /* And carry it out     */
}                            /* As long as possible. */
```

Note that all single-line comments start at some specific column and end with the closing "*/" tabbed to column 72 on the line. Closing the comment at the right-hand margin makes it more readable than if the "*/" were next to the comment text itself:

```
while (!finish()) {             /* Main sequence: */
    inquire();                  /* Get user request */
    process();                  /* And carry it out */
}                               /* As long as possible. */
```

In general, you should use one-line comments to document variable definitions and block comments to describe the computation processes. The above comments should actually have been written as a block comment:

```
/*
 * Main sequence: get and process
 * all user requests.
 */
while (!finish()) {
    inquire();
    process();
}
```

5.0 Function Declarations

Each function should be defined beginning in column 1 (to simplify searches for the function's definition). If the function returns a value or is static, that should be alone on the proceeding line.

Each formal parameter should be declared, with a comment, on a separate line. If the function uses any external variables or functions (that do not return integers), these should be declared with other local variables. This is particularly beneficial to someone reading code written by another.

The format for the function declaration may be illustrated as follows:

```
char *
    savest(string)
    char        *string;        /* String to save       */
    /*
     * Savest saves its argument string in free storage,
     * returning a pointer to the allocated datum.
     * It returns NULL if the allocation fails.
     */
    {
        register char   *ptr;
        extern char     *malloc();

        if ((ptr = malloc(strlen(string) + 1)) != NULL)
            strcpy(ptr, string);
        return (ptr);
    }
```

Note that, in the example above, the function description followed the formal definition itself. Another acceptable style precedes the function by a block comment.

```
/*
 * match(string, pattern)
 *
 * If the pattern is an initial substring of string,
 * return a pointer to the first character of the
 * string s beyond those matching the pattern,
 * Otherwise, return NULL.  Thus:
 *     match("abcde","abc")
 * returns a pointer to the 'd' in the first string;
 *     match("abcde","bc")
 * returns NULL.
 */

char *
match(string, pattern)
register char       *string;        /* Source       */
register char       *pattern;       /* for match    */
{

    while (*string == *pattern && *string != EOS) {
        pattern++;
        string++;
    }
    return ((*pattern == EOS) ? string : NULL);
}
```

Note that, in this format, the block comment is separated from the function definition by a blank line.

6.0 Structure and Variable Declarations

Structures are one of the most important features of C. They enhance the logical organization of your code, offer consistent addressing, and will generally increase the efficiency and performance of your programs by a significant amount.

In general, if there are two or more "things" in your program that are addressed by the same index, they should be defined by a common structure. This gives you great freedom to allow the program to evolve (by adding another "thing" to the structure, for example), or to modify storage allocation (from pre-compiled to dynamic allocation).

For example, if your program processes symbols -- where each symbol has a name, type, flags, and an associated value, you shouldn't define separate vectors:

```
char *name[NSYMB];
int  type[NSYMB];
```

```
    int  flags[NSYMB];
    int  value[NSYMB];
```

but, rather,

```
    typedef struct symbol {
        char *sy_name;
        int  sy_type;
        int  sy_flags;
        int  sy_value;
    } SYMBOL;

    SYMBOL symboltable[NSYMB];
```

All structures should be defined by typedef's. Note, also, the use of a header ("sy_") to identify members of the SYMBOL structure.

The local variables used by a function should have names that do not duplicate global names.

7.0 Compound Statements

Compound statements carry out the calculations required by the C program. They are lists of statements enclosed in braces. They should be tabbed over one more than the tab position of the compound statement introducer itself. (Four space indentation is recommended, although it is certainly more convenient to use the hardware-provided eight position tab stops. If you change your mind in the middle of a program, you should have the courtesy to re-edit the rest of the file so it is consistent.)

The opening left brace should be at the end of the line beginning the compound statement and the closing right brace should be alone on a line, tabbed under the beginning of the compound statement. Note that the left brace beginning a function body is the only occurrence of a left brace which is alone on a line. This is the "Indian Hill" style, also present in Kernighan and Ritchie's book. (Other style sheets recommend placing the opening left brace alone on the line following the statement opener. Choose one style; be consistent. This subject will be discussed further in a subsequent section.)

The right brace before the while of a do-while statement is the only place where a closing right brace is not alone on a line:

```
    do {
        stuff();
    } while (cond != FALSE);
```

It is good practice always to provide braces, even when they are are not required by the language:

```
if (abc < def) {
    lesser();
}
else if (abc == def) {
    equal();
}
else {
    greater();
}
```

This prevents surprises when you add debugging statements.

Never, never, write nested conditionals or loops without braces:

```
for (dp = &values[0]; dp < top_value; dp++)
    if (dp->d_value == arg_value
     && (dp->d_flag & arg_flag) != 0)
        return (dp);
return (NULL);
```

While the above is correct C, it is unmaintainable. It should always be written as

```
for (dp = &values[0]; dp < top_value; dp++) {
    if (dp->d_value == arg_value
     && (dp->d_flag & arg_flag) != 0) {
        return (dp);
    }
}
return (NULL);
```

If the span of a block is large (more than about 40 lines) or there are several nested blocks, closing braces should be commented to indicate what part of the process they delimit:

```
for (sy = sytable; sy != NULL; sy = sy->sy_link) {
    if (sy->sy_flag == DEFINED) {
        ...
    }               /* if defined                    */
    else {
        ...
    }               /* if undefined                  */
}                   /* for all symbols               */
```

Each line should contain one and only one statement. The only exception to this is the "else if" construction as shown above. In a sequence of "if ... else if ..." statements, there should always be a terminating "else" even if it is merely a dummy statement. Note especially that an if statement and its associated conditionally executed statement appear on separate lines.

If a for or while statement has a dummy body, the ';' must go on the next line:

```
/*
 * Locate end of string
 */
for (charp = string; *charp != EOS; charp++)
    ;
```

There are few more insidious bugs than an extra ';' tacked on the end of a for loop statement. Everything will compile normally and the code might even work for some cases, but -- because of the invisibility of the ';' -- the bug will be very difficult to track down.

There should always be a blank between reserved words and their opening parentheses, e.g., "if (condition)" rather than "if(condition)". There should also be parentheses around the objects of sizeof and return.

If the conditional test in an if statement is so complex that it requires more than one line, break it at an && or ||, and line up the expressions so the tests line up as well:

```
if (a == b
 && b == c) {
    printf("a == c");
}
```

If the conditional test extends over one line, always enclose the conditionally-executed statement in braces.

The above is a special case of a more general recommendation that you break statements across lines at meaningful boundaries, and attempt to align the components to make the meaning clear. For example, the following sequence computes the length of an RMS logical record.

```
r->lrecl = r->rab.rab$w_rsz /* Record size from RAB */
 + ((hbyte != EOS) ? 1 : 0) /* If header byte       */
 + ((tbyte != EOS) ? 1 : 0) /* If trailer byte      */
 - offset                   /* For Fortran hacking  */
 + hnewline                 /* For VFC hacking      */
 + tnewline;                /* For VFC hacking      */
```

Switch statements offer a good alternative to multiple if...else sequences. Each case appears by itself on a line, tabbed under the switch itself. The break that terminates a case should be followed by a blank line. The "fall through" feature of C's switch statement should rarely, if ever, be used. If it is needed, it must be commented for further reference:

```
count = 0;
while ((c = getchar()) != EOF) {
    switch (c) {

    case '\n':      /* Newline,                    */
        lines++;    /* count lines                 */
```

```
            /*
             * Fall through to "end of word" case
             */
        case '\t':      /* Tabs, newlines, and blanks   */
        case ' ':       /* Form words.                  */
            words += count;
            count = 0;  /* Don't count multiple runs    */
            letters++;  /* But count all "white space"  */
            break;

        default:        /* All the rest form a word     */
            letters++;
            count = 1;
            break;
        }
    }
    words += count;     /* Fix count of last word       */
```

The above implements the central algorithm of a "word count" routine where a newline, blank, or tab terminates a word, but multiple blanks do not increase the number of words.

Note that the break following the last case is redundant, but should be provided to make the programmer's intent clear. In general, the default case should be last.

All switch statements should have a default case, which may merely be a "fatal error" exit.

8.0 Expressions and Operators

C is an expression language. This means -- in essence -- that the assignment statement "a = b" itself has a value which can be embedded in a larger context. This should be used very sparingly. For example,

```
while ((value = *pointer++) != 0) {
    process(value);
}
```

shows a standard C idiom which all programmers should recognize. It is essential, however, that you do not carry this to extremes by embedding multiple assignments (or other side-effects) in a statement.

Blanks should surround all binary operators except those which compose primaries, (".", "->"). No blanks should separate a unary operator (such as '-', '&', '[]', '!') from its operand. Sizeof and return are exceptions to this rule.

Some judgement is called for here as there are a few situations when complex expressions become clearer when inner constructions don't have spaces. For example,

```
x = (a*b) + (c*d);
```

Blanks should appear after commas in argument lists to help separate the arguments visually. On the other hand, macros with arguments and function calls should not have a blank between the name and the left parenthesis.

Side effects within expressions should be used sparingly. No more than one operator with a side-effect ("=", "op=", "++", "--") should appear within an expression. It is very easy to misunderstand the rules for C compilation and get side-effects compiled in the wrong order. For example,

```
func(*ptr++, *ptr++);
*ptr = *ptr++;
*ptr++ = *ptr;
```

Are not necessarily going to do what you expect.

The old versions of the assigned operators ("=+", etc.) must not be used. Always surround assigned operators by spaces. "x=*foo" is interpreted as "x = x * foo" (even if foo is a pointer).

The comma operator should be used exceedingly sparingly. One of the few appropriate places is in a for statement:

```
for (sum = 0, ptr = &array[0];
        ptr < &array[A_MAX];) {
    sum += *ptr++;
}
```

Since C has some unexpected operator precedence rules, all expressions involving mixed operators should be fully parenthesized. This is especially true when mask operators (&, |, and ^) are combined with shifts.

9.0 Naming Things

When a program must be used as part of a larger context, whether it be a subroutine library, or an independent program within an application package, the programmer's creativity in defining mnemonic names must be subservient to the needs of the group as a whole. The following suggestions for program and variable names in large projects were taken from a C programming standards manual by Ray Van Tassle:

9.1 Naming Rules

- Application program names should follow a standard format, such as:
  The 1st 2 or 3 characters = subsystem code
  The rest = unique meaningful identifier

- Names (variables, structs, unions, and procedures) are lower-case, unique in the first eight characters. (Some C systems require names to be unique in the first six characters.)

  External names must be unique in the first six characters.

  If the first letter in an external name is an underscore '_' it indicates a system-internal name, (such as a routine within a file-management I/O system). Applications programs should not use this; as it implies system-level programming. Trailing underscores should also be avoided.

  (Note that this may conflict with variables defined by your operating systems. For example, on RSX-11M, the operating system file management routines use -- in effect -- a leading underscore.

  Longer names and underscore should be used freely to improve readability and understandability.

  Upper-case and lower-case may <u>not</u> be mixed in a name.

  Names more than four characters long should differ by at least two characters:
  ```
  systst, sysstst /* are easily confused */
  ```
  Constants (things that are in a "#define") should be in all upper-case.

  All names must be unique, ignoring case. In other words, even though C knows that "this" is different from "THIS", do not do it. Also, do not have a variable and a typedef (or struct) with the same name, even though C also allows this. These make the program very difficult to follow.

9.2 Choosing names

Names should be meaningful. Abbreviations should also be meaningful, and should be chosen by some uniform, rational scheme.

- Each variable and name must have an invariant usage and meaning throughout the program.

- Names should not be re-defined in inner blocks. Nor should global names be redeclared within a function.
- Standard meaningful names for local (temporary) variables include:

i, j, k	indexes
c, ch	character
n, m	counters
p, q, a, b	pointers
s	strings

9.3 Names for structs, unions, and defines

Consider using "typedef" for struct's and union's. This helps both reader of the code and type checking programs such as LINT.

The name is composed of three parts:

prefix: 1, 2, or 3 characters, related to the sub-system, data-base, or struct.

body: the name of the entity.

suffix: indicates the "type" of the name:

A = array
T = type
S = size
N = number of elements (in an array)
L = limit (other than array elements)
D = "type has been defined" flag
TD = same as D

For a member of a struct, the prefix should be related to the body of the struct name.

These rules are more restrictive for structs and unions that are system-wide. Internal structs, for example, might have only a single character prefix.

A define with the "D" or "TD" suffix is required for all structs that are in an include file. For example:

```
#ifndef VH_UNR_D    /* declare this only once */
#define VH_UNR_D
struct  VH_UNR_T {
   int unr_unst;
   .....
}
#endif
```

Try to use an underscore between prefix, body, and suffix.

9.4 Pointers

Pointers should be declared and used as "pointer to a thing of type X". Do not, for example, use a variable which is declared as "pointer to int" to point to a char, even though the compiler and/or machine will let you do it.

9.5 Standard Defined-names

In writing a program composed of a number of files, a package-wide header file simplifies program maintenance. The following is extracted (with a few changes) from a header file defined by Ray Van Tassle:

```
GLOBAL    - The defining instance of a global variable
IMPORT    - Reference to an external defined in another module
LOCAL     - Static, defined either inside or outside function
VOID      - For functions returning nothing

TINY      - An 8-bit signed integer
UTINY     - An 8-bit unsigned integer

BOOL      - A Boolean quantity, tested for only zero/non-zero

TRUE      - Boolean true
FALSE     - Boolean false

NULL      - For comparison or assignment of pointers
EOS       - The end of string marker
EOF       - End-of-file
NOINIT    - For extern's that don't need initialization

TRYOUT    - Switch for compiling a main routine for testing.
DEBUG     - Switch for compiling debugging code.

 /*
  * The following define the largest number that
  * can be stored in a variable of a specified type.
  */
MAX_INT   - The largest positive number stored in an int
MIN_INT   - The largest negative number stored in an int
```

```
MAX_UINT - The largest positive number (unsigned int)
MAX_LNG  - The largest positive number (long int)
MIN_LNG  - The largest negative number (long int)
MAX_ULNG - The largest positive number (unsigned long)

 /*
  * The following should be used ONLY when unavoidable
  */
BITS8    - A variable that MUST be exactly 8 bits wide
BITS16   - A variable that MUST be exactly 16 bits wide
BITS32   - A variable that MUST be exactly 32 bits wide

 /*
  * The following should be used ONLY in "system-level" routines
  * that must be machine dependent.
  */
BITS_CHR - The number of bits in a char
BITS_INT - The number of bits in an integer
BITS_LNG - The number of bits in a long int
BITS_FLT - The number of bits in a float
BITS_DBL - The number of bits in a double-float
```

All binary bitwise operations must be done on one of the "BITS" data-types.

10.0 Portability

Portability means that a source file can be compiled and executed on different machines, operating systems, and/or compilers with either no source file changes or, at most, changes to system-specific header files. In writing portable software, the following should be understood:

- Most C compilers predefine symbols that may be used to isolate machine-dependent code. The following list may be helpful:
  - Decus C defines "pdp11", "decus", "rsx" (or "rt11").
  - Vax-11 C defines "vax", "vms", and "vax-11c"
  - Venix defines "pdp11", and "unix"
  - A compiler for the Dec-20 defines "TOPS20" and "PDP10"
- Some things are inherently non-portable. For example, a hardware device handler can, in general, not be transported between operating systems.

- Different machines have different word sizes. While the language standard guarantees that "long int" is at least as long as "int" and "short int" are never longer than "int", it does not guarantee any specific word length. Note also that pointers and integers are not necessarily the same size; nor are all pointers the same size.

- Word size and constants can interact in unpleasant ways. For example,

```
int x;
x &= 0177770;
```

  Clears the low-order 3 bits of an integer on a PDP-11. However, on a Vax, it will also clear the upper half-word. Instead, you should use:

```
x &= ~07;
```

  Which is portable.

- Beware of code that takes advantage of two's complement arithmetic. In particular, optimizations that replace division or multiplication with shifts should be avoided.

- Watch out for the PDP-11 signed character, which becomes unsigned on other machines. Also, do not presuppose any specific byte ordering within words.

- Do not default Boolean tests. Use

```
if (func() != FALSE) {
```

  Instead of

```
if (func()) {
```

  A particularly insidious example of incorrect code is:

```
if (strcmp(s1, s2)) {
    /* different */
}
```

  Always write

```
if ((strcmp(s1, s2) != 0) {
    /* different */
}
```

  Decus C provides streq() for this purposes. On other systems, you can easily write a macro:

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

  One counter example to this is generally made for

predicates, functions which have no other purpose than to return TRUE or FALSE, and which are named so that the meaning of a TRUE return is absolutely obvious. For example, a routine should be named "isvalid()", not "checkvalid()".

- Be very suspicious of numeric values appearing in the code. Almost all constants would be better expressed as #defined quantities.

- Any unsigned type other than unsigned int should be identified by a typedef, as these are highly compiler dependent. As noted above, large programs should have a central header file which encapsulates machine-dependent information.

- Become familiar with the standard library and use it for string and character manipulation. Do not reimplement standard routines as the person reading your code must then figure out whether you're doing something special in the reimplemented stuff. Home-brew "standard" routines are a fruitful source of bugs as your routines might be called by other parts of the library. Also, the standard library hides non-portable details that you might not (and generally should not) be aware of.

11.0 Miscellaneous

This section contains a fairly disorganized list of hints, some of which appear in other sections of this style sheet. They are not in any specific order.

- Don't change syntax via macro substitution. It makes the program unintelligible to all who come after.

- There is a time and place for embedded assignment statements. In some cases, this is the best way to specify the algorithm. However, it is not your responsibility to second-guess the compiler by packing code as tightly as possible. For example:

```
a = b + c;
d = a + r;
```

should not be rewritten as:

```
d = (a = b + c) + r;
```

Even though the latter may save one instruction.

- Don't overuse the ternary "(cond) ? a : b" operator The condition should always be enclosed in parentheses. Nested ternary operators should be avoided if possible.

- Goto statements should be used sparingly. The main place where they are useful is in breaking out of several levels of switch/for/while nesting. If a goto is needed, the accompanying label should be at the left margin with a comment explaining who jumps here. The continue statement is also a source of bugs.

  But, don't be afraid that evil spirits will haunt you if you write the dreaded goto. It is often much clearer to use goto's to escape from an inner loop than by using seemingly random combinations of break, continue, return and default exits from switch statements. To some extent, the lack of a rich set of exit operations is a failure of C, requiring discipline on the part of the programmer.

  Often, the need for goto's and complicated exit conditions is an indication that the inner constructions ought to be redone as a separate function with a success/failure return code.

- In declarations (#defines, structure definitions, or variable definitions), various components should line up. Thus:

```
#define TESTING         1
#define PRODUCTION      2
```

- The components of a structure should be uniquely "tagged" using a one-or two character marker followed by '_'. For example,

```
typedef struct list_element {
    struct list_element  *l_next;
    int                   l_value;
} LIST_ELEMENT;
```

- When the storage structure or type of a variable is important, always state it explicitly. In particular, use "auto" if you are going to use the address of a variable using '&'. Declare integer parameters as "int", rather than letting them default.

- Sometimes it is impossible to avoid doing something tricky. (And sometimes you just can't resist the temptation.) At the very least, put enough documentation in the code to warn the poor soul who comes after you.

- Try to write code that is clear and safe, rather than something that "seems" easier to compile. Make sure local variables are local (or static) so things won't blow up when you compile with other modules.

- Try to keep the flow of control through your program apparent. Where this is governed by separately-compiled tables (such as a finite-state parser), embed comments in the parser table to aid the maintainer.

- Use register variables wherever possible. They are especially efficient when used as structure or array pointers. Since offsets within a structure are known at compile time, the compiler can generate extremely efficient code.

  For example, suppose a program is processing a collection of elements which have a value and a set of flag bits. The "simple" solution would be:

```
int     value[MAX];
long    flags[MAX];
int     array_max;

int
lookfor(arg_val, arg_flag)
int         arg_val;
long        arg_flag;
/*
 * Return index to the element with the same
 * value and at least one matching flag bit.
 * Return -1 on failure.
 */
{
        int     i;

        for (i = 0; i < array_max; i++) {
            if (value[i] == arg_val
             && (flag[i] & arg_flag) != 0) {
                return (i);
            }
        }
        return (-1);
}
```

  The inner loop of the above requires turning the index "i" into a pointer twice. The above should generally be rewritten as:

```
typedef struct data {
    int     d_value;
    long    d_flag;
} DATA;
DATA        values[MAX];
DATA        *top_value;
```

```
DATA *
lookfor(arg_value, arg_flag)
int         arg_value;
long        arg_flag;
/*
 * Return a pointer to the element with the same
 * value and at least one matching flag bit.
 * Return NULL on failure.
 */
{
        register DATA       *dp;

        for (dp = &values[0]; dp < top_value; dp++) {
            if (dp->d_value == arg_value
             && (dp->d_flag & arg_flag) != 0) {
                return (dp);
            }
        }
        return (NULL);
}
```

Note the use of redundant braces in the above programs.

- If a function manipulates a database stored in a separate file, the routines that manipulate (generate and access) this database should be isolated from other routines. The internal structure of the data base should also be defined. If the database format is likely to change, a release date or version should be buried in the database and precompiled into the manager software. The program should check the validity of the release date when the package opens the database.

- If a file contains the main routine of a program, that should be the first function in the file. On Unix and VMS, where programs may be called as sub-processes, it is important that all programs exit by calling exit(). On Unix, use "exit(0)" for success and a non-zero value for failure. The following construction may be useful:

```
#ifdef      vms
#include    <ssdef.h>
#endif
...
#ifdef  vms
exit(SS$_NORMAL);
#else
exit(0);
#endif
```

- In the condition portion of an if, for, while, etc., side effects whose effect extends beyond the extent of the guarded statement block should be avoided. For example, consider:

```
if ((c = getchar()) != EOF) {
    guarded-statements
}
other-statements
```

It is natural to think of variable "c" being "bound" to a value only within "guarded-statements." Its value should not be presumed upon entrance to "other-statements." Use of a variable set or modified inside a condition outside the range of statements guarded by the condition is in general quite distracting.

- You should not use || and && with right-hand operands having side-effects. For example,

```
if ((fildes = fopen("file.nam", "r")) == NULL
 || readin(fildes) != SUCCESS) {
    bug("something's wrong somewhere.");
}
```

Whenever sequences contain both || and &&, parentheses should be used for clarity.

- Routines should be kept reasonably short. It is important for the maintainer to be able to read and comprehend all of the routine at one glance. In general, a routine processes one or more inputs and generates one or more outputs, where each of the inputs and outputs can be concisely described.

Signs that a routine is too long, and ought to be split up, are: length greater than 100 lines (two pages), heavy use of localized variables (whose active scope is less than the entire routine), or conditional or loop statements nested more than four levels.

Even when processing is linear (do first part, do second part, etc.), it is often helpful to the maintainer to break the routine into separate pieces:

```
main(argc, argv)
int         argc;
char        *argv[];
{
        setup(argc, argv);
        process();
        finish();
}
```

On many Dec operating systems, the setup() and finish() modules can be compiled into overlay structures, leaving more room for in-memory data.

- Use of globals should be minimized by judicious use of parameters.

- In general, a routine should be designed with a "natural", easily-remembered calling sequence. Routines with more than five arguments are not recommended. Routines with "op-code" arguments, where one argument determines the interpretation and functions of the others, are also not recommended (though they often prove useful as internal routines to a package, they should not be part of a package's documented interface.)

- Datatype compatibility should be practiced where possible. This can be facilitated by use of C's typedef facility, by explicit type casting, or by the use of the union datatype.

  A package which returns a pointer to a structure whose format need not be known outside of that package may return a "generic pointer" (char *). The C language specifically guarantees that any pointer may be converted to a char * and back again without harm.

- Use #defines to eliminate magic numbers. Use compile-time computation to combine magic numbers into others:

```
#define ARRAY_A_SIZE 123
#define ARRAY_B_SIZE 456
#define BOTH (ARRAY_A_SIZE + ARRAY_B_SIZE)
```

  If you change ARRAY_A_SIZE, the compiler with change BOTH without your further intervention.

- Some experience is needed to decide what to put in a for statement and what to put in the loop body. In general, you should put what is needed to control the loop in the for, and the process itself in the body. Also, you should be disciplined about using break, continue, and goto to control "unusual" break-out cases. For example, the following code searches a symbol table for an unused element:

```
for (sp = &sym[0]; sp < &sym[MAXSYM]; sp++) {
    if ((sp->sy_flag & UNUSED) != 0)
        goto found;
}
error_message("No room in symbol table");
return (FALSE);

found:
    /* ... here to process symbol          */
    return (TRUE);
```

  In this case, the most natural way to write the code is to use a goto for the "normal" case. While the above could be handled by a flag (or auxiliary test), the solution seems less intuitive:

```
for (sp = &sym[0]; sp < &sym[MAXSYM]; sp++) {

    if ((sp->sy_flag & UNUSED) != 0)
```

```
            break;
    }
    if (sp >= &sym[MAXSYM]) {
        error_message("No room in symbol table");
        return (FALSE);
    }
    else {
        /* ... here to process symbol                    */
        return (TRUE);
    }
```

- The first three register variables, in lexicographic order, should be ones for which the most gain can be gotten.

- While C distinguishes between upper- and lower-case in variables and keywords, the programmer should maintain discipline. Global symbols should never require case distinction as they will not work properly on many operating systems. You should also avoid using the same name for different quantities.

  Never require the reader to see difference between "1", "l" (letter), and "I", or "O", "Q", and "0". The C language "long constant" identifier ("1l" is a long integer if the second character is the letter 'L') offers a good example of a practice to avoid (use "1L" instead).

12.0 Other Issues

The style sheet presented here differs in a few minor respects with other suggested C styles:

- I have recommended writing the closing "*/" of a single line comment at column 72 to enhance readability. The eye need not "read" and "understand" the content of the comment terminator, but can treat it as the edge of a page.

- Several other style sheets recommend the brace syntax:

```
if (cond)
{
    statements;
}
```

  Another recommendation is similar to the above except that the braces are aligned with the conditionally-executed statements:

```
if (cond)
    {
    statements;
    }
```

This follows the structured programming methodology that "begin" and "end" are at the same indentation level. It will be discussed further in the next section.

The syntax recommended in this manual (with the left brace on the same line as the conditional) seems, in the author's eyes, to bind the left brace closer to the conditional than does the "left brace on a new line" format. Also, Left braces don't appear in the same column as right braces and are, hence, more visually distinctive. Finally, the right brace is aligned vertically with the clause introducer (if/while/etc.) with no intruding text. This seems to make things more visible.

13.0 Reexamining Indentation

When an early draft of this style sheet was reviewed, a colleague, Jeff Lomicka, took exception the the recommendations for indentation.

Here is an alternative indentation style presented with its own rationale. You may choose your style accordingly, but be prepared to defend it.

A program is a sequential execution of simpler functions, each of which is broken up into more primitive functions until the primitives become directly executable. A compound statement is the same kind of entity as is a single statement or a function call, and should therefore be treated equally.

The goal of proper indentation is to separate visually the level of detail at which the program is viewed, and to permit the reader easily to associate related elements of the program with each other. For example, we need to associate an "if" with its "else", and to be able to determine what are the contents of the if-clause and else-clause.

The general formatting rules are:

- Statements executed sequentially are all at the same indentation level.
- If a statement includes other statements, such the "while" loop body or the "then" and "else" clauses of a conditional, these statements are indented to the next block level.
- Braces are part of the statement, and are always displayed at the same indentation level as the code they contain.
- This improves the readability of the program, since each compound statement easily identified as a primitive function, separate from the control structure that controls

its execution. In traditional top-down fashion

```
if (conditional)
    statement;
else
    statement;
```

is seen when reading a passage of code at one level of detail, and a close look can reveal the details of the statements:

```
if (conditional)
    { /* when executed and what is done here    */
    statements;
    }
else
    { /* when executed and what is done here    */
    statements;
    }
```

A reader is therefore not forced to see the inner block details when trying to understand only the outer block. Note that when reading the code at the outer block's level of detail, only the introducing comment needs to be read to discern the purpose of a compound statement.

These rules are modified according to the same considerations as listed earlier, as seen in the else-if. For example:

```
while (conditional)
    { /* when executed and what is done here        */
    statements;
    }

for (s1; s2; s3)
    { /* when executed and what is done here        */
    statements;
    }

if (conditional)
    { /* when executed and what is done here        */
    statements;
    }
else if (conditional)
    { /* when executed and what is done here        */
    statements;
    }
else
    { /* when executed and what is done here        */
    statements;
    }
```

Note how these rules effect switch statements:

```
switch (c)
```

```
    {
case 1: /* when executed and what is done here      */
    statements;
    break;
case 2: /* when executed and what is done here      */
    statements;
    break;
}
```

The purpose of a typographical style is to present the semantic elements of your program in a way that is understandable by your readers.

The C programming language can be very deceptive. Although it has every characteristic of other block structured languages, because of the way it "looks", it must be treated differently. Many programmers started using Algol derivatives, such as Simula: languages with BEGINs and ENDs. In such languages, BEGIN and END must be prominent as any declaration -- even a function -- could follow any BEGIN. (Later versions of C, though not Decus C, permit variables to be declared following a '{'.) There was thus little difference between single statements and whole programs. In these languages, keywords were always in upper case, library routines would have their first letter capitalized, and user defined variables and functions were in lower case. Everybody did things that way.

While, superficially, C doesn't look very different, it is so in some deeper sense. Those curly braces look like they want to disappear. The blocking appears to want to be done with indentation alone. Since you cant see the braces anyway, it probably doesn't make that much difference where they are, so long as the contents of the blocks are properly indented. There doesn't seem to be any real difference in readability.

Note also that C has at least four separate "flavors" of braces: structure definition delimiters, function delimiters, if/for/while/do delimiters, and switch block delimiters. Since there is only one construct terminator, '}', it becomes more important for the reader to be able to scan up and immediately locate the construct initiator. (In some other languages, such as Bliss, each construct, such as IF, has an unique terminator, such as FI. While this helps prevent runaway syntax errors, it also requires the programmer to remember more information.)

Responding to the difference in language syntax, programmers develop different programming habits. For example, an Algol programmer might think of an IF statement, in general, as:

```
IF condition THEN statement ELSE statement;
```

(with one statement in each clause), while a C programmer might think of an IF statement as:

```
IF condition THEN statements ELSE statements END-IF;
```

Where in C, the THEN is implied by the end of the condition, the braces around the THEN clause are a syntatic nuisance, and the ENDIF is represented by the closing brace on the ELSE clause.

We can do the same with loops.

```
Algol:  WHILE condition DO statement;
C:      WHILE condition DO statements END-WHILE;
```

Here too, the patterns we look for when reading the code are different. The END-IF and END-WHILE are represented, in C, by '}' which requires typographical prominence and must be kept visually distinct from the visually similar '{'. The C style:

```
if (condition) {
    statements;
}
```

is thus more understandable.

But, of course, programmers are different in their needs, backgrounds, and motivations. Essential, however, is the need to define a style, understand it, use it, and know when to violate it to attain the overriding goal of clarity and communication.

14.0 Summary

The following extended -- and artificial -- example shows most of the recommended decisions.

```
/*
 * A C Style Summary Sheet                Block comment
 * abstracted from one                    describes a file
 * by Henry Spencer,                      or section of
 * University of Toronto,                 code.
 * Department of Zoology
 */

#include <stdio.h>                        Header files
#include "local.h"                        don't nest

typedef int     SYTYPE;                   Global definitions
typedef struct symtab {                   struct's use typedef's
  struct symtab *s_next;          /* Link entries          */
  char  *s_name;                  /* Symbol name           */
  SYTYPE s_type;                  /* Symbol type           */
#define   TY_UNK 0                /*   unknown             */
#define   TY_INT 1                /*   integer             */
#define   TY_STR 2                /*   char *              */
  union {
    int         i;                /* Integer               */
    char        *s;               /* String                */
```

```
  } svalue;
} SYMBOL;                                   Typdef's capitalized
SYMBOL  *sy_head = NULL;                    Explicit initialization

/*
 * sylookup(text)
 *
 * Look for a word in the symbol table,
 * return a pointer to the symbol if found.
 * return NULL if not found.
 */

static SYMBOL *                             What is returned
sylookup(text)                              Name at first column
char    *text;  /* Symbol name             */
{
    register SYMBOL *syp;

    for (syp = sy_head; syp != NULL; syp = syp->s_next) {
        if (strcmp(text, syp->s_name) == 0)
            return (syp);
    }
    return (NULL);
}


/*
 * syprint(text)
 *
 * If the argument is in the symbol table, print
 * the associated value, else print "not found".
 */

syprint(text)                               Doesn't return a value
char    *text;  /* Symbol name             */
{
    register SYMBOL     *syp;

    printf("%s: ", text);
                                            The following shows
                                            an acceptable embedded
                                            assignment, but don't
                                            default the NULL test.
                                            Use braces even for a
                                            single statement.
    if ((syp = sylookup(text)) == NULL) {
        printf("%s: not found\n", text);
    }
    else {                                  Braces here, too.
        switch (syp->s_type) {
        case TY_UNK:
            printf("unknown");
            break;
                                            Blank line after break
        case TY_INT:
```

```
            printf("%d", syp->s_value.i);
            break;

        case TY_STR:
            printf("%s", syp->s_value.s);
            break;

        default:                            Always have a default
                                            Message before abort
            printf("? bad type %d\n", syp->s_type);
            abort();
        }
    }
    printf("\n");
}
```

UNIX Hints and Kinks

Armando P. Stettner
Digital Equipment Corporation
Merrimack, NH
Dorothy Geiger, Session Chairperson
Cal Poly
San Luis Obispo, CA

Reported by Dorothy Geiger, DECUS Scribe Service

The UNIX Hints and Kinks session was led by Armando P. Stettner and included panelists Joe Sventek, Norman Wilson, Bill Burns, and Vance Vaughn. Questions were as follows:

Q. Does UNIX have support for networking?
A. VMS/UUCP is being examined, as is DECnet for VAX/UNIX. Software is available to support "tar" under VMS and TCP/IP under UNIX.

Q. Does 4.2Bsd have new networking software?
A. 4.2Bsd provides rich support for interprocess communications between machines, including named sockets for servers, TCP/IP, and Ethernet. In addition, processes do not require the same parent for interprocess communications.

Q. What is the Software Tools Group's Virtual Operating System?
A. VOS is NOT UNIX, it is an entire program development environment which is based on the Software Tools book by Kernighan and Plauger. It provides a variety of shells, tools and utilities, and has been implemented for various operating systems such as RSX and VMS. The VOS software is available on the DECUS SIG tapes.

Q. Will DIGITAL's release of UNIX have system performance and monitoring tools?
A. DIGITAL's current release includes "vmstat", "iostat" and kernel profiling. There are no firm current plans for more.

Q. Can files be accessed across the Network?
A. 4.2Bsd provides for symbolic "links" across network nodes. In addition, work has been done elsewhere to allow file systems to be "mounted" across nodes. 4.3Bsd has transparent file access as a design goal.

Q. Will DIGITAL's VAX/UNIX release have the same basic "goodies" as 4.2Bsd?
A. No decision has been made on 4.1Bsd vs 4.2Bsd. Because 4.2Bsd is a new release, assessments of suitability and reliability are premature at this time.

Q. How does one get 4.xBsd?
A. 4.xBsd may only be supplied to holders of Bell UNIX source licenses.

Q. Is DIGITAL shipping UNIX source licenses?
A. No, pending resolution of legal questions with Bell.

Q. Does DIGITAL support UNIX in the VAX cluster world?
A. Not at the present time.

Q. It would be highly desirable for DIGITAL to provide support in this area.
A. Noted. (APS)

Q. Are there bugs in the 4.2Bsd DMF-32 driver?
A. None are known to the panel. However, be aware that the DMF-32 driver only uses the serial asynchronous portion of the DMF-32, and that the DIGITAL board have modem control on only two lines of the eight on the board.

Q. Will DIGITAL continue to provide free source to device drivers as has been done in the past?
A. Hopefully.

Q. What is the state of Berkeley INGRESS?
A. Good. The VAX architecture necessitates fewer processes per user than the PDP-11 architecture. This greatly enhances performance. INGRESS is available on the current 4.xBsd tape.

Q. Are there any changes in 4.2Bsd from 4.1Bsd?
A. Many. For example, Bsd4.2 has a new file system which is much better for high I/O bandwidth applications such as VLSI design graphics.

Q. What about the new F77 compiler on 4.2Bsd?
A. The old compiler produced code which executed about half as fast as the VMS Fortran compiler. With the new compiler, execution speed is about the same as VMS Fortran, but compile times are SLOW.

Q. Are there incompatibilities between 4.2Bsd and 4.1Bsd?
A. Executable images will run on both systems. Problems arise when munging on directories, since directory formats have changed.

Q. Does 2.9Bsd include job process control via C shell?
A. Yes.

Q. How does System V differ from 4.xBsd?
A. System V supports a two UNIBUS configuration on the VAX-11/780 with distinct restrictions on device placement. 4.2Bsd supports a four UNIBUS configuration with no restrictions on device placement. In addition, VAX-11/750 support is new on System V but not on 4.2Bsd.

Writing C Code for VAX/VMS and UNIX Systems

David Moore
Digital Equipment Corporation
Nashua, NH
Jim Livingston, Session Chairperson
Measurex Corporation
Cupertino, CA

Reported by Todd Spangler, DECUS Scribe Service

There are several problems when writing portable C code. There is no formal standard for C. Confusion exists between boundaries of C and UNIX and there is a general lack of awareness of the problem. David Moore of Digital Equipment Corporation described the differences between UNIX C and VAX/VMS C. VAX/VMS does not use UNIX-style file specifications, fork and exec sort utilities on standard I/O, command line parsing (piping and file redirection), and other routines not in the emulation set. On VMS, RMS provides an RTL stream which is compatible with that of the UNIX stream system. Exec-family RTL routines only provide sharable reads between parent/child directories and only initialization with "=" is supported. Also supported are the += and *= operators.

To help maintain the portability of C code, it is advisable to keep track of size of data items. On the PDP-11, a long variable is equivalent to a short variable as far as memory is concerned. On VAX, a short variable is half the size of a long variable. The memory order and continuity must also be kept track of as VMS can have variables that exist but that do not exist in storage. When referring to this variable with a pointer, there will be an error message created since the variable cannot be found. On VMS, the layout of program address space is important, especially uninitialized pointers, end, edata, and etext. In UNIX, there is a zero pointer, but on VMS this pointer is protected. Unlike UNIX, characters are signed on VMS. Pointer/integer exchanges are possible, but not portable due to size conflicts. External identifiers on VMS are 31 bits long and on UNIX are 6 to 8 bits long. Unlike VMS, it is possible to have holes in the structure alignment. The order of operation on VMS is only forced when using the COMMA, logical AND,and bitwise or logical OR. To be safe, one should not rely on character set dependencies (VMS uses 7 bit ASCII). VMS does not have an ASM program.

Basically, rules to follow in making C code as portable as possible are to use DEFINE as much as possible and stay away from manifest constants. Make use of common header (.h) files in order to keep some record of system dependent constructs, use SIZE OF to keep track of data, and use common routines with all C libraries.

When writing non portable C, one should choose the right support environment. Making use of all the compiler capabilities is important. Using symbols like 'vax', 'vms', and using include text libraries will improve optimization. Allowing the compiler to manage temporary files and assign registers will increase portability. Using constant folding, DEFINE, MODULE, INCLUDE libraries, passing constants by reference, compiler listings including symbol tables, cross references, and preprocessor substitutions will also be helpful. The proper use of PERROR RTL routines to diagnose errors is desirable. With these things in mind, one can consider the system, use the suggestions, and be on the road to writing portable and non-portable C code.

Networking With UNIX in Tokyo

H. Ishida, J. Adachi, T. Kimura, K. Takano
University of Tokyo
Tokyo, Japan
Mark Bartelt, Session Chairperson
HSC Research Development Corporation
Toronto, Ontario

Reported by Jeff Stapleton, DECUS Scribe Service

At the University of Tokyo, the main computer system is a HITACHI main frame. It supports eight processors, each of which is faster than IBM's fastest machine. The system contains 96 megabytes of main store, and supports over five thousand graduate students and faculty members.

The HITACHI has access to over 100 disk storage devices. A laser printer is the only output device, since it is the only printer fast enough for the system. A switching packet designated DDX is used for communications over telephone lines to other university systems. Although the HITACHI is highly flexible, it is not user-friendly. To help alleviate this problem, four UNIX systems have been added. These include a VAX-11/780, a VAX-11/730, an LSI-11/23, and an Intel MDS.

The 780 has four megabytes of memory, the 730 has one megabyte, while the LSI-11 supports 256 kilobytes and the Intel has 64 kilobytes. Users of the system seem to prefer the VAX. The main features of the HITACHI are the large number of available disks, the laser printer, the DDX network packet switching, and the substantial computing power of the system.

In Japan, UNIX is becoming so popular that several manufactures are implementing. These include such familiar names as Toshiba, Sharp, Sard, Densam, CEC and ASR. Most manufacturers use Kanji word processing. Because of this, a standard is being worked on for this.

In Tokyo, electricity is extremely expensive. For this reason, the university system is shut down on weekends and holidays. A complaint on the VAX was that the power up has to be done with three different switches. They wanted to be able to turn the VAX on and off by only throwing one switch and to make this automatic. Digital would not allow the university to change the circuitry on the VAX, so they use a remote physical plunger to throw the switches.

The automatic operation of the VAX/UNIX system works as follows:

1. set up the weekly schedule
2. the timer activates a sequence controller that turns on; (a) the air conditioner, (b) magnetic tape and disk drives, (c) and finally the CPU.

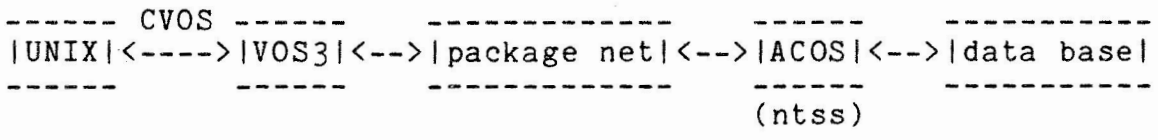
The automatic shutdown is as follows:

1. stop UNIX,
2. stops the disk and magnetic tape drives,
3. and turns off the air conditioner.

The speaker suggested that this automatic procedure should be standard on all VAXs.

The networking system supports two specific packages, CVOS and CVOS2. CVOS calls VOS3, which is the operating system of the host computer, (the HITACHI). CVOS uses a single asynchronous line and is designed for a single user. It allows communication to the laser printer and the Kanji word processor. CVOS2 is similar to CU (call UNIX) and is a cluster controller emulator. This package uses a bi-synchronous line and is designed for several users. Both CVOS and CVOS2 are used for interactive computing.

As an example of interactive programming, suppose you are on the University of Tokyo's system and want to access a data base at the University of Tohoku. The schematic would look like:



The command example given was:

```
CVOS < TOHOKU > logfile
```

where the Tohoku file would appear as:

```
logon uid1/pwd1   <-- VOS3
ntss  TOHOKU      <--  "
uid2              <-- ACOS
pwd2              <--  "
dbname            <--  "
```

then, the system would automatically switch to terminal input.

The batching capacity of CVOS for UNIX can be done either from the terminal or by copying a file. A daemon process in UNIX allows CVOS to spool programs. It can receive batch jobs, output to a log file, output the users job to a file or spool back to VOS3.

Outside users wanted to access the VAX from the DDX. An attempt was made to arrange this, but there were so many problems that the university finally had to give up. Full duplexing was impossible because of unfriendly hardware. The IBM-like OS was difficult to program. Another problem was how to transmit a BREAK to the UNIX environment. The biggest problem was the need to add a carriage return to the transmission. A CR could be added to the end of a data transmission, but no way was found to add the carriage return to a UNIX message. The difficulties involved finally overcame the effort and the project was abandoned.

UNISIG Operating Principles

1 September 1983

1. Name and Purpose

The name of the SIG is "UNISIG" (formerly the "Special Software and Operating Systems Special Interest Group"). The group will deal with, and provide opportunities for, user-user and user-Digital communications about issues related to (1) use of the UNIX* operating system on PDP-11's and VAXes, and (2) UNIX-inspired software (any set of tools providing a UNIX-like environment) which runs under Digital operating systems.

2. Membership

Membership in UNISIG is open to all DECUS members with an interest in UNIX or UNIX-related issues.

3. Steering Committee

The UNISIG steering committee is responsible for the ongoing operation of UNISIG, and for any long-range planning necessary to enhance UNISIG's ability to meet the needs of its members. The steering committee will include, but not necessarily be limited to, the following officers:

Chairman. Interfaces with DECUS on UNISIG-related matters; chairs steering committee meetings and UNISIG business meetings; has responsibility for the orderly operation of UNISIG.

Librarian. Maintains library or user-contributed software; coordinates distribution of UNISIG tapes.

LUG Coordinator. Assists DECUS members in the formation of UNISIG-related local user groups.

Newsletter Editor. Solicits material for UNISIG newsletters; edits and produces newsletters.

Site Survey Coordinator. Maintains a database of site configurations, making information available to members on an as-needed basis.

Standards Coordinator. Deals with any standards-related issues relevant to UNISIG; keeps membership apprised of status of interesting developments.

Symposia Coordinator. Works with DECUS symposia committee during symposium scheduling, to handle all UNISIG-related sessions; writes UNISIG's submissions for the "Call for Papers" and the preliminary program.

---

*UNIX is a trademark of Bell Laboratories.

Usenix Liaison. Reports to UNISIG on activities of the USENIX Organization, as well as other UNIX-related user groups.

In addition, UNISIG's Digital counterpart is a de facto member of the UNISIG steering committee.

4. Appointment of Officers

When a steering committee vacancy exists, the chairman may appoint one or more persons to fill the position, subject to the approval of the UNISIG steering committee. New steering committee positions may be created in a similar fashion if a need for such positions is deemed to exist.

If the position of chairman should become vacant, a new chairman may be appointed by the steering committee.

5. Removal of Officers

UNISIG officers may be removed by a sixty percent vote of the UNISIG steering committee, or by a majority vote of the UNISIG membership in an election.

6. Elections

In the event that there are multiple volunteers for one or more steering committee positions, and if the possibility of a shared position appears to be be impractical or unworkable, an election may be held to fill those positions. An election may also be held to remove a steering committee member from office, or to amend these operating principles.

An election will be held upon a majority vote of the steering committee, or upon petition of five percent of the membership. A member wishing to call an election will be permitted to use the UNISIG newsletter for the purpose of circulating call-for-election petitions.

Informal referendums or polls may be used by UNISIG officers to attempt to gauge the feelings of the UNISIG membership on various issues.

7. Amendments

These operating principles may be amended, modified, or superseded by a majority vote of the UNISIG membership.

UNISIG Steering Committee

Chairman

Mark Bartelt
HSC Research Development Corporation
555 University Avenue
Toronto, Ontario M5G 1X8 CANADA

416-594-5955

Newsletter Editors

Don Crabb
University of Chicago
Computation Center
5737 South University Avenue
Chicago, IL 60637

312-962-7173

William Toth
Harvard-Smithsonian Center for Astrophysics
60 Garden Street, P-353
Cambridge, MA 02138

617-495-7181
{ihnp4|allegra|genrad|amd70|ima}!wjh12!hscfa!toth

Librarian

Carl Lowenstein
UCSD P-001
La Jolla, CA 92093

619-294-3678
ucbvax!sdcsvax!mplvax!cdl

Symposia Coordinator

Dorothy Geiger
Amdahl Corporation
P.O. Box 3470, MS 316
Sunnyvale, CA 94088-3470

Site Survey Coordinator

Paul Graham
Sage Computer Technology
4905 Energy Way
Reno, NV 89502

702-322-6868

Handout Editor

Don Crabb
University of Chicago
Computation Center
5737 South University Avenue
Chicago, IL 60637

312-962-7173

USENIX Liaison
William Toth
Harvard-Smithsonian Center for Astrophysics
60 Garden Street, P-353
Cambridge, MA 02138

617-495-7181
{ihnp4|allegra|genrad|amd70|ima}!wjh12!hscfa!toth

Standards Coordinator
Vacant

LUG Coordinator
Vacant

Digital Counterpart
Jim Barclay
Digital Equipment Corporation
MK2-1/H10
Merrimack, NH 03054

603-884-7256
decvax!jmb

Digital UNIX Guru
Armando Stettner
Digital Equipment Corporation
MK2-1/H10
Merrimack, NH 03054

603-884-2914
decvax!aps