Norsk Data

CC-100 and CC-500
C-Compiler ND-100/500
User Manual

ND-60.214.01

CC-100 and CC-500
C-Compiler ND-100/500
User Manual

ND-60.214.01

NOTICE

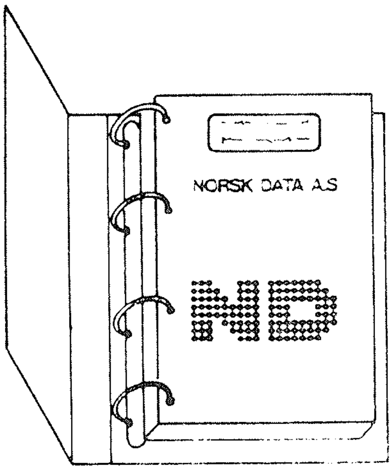
The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

The information described in this document is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A/S.

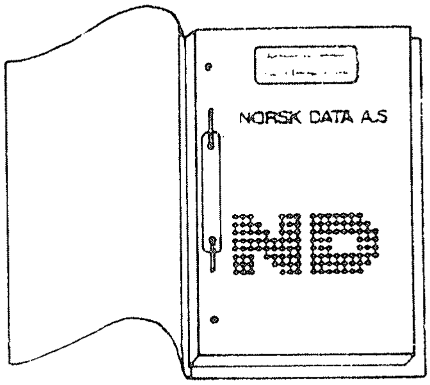
This manual is in loose leaf form for ease of updating. Old pages may be removed and new pages easily inserted if the manual is revised.

The loose leaf form also allows you to place the manual in a ring binder (A) for greater protection and convenience of use. Ring binders with 4 rings corresponding to the holes in the manual may be ordered in two widths, 30 mm and 40 mm. Use the order form below.

The manual may also be placed in a plastic cover (B). This cover is more suitable for manuals of less than 100 pages than for large manuals. Plastic covers may also be ordered below.



A. Ring Binder



B. Plastic Cover

Please send your order to the local ND office or (in Norway) to:

Norsk Data A.S
Graphic Center
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

ORDER FORM

I would like to order

..... Ring Binders, 30 mm, at nkr 20,- per binder

...... Ring Binders, 40 mm, at nkr 25,- per binder

...... Plastic Covers at nkr 10,- per cover

Name ....

Company ....

Address ....

....

City ....

PRINTING RECORD

Printing	Notes
01/85	Version 01

CC-100 and CC-500 C-Compilers for ND-100/500 User Manual
Publ.No. ND-60.214.01

Norsk Data

Norsk Data A.S
Graphic Center
P.O.Box 25, Bogerud
0621 Oslo 6, Norway

Manuals can be updated in two ways, new versions and revisions. New Iversions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the ND Bulletin and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

These forms and comments should be sent to:

Documentation Department
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Requests for documentation should be sent to the local ND office or (in Norway) to:

Graphic Center
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Preface:

THE PRODUCT

This manual covers the programming language C - or perhaps more known as the "UNIX" programming language - as described in the book:

The C Programming Language
by
Brian W. Kernighan and Dennis M. Ritchie
Bell Telephone Laboratories, Incorporated
©1978
Prentice-Hall Software Series
ISBN 0-13-110163-3

This implementation is made by the University of Luleå, and IAR Systems AB, Sweden, in cooperation with Norsk Data A.S.

The compiler and its accompanying libraries are available for the ND-100 and the ND-500 computers, running under the operating system SINTRAN III VSX and 500/VSX.

Product numbers: ND-10760, for the ND-100, and
ND-10761, for the ND-500.

THE READER

This manual is intended for the experienced programmers, having either good knowledge of the C language from the above mentioned publication, or having good experience with the ND-computers and system software.

PREREQUISITE KNOWLEDGE

The readers are expected to have extended programming experience, good knowledge of ND-computers and system utilities as editors and program-linkage-loaders, as well as file-handling and related topics.

RELATED MANUALS

SINTRAN III Reference Manual	ND-60.128
Symbolic Debugger	ND-60.158
ND-100 ND Relocatable Loader (NRL)	ND-60.066
ND-100 BRF-LINKER	ND-60.196
ND-500 LOADER/MONITOR	ND-60.136

TABLE OF CONTENTS

Section Page

Section Page

Section Page

APPENDIX

1 The C Language.

The C programming language was developed at Bell Laboratories and was originally used to implement the UNIX operating system.

The reasons for using C for general purpose programming are several:

> C combines high and low-level features which makes it a more "complete" language than for example Pascal and FORTRAN.

Due to the language design it is relatively easy to make C compilers produce efficient code so that assembly language will seldom be needed.

There is one (only one) recognized standard for the C language ("The C Programming Language" by Kernighan and Ritchie).

Perhaps the most important feature of C is that it has proved to be a very portable language, virtually independent of operating systems and CPU wordlength.

This is more important than ever before since a good piece of software which you may have invested several years of development in, is likely to "survive" changing hardware environments.

> With CC-100/500 users of ND-computers can join the
> rest of the computing world, and that without
> changing operating system, editor etc.!!

The name of the ND-100 and ND-500 C compiler is

@CC-100 and @CC-500

respectively.

1.1 Relocatable Libraries.

The relocatable libraries includes all system functions described in 7.

The CC-HEADER file must be loaded prior to the user's object files, and the libraries and TRAILER files afterwards. This is necessary to set up the proper initialization and termination routines.

On the ND-100, the libraries are separated in 1- and 2-bank versions. The default compiler option is 2-bank, hence all loading must use the CC-2 library files. It is not allowed to mix 1- and 2-bank routines, the NRL or BRF-LINKER will give an error message.

To compile in 1-bank mode, the compiler-option "-s" must be given.

ND-100

CC-1HEADER-A:BRF	1-bank version	header file
CC-1BANK-A:BRF	1-bank version	library file
CC-1TRAILER-A:BRF	1-bank version	trailer file
CC-2HEADER-A:BRF	2-bank version	header file
CC-2BANK-A:BRF	2-bank version	library file
CC-2TRAILER-A:BRF	2-bank version	trailer file

On the ND-500 there is only one mode, thus one set of library files are necessary.

ND-500

CC-HEADER-A:NRF
CC-LIBRARY-A:NRF

1.2 Standard Library Definitions Include Files.

To get the proper symbols defined for the standard functions described in 7 the source-program must contain at least once a reference to the "header"-file, using the #INCLUDE "<file-name>" preprocessor statement. Please observe that an header-file must only be referred to once, otherwise duplication of symbols will occur.

Further, the name of the include-file must be enclosed either in a set of arrow brackets, <filename>; or a set of double quotes "filename".

The arrow-brackets will direct the compiler to locate the include-files stored under the user C-INCLUDE, the double-quotes will use the standard file-search function in the operating system: first search among the files in the current user's own file-catalogue, if not found there, the search continues at user SYSTEM.

The header files will have the filetype :H, and exist in both ND-100 and ND-500 versions. The compilers will automatically select the proper kind.

The installation procedure described in Appendix C will store the standard header-files under the user "C-INCLUDE".

The names of the standard header files and their funtions are:

ERRNO:H	C runtime error number macro definitions.
STDIO:H	A file containing I/O macro definitions.
CTYPE:H	Useful macro definitions like toupper, isalpha etc.
MATH:H	Declares external math functions.
FCNTL:H	File control block used with the OPEN function.

SETJMP:H	Functions for saving and restoring the stack environment, useful for dealing with errors and interrupts encountered in low-level subroutines.
VARARGS:H	Macroes for writing portable procedures which accepts a variable number of arguments.

These header files required for CC-500 only:

MEMORY:H	Memory allocation routines.
STRING:H	A collection of usefull string manipulations functions.

2 Sample Session.

In this section a small C program is compiled and loaded, showing both the ND-100 and ND-500 procedure.

The program has been taken from the book previously mentioned, and should be fairly typical of a program written in C.

It is assumed that the program has been stored in the file CAT:C

```
# include <stdio.h>
main( argc, argv )    /* concatenate files */
  int argc;
  char *argv[];
  {
    int i;
    char c;
    FILE * fp;
    if( argc == 1 )
      {
        printf( "Usage: cat < file 1 > [ < file 2 > ]" );
        printf( "... [ < file N > ]\n" );
        exit();
      }
    for( i = 1; i < argc; ++i )
      {
        fp = fopen( argv[ i ], "r" );
        if( fp == NULL )
          {
            printf( "Cannot open %s\n", argv[ i ] );
            break;
          }
        while( ( c = getc( fp ) ) != EOF ) putchar( c );
        fclose( fp );
      }
  }
```

The program concatenates the contents of one or more files, to the standard output device, the terminal. The name(s) of the file(s) must be given on the command line, where also the output file may be redirected using the >file option.

For ND-100:

```
@cc-100 cat:c
@nrl
*prog-file "CAT"
*load cc-2header, cat, cc-2bank, cc-2trailer
*exit
@
```

For ND-500:

```
n500:cc-500 cat:c
```

```
                          n500:linkage-loader
                          nll:set-domain "cat"
                          nll:load-segment cc-header, cat, cc-library
                          nll:exit
                          n500:
```

To run the program give the command:

```
    @cat cat:c        or       n500:cat cat:c
```

and see what happens (the program should print a copy of the file CAT:C). Then try :

```
    @cat                       n500:cat
```

And you should get the message "Usage : cat < file 1 > ..." indicating that the program is not activated the proper way.

```
    @cat file
```

where "file" does not exist and the program will tell you that it has failed to open the file "file".

To catenate several files into another file the command would look as

```
    @cat file-a file-b file-c >file-abc
```

3 Running CC-100/500.

3.1 Compiling ND-100 programs.

The ND-100 C compiler is invoked by:

```
      @CC-100  [ -flags ] sourcefilename:C
```

Note: The source filename must have the extension ":C".

The currently implemented flags and their meaning are:

-b Compile in library mode.

-c Send comments through the preprocessor.

-dSYM Define symbol SYM. Equal to: #define SYM 1

-dSYM=nn Define symbol SYM Equal to: #define SYM nn

-w Suppress compiler warnings.

-e The compiler will only process macro definitions. The result will appear on stdout.

-i(DIR) Add directory (DIR) to "#include" search list.

-l FIL A merged list of the C program and the corresponding assembly code is written onto the file FIL:LST. ND-100 only.

-o FIL The file FIL:BRF will receive the object code instead of the default sourcefile:BRF

-s Compile in single bank mode. ND-100 only.

-uSYM Undefine symbol SYM. Only useful for disabling the predefined symbols (SIN3 or ND_100/ND500).

Examples: @CC-100 -B -O (OBJ)ATOB ATOB:C

This (ND-100) example shows a compilation of the file ATOB:C in library mode and the object code is redirected to the file (OBJ)ATOB:BRF.

If the object-file does not exist, it will be created using the same name as the source-file, but with the file-type :BRF.

3.2 Loading ND-100 C programs.

The code produced by CC-100 can be made into :PROG files by using NRL or BRF-LINKER.

Below is the sequence to use:

```
@NRL
*PROG-FILE <your own :PROG file>
*LOAD CC-?HEADER-A
*LOAD <your own files>
*LOAD CC-?BANK-A
*LOAD CC-?TRAILER
*EXIT
```

Note that the question mark ("?") denotes that this character should be either "1" or "2" depending on if one or two-bank code has been generated.

Note that one-bank code is generated by activating the "-s" command line option at compile-time.

The size of the heap (for "malloc" and "free") is by default set to 30000B but can also be manually set in @NRL by using:

```
*DEFINE #HEAPZ <value>
```

Note that this must be done before loading takes place!

The size of the run-time stack is:

```
1-bank load: <stacksize> =  <lowest COMMON address> - #HEAPZ -
                            <highest load address>

2-bank load: <stacksize> =  177777B - #HEAPZ -
                            <highest data load address>
```

3.3 Compiling ND-500 programs.

The ND 500 C compiler is invoked by:

```
n500:CC-500  [ -flags ] sourcefilename:C
```

Note: The source filename must have the extension ":C".

The currently implemented flags and their meaning are:

-b Compile in library mode.

-c Send comments through the preprocessor.

-dSYM Define symbol SYM. Equal to: #define SYM 1

-dSYM=nn Define symbol SYM Equal to: #define SYM nn

-w Suppress compiler warnings.

-e The compiler will only process macro definitions. The result will appear on stdout.

-i(DIR) Add directory (DIR) to "#include" search list.

-o FIL The file FIL:NRF will receive the object code instead of the default sourcefile:NRF.

-uSYM Undefine symbol SYM. Only useful for disabling the predefined symbols (SIN3 or ND_100/ND500).

-l Compile the program and leave the assembler language output on a corresponding file with extension :A5. ND-500 only.

-g Compile in debug mode. ND-500 only.

This (ND-500) example shows a compilation of the file ATOB:C where the code will be put on the file ATOB:NRF

```
n500:CC-500  ATOB:C
```

If the object-file does not exists, it will be created using the same name as the source-file, but with the file-type :NRF.

3.4 Loading ND-500 C programs.

The code produced by CC-500 can be made into executable domains by using the ND-500 LINKAGE-LOADER.

Below is the sequence to use:

```
n500:LINKAGE-LOADER
NLL: SET-DOMAIN "Your execute domain"
NLL: LOAD-SEGMENT CC-HEADER-A
NLL: LOAD-SEGMENT <Your own files>
NLL: LOAD-SEGMENT CC-LIBRARY-A
NLL: END-DOMAIN
NLL: EXIT
```

The size of the heap (for "malloc" and "free") is by default set to 50000B but can also be manually set in LINKAGE-LOADER by using:

```
*DEFINE-ENTRY HEAP <size> D
```

Note that this must be done before loading CC-LIBRARY!

The size of the run-time stack is by default set to 50000B but can be changed in the same way as the HEAP:

```
*DEFINE-ENTRY STACK <size> D
```

3.5 Using the SYMBOLIC DEBUGGER on ND-500 C programs.

On the ND-500 the debug information is generated by the compiler option '-g'.

The loading sequence is the same as in the previous example.

The following little program will be referenced to in the debugging examples:

```
/* print Fahrenheit-Celsius  table for f= 0, 20, ... 300 */
main () {
int lower, upper, step;
float fahr, celsius;
lower = 0;                     /* lower limit of temp table*/
upper = 300;                   /* upper limit of temp table*/
step  = 20;                    /* increment step size      */
fahr  = lower;
while (fahr <= upper) {
      celsius= (5.0 / 9.0 ) * (fahr - 32.0);
      printf("%4.0f %6.1f\0", fahr, celsius;
      fahr = fahr + step;
      }
}
```

NB The example does not intend to demonstrate the elegance of a C program, but is just simple enough to be used for the debugging purpose. (The program is taken from the book " The C Programming Language" mentioned in the preface of this manual).

After compiling and loading the program, then the debugger is activated by:

```
 1  n500:DEBUGGER <program>
 2  ND500 SYMBOLIC DEBUGGER VERSION ....
 3  START AT 01000000004B
 4  *break 13
 5  *run
 6  BREAK AT MAIN.13
 7  *display
 8  LOWER = 0      UPPER = 300      STEP = 20       FAHR  = 0.0
 9  CELSIUS= -7.13053E+29
10  *continue
11  0        -17.8
12  BREAK AT MAIN.13
13  *display celsius
14  CELSIUS =  -1.77778E+01
15  *exit
16  n500:
```

line 1 activates the DEBUGGER with the user domain
4 sets a BREAK-POINT at line no 13 in the source file,
5 starts execution,
6 the Debugger informs that the line has been reached,
7 give the command to display all local variables,
10 continue execution,
11 output from the program,
12 the break-point has been reached again,
13 now, display only the variable CELSIUS,
14 the full fl.pt format is shown.
15 terminate the DEBUGGER

3.5.1 How to Look At and to Set Values to Variables

The command "display" without arguments shows all variables and parameters of the scope you currently visit. Also, the values of simple variables are shown. You can access all global variables by name, even when you are inside functions.

You will get into problems if you have two or more names that differ only in letter cases, because the debugger makes no difference between lower and upper case characters.

3.5.2 Simple Variables

Assumed: char count, letter;

```
    display count, letter           Show values
    set count = -15                 Set value
    display addr (count)            Show address
```

3.5.3 Pointers

Assumed: char *letterp = &letter;

It is not possible to use some C conventions, as letterp[0], *letterp, &letterp, &letter, and ind (letterp + 1).

```
    display letterp                      Show value of letterp
    display ind (letterp)                Show value of letter
    set ind (letterp) = #a               Set value of letter to 'a'
```

3.5.4 Character Strings

Assumed: char *message = 'now is the time';

There isn't today any convenient way, in the debugger, to look at a string pointed to by the *message.

```
    display ind(message)                 Show the value 'n'
    display ind(message + 1)             This is illegal!
    look-at-data ind(message)            Show the first part of string
```

3.5.5 Arrays

Assumed: int mat [10,10], vec[10];

It is not possible to treat matrix and vector names as pointers.

```
    display mat, vec                     Show all element values
    display mat[5,2], vec[9]             Show element values
```

3.5.6 Structures and Unions

Assumed: struct { char ch; int i } s;

```
    display s                            Show all values
    set s.i = 226                        Set one of the values
```

3.5.7 Bit Fields

Assumed: struct { unsigned f3: 3; f16: 16; } bf;

```
    display bf                           Show all values
    set bf.f16 = 226                     Set one of the values
```

3.5.8 Enumeration Variables

Assumed: enum { black, green, white } colour;

display colour	Show value
set colour = white	Set value

3.5.9 Variables in Inner Blocks

Assumed: { int chcount; chcount = ... }

In the current version of the debugger, the inner blocks of functions hae no scope of their own, Therefore, there are only two levels of scope; global scope and function scope. When inside functions, all the variables of the function are available for inspection and change.

To reduce the possibility of duplicate names, the variables of the inner block will be suffixed with a hashmark (#) and the line number where the block begins. (The remaining possible problem, is the rare case when two different inner block variables with the same names are declared on the same line).

display count#125	Show value of a typical inner block variable

3.5.10 Pointers to functions

Assumed: int (*funcp)();

display funcp	Show start address of the routine that funcp points to

3.6 Compiler diagnostics.

There are three kinds of error messages from the compiler:

Warnings: the compiler warns you that a construction is "dangerous" in some way.

Try for example to compile a program where a character is added to a pointer. If you know what you are doing it might be OK to run the program, but on the other hand it might not. Warnings can be suppressed by giving the "-w" option when invoking the compiler.

Errors: these are ordinary errors and in most cases the compiler will tell you what is wrong.

Compiler errors: if the compiler enters never-never-land in its attempt to compile some strange constructions it will tell you what went wrong, perhaps suggest some code modification, and abort.

If a compiler error occurs before any other type of error has been encountered please take a copy of your source program, add a description of the error message and send it to the nearest technical support center.

4 The Command Line.

When starting a C program, the command line (the contents after @<prog-name>) is handled to almost standard UNIX format.

4.1 Startup functions ARGC and ARGV.

The main routine is called on by the startup facility with parameters "argc", "argv" where argc is the number of items on the command line (the command name included), and argv is a pointer array where the pointers points to the "item strings" on the command line. Argv[1] is a pointer to the first parameter, argv[2] to the second and so on. The difference compared to UNIX is that argv[0] points to the entire command line not to the command name, because SINTRAN "eats" the command name.

4.2 Redirection of standard Input and Output files.

Redirection of I/O, <input-file-name and >output-file-name is also possible. Default file type is :SYMB. If the output file referenced does not exist it will be created automatically as a :SYMB file.

Example: Assume program name is PROG.

@PROG <input	redirects the program to read from the file named INPUT:SYMB instead of the terminal.
@PROG >output:data	redirects the program to write data to the file named OUTPUT:DATA, instead of the terminal.

5 Implementation Notes.

5.1 Identifiers.

In the ND-100 C compiler internal identifiers have 12 significant characters while the ND-500 version has 30 significant characters.

For external names the ND-100 version limits the number of significant characters to 7, and no distinction is made between upper and lower-case during linkage.

5.2 Data representation.

The various data types have the following length:

	CC-100	CC-500
char	16 bits [1]	8 bits
short	16 bits	16 bits
int	16 bits	32 bits
enum	16 bits	32 bits
unsigned	16 bits	32 bits
unsigned char	16 bits	8 bits
unsigned short	16 bits	16 bits
long	32 bits	32 bits
unsigned long	32 bits	32 bits
float	48 bits	32 bits
double	48 bits [2]	64 bits
pointers	16 bits	32 bits

[1] Characters are stored as integers in main memory but are truncated to 8 bits when written onto files or streams.

[2] Doubles and floats are considered as equivalent in this implementation because of efficiency reasons. (ND-100 does not have double precision arithmetic in the hardware.)

5.3 Register declarations.

Register declarations are permitted although they are immediatly converted to auto.

5.4 Include files.

Include (#include) files have similar syntax compared to UNIX and CP/M implementations (i.e. "name.ext" is automatically converted to "NAME:EXT") in order to increase portability between ND and other computers.

Include files must as under UNIX be surrounded by angle brackets or double quotes.
These characters have the following meaning for CC-100/500:

```
"file"          =>  Search for: FILE

<file>          =>  ND-100:  Search for: (C-INCLUDE)100-FILE
                    ND-500:  Search for: (C-INCLUDE)500-FILE
```

5.5 Pre-defined "#define" symbols.

When CC-100 is started an implicit declaration of "#define ND_100" and "#define SIN3" is performed whereas CC-500 define the symbols ND500 and SIN3.

This feature can be used in conjunction with "#ifdef" to enhance portability of the source code.

6 Deviations From Standard C.

Probably the best C Reference manual available is the afore-mentioned "The C Programming Language". Since the publication of that book back in 1978, a number of small changes have been made to C. Some of these are described in a one-page Bell document distributed with UNIX Version 7 and UNIX System III.

This section briefly describes these changes as well as some particular deviations in this implementation.

6.1 Unsigned values.

An addition to the original C definition is that the reserved word "unsigned" may also be used on char, short and long variables.

6.2 The "void" data type.

The purpose of the void data type is to declare that a function does not generate any return-value;

```
void funcname(a,b,c)
```

6.3 Enumeration types.

The enumeration type is an unique data type borrowed from Pascal. Enumeration types are used to get automatic sequencing of named constants, and by using **casts** they can be used in expressions. Probably the best use of the enumeration type is in switch-statements. The syntax reassembles that of a **structure** or **union** declaration.

```
enum car { saab, pontiac, mercedes };
```

Establishes an enumeration type "car" with values "saab", "pontiac" and "mercedes", with values 0, 1 and 2 respectively.

```
enum car vehicle, *vp;
```

Declares that **vehicle** is a car, and **vp** points to one.

```
if( vehicle == saab ) vehicle = mercedes;
vp = &vehicle;
```

These are thus two valid statements. As with structures and unions, the enumeration type need never be named explicitly. Normally, the constants begin at 0 and increase by 1; a name followed by "=" and a constant is given that value, and the progression continues from the assigned value. The names of enumerations in the same scope must all be distinct from each other and from the names of ordinary variables; in this way they are different from structures and unions.

6.4 Structure and union assignments.

Structures and unions may be assigned to one another as long as both sides of the assignment are of the same type. They may also pe passed as arguments to functions and returned as function values. Thus the expression to the left of the dot need no longer be a **lvalue**; it may also be a function returning a structure or union.

6.5 Static declarations.

The ND-100 implementation of the C language requires that the storage class "static" is known by the compiler when a static identifier is referenced.

This is easily solved by using "forward" declarations of functions that appear later in the file (which also conforms to the C standard):

```
static foo();  /* Forward declaration of "foo" */

main()
  {
    foo();  /* Reference to "foo" */
  }

static foo()
  {
     /* Body of "foo" */
  }
```

6.6 Pre-processor directives.

Both compilers have implemented the standard pre-processor directives as described in "The C programming language", that is:

```
#define identifier token-string
#define identifier( identifier, ... ,identifier) token-string

#undef identifier

#include "filename"
#include <filename>

#if constant-expression

#ifdef identifier

#ifndef identifier

#else

#endif

#line constant identifier
```

ND-100 special:

```
#lstcod <+|->
```

The lstcod directive has been included to aid debugging of C programs as well as the compiler itself. "#lstcod +" activates a list of the generated code in mnemonic form whereas "# lstcod -" disables this listing.

Note that this directive is a counting one (i.e two "# lstcod -" needs two or more "# lstcod +" to enable listing again. Also note that the "-l FIL" command line option performs an implicit "# lstcod +".

7 CC-100/500 System and I/O Library.

7.1 System and I/O Libraries.

The C language does not include Input and Output statements as a part of the language, but relies on a set of functions to be called upon to perform such operations.

The CC-100/500 I/O-libraries contain most UNIX standard functions, and on the following pages there is a list of the available functions documented in the form they usually are on a UNIX system.

The number in parenthesis after each function name is actually referring to the name of the chapter in the UNIX programmer's guide (System V). NB ! Not a part of this manual.

The heading NAME contains the names of the functions described, in some cases several related functions are described on the same page.

The heading SYNOPSIS gives the declarations of the number of arguments and their types in functions described, as it appears in the #include file <stdio.h>. In some cases the name of a special #include file is specified, containing the definitions that must be declared in the user program before refering to the actual function.

The heading DESCRIPTION gives an explanation of the function(s) described, legal values of arguments, and the results expected.

The heading RETURN VALUE explains the type of result to be expected by the function that is called; and if not successful executed, the error name and a short explanation of the cause of failure. The error name refers to the list of names defined in the include file <errno.h>. See explanation in section INTRO(2) on page 26.

In the headings DIAGNOSTICS and NOTES some special precautions and particularities of the functions are explained.

7.2 System Calls and Error Numbers - INTRO(2).

SYNOPSIS #include <errno.h>

LIST OF FUNCTIONS

Name	Appears on	Page	Description
close	close(2)	35	Close a File Descriptor
creat	creat(2)	29	Create a New file, or Rewrite Existing
exit	exit(2)	37	Terminate Program
lseek	lseek(2)	34	Move Read/Write File Pointer
open	open(2)	30	Open for Reading or Writing
read	read(2)	32	Read from File
unlink	unlink(2)	36	Remove Directory Entry
write	write(2)	33	Write on a File
_exit	exit(2)	37	Terminate Program without Cleanup

DESCRIPTION

The following sections describes all of the system calls available in the relocatable library-files.

Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

An error number is also made available in the external variable *errno*, and if the operating system has indicated an error code this is made available in the external variable *OSerrno* (otherwise *OSerrno* is cleared whenever *errno* is set).

Errno is not cleared on successful calls, so the error numbers should be tested only after an error has been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls.

The following is a list of the *errno* error numbers that are used in this implementation, and their names as defined in <errno.h>. For the *OSerrno* error codes, please consult the documentations of the operating system.

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden by the file protection system of the operating system.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't.

5 EIO I/O error

Some physical I/O error occured during a read or write. This error may in some cases occur on a call following the one to which it actually applies.

9 EBADF Bad file number

Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

12 ENOMEM Not enough space

A program asks for more space than the system is able to supply (used internally by malloc(3C)).

13 EACCES Permission denied

An attempt was made to access a file in a way forbidden by the protection system.

17 EEXIST File exists

An existing file was mentioned in an inappropriate context.

22 EINVAL Invalid argument

Some invalid argument (e.g., reading or writing a file for which lseek has generated a negative pointer). Also set by the functions in the math package (3M).

23 ENFILE File table overflow

The system's table of open files is full, and temporarily no more opens can be accepted.

24 EMFILE Too many open files

The open-file-count limit of the operating system has been reached.

27 EFBIG File too large

The file tried to grow past a file space limit of the file system.

33 EDOM Math argument

The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large

The value of a function in the math package (3M) is not representable within machine precision.

DEFINITIONS Unless specifically stated otherwise, the null file name is treated as if it named a non-existent file.

SEE ALSO intro(3).

NOTE The system calls open and unlink in this implementation accept the usual SINTRAN III abbreviations of file names. This is non-standard, and the use thereof might decrease the portability of programs.

7.2.1 CREAT(2) - Create a New File or Rewrite an Existing One.

NAME creat

SYNOPSIS

```
int creat (file, mode)
char *file;
int mode;
```

DESCRIPTION *Creat* creates a new ordinary file or prepares to rewrite an existing file named by the file name pointed to by *file*.

The file name must not be abbreviated. If no file type is given, type SYMB is assumed.

Mode is not used in this implementation. 0644 is a common standard value of *mode* in most UNIX implementations.

If the file exists, the length is truncated to 0. Otherwise, the file is created.

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing. The file pointer is set to the beginning of the file. No program may have more than 20 files open simultaneously.

Creat will fail if one or more of the following are true:

[EACCES] The file does not exist and the directory in which the file is to be created does not permit writing.

[EACCES] The file exists and access permission is denied.

[EMFILE] Twenty (20) file descriptors are currently open or some open-file-count limit in the operating system is exceeded.

[ENOENT] Error in some component of the file name.

RETURN VALUE Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO close(2), lseek(2), open(2), read(2), write(2).

7.2.2 OPEN(2) - Open for Reading or Writing.

NAME open

SYNOPSIS

```
#include <fcntl.h>
int open (file, oflag [ , mode ] )
char *file;
int oflag, mode;
```

DESCRIPTION *File* points to a file name naming a file. The name may be abbreviated if the *oflag* O_CREAT is not given. If no file type is given, type SYMB is assumed.

Open opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*.

Oflag values are constructed by or-ing flags from the following list (of the first three flags below, exactly one must be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file is created and a value may be given to *mode*. This value is ignored in this implementation. The file name must not be abbreviated (in combination with this flag only).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

RETURN VALUE Upon successful completion, a non-negative integer, namely a file descriptor, is returned.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

The file pointer used to mark the current position within the file is set to the beginning of the file.

No program may have more than 20 file descriptors open simultaneously.

ERROR CODES The named file is opened unless one or more of the following are true:

[ENOENT]	O_CREAT is not set and the named file does not exist.
[EACCES]	Oflag permission is denied for the named file.
[ENFILE]	Twenty (20) file descriptors are currently open.
[EMFILE]	Some open-file-count limit in the operating system is exceeded.
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[ENOENT]	Error in some component of the file name.
[ENOENT]	O_CREAT is not set, and there are more than one file name with the given file name as abbreviation.
SEE ALSO	close(2), creat(2), lseek(2), read(2), write(2).

7.2.3 READ(2) - Read from File.

NAME read

SYNOPSIS

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Fildes is a file descriptor obtained from a creat or open system call.

Read attempts to read nbyte bytes from the file associated with fildes into the buffer pointed to by buf.

On devices capable of seeking, the read starts at a position in the file given by the file pointer associated with fildes. Upon return from read, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, read returns the number of bytes actually read and placed in the buffer; this number may be less than nbyte if the number of bytes left in the file is less than nbyte bytes. A value of 0 is returned when an end-of-file has been reached.

When reading from the terminal, the following characters have a special meaning:

ctrl-@	- End-of-file
ctrl-A	- Remove previous character in line.
ctrl-Q	- Clear the current line.
ctrl-R	- Rewrite the line as it now looks.

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read.

Otherwise, a -1 is returned and errno is set to indicate the error.

[EBADF] Fildes is not a valid file descriptor open for reading.

SEE ALSO creat(2), open(2).

7.2.4 WRITE(2) - Write on a File.

NAME write

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION Fildes is a file descriptor obtained from a creat or open system call.

Write attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the fildes.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from write, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the file was opened with the O_APPEND flag, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] Fildes is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write a file that would exceed a space limit of the file system.

RETURN VALUE Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and errno is set to indicate the error.

SEE ALSO creat(2), lseek(2), open(2).

7.2.5 LSEEK(2) - Move Read/Write File Pointer.

NAME lseek

SYNOPSIS

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

DESCRIPTION *Fildes* is a file descriptor returned from a *creat* or *open* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

RETURN VALUE a non-negative integer indicating the file pointer value is returned.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

[EBADF] *Fildes* is not an open file descriptor.

[EINVAL] *Whence* is not 0, 1 or 2.

[EINVAL] The resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO creat(2), open(2).

7.2.6 CLOSE(2) - Close a File Descriptor.

NAME **close**

SYNOPSIS

```
int close (fildes)
int fildes;
```

DESCRIPTION *Fildes* is a file descriptor obtained from a *creat* or *open* system call. *Close* closes the file descriptor indicated by *fildes*.

RETURN VALUE Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

[EBADF] *Close* will fail if *fildes* is not a valid open file descriptor.

SEE ALSO creat(2), open(2).

7.2.7 UNLINK(2) - Remove Directory Entry.

NAME unlink

SYNOPSIS

```
int unlink (file)
char *file;
```

DESCRIPTION *Unlink* removes the directory entry named by the file name pointed to by *file*. The name may be abbreviated.

The named file is unlinked unless one or more of the following are true:

[ENOENT] The named file does not exist.

[EACCES] The file is open, or removal permission is denied for the named file.

[ENOENT] There are more than one file name with the given file name as abbreviation.

[ENOENT] Error in some component of the file name.

RETURN VALUE Upon successful completion, a value of 0 is returned.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO close(2), open(2).

7.2.8 EXIT(2) - Terminate Program.

NAME **exit, _exit**

SYNOPSIS

```
void exit (status)
int status;
void _exit (status)
int status;
```

DESCRIPTION *Exit* terminates the calling program with the following consequences:

All of the file descriptors open in the calling program are closed.

The C function *exit* may cause cleanup actions before the program exits. The function _exit circumvents all cleanup.

7.3 Standard I/O Subroutines and Libraries - INTRO(3)

SYNOPSIS #include <stdio.h>

LIST OF FUNCTIONS

Name	Appears on Page		Description
abs	abs(3C)	41	Return Integer Absolute Value
atof	atof(3C)	42	Convert ASCII string to Float.-Point Value
atoi	strtol(3C)	50	Convert String to Integer, Base 10
atol	strtol(3C)	50	Convert String to Long Integer, Base 10
calloc	malloc(3C)	54	Main Memory Allocator, gives zeroed mem.space
ecvt	ecvt(3C)	46	Convert Fl.Pt Number to string
errno	perror(3C)	59	Error Number
fcvt	ecvt(3C)	46	Convert Fl.Pt Number to Fortran F-format
free	malloc(3C)	54	Main Memory Allocator, Free Block
frexp	frexp(3C)	52	Manipulates Parts of Fl.Pt Numbers
gcvt	ecvt(3C)	46	Convert Fl.Pt Number to Fortran F or E-format
isalnum	ctype(3C)	45	Classify if Char is Alphanumeric
isalpha	ctype(3C)	45	Classify if Char is Letter
isascii	ctype(3C)	45	Classify if Char is Ascii
isatty	isatty(3C)	53	Find if File is a Terminal
iscntrl	ctype(3C)	45	Classify if Char is Control Char
isdigit	ctype(3C)	45	Classify if Char is Digit
isgraph	ctype(3C)	45	Classify if Char is Printable except Space
islower	ctype(3C)	45	Classify if Char is Lowercase
isprint	ctype(3C)	45	Classify if Char is Printable
ispunct	ctype(3C)	45	Classify if Char is Punctuation char
isspace	ctype(3C)	45	Classify if Char is Space (blank)
isupper	ctype(3C)	45	Classify if Char is Uppercase
isxdigit	ctype(3C)	45	Classify if Char is Hex Digit
ldexp	frexp(3C)	52	Manipulates Parts of Fl.Pt Numbers
longjmp	setjmp(3C)	60	Restore Stack Environment
malloc	malloc(3C)	54	Main Memory Allocator, gives mem.space
memccpy	memory(3C)	56	Memory Operations, Copy until Char
memchr	memory(3C)	56	Memory Operations, Find Char in String
memcmp	memory(3C)	56	Memory Operations, Compare
memcpy	memory(3C)	56	Memory Operations, Copy Char
memset	memory(3C)	56	Memory Operations, Set Chars
mktemp	mktemp(3C)	58	Make Unique File Name.
modf	frexp(3C)	52	Manipulates Parts of Fl.Pt Numbers
OSernno	perror(3C)	59	Operating System Error Number
perror	perror(3C)	59	Print Error Message on stderr
realloc	malloc(3C)	54	Main Memory Allocator, Change Size
setjmp	setjmp(3C)	60	Save Stack Environment
strcat	string(3C)	47	Appends a Sting to another String
strchr	string(3C)	47	Find First Occurence of Char
strcmp	string(3C)	47	Compare two Strings
strcpy	string(3C)	47	Copy Strings
strcspn	string(3C)	47	Find Number of Non-Matching Chars
strlen	string(3C)	47	Return Length of String
strncat	string(3C)	47	Appends a Sting of N char to another
strncmp	string(3C)	47	Compare two Strings of N char

strncpy	string(3C)	47	Copy Strings of N char
strpbrk	string(3C)	47	Find Position of First Matching Char
strrchr	string(3C)	47	Find Last Occurance of Char
strspn	string(3C)	47	Find Number of Matching Chars
strtok	string(3C)	47	Return Tokens from String
strtol	strtol(3C)	50	Convert String to Long Integer
swab	swab(3C)	51	Swap Bytes
sys_errlist	perror(3C)	59	Error Message Table
sys_nerr	perror(3C)	59	Largest Error Number in Error Table
toascii	conv(3C)	43	Translate Characters to Ascii
tolower	conv(3C)	43	Translate Characters to Lowercase
toupper	conv(3C)	43	Translate Characters to Uppercase
varargs	varargs(3)	61	Variable Argument List
_tolower	conv(3C)	43	Translate Characters to Lowercase (macro)
_toupper	conv(3c)	43	translate characters to uppercase (macro)

DESCRIPTION This section describes functions found in various libraries, other than those functions that directly invoke operating system primitives, which are described in Section 2 of this library documentation. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library. Declarations for some of these functions may be obtained from #include files indicated on the appropriate pages.

(3M) These functions constitute the Math Library. Declarations for these functions may be obtained from the #include file <math.h>.

(3S) These functions constitute the "standard I/O package" (see stdio(3S)). These functions are in the Standard C Library already mentioned. Declarations for these functions may be obtained from the #include file <stdio.h>.

DEFINITIONS

character is any bit pattern able to fit into a byte on the machine.

null-character is a character with value 0, represented in the C language as '\0'.

character array is a sequence of characters.

null-terminated character array is a sequence of characters, the last of which is the null character.

string is a designation for a null-terminated character array.

null-string — is a character array containing only the null character.

NULL pointer — is the value that is obtained by casting 0 into a pointer.

The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. NULL is defined as 0 in <stdio.h>; the user can include his own definition if he is not using <stdio.h>.

SEE ALSO intro(2), stdio(3S).

NOTE The functions fopen and freopen(3S) in this implementation accept the usual SINTRAN III abbreviations of filenames. This is non-standard, and the use thereof might decrease the portability of programs.

7.3.1 ABS(3C) - Return Integer Absolute Value.

NAME abs

SYNOPSIS

```
int abs (i)
int i;
```

DESCRIPTION *Abs* returns the absolute value of its integer operand.

NOTES In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

7.3.2 ATOF(3C) - Convert ASCII String to Floating-Point Number.

NAME **atof**

SYNOPSIS

```
double atof (nptr)
char *nptr;
```

DESCRIPTION

Atof converts a character string pointed to by *nptr* to a double-precision floating-point number.

The first unrecognized character ends the conversion.

Atof recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional e or E followed by an optionally signed integer.

If the string begins with an unrecognized character, *atof* returns the value zero.

SEE ALSO scanf(3S).

7.3.3 CONV(3C) - Translate Characters.

NAME **toupper, tolower, _toupper, _tolower, toascii**

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and tolower

have as domain the range of getc(3S): the integers from -1 through 255.

If the argument of toupper represents a lower-case letter, the result is the corresponding upper-case letter.

If the argument of tolower represents an upper-case letter, the result is the corresponding lower-case letter.

All other arguments in the domain are returned unchanged.

_toupper and _tolower

are macros that accomplish the same thing as toupper and tolower but have restricted domains and are faster.

_toupper requires a lower-case letter as its argument; its result is the corresponding upper-case letter.

_tolower requires an upper-case letter as its argument; its result is the corresponding lower-case letter.

Arguments outside the domain cause undefined results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO ctype(3C), getc(3S).

7.3.4 CTYPE(3C) - Classify Characters.

NAME **isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii**

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false.

Isascii is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (-1 - see *stdio*(3S)).

isalpha *c* is a letter.

isupper *c* is an upper-case letter.

islower *c* is a lower-case letter.

isdigit *c* is a digit [0-9].

isxdigit *c* is a hexadecimal digit [0-9], [A-F] or [a-f].

isalnum *c* is an alphanumeric (letter or digit).

isspace *c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.

ispunct *c* is a punctuation character (neither control nor alphanumeric).

isprint *c* is a printing character, code 040 (space) through 0176 (tilde).

isgraph *c* is a printing character, like *isprint* except false for space.

iscntrl *c* is a delete character (0177) or an ordinary control character (less than 040).

isascii *c* is an ASCII character, code less than 0200.

DIAGNOSTICS If the argument to any of these macros is not in the domain of the function, the result is undefined.

7.3.5 <u>ECVT(3C) - Convert Floating-Point Number to String.</u>

NAME ecvt, fcvt, gcvt

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

<u>Ecvt</u> converts <u>value</u> to a null-terminated string of <u>ndigit</u> digits and returns a pointer thereto.

The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through <u>decpt</u> (negative means to the left of the returned digits). The decimal point is not included in the returned string.

If the sign of the result is negative, the word pointed to by <u>sign</u> is non-zero, otherwise it is zero.

<u>Fcvt</u> is identical to <u>ecvt</u>, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by <u>ndigit</u>.

<u>Gcvt</u> converts the <u>value</u> to a null-terminated string in the array pointed to by <u>buf</u> and returns <u>buf</u>. It attempts to produce <u>ndigit</u> significant digits in Fortran F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO printf(3S).

NOTES The return values point to static data whose content is overwritten by each call.

7.3.6 STRING(3C) - String Operations.

NAME **strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok**

SYNOPSIS

```
#include <string.h>
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2; int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments *s1*, *s2* and *s* point to strings (arrays of characters terminated by a null character).

The functions *strcat*, *strncat*, *strcpy* and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string s2 to the end of string s1.

Strncat appends at most n characters. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as s1 is lexiographically less than, equal to, or greater than s2.

Strncmp makes the same comparison but looks at at most n characters.

Strcpy copies string s2 to s1, stopping after the null character has been copied.

Strncpy copies exactly n characters, truncating s2 or adding null characters to s1 if necessary. The result will not be null-terminated if the length of s2 is n or more. Each function returns s1.

Strlen returns the number of characters in s, not including the terminating null character.

Strchr (strrchr) returns a pointer to the first (last) occurrence of character c in string s, or a NULL pointer if c does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string s1 of any character from string s2, or a NULL pointer if no character from s2 exists in s1.

Strspn (strcspn) returns the length of the initial segment of string s1 which consists entirely of characters from (not from) string s2.

Strtok considers the string s1 to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string s2.

The first call (with pointer s1 specified) returns a pointer to the first character of the first token, and will have written a null character into s1 immediately following the returned token.

The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with the first argument a NULL pointer) will work through the string s1 immediately following that token.

In this way subsequent calls will work through the string s1 until no tokens remain.

The separator string s2 may be different from call to call. When no token remains in s1, a NULL pointer is returned.

NOTES For user convenience, all these functions are declared in the #include <string.h> header file.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

7.3.7 STRTOL(3C) - Convert String to Integer.

NAME strtol, atol, atoi

SYNOPSIS

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;

long atol (str) char *str;

int atoi (str) char *str;
```

DESCRIPTION

Strtol returns as a long integer the value represented by the character string str. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters are ignored.

If the value of ptr is not (char **)NULL, a pointer to the character terminating the scan is returned in *ptr. If no integer can be formed, *ptr is set to str, and zero is returned.

If base is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if base is 16.

If base is zero, the string itself determines the base thus: After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment, or by an explicit cast.

Atol(str) is equivalent to strtol(str, (char **)NULL, 10).

Atoi(str) is equivalent to (int) strtol(str, (char **)NULL, 10).

SEE ALSO atof(3C), scanf(3S).

NOTES Overflow conditions are ignored.

7.3.8 SWAB(3C) - Swap Bytes.

NAME **swab**

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION *Swab* copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*-1 instead. If *nbytes* is negative *swab* does nothing.

7.3.9 FREXP(3C) - Manipulate Parts of Floating-Point Numbers.

NAME frexp, ldexp, modf

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the mantissa of a double value as a double quantity, x, of magnitude less than 1 and stores indirectly, in the location pointed to by eptr, an integer n such that value = x*2**n.

Ldexp returns the quantity value*2**exp.

Modf returns the signed fractional part of value and stores the integral part indirectly in the location pointed to by iptr.

7.3.10 ISATTY(3C) - Find If File Is a Terminal.

NAME **isatty**

SYNOPSIS

```
int isatty (fildes)
int fildes;
```

DESCRIPTION *Isatty* returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

7.3.11 MALLOC(3C) - Main Memory Allocator.

NAME malloc, free, realloc, calloc

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION *Malloc* and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches.

It tries to fetch more memory from the memory allocation system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block.

The contents will be unchanged up to the lesser of the new and old sizes.

If no free block of *size* bytes is available in the storage area, then *realloc* will ask *malloc* to enlarge the area by *size* bytes and will then move the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of malloc to do storage compaction.

Calloc allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS Malloc, realloc and calloc return a NULL pointer if there is no available memory or if the area has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by ptr may be destroyed.

NOTE Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

SEE ALSO The size of the available memory can be adjusted at link-time which is described in the section: Loading C programs.

7.3.12 MEMORY(3C) - Memory Operations.

NAME **memccpy, memchr, memcmp, memcpy, memset**

SYNOPSIS

```
#include <memory.h>
char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area s2 into s1, stopping after the first occurrence of character c has been copied, or after n characters have been copied, whichever comes first. It returns a pointer to the character after the copy of c in s1, or a NULL pointer if c was not found in the first n characters of s2.

Memchr returns a pointer to the first occurrence of character c in the first n characters of memory area s, or a NULL pointer if c does not occur.

Memcmp compares its arguments, looking at the first n characters only, and returns an integer less than, equal to, or greater than 0, according as s1 is lexicographically less than, equal to, or greater than s2.

Memcpy copies n characters from memory area s2 to s1. It returns s1.

Memset sets the first n characters in memory area s to the value of character c. It returns s .

NOTES For user convenience, all these functions are declared in the #include <memory.h> header file.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

7.3.13 MKTEMP(3C) - Make a Unique Filename.

NAME mktemp

SYNOPSIS

```
char *mktemp(template)
char *template;
```

DESCRIPTION

Mktemp replaces template by a unique file name, and returns the address of the template.

The template should look like a file name with between six and nine trailing X's, which will be replaced with a letter, the terminal number (3 digits) of the user process, and as much of the string ":temp" as possible. This mean that you will get file type ":t", if you have six trailing X'es, and that you will get file type ":temp", if you have nine trailing X'es.

DIAGNOSTICS

If every letter (a through z) thus inserted leads to an existing file name, mktemp will have shortened your string to zero length upon return (i.e., the first character is set to '\0'). All other detected errors are indicated in the same way.

NOTE

The replacement of templates depends on the operating system. The description above is specific for SINTRAN III, and differs somewhat from the mktemp descriptions of other library implementations.

7.3.14 PERROR(3C) - System Error Messages.

NAME **perror, errno, OSerrno, sys_errlist, sys_nerr**

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern int OSerrno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function.

The argument string s is printed first, then a colon and a blank, then the message and a new-line.

If OSerrno is not zero, a second message and a new-line follows.

To be of most use, the argument string should include the name of the program that incurred the error.

errno

OSernno

The error numbers are taken from the external variables errno and OSerrno, who are set when errors occur but are not cleared when non-erroneous calls are made. The OSerrno variable is set by library routines to the error number of the operating system due to which errno is set. If they set errno of other reasons, OSerrno is cleared.

sys_errlist

sys_nerr

To simplify variant formatting of messages, the array of message strings sys errlist is provided; errno can be used as an index in this table to get the first message string without the new-line. Sys nerr is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO intro(2).

7.3.15 SETJMP(3C) - Non-Local Goto.

NAME setjmp, longjmp

SYNOPSIS

```
#include <setjmp.h>
int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp buf*, is defined in the <*setjmp.h*> header file), for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*.

Longjmp cannot cause *setjmp* to return the value 0.

If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

WARNING If *longjmp* is called when *env* was never primed by a call to *setjmp*, or when the last such call is in a function which has since returned, absolute chaos is guaranteed.

7.3.16 VARARGS(3) - Variable Argument List

NAME varargs

SYNOPSIS

```
#include <varargs.h>
function(va_alist)
va_dcl
va_list pvar;
va_start(pvar);
f = va_arg(pvar, type);
va_end(pvar);
```

DESCRIPTION This set of macroes provides a means of writing portable procedures that accept variable argument lists. Routines having variable arguments lists (such as printf(3)) that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for va_alist. Note that there is no semicolon after va_dcl.

va_list is a type which can be used for the variable pvar, which is used to travererse the list. One such variable must always be declared.

va_start(pvar)
is always called to initiate pvar to the beginning of the list.

va_arg(pvar,type)
will return the next argument in the list pointed to by pvar. Type is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(pvar) is used to finish up.

Multiple traversals, each bracketed by va_start ... va_end, are possible.

EXAMPLE

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
     va_list ap;
     char *file;
     char *args[100];
     int argno;

     va_start)ap);
     file = va_arg(ap, char *);
     while (args[ argno++ ] = va_arg)ap, char *))
         ;
     va_end(ap);
     return execv(file, args);
}
```

NOTES

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, *execl* passes a 0 to signal the end of the list. *Printf* can tell how many arguments are supposed to be there by the format.

7.4 Standard Buffered Input/Output Package - STDIO(3S)

SYNOPSIS

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

LIST OF FUNCTIONS

Name	Appears on Page		Description
clearer	ferr(3S)	76	Reset error and EOF indicators
fclose	fclose(3S)	75	Close a Stream
feof	ferror(3S)	76	Test if EOF
ferror	ferror(3S)	76	Test if error
fdopen	fopen(3S)	71	Associate Stream with File Descriptor
fflush	fclose(3S)	75	Write out Buffered Data for Stream
fgetc	getc(3S)	65	Get Next Character (function)
fgets	gets(3S)	67	Get String from stream
filno	ferror(3S)	76	Get File Descriptor of Stream
fopen	fopen(3S)	71	Open a Stream
fprintf	printf(3S)	77	Print Formatted Output on Stream
fputcar	putc(3S)	68	Put Char on Stream (function)
fputs	putc(3S)	68	Put String on Stream
fread	fread(3S)	73	Array Input
freopen	fopen(3S)	71	Attach Preopen Stream to stdin/err/out
fscanf	scanf(3S)	81	Convert Formatted Input from stream
fseek	fseek(3S)	74	Set Position of next in/output on stream
ftell	fseek(3S)	74	Returns the Offset of Current Byte
fwrite	fread(3S)	73	Array Output
getc	getc(3S)	65	Get Next Character (macro)
getchar	getc(3S)	65	Get Next Character (macro)
gets	gets(3S)	67	Get String from *stdin*
getw	getc(3S)	65	Get Word, eg.integer (macro)
printf	printf(3S)	77	Print Formatted Output on *stdout*
putc	putc(3S)	68	Put Character (macro)
putchar	putc(3S)	68	Put Character (macro)
puts	putc(3S)	68	Put String on *stdout*
putw	putc(3S)	68	Put Word eg. integer
rewind	fseek(3S)	74	Set Position to the beginning of stream
scanf	scanf(3S)	81	Convert Formatted Input from *stdin*
setbuf	setbuf(3S)	85	Assign Buffer to a Stream
sprintf	printf(3S)	77	"Print" Formatted Output on *string*
sscanf	scanf(3S)	81	Convert Formatted Input from *string*
ungetc	ungetc(3S)	86	Push Char Back Into Stream

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly.

The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

The SINTRAN III file system differs from C and UNIX in the handling of text files. It uses two characters (\r plus \n) as line delimiters where C uses only one (\n), and it uses parity bits in characters where C uses none. This means that text files have to be converted by the library at file reading and file writing. Of course, these conversions must not be done for binary files, so we need a convention that tells the stdio library if the file is a binary file or a text file.

The stdio library assumes that files are text files as a default, but the binary mode may be forced to a file by adding the character 'b' to the *type* parameter of the *fopen*, *freopen*, and *fdopen*(3S) calls. Most C implementations need not worry about the difference between text files and binary files, so this is a non-standard convention.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file.

A constant NULL (0) designates a nonexistent pointer.

An integer constant EOF (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *clearerr*, and *fileno*.

SEE ALSO open(2), close(2), lseek(2), read(2), write(2), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), ungetc(3S).

DIAGNOSTICS Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

7.4.1 GETC(3S) - Get Character or Word From Stream.

NAME **getc, getchar, fgetc, getw**

SYNOPSIS

```
#include <stdio.h>
int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next text character from the named input *stream*. It also moves the file pointer, if defined, ahead one character in *stream*. *Getc* is a macro and so should not be used if a function is necessary; for example one should not have a function pointer point to it.

Getchar returns the next character from the standard input stream, *stdin*. As in the case of *getc*, *getchar* is a macro.

Fgetc performs the same function as *getc*, but is a genuine function. *Fgetc* may run more slowly than *getc*, but may take less space per invocation.

Getw returns the next word (i.e. integer) from the named input *stream*. The size of a word varies from machine to machine. It returns the constant EOF upon end-of-file or error, but as that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* increments the associated file pointer, if defined, to point to the next word. *Getw* assumes no special alignment in the file.

SEE ALSO fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS These functions return the integer constant EOF at end-of-file or upon an error.

NOTES Because *getc* is implemented as a macro, it may incorrectly treat a *stream* argument, causing side effects. In particular, getc(*f++) may not work sensibly. *Fgetc* should be used instead. Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and may not be read using getw on a different processor.

7.4.2 GETS(3S) - Get a String From a Stream.

NAME gets, fgets

SYNOPSIS

```
#include <stdio.h>
char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, stdin, into the array pointed to by s, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the stream into the array pointed to by s, until n-1 characters are read, or a new-line character is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS If end-of-file is encountered and no characters have been read, no characters are transferred to s and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise s is returned.

7.4.3 PUTC(3S) - Put Character or Word On a Stream.

NAME putc, putchar, fputc, putw

SYNOPSIS

```
#include <stdio.h>
int putc (c, stream)
char c;
FILE *stream;

int putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character c onto the output stream (at the position where the file pointer, if defined, is pointing). Putchar(c) is defined as putc(c, stdout). Putc and putchar are macros.

Fputc behaves like putc, but is a function rather than a macro. Fputc may run more slowly than putc, but may take less space per invocation.

Putw writes the word (i.e. integer) w to the output stream (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. Putw neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream stderr, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream stderr is by default unbuffered, but use of freopen(see fopen(3S)) will cause it to become buffered or line-buffered.

When an output stream is unbuffered information is queued for writing on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block; when it is line-buffered each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Setbuf(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This will occur if the file *stream* is not open for writing, or if the output file cannot be grown. Because EOF is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

NOTES

Because *putc* is implemented as a macro, it may incorrectly treat a *stream* argument causing side effects. In particular, putc(c, *f++); may not work sensibly. *Fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor. For this reason the use of *putw* should be avoided.

7.4.4 PUTS(3S) - Put a String On a Stream.

NAME puts, fputs

SYNOPSIS

```
#include <stdio.h>
int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by s, followed by a new-line character, to the standard output stream stdout.

Fputs writes the null-terminated string pointed to by s to the named output stream.

Neither function writes the terminating null character.

DIAGNOSTICS Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES Puts appends a new-line character while fputs does not.

7.4.5 FOPEN(3S) - Open a Stream.

NAME **fopen, freopen, fdopen**

SYNOPSIS

```
#include <stdio.h>
FILE *fopen (file, type)
char *file, *type;

FILE *freopen (file, type, stream)
char *file, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by file and associates a stream with it. Fopen returns a pointer to the FILE structure associated with the stream.

File points to a character string that contains the name of the file to be opened. This name must not be abbreviated if the type string contains any of the characters 'w' or 'a'. If no file type is given, type SYMB is assumed.

Type is a character string having one of the following values (possibly modified by appending or substituting characters as detailed further below):

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of- file

All of these values of type assume that the file contains text, so the parity bit may be removed and conversions between internal (\n) and external (\r followed by \n) line separation characters may be done.

To achieve a correct handling of binary data, you have to append the character 'b' (for binary) to the type string, thus giving type one of the values "rb", "wb", "ab", "r+b", "w+b", or "a+b".

When the type string contains one of the characters 'w' or 'a', no abbreviations of the file name are accepted. If those characters are given in upper case, i.e. as 'W' or 'A', abbreviations of the file name will be accepted.

These upper-case alternatives are non-standard, but faciliate the making of applications that are local to the SINTRAN III operating system, where file name abbreviations are frequently used.

Freopen substitutes the named file in place of the open stream. The original stream is closed, regardless of whether the open ultimately succeeds. Freopen returns a pointer to the FILE structure associated with stream.

Freopen is typically used to attach the preopened streams associated with stdin, stdout and stderr to other files.

Fdopen associates a stream with a file descriptor obtained from open or creat, which will open files but not return pointers to a FILE structure stream which are necessary input for many of the section 3S library routines. The type of stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening fseek or rewind, and input may not be directly followed by output without an intervening fseek, rewind, or an input operation which encounters end-of-file.

A file that is opened for append (i.e., the type string contains an "a" or "A") can be used in the same way as a file that is opened for write (i.e, the type string contains a "w" or "W"). Only in the case that the file already existed, some differences may be noted: The file pointer of the file opened for append will be initially set to the end of the file, while the file opened for write will be truncated to zero length.

SEE ALSO open(2), fclose(3S).

DIAGNOSTICS Fopen and freopen return a NULL pointer on failure.

NOTE The use of the characters 'b', 'W', and 'A' in type strings are non-standard, and thus non-portable constructions.

7.4.6 FREAD(3S) - Array Input/Output.

NAME fread, fwrite

SYNOPSIS

```
#include <stdio.h>
int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies, into an array beginning at *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*.

Fread stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically *sizeof*(**ptr*) where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), stdio(3S).

NOTE Because the internal data representation of *chars* is a word while the external is byte on ND-100 *fread* and *fwrite* have been omitted in the ND-100 library.

DIAGNOSTICS *Fread* and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

7.4.7 FSEEK(3S) - Reposition a File Pointer In a Stream.

NAME fseek, rewind, ftell

SYNOPSIS

```
#include <stdio.h>
int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

Rewind is equivalent to *fseek*(*stream*, OL, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO lseek(2), fopen(3S).

DIAGNOSTICS *Fseek* returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal.

WARNING Although on the UNIX System (it is the same on the SINTRAN III System) an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX Systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such a offset, which is not necessarily measured in bytes.

7.4.8 FCLOSE(3S) - Close or Flush a Stream.

NAME **fclose, fflush**

SYNOPSIS

```
#include <stdio.h>
int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

Fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO close(2), exit(2), fopen(3S), setbuf(3S).

7.4.9 FERROR(3S) - Stream Status Inquiries.

NAME ferror, feof, clearerr, fileno

SYNOPSIS

```
#include <stdio.h>
int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno(stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero when EOF has previously been detected reading the named input stream, otherwise zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named stream, otherwise zero.

Clearerr resets the error indicator and EOF indicator to zero on the named stream.

Fileno returns the integer file descriptor associated with the named stream; see open(2).

NOTE All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO open(2), fopen(3S).

7.4.10 PRINTF(3S) - Print Formatted Output.

NAME **printf, fprintf, sprintf**

SYNOPSIS

```
#include <stdio.h>
int printf (format [ , arg ] ...  )
char *format;

int fprintf (stream, format [ , arg ] ...  )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ...  )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*.

Fprintf places output on the named output *stream*.

Sprintf places "output", followed by the null character (\0) in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available.

Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

char % Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

flags zero or more *flags*, which modify the meaning of the conversion specification.

field width An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;

precision — A *precision* that gives the minimum number of digits to appear for the d, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e and f conversions, the maximum number of significant digits for the g conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.

long — An optional l specifying that a following d, o, u, x, or X conversion character applies to a long integer *arg*.

asterisk (*) — A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The *flag* characters and their meanings are:

- — The result of the conversion will be left-justified within the field.

+ — The result of a signed conversion will always begin with a sign (+ or -).

blank — If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

# — This flag specifies that the value is to be converted to an "alternate form."

For c, d, s, and u conversions, the flag has no effect.

For o conversion, it increases the precision to force the first digit of the result to be a zero.

For x (X) conversion, a non-zero result will have 0x (0X) prefixed to it.

For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it).

For g and G conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (x and X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion.

The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes.

The default precision is 1.

The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[-]d.ddde+ _dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer *arg* has the value zero, the result is undefined. A *null* arg will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by printf and fprintf are printed as if putc(3S) had been called.

EXAMPLES To print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to null-terminated strings:

```
printf("%s,  %s %d,  %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO ecvt(3C), putc(3S), scanf(3S), stdio(3S).

7.4.11 SCANF(3S) - Convert Formatted Input.

NAME **scanf, fscanf, sscanf**

SYNOPSIS

```
#include <stdio.h>
int scanf (format [ , pointer ] ...  )
char *format;

int fscanf (stream, format [ , pointer ] ...  )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ...  )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream stdin.

Fscanf reads from the named input stream.

Sscanf reads from the character string s.

Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string format described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.

2. An ordinary character (not %), which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional l or h indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

convertion codes:

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the scanset, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex, (), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first*-*last*, thus [0123456789] may be expressed [0-9].

Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically.

The conversion characters d, u, o, and x may be preceded by l or h to indicate that a pointer to long or to short rather than to int is in the argument list. Similarly, the conversion characters e , f , and g may be preceded by l to indicate that a pointer to double rather than to float is in the argument list.

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

EXAMPLES The call:

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to

i	the value	25,
x	the value	5.432,
name	will contain	thompson\0.

Another example:

```
int i; float x; char name[50];
scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign

i	the value	56,	as integer,
x	the value	789.0	as floating-point
	skip	0123,	
name	the value	56\0	as a string of characters.

The next call to *getchar* (see *getc*(3S)) will return the character 'a'.

SEE ALSO

atof(3C), getc(3S), printf(3S), strtol(3C).

NOTES

Trailing white space (including a new-line) is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

7.4.12 SETBUF(3S) - Assign Buffering To a Stream.

NAME setbuf

SYNOPSIS

```
#include <stdio.h>
void setbuf (stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by buf to be used instead of an automatically allocated buffer.

If buf is a NULL character pointer input/output will be completely unbuffered.

A constant BUFSIZ, defined in the <stdio.h> header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from malloc(3C) at the time of the first getc or putc(3S) on the file, except that the standard error stream stderr is normally not buffered.

Output streams directed to terminals are always line-buffered unless they are unbuffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

7.4.13 UNGETC(3S) - Push Character Back Into Input Stream.

NAME ungetc

SYNOPSIS

```
#include <stdio.h>
int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc* call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If *c* equals EOF, *ungetc* does nothing to the buffer and returns EOF.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

In order that *ungetc* perform correctly, a read statement must have been performed prior to the call of the *ungetc* function. *Ungetc* returns EOF if it can't insert the character. In the case that *stream* is *stdin*, *ungetc* will allow exactly one character to be pushed back onto the buffer without a previous read statement.

7.5 Mathematical Library Functions - INTRO(3M)

DESCRIPTION These functions constitute the math library. Declarations for these functions may be obtained from the include file <math.h>.

LIST OF FUNCTIONS

Name	Appears on	Page	Description
acos	sin.3M	93	Trigonometric functions
asin	sin.3M	93	Trigonometric functions
atan	sin.3M	93	Trigonometric functions
atan2	sin.3M	93	Trigonometric functions
cabs	hypot.3M	91	Euclidean distance
ceil	floor.3M	89	Ceiling functions
cos	sin.3M	93	Trigonometric functions
cosh	sinh.3M	94	Hyperbolic functions
exp	exp.3M	88	Exponential
fabs	floor.3M	89	Absolute value
floor	floor.3M	89	Floor
gamma	gamma.3M	90	Log gamma function
hypot	hypot.3M	91	Euclidean distance
j0	j0.3M	92	Bessel functions
j1	j0.3M	92	Bessel functions
jn	j0.3M	92	Bessel functions
log	exp.3M	88	Logarithm
log10	exp.3M	88	Logarithm
pow	exp.3M	88	Power
sin	sin.3M	93	Trigonometric functions
sinh	sinh.3M	94	Hyperbolic functions
sqrt	exp.3M	88	Square root
tan	sin.3M	93	Trigonometric functions
tanh	sinh.3M	94	Hyperbolic functions
y0	j0.3M	92	Bessel functions
y1	j0.3M	92	Bessel functions
yn	j0.3M	92	Bessel functions

7.5.1 EXP(3M) - Exponential, Logarithm, Power, Square root.

NAME exp, log, log10, pow, sqrt

SYNOPSIS

```
#include <math.h>
double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of x.

Log returns the natural logarithm of x; log10 returns the base 10 logarithm.

Pow returns x rised to the y:th power.

Sqrt returns the square root of x.

SEE ALSO hypot(3M), sinh(3M), intro(3M)

DIAGNOSTICS pow return a huge value when the correct value would overflow; errno is set to ERANGE. Pow returns 0 and sets errno to EDOM when the second argument is negative and non-integral and when both arguments are 0.

7.5.2 FLOOR(3M) - Absolute value, Floor, Ceiling Functions.

NAME fabs, floor, ceil

SYNOPSIS

```
#include <math.h>
double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

Fabs returns the absolute value |*x*|.

Floor returns the largest integer not greater than *x*.

Ceil returns the smallest integer not less than *x*.

SEE ALSO abs(3)

7.5.3 GAMMA(3M) - Log Gamma Function.

NAME gamma

SYNOPSIS

```
#include <math.h>
double gamma(x)
double x;
```

DESCRIPTION *Gamma* returns ln |*G*(|*x*|)|, where *G* denotes the gamma function. The sign of *G*(|*x*|) is returned in the external integer *signgam*.

The following C program might be used to calculate *G*:

```
    y = gamma(x);
    if (y > 88.0)
         error();
    y = exp(y);
    if(signgam)
         y = -y;
```

DIAGNOSTICS A huge value is returned for negative integer arguments.

NOTES There should be a positive indication of error.

7.5.4 HYPOT(3M) - Euclidean Distance.

NAME **hypot, cabs**

SYNOPSIS

```
#include <math.h>
double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

DESCRIPTION *Hypot* and *cabs* returns

```
sqrt(x*x + y*y),
```

taking precautions against unwarranted overflows.

SEE ALSO exp(3M) for *sqrt*

7.5.5 JO(3M) - Bessel Functions.

NAME j0, j1, jn, y0, y1, yn

SYNOPSIS

```
#include <math.h>
double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
double x;
```

DESCRIPTION These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS Negative arguments cause *y0*, *y1*, and *yn* to return a huge negative value and set *errno* to EDOM.

7.5.6 SIN(3M) - Trigonometric Functions.

NAME **sin, cos, tan, asin, acos, atan, atan2**

SYNOPSIS

```
#include <math.h>
double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

Sin, _cos_ and _tan_

returns trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range -_pi_/2 to _pi_/2.

Acos returns the arc cosine in the range 0 to _pi_.

Atan returns the arc tangent of _x_ in the range -_pi_/2 to _pi_/2.

Atan2 returns the arc tangent of _x_/_y_ in the range -_pi_ to _pi_.

7.5.7 SINH(3M) - Hyperbolic Functions.

NAME **sinh, cosh, tanh**

SYNOPSIS

```
#include <math.h>
double sinh(x)

double cosh(x)
double x;

double tanh(x)
double x;
```

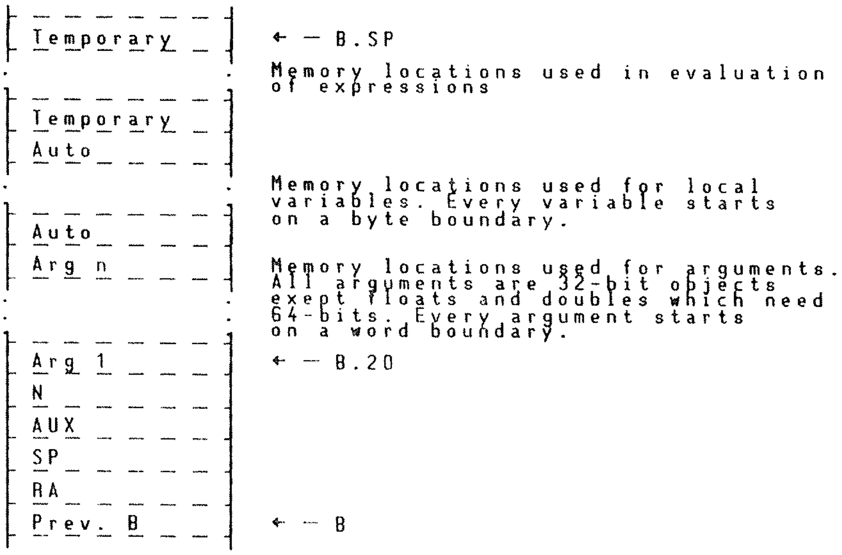
DESCRIPTION These functions compute the designated hyperbolic functions for real arguments.

Successful interfacing with other languages, is based on good understanding of the calling sequence, and the layout of the stackframe in the environment of the calling and called routine.

The following presentation starts with a general description of the stackframe layout produced by the CC-500 compiler. It is continued with an example where a routine written in C calls another C routine. The example is supposed to illustrate the calling sequence.

Currently only routines written in ND-500 Assembler can be added to programs written in C. The presentation is therefore ended with some guidelines showing how these routines should be written.

The stack is growing against higher memory addresses. Higher addresses are upwards in the following description.



Memory locations used in evaluation of expressions

Memory locations used for local variables. Every variable starts on a byte boundary.

Memory locations used for arguments. All arguments are 32-bit objects exept floats and doubles which need 64-bits. Every argument starts on a word boundary.

The following example illustrates the calling sequence produced by CC-500.

Concider the following situation:

The routine "foo" will call the routine "bar" with three arguments

```
foo() {
      int i, j, k;
      i = bar ( j, k, 4);
}
```

The following code will be generated for the routine "foo"

```
foo:     ents      FU1            % create a stackframe for
                                  % foo, FU1 is the size in bytes
         w move    b.24,b.20+FU1  % Move first argument into the
                                  % stackframe which will be
                                  % created by the called
                                  % routine.  B.20 is the
                                  % location of the first
                                  % argument in "bar".
         w move    b.28,b.24+FU1  % The second argument
         w move    4,b.24+FU1     % And the third.
         call      bar,0          % Notice that the hardware
                                  % mechanisme for parameter
                                  % passing is not used.
         w move    r1,b.20        % The return value is passed
         ret                      % in register R1.
FU1:     equ       32
```

The called routine "bar" consists of the following source code:

```
bar ( s, t, u ) {
         return ( s + t + u ); }
```

For this the CC-500 will generate the following code:

```
bar:     ents      FU1            % Create stackframe
         w add3    b.20,b.24,r1   % The arguments s, t, and u has
                                  % the following addresses
                                  % b.20, b.24, and b.28
         w1 +      b.28           % The return value is now in R1
         ret
FU1:     equ       32
```

After the arguments are moved into the area which will be the new stackframe but before the call is executed, the situation can be described with the following figure:

```
|- - - - - - - -|
| 4             |
| k             |
| Undefined     |
| Undefined     |
| Undefined     |
| Undefined     |
| Undefined     |
| k             |      <-- B.SP
| j             |
| i             |      <-- B.20
| N             |
| AUX           |
| SP            |
| RA            |
| Prev. B       |      <-- B
|- - - - - - - -|
```

After execution of the ENTS instruction in the called routine "bar" the extended stack has the following shape:

```
|- - - - - - - -|
| 4             |      <-- B.SP
| k             |
| j             |
| N             |
| AUX           |
| SP            |
| RA            |
| Prev. B       |      <-- B
| k             |
| j             |
| i             |
| N             |
| AUX           |
| SP            |
| RA            |
| Prev. B       |
|- - - - - - - -|
```

The called assembler routine is supposed to begin with an ENTS (enter stack) instruction. ENTS has one parameter, the size in bytes, for the stackframe which will be generated.

The size of the frame must at least be large enough to contain the default part (20 bytes) as well as the arguments of the routine. The first argument will be found at address B.20.

The called routine is able to return a value by loading the register R1 with the actual value before executing the return instruction.

Currently only ND-100 assembly routines can be added to C programs and the technique can be studied in the section "C stack layout"

In order to give the advanced user some ideas on how the run-time system is designed the following information has been include.

Consider the following situation:

```
foo() {
        int j, i, k;
        i = bar( j, k, 4 );
              etc.......
```

will generate the sequence:

```
foo,    lda      *-1            % size of autos
        copy     ad1 sp dx      %address where execution is
                                %continued after the call to
        jmp    i (*csv          %the C enter routine csv.
        saa      4              %push  the  value 4
        sta    i ,b TOS         %TOS is equal to -3
        min      ,b TOS
        lda      ,b 3           %push k
        sta    i ,b TOS
        min      ,b TOS
        lda      ,b 1           %push j
        sta    i ,b TOS
        min      ,b TOS
        sat      3              %size of parameter block
        jpl    i (bar           %call to bar
        sta      ,b 4           %store result in i
        etc.......
```

and in routine bar:

```
bar( s, t, u, v ) {
        return( s + t + u );
        }
```

the generated code sequence is:

```
bar,    lda      *-1
        copy     ad1 sp dx
        jmp    i (*csv
        lda      ,b -6
        add      ,b -7
        add      ,b -8
        jmp    i (*cret
```

The stack in routine "bar" before the jump to *csv will look like:

```
|               |
|   undefined   |
|  previous B   |
|      TOS      |
|   undefined   |
|   undefined   |
|   undefined   |   <--------  B
|   j       -   |
|   i       -   |
|   k       v   |
|   4       u   |
|   k       t   |
|   j       s   |
|   undefined   |
|   undefined   |
```

and after the setup routine *csv it will look like:

```
|                   |
|     undefined     |
|  previous B foo   |
|        TOS        |
|  parsiz to bar    |
|     undefined     |
|     undefined     |
|     j        -    |
|     i        -    |
|     k        v    |
|     4        u    |
|     k        t    |
|     j        s    |
|  previous B bar   |
|     undefined     |
|     TOS (bar)     |
|     undefined     |
|     undefined     |
|     undefined     |   <--------  B
|         w         |
|         x         |
```

Shortly, in your assembly routine do

```
myroutine,      saa     5               % Number of "auto" words
                copy    ad1 sp dx
                jmp   i (*csv
                -
                -
                lda     (retvalue
```

or if the retvalue is a pointer:

```
                ldx     (retvalue
```

or if the retvalue is a long:

```
                ldd     (retvalue
```

or if the retvalue is a float or double:

```
                ldf     (retvalue
```

terminating with

```
                jmp   i (*cret
```

Comments: /* ... this is a comment, ...
may extend over several lines, terminated by */

Identifiers: CC-100 upto 12 significant characters,
A - Z, a - z, 0 - 9 and underscore (_),
first letter must be alphabetic.

CC-500 upto 30 significant characters,
A - Z, a - z, 0 - 9, and underscore (_),
first letter must be alphabetic.

External identifiers: 7 characters, no distinction between upper and lower case letters.

C treats words of upper and lower case as different identifiers. All reserved words in C, as type declarations, standard function names, etc, must be given in lower case.

Constants:

	129	**decimal**	(16-bit)	123489	**decimal**	(32-bit)
	0123	**octal**	(16-bit)	012345	**octal**	(32-bit)
	0x13FF	**hex**	(16-bit)	0x13FF	**hex**	(32-bit)
or	0X13FF	**hex**	(16-bit)	0X13FF	**hex**	(32-bit)

	129l	**long decimal**	(32-bit)
or	129L	**long decimal**	(32-bit)
	0123l	**long octal**	(32-bit)
or	0123L	**long octal**	(32-bit)
	0x13FFl	**long hex**	(32-bit)
or	0x13FFL	**long hex**	(32-bit)

123.0	**float**	(64-bit)	all float.point constants
123.E6			are treated as **double**
123.e-6			

´a´ character (8-bit)
´\c´ non-graphic character (8-bit)

\n	new line	(LF)
\t	tab	(HT)
\´	apostrophe	(´)
\\	backslash	(\)

´\ddd´ char.numeric value (ASCII)

\014	formfeed	(FF)	octal format
\13	carriage return	(CR)	decimal format
\000	string terminator	(NUL)	
\0X7	bell char	(BEL)	hex format

"abc" char. string, all strings are terminated
"" empty string. by NUL-char ´\0´

```
Type specifiers:                     CC-100                              CC-500
                    int       16-bit integer unsigned             32-bit unsigned
                    short     16-bit integer                      16-bit unsigned
                    long      32-bit integer                      32-bit integer
                    float     48-bit fl.pt                        32-bit fl.pt
                    double    48-bit fl.pt                        64-bit fl-pt
                    char      16-bit (input/output 8 bits)        8-bit
                    pointer   16-bit                              32-bit
                    enum      16-bit                              32-bit
                    struct    variable size, (record type)        same
                    union     variable size, (record type)        same
                    typedef   user named type of previous declarations
                    unsigned 16-bit integer                      32-bit integer
                    void func.name     defines no return-value from function

                   Short, long, and unsigned can be combined with int, and char.

Storage classes:    auto       local variables        /* storage class     */
                    static     global variables       /* prefixes type     */
                    extern     global variables       /* declaration, as   */
                    register   treated as auto        /* static int ...    */

Declarations:       int identifer;
                    short int digits;
                    long x, y, z; long int maxnumber;
                    char w, *w;
                    short char letter;
                    float f1; double value

Arrays:             int table[10];          /*                            */
                    float result[50];       /*  arrays i C begins with    */
                    char text[100];         /*  element 0, both for       */
                    int multi[10] [10];     /*  vectors and matrices      */
                    float mix[5] [5];       /*                            */
                    char screen[24] [80]; /*                            */

Accessing arrays:  table[0];               /*  elements can be accessed */
                   table[10];              /*  using constants, vari-   */
                   screen[line] [pos];     /*  ables, and expressions   */

Initialization:    int     a=5
                   int     table[10] =( 1,2,3,4,5,6,7,8,9,10)/;
                   float   pi=3.14169;
                   char    NUL='\0';, LF ='\014';
                   char    *msg="message";

Structure definitions:
                    struct name {
                                  int  day, mth, year;
                                  char mtn_name [4];
                                }
Structure declaration:
                    struct name identifier;
                    struct date d = [ 31, 12, 1984, "DEC "];
                    struct date *pd;   /*  pd points to date structure */
```

Structure accessing:

```
d.mth_name = "MAY "      /*  dot-notation addressing */
```

Operators:

	unary:		binop:	
*	indirect	*msg	multiply	a*5
/			divide	a/5
-	negation	-1	subtract	a-5
+			add	a+5
%			remainder(modulo)	a%5
&	address of	&msg		
sizeof	number of bytes for a type	=sizeof(msg)		

Relational:

>	less than
>=	less and equal
==	equal
<	greater than
<=	greater and equal
!=	not equal

Logical:

&& terminate evaluation if the expression is TRUE

```
if ( character == ´0´ && character == ´9´ )
```

|| terminate evaluation if the expression is FALSE

```
if ( c==´ ´ || c==´\n´ || c==´\t´ )
```

Assignment:

	=	a=5;	a is assigned the value of 5
	+=	a+=5;	a is incremented by 5
	-=	a-=10;	a is decremented by 10
	*=	a*=b+1;	is treated as if a = a * (b + 1)
	++	a++;	assigns value after increment by 1
or		++a;	assigns value, then increment by 1
	--	a--;	assigns value, after decrement by 1
or		--a;	assigns value, then decrement by 1

Bit wise operators:

&	and
|	inclusive or
^	exclusive or
<<	shift left
>>	shift right
~	one´s complement (unary)

```
Statements:          if  (expr) statement;
                     if  (expr) statement; else statement;
                     for (initial-expr; terminate-expr; increment-expr;)
                           statement;
                     while  (expr) statement;
                     do  statement; while (expr);
                     switch (expr);
                       {case-statement(s); and/or default-statement;}
                     case constant-expr: statement;
                     default: statement;
                     break;
                     continue;
                     return (expr);
                     goto label;
                     label: statement;
                     ;  null or empty statement
```

A statement can be a single expression, or a group of expressions within curly brackets (compound statements).

```
Compound statements:{ expression; ....; expression; }
                                         /* may extend over several lines */
```

Each expression must be terminated by a semicolon (;).

```
Conditional statement:?:
                    (expr) ? (expr) : (expr);

                    eg.  z=( a > b ) ? a : b              /* max of a or b */
                    is:  if (a > b ) z = a; else  z = b;

Preprocessor:        #define  identifier constant-expr
                     #define  identifier(parameter,...)  macro-body
                     #undef   identifier
                     #include "filename"        /* user files or SYSTEM */
                     #include <filename>        /* user (C-INCLUDE)     */
                     #if      constant-expression
                     #ifdef   identifier
                     #ifndef  identifier
                     #else
                     #endif
                     #line    constant-identifier
                     #lstcod  list assembly code on/off (+|-)
```

Index

*************** SEND US YOUR COMMENTS!!! ***************

Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card — and an answer to your comments.

Please let us know if you
- * find errors
- * cannot understand information
- * cannot find information
- * find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!

************ HELP YOURSELF BY HELPING US!! ************

Manual name: CC-100 and CC-500 C-Compiler ND-100/500
User Manual

Manual number: ND-60.214.01

What problems do you have? (use extra pages if needed) ______________________

Do you have suggestions for improving this manual ? ______________________

Your name: ______________________ Date: __________

Company: ______________________ Position: __________

Address: ______________________

What are you using this manual for ? ______________________

NOTE!
This form is primarily for documentation errors. Software and system errors should be reported on Customer System Reports.

Send to:
Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Norsk Data's answer will be found on reverse side →

Answer from Norsk Data ______________________________

____

____

____

____

____

____

____

____

____

____

____

____

____

Answered by______________________________Date ______________

- -

□

Norsk Data A.S

Documentation Department
P.O. Box 25, Bogerud
0621 Oslo6, Norway

Systems that put people first

NORSK DATA A.S OLAF HELSETS VEI 5 P.O. BOX 25 BOGERUD 0621 OSLO 6 NORWAY
TEL.: 02 - 29 54 00 - TELEX: 18284 NDN