MH

1559

Bell Laboratories Cover Sheet for Technical Memorandum

Title: **The *printf* Family**

Date: **April 25, 1980**

Other Keywords:

TM: **80-3644-2**

Author(s)	Location	Extension	Charging Case:	49579-980
Andrew Koenig	**MH 7D-301**	**5570**	Filing Case:	**40125-3**

ABSTRACT

This memorandum is a description of the *printf*, *fprintf*, and *sprintf* functions in the C Standard Library. It consists of an informal tutorial, a formal definition, and a listing of C implementations of the functions. The tutorial should be useful for people with some knowledge of the C language who want to be able to use *printf* effectively; the balance supplies more detailed information of a sort that is most useful for implementors.

Pages Text: 23	Other: 0	Total: 23	WORKING COPY
No. Figures: 0	No. Tables: 0	No. Refs.: 2	

BELL TELEPHONE LABORATORIES, INC.

TM-80-3644-2

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATA FILE COPY
(FORM B-1328)

10 REFERENCE COPIES

AITCHISON,E J
AKMAN,THOMAS D
ARTHURS,E
BARTON,M E
BECKETT,J T
BOEHM,PAUL W
CAMPBELL,JEFFRY H
CANNATA,PHILIP M
CHELLIS,ALICIA L
CHRISTENSON-BECKER,MARY
CLARK,DAVID L
COHEN,AARON S
DE GRAAF,E A
*DE JAGER,P S
DEAN,JEFFREY S
DOLOTTA,T A
DURFAN,M R
DWORAK,P S
EISEN,STEVEN R
ELDREDGE,GARY P
FARRELL,JAMES W
FEDER,J
FIORE,RHODA J
FRITZ,THOMAS E
GIPSON,E T,JR
GRAMPP,F T
GUIDI,RIED W
HAIGHT,R C
HALLER,N W
HALL,ANDREW E,JR
HANSEN,R C
HARKNESS,CAROL J
HOPNER,GEORGE G
IMAGNA,C P
JIMENEZ,ZEIMA L
JOHNSTONE,I
KANE,J RICHARD
KUNDU,SUBHAMAY
LAYTON,H J,JR
LOCKITS,E P
LONG,P E
MANTELLOTTO,K A
MAST,C A

KIEDERLOT,R G
GIBSON,S PERI
PAC,T W
PERRY,CATHERINE D
PETERSON,RALPH W
<PHILLIPS,S J
PRIEVE,PERTON G
PUTTRESS,JOHN J
RAILSICK,T M
REPP,K A
ROSENTHAL,VICKI H
ROVEGNO,HELEN D
ROWLAND,BRUCE R
ROY,SUMIT
RUGABER,JON S
SANSIVITZ,A L
SAICHIN,FRED A
SCHMITT,P A
SCHWARTZ,A C
SHULMAN,SHERYL
SLAKA,P F
<SMITH,W T,JR
STOKES,EDWARD R
STUCK,B W
TAGUE,BEVERLY A
VOGEL,GERALD C
WAGNER,MARY E
WEHE,LARRY A
WELSCH,RICHARD J
YACOBELLIS,ROBERT H
78 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

6 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ABRONSON,STEVE M
ABATEMARCO,TERESA M
ABATE,JOSEPH
ACKERMAN,A FRANK
ACKERMAN,J T
ACKHOFF,JOHN M
AHO,ALFRED V
AHRENS,HAINER B
ALBAGLI,V P
ALBERRALE,RICHARD J
ALCALAY,D
ALEXIS,A D,JR

ANDERSON,FREDERICK I
ANDERSON,KATHRYN J
ANDERSON,MILTON M
ANDERSON,R E
ANDERSON,R H
ANDREWS,W J
ANTOLICK,DAVID H
ANTONELLI,CHARLES J
APFELBAUM,MATTHEW A
ARCHER,RUSSELL E,JR
ARMSTRONG,D B
ARMSTRONG,F O,JR
ARNETT,DENNIS I
ARNOLD,GEORGE W
ARNOLD,JAMES O
ARNOLD,PHYLLIS A
ARNOLD,THOMAS F
ARVIDSON,L P
ASFLTINE,EDWARD G
ASMUTH,RICHARD L
ASTHANA,ARNAYA
ATAL,BISHNU S
AULI,DENIS W
AXELSON,A L
BABU,RAJESH KAILAL
BAGGA,YUDHVEER S
BAILEY,CATHERINE T
BAKER,DON
BAKER,MITCHELL R
BALDWIN,GEORGE L
BALENSON,CHRISTINE M
BALLANCE,ROBERT A
BARBATO,ROBERT F
BARCLAY,DAVID K
BARNHARDT,KARL R
BARNS,R L
BAPOPSKY,ALLEN
BARON,ROBERT V
BAER,DAVID L
BATTAGLIA,FRANCES
BAUER,BARBARA T
BAUER,P C
BAUER,HELEN A
BAUGH,C R
BAXTER,LESLIE A
BAYER,D L
BEACHY,MILTON
BEBLO,WILLIAM
BECERRA,PEDRO O
BECKER,CURTIS A
BECKER,JACOB I
BECKER,RICHARD A
BEENAR,JOSEPH A,JR
BRIGHLEY,KEITH A
BENCO,DAVID S

BENSING,JAMES EDWARD
BERENBAUM,ALAN
BERGH,A A
BERGLAND,G D
BERKLEY,M A
BERK,DONALD A
BERNHARDT,RICHARD C
BERNOSKE,BEVERLY G
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BERNSTEIN,PAULA R
BERZINS,ALEXANDER M
BEYER,JEAN-DAVID
BEYLER,ERIC
BHATIA,RAJIV
BIANCHI,M M
BICKFORD,NEIL B
BILASH,TIMOTHY D
BILLINGTON,MARJORIE J
BILOWOS,R M
BIREN,IRMA R
BISHOP,J DANIEL
BISHOP,THOMAS P
BISHOP,VERONICA L
BITTNER,B B
BITTRICH,MARY E
BLAKE,GARY D
BLAZIER,S D
BLECHMAN,RONALD I
BLEIER,JOSEF
BLINN,J C
BLOSSER,PATRICK A
BLUM,MARION
BOCHULA,EDWARD J
BOCKUS,ROBERT J
BOCK,NANCY E
BODDIE,JAMES R
BODEN,F J
BOESE,J O
BOGART,F J
BOGART,THOMAS G
BOIVIE,RICHARD H
BOLSKY,MORRIS I
BONANNI,L E
BOND,F C
BORDELON,EUGENE P
BORG,KEVIN E
BORISON,ELLEN A
BORYAK,SURESH P
BOSE,DEBASISH
BOSTON,RONALD E
BOSWELL,PAULA S
BOULIN,D M
BOUNA,HERMAN J
BOURNE,STEPHEN R

BRADFORD,EDWARD G
BRADLEY,M HELEN
BRADLEY,R H
BRANDT,RICHARD B
BRAUNE,DAVID P
BRAUN,DAVID A
BREILAND,JOHN R
BRENSKI,EDWIN F
BRESLER,RENEE A
BRIGGS,GLORIA A
BRITT,WARREN D
BROAD,MARTHA M
BRONSTEIN,E
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROYMAN,INNA
BROWNING,JASON DAVID
BROWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BROWN,MARK S
BROWN,STUART G
BROWN,W B
BROWN,W STANLEY
BRUECKNER,DOUGLAS E
BRYANT,DAVID J
BUCK,I D
BULLEY,P M
BURGESS,JOHN T,JR
BURG,F M
BURIC,MILORAD R
BURKE,MICHAEL E
BURKE,P J
BURNETTE,W A
BURNETT,DAVID S
BURNPT,ROSE M
BUROFF,STEVEN J
BURROWS,THOMAS A
BUTLETT,DARRELL L
BUTTON,BEVERLY
BYERLEE,P W
BYORICK,ROBERT S
BYRNE,EDWARD R
CALL,PETER F
CALVERT,KENNETH L
CAMPBELL,MICHAEL R
<CANADAY,RUDD H
CANDPEA,RONALD D
CAREY,J E
CAREY,J H
CARMAN,JOHN H
CARRIGAN,RAYMOND J
CARR,DAVID C
CARTER,DONALD H
CASPERS,BARBARA E
CASTELLANC,MARY ANN

MC CABE,P S
MC PHERSON,A F
MEE,C,III
MILLER,JO ANNE H
MITZE,ROBERT W

MIKUS,FREDERICK
MILLSON,C E,JR
MITAY,M
MMIFATA,P T
MOSCN,IRVING

BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,RICHARD L
BENNETT,WILLIAM C
BENOWITZ,P

BOWIER,PAT
BOYCE,K J
BOYCE,W M
BOYER,PHYLLIS J
BOYLE,GERALD C

CATO,H E
CAVINESS,JOHN D
CELLER,GEORGE K
CERMAK,I A
CHAFFEE,N F

* NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT CA ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

1387 TOTAL

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
364-SUP 3644

COVER SHEET TO:
COPICP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE
COPIIO = COMPUTER INPUT-OUTPUT AND UTILITIES PROGRAMMING
UNPICL = C LANGUAGE

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

NO CORRESPONDENCE FILES
NO 1A127

TM-80-3644-2
TOTAL PAGES 24

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

subject: **The *printf* Family**
Case: **49579-980**
File: **40125-3**

date: **April 25, 1980**

from: **Andrew Koenig**
MH 3644
7D-301 x5570
3644-800425.01MF

TM: **80-3644-2**

MEMORANDUM FOR FILE

1. INTRODUCTION

Every useful program needs to produce output of some kind, and the usual way for a C programmer to produce human-readable output from a program is to use the *printf* function. As a result, *printf* is probably the most commonly used program in the C library.

Unlike the description in the reference manual, which is best used to jog the memory and resolve arguments, this document is intended for use by people who do not have a detailed knowledge of the mysteries of *printf*.

2. A SIMPLE EXAMPLE

Consider the following program:

```
main()
{
        printf ("Hello there\n");
}
```

The output from this program is:

```
Hello there
```

followed by a new-line character.

The first argument to *printf* is always a *format*. A format is a character string that describes the form of the output that will result from a call to *printf*. *Printf* works by copying characters from the format to the standard output until either the end of the format is reached or a % character is encountered. Instead of printing a % it finds in the format, *printf* looks at a few characters following the % for instructions as to how to convert its next argument. The converted argument is printed in place of the % and the next few characters. Since the format in this example does not contain a %, the output from *printf* is exactly those characters given in the format. Notice that if you use *printf*, you must explicitly specify every character you want to appear in the output, *including the new-line that ends each line*. It is a common error for beginners to forget the \n that usually terminates a format string. The \n is not supplied automatically because it is frequently useful to produce a single line of output by multiple calls to *printf*.

3. SIMPLE FORMAT TYPES

3.1 Integer Formats

3.1.1 Decimal integers. The most common non-trivial use of *printf* is to print integer values in decimal form. For example, the following call:

```
printf ("2 + 2 = %d\n", 2 + 2);
```

will *print*

```
2 + 2 = 4
```

followed by a new-line (in future examples, we will not explicitly state the presence of a new-line in the output). Of course, most people can put two and two together without the aid of a computer; the purpose of this example is to illustrate the most common form of the **%d** format item.

This example shows a form that will become increasingly familiar in future examples. Every format item is introduced by a % sign, which is followed, not always immediately, by a character, called the *format code*, giving the type of conversion. Other characters may optionally appear between the % and the format code; they serve to modify the conversion in ways that are detailed later.

Note that this example uses a second argument to *printf*. Each format item will usually have a corresponding argument.

The **%d** format item is a request to print an integer. There must be a corresponding **int** argument. The decimal value of the integer, with no leading or trailing spaces, replaces the **%d** as the format is copied to the output. If the integer is negative, the first character of the output value is a – sign.

3.1.2 Unsigned integers. You can print an integer as if it were of **unsigned** type by using the **%u** format item instead of the **%d** format item.

3.1.3 Octal and hex. It is frequently useful to be able to print integer values in base 8 or 16. This is accomplished by the **%o**, **%x**, and **%X** format items. The **%o** item specifies octal output, and the **%x** and **%X** items both specify hexadecimal output. The only difference between **%x** and **%X** is that the **%x** item uses the letters **a**, **b**, **c**, **d**, **e**, and **f** for digit values from 10 through 15, and the **%X** item uses **A**, **B**, **C**, **D**, **E**, and **F**. Octal and hex values are always unsigned.

An example:

```
printf ("%d decimal = %o octal = %x hex\n", 108, 108, 108);
```

will print

```
108 decimal = 154 octal = 6c hex
```

If **%X** were used instead of **%x**, the output would be

```
108 decimal = 154 octal = 6C hex
```

3.2 Character Formats

3.2.1 Single characters. Another frequent use of *printf* is to print character data.

```
printf ("%c", c);
```

is equivalent to

```
putchar (c);
```

but has the added flexibility of being able to insert the value of the character *c* into the string being printed. For example:

```
printf ("par%cty\n", 'i');
```

will print

```
parity
```

3.2.2 Character strings. Printing single characters is usually not as useful as printing entire strings of characters. The %s format item is used for printing strings: the corresponding argument must be a character pointer, and characters are printed starting at the location addressed by the argument until a null character ('\0') is encountered. Here is an example of how the %s format item might be used:

```
int n;
char *s, *r;


r = "is";
s = "";
if (n != 1) {
        s = "s";
        r = "are";
}
printf ("There %s %d item%s in the list.\n", r, n, s);
```

This could, of course, be shortened:

```
printf ("There %s %d item%s in the list.\n", n!=1? "are": "is", n, n!=1? "s": "");
```

The idea is that either is or are will be substituted for the first %s, and that either s or the null string will be substituted for the second %s.

When you use the %s format item, it is important that the strings you print be terminated by a null character ('\0'). That is the only way that *printf* can find the end of the string. If a string that is given to the %s item is not properly terminated, *printf* will continue printing characters until it finds a '\0' somewhere in memory – the output may be very long indeed!

Since a NULL pointer does not point to a string of characters,

```
printf ("%s\n", NULL);
```

is illegal.

3.3 Floating-Point Formats

Three format items provide for printing floating-point values: %g, %f, and %e.

Perhaps the most useful floating-point format item is %g. This item causes the corresponding value (which must be **float** or **double**) to be printed, with trailing zeroes removed, to a precision of six significant figures. Thus

```
double atan();
printf ("Pi = %g\n", 4 * atan (1.0));
```

would print

```
Pi = 3.14159
```

and

```
printf ("%g %g %g %g\n", 1.0/1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 0.0);
```

would print

```
1 0.5 0.333333 0.25 0
```

Notice that nonzero values of magnitude less than 1 are printed with a single leading zero before the decimal point.

If the magnitude is greater than 999999, printing the value in the format just described would require either printing more than six significant digits or displaying an incorrect value. The %g format item resolves this problem by printing the value in "scientific notation":

printf ("%g\n", 123456789.0);

prints

1.23457e+08

This value should be interpreted as 1.23457×10^8.

A related problem occurs when printing values with a very small magnitude. When the magnitude gets small enough, the number of characters required to represent the value gets uncomfortably large. For example, it is ungainly to write $\pi \times 10^{-10}$ as **0.000000000314159**; it is both more compact and easier to read if written as **3.14159e-10**. The two forms have the same length when the exponent is −4 (for example: **0.000314159** as opposed to **3.14159e-04**); in this case the **%g** format item selects the decimal form, because it is more familiar to most people.

There are some occasions where it may be useful to insist that floating-point values always be written with an explicit exponent. This is the purpose of the **%e** format item. Thus, the value of π written under **%e** format is **3.141593e+00**. Notice that the **%e** format item prints six digits *after the decimal point*, rather than printing six significant figures.

In a similar way, the **%f** format item forces the value to be printed *without* an explicit exponent, so π appears as **3.141593**. Again, the **%f** format item prints six digits after the decimal point.

Some computer systems require that if an exponent appears in input data, it must be introduced by E rather than e. If you want to produce output that can be read by such a system, you can use **%E** or **%G** format. These format items behave the same as their lower-case counterparts, except that E instead of e will introduce the exponent.

3.4 Printing a %

The **%%** format item is used to print a **%** character. It is unique in that it is used *without* a corresponding argument. Thus, the statement

printf ("%% %d prints a decimal value\n");

prints

%d prints a decimal value

4. MODIFIERS

Many applications require more flexibility than is provided by the format specifications described so far. *Printf* therefore accepts additional characters that modify the meaning of a format item. These characters appear between the % and the following format code.

4.1 Length Modifier

Integers come in three lengths: **short**, **long**, and plain. If a short integer appears as an argument to any function, including *printf*, it is automatically expanded to a plain integer, but we still need a way to tell *printf* when a long integer is to be printed. This is done by inserting an **l** immediately before the format code, effectively creating **ld**, **lo**, **lx**, and **lu** as new format codes. These modified codes behave exactly the same way as their unmodified counterparts, except that they demand a long integer to correspond with them. Note that using the **lu** format item results in printing a long integer as if it were **long unsigned**, even though that type does not exist in all C implementations. The **l** modifier is meaningless for other than integer format codes.

4.2 Field Width

Programmers frequently like to print numbers in columns. This is difficult using only those features of **printf** discussed so far, because the number of characters that represent a value depends on the value.

Printing values in fixed-width fields is made easier by the *width modifier*. This takes the form of an integer that appears between the % and the following format code, and specifies the *minimum* number of characters that should be printed by the format item so modified. If the value being

printed does not fill the field, blanks will be added on the left to make the value wide enough (See **Flags**, following, if you want the blanks to appear on the right). If the value printed is too big for the field, the field is expanded appropriately. *The width modifier never causes truncation of a field.* If you are using the width modifier to line up columns of figures, and a value is too large for its column, subsequent values on that row will be displaced to the right to make room for the one that was too large.

The width modifier is effective for all format codes.

4.3 Precision

Use the precision modifier if you want to control the number of digits that appear in the representation of a number, or limit the number of characters printed from a string. It consists of a decimal point, followed by a string of digits, and must appear before the format code and length modifier (if present), and after the % and width modifier (if present). The exact meaning of the precision modifier varies with the format code:

- For the integer format items **%d**, **%o**, **%x**, and **%u**, it specifies the *minimum* number of digits that should be printed. If the vale doesn't need that many digits, leading zeroes will be supplied. Thus, 24-bit values might reasonably be printed using **%.6x** or **%.8o** format items, and

  ```
  printf ("%.2d/%.2d/%.4d\n", 3, 12, 1982);
  ```

  would print

  ```
  03/12/1982
  ```

- For **%e**, **%E**, and **%f** format items, the precision specifies the number of digits after the decimal point. Unless the flags (q. v.) specify otherwise, the decimal point does not appear unless the precision is greater than zero.

- For **%g** and **%G** format items, the precision specifies the number of significant digits to be printed. Unless the flags (q. v.) specify otherwise, trailing zeroes are removed, and the decimal point is deleted if no digits follow it.

- For **%s** format items, the precision specifies the maximum number of characters to be printed from the corresponding string. If a null character is encountered before the requisite number of characters have been printed, the string is considered to have ended. For example, path component names are frequently stored as elements of a 14-character character array. If the component name has fewer than 14 characters, the remainder of the array is filled with null characters, but if the name has its maximum length, no null character terminates the array. Such a name might be printed as follows:

  ```
  char dirname[14];
  printf ("... %.14s ...", ... , dirname, ...);
  ```

  This ensures that the name is printed properly, regardless of its length. Using a format item of **%14.14s** would guarantee that exactly 14 characters would be printed, regardless of the length of the name (although the padding blanks would still appear on the *left*; see **Flags** to find out how to put them on the right).

- The precision is ignored for **c** and **%** format items.

4.4 Flags

Between the % and the field witch, characters may appear that modify the effect of the format item slightly. These are called *flag characters*. The flag characters and their meanings are as follows:

- **–** This flag is meaningful only if a width is present. In that case, any padding blanks will appear on the right rather than on the left.

- **+** This flag specifies that every numeric value printed should have a sign as its first character. Thus non-negative values will appear with a + as the first character. It bears no relationship to the – flag.

b When a blank (represented in this document as **b**) is used as a flag, it means that a single blank is to appear before a numeric value if its first character is not a sign. This is most useful for making left-justified columns of numbers line up without using + signs. If the + and b flags appear with the same format item, the + flag takes precedence.

This flag alters the format of numeric values slightly, in a way that depends on the particular format item. Its effect on the **%o** format item is to increase the precision, if necessary, just enough that the first digit that is printed is **0**. The idea is to permit octal values to be printed in the format in which most C programmers are used to seeing them. Note that **%#o** and **0%o** are not the same, as the latter would cause 0 to be printed as **00**. Similarly, the **%#x** and **%#X** format items cause the value to be preceded by **0x** and **0X**, respectively, for the same reason. The effect of the # flag on floating-point formats is twofold: first, it causes the decimal point always to be printed, even if there are no digits after it; second, it stops the **%g** and **%G** formats from suppressing trailing zeroes. The **%#c**, **%#d**, **%#s**, **%#u**, and **%#%** formats mean the same as **%c**, **%d**, **%s**, **%u**, and **%%**, respectively.

The flags are all independent of each other, except for **b** and **+**.

5. VARIABLE FIELD WIDTH AND PRECISION

It is frequently useful to be able to specify the field width and precision as an expression, rather than as part of the format. This is accomplished by replacing the field width, the precision, or both by an *. In this case, *printf* takes the actual value(s) to be used from its argument list before it fetches the value to be printed. For example,

```
printf ("%*d\n", 5, x);
```

has the same effect as

```
printf ("%5d\n", x);
```

If the * convention is used for both field width and precision, the field width argument appears first, followed by the precision argument, and finally by the value to be printed. Thus

```
printf ("%*.*s", 12, 5, str);
```

has the same effect as

```
printf ("%12.5s", str);
```

which prints the first five characters of *str* (or fewer, if *strlen*(*s*)<5), preceded by enough blanks to bring the total number of characters printed to twelve. As another example,

```
printf ("%*%", n);
```

prints $n-1$ spaces followed by a %.

If an * is used for the field width, and the corresponding value is negative, the effect is as if the − flag were also present. Thus, in the example above, if n is negative, the output will be a % followed by $1-n$ spaces.

6. FPRINTF AND SPRINTF

Printf has two relatives, *fprintf* and *sprintf*, that are also very useful.

Where *printf* is restricted to writing on the standard output, *fprintf* can write on any output file. The specific file to be used is given to *fprintf* as its first argument: it must be a FILE pointer. Thus,

```
printf (...);
```

and

```
fprintf (stdout, ...);
```

have the same meaning.

The third family member, *sprintf*, is used when the output is to go somewhere other than a file. The first argument to *sprintf* is a character array in which *sprintf* will place its output. It is the programmer's responsibility to ensure that this array is large enough to contain the output that *sprintf* will generate. The remaining arguments are identical to those of *printf*. The output of *sprintf* is always terminated by a null character, but note that there may be embedded null characters if the %c format item is used.

7. RETURN VALUE

All three functions return the number of characters transmitted. In the case of *sprintf*, the count does not include the null character at the end of the output. If *printf* or *fprintf* encounters an I/O error while attempting to write, it will return some negative value. In this case, it will be impossible to determine how many characters were written.

8. FORMAL SPECIFICATION

This section is recommended reading for masochists. It contains a complete description of the behavior of *printf*, in much more formal terms than the preceding sections. Read it only after you are convinced you understand those sections thoroughly.

8.1 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;
```

8.2 Description

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places 'output' in the character array pointed to by *s*, followed by the character '\0'. The array must be large enough.

Each of these functions returns the number of characters transferred (*sprintf* does not count the terminating '\0'), or a negative value if the operation was unsuccessful. If the number of characters transferred is too large to fit in an int, the value returned is undefined.

Each function makes a single pass over the *format*, stopping when the end of the *format* is reached. That process is performed by repeating the following steps:

- Characters are copied from the *format* to the output stream until a % is encountered in the *format*. The % is not copied. If the end of the *format* is reached before a % is encountered, the function terminates normally.
- A substring of the *format* is found, starting from the %, which is a <conversion specification>. The substring, if one exists, is unique by virtue of the definition given below of a <conversion specification>. If such a substring cannot be found, the *format* is in error and the results are undefined.
- The conversion implied by the <conversion specification> is determined. This process fetches an *arg* for each * in the <conversion specification>, if any. If two *'s are present, the first *arg* fetched corresponds to the <field width>. Each *arg* fetched during this step must be an int.
- The next *arg* is fetched (unless the <conversion letter> is %), converted to a character string according to the <conversion specification>, and written to the output.

The syntax of a <conversion specification> is as follows:

<conversion specification> ::= % <modifiers> <conversion code>

<modifiers> ::= <flags> <field width> <precision>

<flags> ::= <empty> | <flags> <flag>

<flag> ::= + | − | # | <blank>

<field width> ::= <empty> | <field width integer> | *

<precision> ::= <empty> | . <precision part>

<precision part> ::= <unsigned integer> | *

<unsigned integer> ::= <empty> | <unsigned integer> <digit>

<field width integer> ::= <nonzero digit> | <field width integer> <digit>

<conversion code> ::= <conversion modifier> <conversion letter>

<conversion letter> ::= **c** | **d** | **e** | **f** | **g** | **o** | **s** | **x** | **u** | **E** | **G** | **X** | **%**

<conversion modifier> ::= <empty> | **l** | **h**

<digit> ::= **0** | <nonzero digit>

<nonzero digit> ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<empty> ::=

<blank> ::= ␣ (a space)

It is an error for there to be insufficient *arg*s to satisfy the *format*. The result in this case is undefined.

For each <conversion specification>, a *field width* and a *precision* are determined from the <field width> and <precision>, as follows:

- If the <field width> is <empty>, the field width is 0. If the <field width> is a <field width integer>, the field width is the value of which that <field width integer> is a decimal representation. Otherwise, the <field width> is *, and the field width is the absolute value of the corresponding *arg*.
- If the <precision> is <empty>, the precision takes a *default* value which depends on the <conversion letter>. If the <precision> contains a <unsigned integer>, the precision is the value of which that <unsigned integer> is a decimal representation, or 0 if the <unsigned integer> is <empty>. Otherwise, the <precision> is .*, and the precision is the value of the corresponding *arg*. The result is undefined for negative values of the *arg*.

The <flags> are interpreted as follows:

- If the <flags> contain a −, or the <field width> is a * with a corresponding negative *arg*, *left adjustment* (right padding) is in effect. Otherwise, *right adjustment* (left padding) is in effect.
- If the <flags> contain a +, the result of a **d**-, **e**-, **f**-, **g**-, **E**-, or **G**-conversion will always have a sign (+ or −) as its first character. Otherwise, a sign will appear only if it is −.
- If the <flags> contain a <blank>, the result of a **d**-, **e**-, **f**-, **g**-, **E**-, or **G**-conversion will have a blank prepended to it if its first character is not a sign.
- The effect of a # in the <flags> depends on the conversion, and is described separately for each conversion.

Note that if the <flags> contain a <blank> or a +, the width of the result of an **e** conversion of a value whose exponent fits in two digits does not otherwise depend on the value.

The <conversion modifier> has effect only if the <conversion letter> is **d**, **o**, **x**, **u**, or **X**. In this case, a <conversion modifier> of **l** indicates that the *arg* corresponding to the <conversion specification> is a **long**. A <conversion modifier> of **h** may appear but has no effect.

The meanings of the <conversion letter>s are:

d,o,u,x,X The integer *arg* is converted to signed decimal (**d**), or unsigned decimal (**u**), octal (**o**), or hexadecimal (**x** or **X**) notation. The letters **abcdef** are used for **x**-conversion, and the letters **ABCDEF** for **X**-conversion. The *arg* is taken as an **int** of appropriate length for **d**-conversion, and as an **unsigned** of appropriate length for the others. The precision specifies the minimum number of digits that will appear in the result. If the *arg* is too small to be contained in that number of digits, an appropriate number of leading zeroes will appear. The default precision for these format codes is one. Converting a zero *arg* with a precision of zero will give a result with no digits. If the <flags> contain a #, the precision for an **o**-conversion will be increased by the smallest amount necessary to ensure that the first digit of the result is **0**, and the string '**0x**' ('**0X**') will be prepended to the result of converting a non-zero value in **x**- (**X**-) conversion.

f The **double** *arg* is converted to decimal notation. The number of digits after the decimal point is equal to the precision. A leading zero is inserted before the decimal point if no digits would otherwise appear there. The default precision is 6. If the precision is 0 and the <flags> do not contain a #, the result does not contain a decimal point.

e,E The **double** *arg* is converted in the style '[±]d.ddde±dd' where there is exactly one digit before the decimal point and the number of digits after the decimal point and before the exponent is equal to the precision. The digit before the decimal point is only zero if the *arg* is zero. In that case, the exponent always appears as **e+00** (or **E+00**). The default precision is 6. If the precision is 0 and the <flags> do not contain a #, the result does not contain a decimal point. The number of digits in the exponent is the larger of 2 and the smallest number of digits necessary to contain the exponent value. The **E** format code produces a number with **E** instead of **e** introducing the exponent.

g,G The **double** *arg* is converted in a style similar to **f** or **e** (or **E** in the case of a **G** format code). The default precision is 6. Let n be the precision, let s be the result of converting the *arg* according to %.($n-1$)e, and let e be the value of the exponent of s.

If $e \geq n$ or $e \leq -5$, then let r be s. If r contains a decimal point, and the <flags> do not contain a #, the longest possible string of zeroes in r ending immediately before the **e** (or **E**) is deleted, and if the character before the **e** (or **E**) is now a decimal point, it too is deleted. The result is r.

If $e < n$ and $e > -5$, then let r be the result of converting the *arg* according to %.($n-e-1$)f. If r contains a decimal point, and the <flags> do not contain a #, then any trailing zeroes in r are deleted, and if the last character in r is now a decimal point, it too is deleted. The result is r.

c The result is the character *arg*. The precision is ignored.

s *Arg* is taken to be a string (character pointer). The result is an initial substring of the string, whose length is the smaller of the precision and the length of the string. The default precision is infinite.

% No *arg* is converted; the result is a %.

Once the conversion corresponding to a format specification has been accomplished, the result of that conversion is padded with blanks, if necessary, to the field width. If left-justification is in effect, the blanks appear on the right; otherwise, they appear on the left.

9. A SAMPLE IMPLEMENTATION

The following is a C program which is believed to be an accurate implementation of what has been described in the previous sections.

9.1 param.h

This is a sample *param.h* file with definitions appropriate for use with the VAX−11. The meanings of the various definitions should be apparent from the comments.

```
/* Maximum number of digits in any integer representation */
#define MAXDIGS 11

/* Largest (normal length) positive integer */
#define MAXINT 2147483647

/* A long with only the high−order bit turned on */
#define HIBIT 0x80000000L

/* Convert a digit character to the corresponding number */
#define tonumber(x) ((x)− '0')

/* Convert a number between 0 and 9 to the corresponding digit */
#define todigit(x) ((x)+ '0')

/* Data type for flags */
typedef char bool;

/* Maximum total number of digits in E format */
#define MAXECVT 17

/* Maximum number of digits after decimal point in F format */
#define MAXFCVT 60

/* Maximum significant figures in a floating−point number */
#define MAXFSIG 17

/* Maximum number of characters in an exponent */
#define MAXESIZ 4

/* Maximum (positive) exponent */
#define MAXEXP 40
```

9.2 The *printf*, *fprintf*, and *sprintf* functions

These functions are what is actually called by the user program. They are essentially interfaces to *_print*, which does the real work. The operation of the macros in *varargs.h* is described in MF78-8234-64[1].

1. Koenig: Variable Length Argument Lists in C; MF78-8234-64, June 12, 1978. This document is available from the Computing Information Library as number UNPL-1268.

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

int _print();
extern FILE *_pfile;

int
printf (va_alist) va_dcl
{
        register char *format;
        register int rc;
        va_list ap;

        va_start (ap);
        format = va_arg (ap, char *);
        _pfile = stdout;
        rc = _print (format, &ap);
        va_end (ap);
        return rc;
}
```

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

int _print();
extern FILE *_pfile;

int
fprintf (va_alist) va_dcl
{
        register char *format;
        register int rc;
        va_list ap;

        va_start (ap);
        _pfile = va_arg (ap, FILE *);
        format = va_arg (ap, char *);
        rc = _print (format, &ap);
        va_end (ap);
        return rc;
}
```

```
#include <stdio.h>
#include <varargs.h>
#include "param.h"

int _print();
extern FILE *_pfile;
extern char *_pstring;

int
sprintf (va_alist) va_dcl
{
        register char *format;
        register iht rc;
        va_list ap;

        va_start (ap);
        _pstring = va_arg (ap, char *);
        format = va_arg (ap, char *);
        _pfile = NULL;
        rc = _print (format, &ap);
        va_end (ap);
        *_pstring = '\0';
        return rc;
}
```

9.3 _print

_print does all the difficult work except the actual conversion of floating-point values to their decimal equivalents. It assumes that these conversions are done by routines named *fcvt* and *ecvt* as described in any of the various editions of the user's manual for the UNIX™ operating system.

Both *fcvt* and *ecvt* take four arguments. The first is the actual value to be converted, in the form of a **double**. The second is an *int* that contains the number of significant digits desired, for *ecvt*, or the number of digits after the decimal point, for *fcvt*. The third and fourth arguments are pointers to **int**s. The **int** values thus addressed will be set to indicate the location of the decimal point and the sign of the result, respectively. The result returned by either of these functions is assumed to be a pointer to a character string which contains the decimal digits of the mantissa, with no further adornment. The string is assumed to be terminated by a null character. The routines are permitted to trim trailing zeroes from the string. For example

```
char *ecvt();
int decpt, sign;
char *p;
p = ecvt (123.45, 8, &decpt, &sign);
```

will set *p* to point to a storage area containing **12345**, **123450**, **1234500**, or **12345000**. *decpt* will be set to 3 to indicate three digits before the decimal point, and *sign* will be set to 0 to indicate a positive result. If the value were negative, *sign* would be set to a non-zero value.

_print assumes that *fcvt* and *ecvt* will round their results appropriately: it does not change any of the digits returned (though of course it adorns them with a decimal point, sign, and exponent).

_print also assumes a library routine names *strlen*, which when passed a pointer to a null-terminated character string, returns the number of characters in the string, excluding the null character.

```
/*
 *      _print: common code for printf, fprintf, sprintf
 */

#include <stdio.h>
#include <ctype.h>
#include <varargs.h>
#include "param.h"

#define max(a,b) ((a) > (b)? (a): (b))
#define min(a,b) ((a) < (b)? (a): (b))

/*
 *      The following two variables are set by our caller.
 *      and used in emitchar.  The convention is that if _pfile
 *      is not NULL, it should be the recipient of our output.
 *      if _pfile is NULL, then the output will be placed in storage
 *      starting at *_pstring.
 */
FILE *_pfile;
char *_pstring;

/*
 *      System-supplied routines for floating conversion
 */
char *fcvt();
char *ecvt();

/* This variable counts output characters. */
static int count;

int _print (format, args)
        char *format;
        va_list *args;
{
        /* Current position in format */
        char *cp;

        /* Starting and ending points for value to be printed */
        char *bp, *p;

        /* Field width and precision */
        int width, prec;

        /* Format code */
        char fcode;

        /* Number of padding zeroes required on the left */
        int lzero;

        /* Number of padding zeroes required on the right */
        int rzero;

        /* Flags - nonzero if corresponding character appears in format */
        bool length;                    /* l */
        bool fplus;             /* + */
```

```
bool fminus;            /* - */
bool fblank;            /* blank */
bool fsharp;            /* # */

/* Values are developed in this buffer */
char buf[max (MAXDIGS, max (MAXFCVT + MAXEXP, MAXECVT) + 1)];

/* Pointer to sign, "0x", "0X", or empty */
char *prefix;

/* Exponent or empty */
char *suffix;

/* Buffer to create exponent */
char expbuf[MAXESIZ + 1];

/* The value being converted, if integer */
long val;

/* The value being converted, if real */
double dval;

/* Output values from fcvt and ecvt */
int decpt, sign;

/* Set to point to a translate table for digits of whatever radix */
char *tab;

/* Work variables */
int k, n, hradix, lowbit;

cp = format;
count = 0;

/*
 *      The main loop - this loop goes through one iteration
 *      for each ordinary character or format specification.
 */
while (*cp)
        if (*cp != '%') {
                /* Ordinary (non-%) character */
                emitchar (*cp++);
        } else {
                /*
                 *      % has been found.
                 *      First, parse the format specification.
                 */

                /* Scan the <flags> */
                fplus = fminus = fblank = fsharp = 0;
        scan:   switch (*++cp) {
                case '+':
                        fplus = 1;
                        goto scan;
                case '-':
                        fminus = 1;
```

```
        goto scan;
case ' ':
        fblank = 1;
        goto scan;
case '#':
        fsharp = 1;
        goto scan;
}

/* Scan the field width */
if (*cp == '*') {
        width = va_arg (*args, int);
        if (width < 0) {
                width = -width;
                fminus = 1;
        }
        cp++;
} else {
        width = 0;
        while (isdigit (*cp)) {
                n = tonumber (*cp++);
                width = width * 10 + n;
        }
}

/* Scan the precision */
if (*cp == '.') {

        /* '*' instead of digits? */
        if (*++cp == '*') {
                prec = va_arg (*args, int);
                cp++;
        } else {
                prec = 0;
                while (isdigit (*cp)) {
                        n = tonumber (*cp++);
                        prec = prec * 10 + n;
                }
        }
} else
        prec = -1;

/* Scan the length modifier */
length = 0;
switch (*cp) {
case 'l':
        length = 1;
        /* No break */
case 'h':
        cp++;
}

/*
 *      The character addressed by cp must be the
 *      format letter - there is nothing left for
 *      it to be.
```

```
*
*       The status of the +, -, #, and blank
*       flags are reflected in the variables
*       "fplus", "fminus", "fsharp", and "fblank".
*       "width" and "prec" contain numbers
*       corresponding to the digit strings
*       before and after the decimal point,
*       respectively. If there was no decimal
*       point, "prec" is -1.
*
*       The following switch sets things up
*       for printing.  What ultimately gets
*       printed will be padding blanks, a prefix,
*       left padding zeroes, a value, right padding
*       zeroes, a suffix, and more padding
*       blanks.  Padding blanks will not appear
*       simultaneously on both the left and the
*       right.  Each case in this switch will
*       compute the value, and leave in several
*       variables the information necessary to
*       construct what is to be printed.
*
*       The prefix is a sign, a blank, "0x", "0X",
*       or null, and is addressed by "prefix".
*
*       The suffix is either null or an exponent,
*       and is addressed by "suffix".
*
*       The value to be printed starts at "bp"
*       and continues up to and not including "p".
*
*       "lzero" and "rzero" will contain the number
*       of padding zeroes required on the left
*       and right, respectively.  If either of
*       these variables is negative, it will be
*       treated as if it were zero.
*
*       The number of padding blanks, and whether
*       they go on the left or the right, will be
*       computed on exit from the switch.
*/

prefix = suffix = "";
lzero = rzero = 0;
```

```
switch (fcode = *cp++) {

/*
 *      fixed point representations
 *
 *      "hradix" is half the radix for the conversion.
 *      Conversion is unsigned unless fcode is 'd'.
 *      HIBIT is 1000...000 binary, and is equal to
 *              the maximum negative number.
 *      We assume a 2's complement machine
 */

case 'd':
case 'u':
        hradix = 5;
        goto fixed;

case 'o':
        hradix = 4;
        goto fixed;

case 'X':
case 'x':
        hradix = 8;

fixed:
        /* Establish default precision */
        if (prec < 0)
                prec = 1;

        /* Fetch the argument to be printed */
        if (length)
                val = va_arg (*args, long);
        else if (fcode == 'd')
                val = va_arg (*args, int);
        else
                val = va_arg (*args, unsigned);

        /* If signed conversion, establish sign */
        if (fcode == 'd') {
                if (val < 0) {
                        prefix = "-";
                        /*
                         *      Negate, checking in
                         *      advance for possible
                         *      overflow.
                         */
                        if (val != HIBIT)
                                val = -val;
                } else if (fplus)
                        prefix = "+";
                else if (fblank)
                        prefix = " ";
        }

        /* Set translate table for digits */
```

```
        if (fcode == 'X')
                tab = "0123456789ABCDEF";
        else
                tab = "0123456789abcdef";

        /* Develop the digits of the value */
        p = bp = buf + MAXDIGS;
        while (val) {
                lowbit = val & 1;
                val = (val >> 1) & ~HIBIT;
                *--bp = tab[val % hradix * 2 + lowbit];
                val /= hradix;
        }

        /* Calculate padding zero requirement */
        lzero = bp - p + prec;

        /* Handle the # flag */
        if (fsharp && bp != p)
                switch (fcode) {
                case 'o':
                        if (lzero < 1)
                                lzero = 1;
                        break;
                case 'x':
                        prefix = "0x";
                        break;
                case 'X':
                        prefix = "0X";
                        break;
                }

        break;

case 'E':
case 'e':
        /*
         *      E-format. The general strategy
         *      here is fairly easy: we take
         *      what ecvt gives us and re-format it.
         */

        /* Establish default precision */
        if (prec < 0)
                prec = 6;

        /* Fetch the value */
        dval = va_arg (*args, double);

        /* Develop the mantissa */
        bp = ecvt (dval,
                min (prec + 1, MAXECVT),
                &decpt,
                &sign);

        /* Determine the prefix */
```

```
e_merge:
        if (sign)
                prefix = "-";
        else if (fplus)
                prefix = "+";
        else if (fblank)
                prefix = " ";

        /* Place the first digit in the buffer */
        p = &buf[0];
        *p++ = *bp != '\0'? *bp++: '0';

        /* Put in a decimal point if needed */
        if (prec != 0 || fsharp)
                *p++ = '.';

        /* Create the rest of the mantissa */
        rzero = prec;
        while (rzero > 0 && *bp!= '\0') {
                --rzero;
                *p++ = *bp++;
        }

        bp = &buf[0];

        /* Create the exponent */
        suffix = &expbuf[MAXESIZ];
        *suffix = '\0';
        if (dval != 0) {
                n = decpt - 1;
                if (n < 0)
                        n = -n;
                while (n != 0) {
                        *--suffix = todigit (n % 10);
                        n /= 10;
                }
        }

        /* Prepend leading zeroes to the exponent */
        while (suffix > &expbuf[MAXESIZ - 2])
                *--suffix = '0';

        /* Put in the exponent sign */
        *--suffix = (decpt > 0 || dval == 0)? '+': '-';

        /* Put in the e */
        *--suffix = isupper(fcode)? 'E' : 'e';

        break;

case 'f':
        /*
         *      F-format floating point.  This is
         *      a good deal less simple than E-format.
         *      The overall strategy will be to call
         *      fcvt, reformat its result into buf.
```

```
         *       and calculate how many trailing
         *       zeroes will be required.  There will
         *       never be any leading zeroes needed.
         */

        /* Establish default precision */
        if (prec < 0)
            prec = 6;

        /* Fetch the value */
        dval = va_arg (*args, double);

        /* Do the conversion */
        bp = fcvt (dval,
              min (prec, MAXFCVT),
              &decpt,
              &sign);

        /* Determine the prefix */
f_merge:
        if (sign && decpt > -prec &&
           *bp != '\0' && *bp != '0')
              prefix = "-";
        else if (fplus)
              prefix = "+";
        else if (fblank)
              prefix = " ";

        /* Initialize buffer pointer */
        p = &buf[0];

        /* Emit the digits before the decimal point */
        n = decpt;
        k = 0;
        if (n <= 0)
              *p++ = '0';
        else
              do      if (*bp == '\0' || k >= MAXFSIG)
                          *p++ = '0';
                    else {
                          *p++ = *bp++;
                          ++k;
                    }
              while (--n != 0);
```

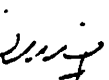
```
        /* Decide whether we need a decimal point */
        if (fsharp || prec > 0)
                *p++ = '.';

        /* Digits (if any) after the decimal point */
        n = min (prec, MAXFCVT);
        rzero = prec - n;
        while (--n >= 0)
                if (++decpt <= 0
                    || *bp == '\0'
                    || k >= MAXFSIG)
                        *p++ = '0';
                else {
                        *p++ = *bp++;
                        ++k;
                }

        bp = &buf[0];

        break;

case 'G':
case 'g':
        /*
         *      g-format.  We play around a bit
         *      and then jump into e or f, as needed.
         */

        /* Establish default precision */
        if (prec < 0)
                prec = 6;

        /* Fetch the value */
        dval = va_arg (*args, double);

        /* Do the conversion */
        bp = ecvt (dval,
                min (prec, MAXECVT),
                &decpt,
                &sign);
        if (dval == 0)
                decpt = 1;

        k = prec;
        if (!fsharp) {
                n = strlen (bp);
                if (n < k)
                        k = n;
                while (k >= 1 && bp[k-1] == '0')
                        --k;
        }

        if (decpt < -3 || decpt > prec) {
                prec = k - 1;
                goto e_merge;
        } else {
```

```
                prec = k - decpt;
                goto f_merge;
            }


        case 'c':
            buf[0] = va_arg (*args, int);
            bp = &buf[0];
            p = bp + 1;
            break;

        case 's':
            bp = va_arg (*args, char *);
            if (prec < 0)
                prec = MAXINT;
            for (n=0; *bp++ != '\0' && n < prec; n++);
            p = --bp;
            bp -= n;
            break;

        case '\0':
            cp--;
            break;

/*      case '%':       */
        default:
            p = bp = &fcode;
            p++;
            break;

        }
        if (fcode != '\0') {
            /* Calculate number of padding blanks */
            int nblank;
            nblank = width
                - (p - bp)
                - (lzero < 0? 0: lzero)
                - (rzero < 0? 0: rzero)
                - strlen (prefix)
                - strlen (suffix);

            /* Blanks on left if required */
            if (!fminus)
                while (--nblank >= 0)
                    emitchar (' ');

            /* Prefix, if any */
            while (*prefix != '\0')
                emitchar (*prefix++);

            /* Zeroes on the left */
            while (--lzero >= 0)
                emitchar ('0');

            /* The value itself */
            while (bp < p)
```

```
                        emitchar (*bp++);

                    /* Zeroes on the right */
                    while (--rzero >= 0)
                        emitchar ('0');

                    /* The suffix */
                    while (*suffix != '\0')
                        emitchar (*suffix++);

                    /* Blanks on the right if required */
                    if (fminus)
                        while (--nblank >= 0)
                            emitchar (' ');
                }
            }

    return (_pfile != NULL && ferror (_pfile))? EOF: count;
}

/* Send a character to the output */
static
emitchar (c)
    char c;
{
    if (_pfile != NULL)
        putc (c, _pfile);
    else
        *_pstring++ = c;
    ++count;
}
```

10. ACKNOWLEDGEMENTS

The formal specification given in section 8 would not have been possible without the help of John Reiser, Dennis Ritchie, and Larry Rosler. These people also produced implementations for the VAX-11, PDP-11, and Honeywell 6000 computers.



Andrew Koenig