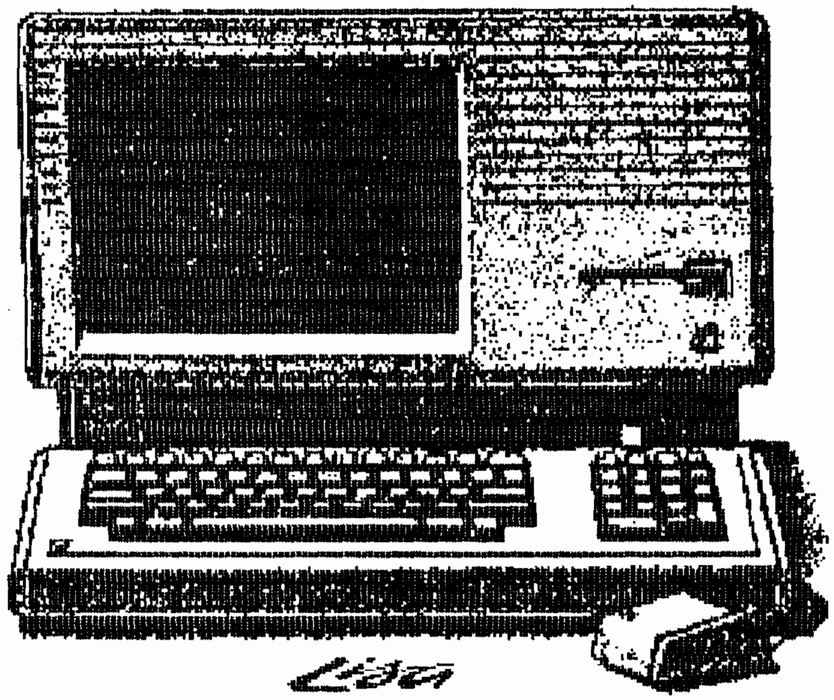
Lisa Pascal 3.0 Systems Software



The Standard Apple Numeric Environment

Contents

The Standard
Apple Numeric Environment

1 Introduction

This manual describes the Standard Apple Numeric Environment (SANE). Apple supports SANE on several current products and plans to support SANE on future products. SANE gives you access to numeric facilities unavailable on almost any computer of the early 1980s--from microcomputers to extremely fast, extremely expensive supercomputers. The core features of SANE are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic [10] as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980s and beyond.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE supports all requirements of the IEEE Standard. SANE goes beyond the specifications of the Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial and scientific calculations.

IEEE arithmetic was specifically designed to provide advanced features for numerical analysts without imposing extra burden on casual users. (This is an admirable but rarely attainable goal: text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to SANE.

2 Data Types

SANE provides three *application* data types (single, double, and comp) and the *arithmetic* type (extended). Single, double, and extended store floating-point values and comp stores integral values.

The *extended* type is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. *Single, double,* and *comp* can be thought of as space-saving storage types for the extended-precision arithmetic. (In this manual, we shall use the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus values can be moved from any of these types to the extended type and back without any loss of information.

2.1 Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- Fixed- or floating-point form,
- Precision,
- Range,
- Memory usage, and
- Speed.

The precision, range, and memory usage for each SANE data type are shown in Table 2-1. Effects of the data types on performance (speed) vary among the implementations of SANE. (See Section 4 for information on conversion problems relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mils, which can be stored exactly in the storage format of the *comp* (for computational) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved by storing numbers in the comp type, which is 20 percent shorter than extended. Nonaccounting applications will normally be better served by the floating-point data formats.

2.2 Values Represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a *sign* (+ or -), an *exponent*, and a *significand*. A represented number has the value

$$\pm\text{significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \le \text{significand} < 2$).

2.3 Range and Precision of SANE Types

This table describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 2-1
SANE Types

Type class	Application			Arithmetic
Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	----	-16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	≈-9.2E18	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm*	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm*	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	≈ 9.2E18	1.1E+4932
Infinities*	Yes	Yes	No	Yes
NaNs*	Yes	Yes	Yes	Yes

* *Denorms* (*denormalized numbers*), *NaNs* (*Not-a-Number*), and *infinities* are defined in Section 7.

Usually numbers are stored in a *normalized* form, to afford maximum precision for a given significand width. Maximum precision is achieved if the high order bit in the significand is 1 (that is, $1 \leq$ significand < 2).

Example

In Single, the largest representable number has

significand	=	$2 - 2^{-23}$
	=	$1.11111111111111111111111_2$
exponent	=	127
value	=	$(2 - 2^{-23}) * 2^{127}$
	≈	$3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	=	1
	=	$1.00000000000000000000000_2$
exponent	=	-126
value	=	$1 * 2^{-126}$
	≈	$1.175 * 10^{-38}$

and the smallest representable positive denormalized number (see Section 7) has

significand	=	2^{-23}
	=	$0.00000000000000000000001_2$
exponent	=	-126
value	=	$2^{-23} * 2^{-126}$
	≈	$1.401 * 10^{-45}$

2.4 Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Single

A 32-bit single format number is divided into three fields as shown below.

```
 1     8                          23                      widths
----------------------------------------------------------
|s|    e      |                   f                      |
----------------------------------------------------------
   msb     lsb msb                                    lsb      order
```

The value v of the number is determined by these fields as follows:

if $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$;

if $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$;

if $e = 0$ and $f = 0$, then $v = (-1)^s * 0$;

if $e = 255$ and $f = 0$, then $v = (-1)^s * \infty$;

if $e = 255$ and $f \neq 0$, then v is a NaN.

See Section 7 for information on the contents of the f field for NaNs.

Double
A 64-bit double format number is divided into three fields as shown below.

```
 1    11                   52                       widths
-------------------------------------------------
|s|    e     |                f                 |
-------------------------------------------------
  msb      lsb msb                          lsb     order
```

The value v of the number is determined by these fields as follows:

if $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$;

if $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$;

if $e = 0$ and $f = 0$, then $v = (-1)^s * 0$;

if $e = 2047$ and $f = 0$, then $v = (-1)^s * \infty$;

if $e = 2047$ and $f \neq 0$, then v is a NaN.

Comp
A 64-bit comp format number is divided into two fields as shown below.

```
 1           63              widths
------------------------------
|s|          d               |
------------------------------
  msb                   lsb  order
```

The value v of the number is determined by these fields as follows:

if $s = 1$ and $d = 0$, then v is the unique comp NaN;

otherwise, v is the two's-complement value of the 64-bit representation.

Extended
An 80-bit extended format number is divided into four fields as shown below.

```
 1    15     1                63                    widths
-----------------------------------------------------
|s|    e    |i|                f                  |
-----------------------------------------------------
  msb     lsb  msb                            lsb    order
```

The value v of the number is determined by these fields as follows:

if $0 <= e < 32767$, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$;

if $e = 32767$ and $f = 0$, then $v = (-1)^s * \infty$, regardless of i;

if $e = 32767$ and $f \neq 0$, then v is a NaN, regardless of i.

3 Arithmetic Operations

SANE provides the basic arithmetic operations for the SANE data types:

- Add.
- Subtract.
- Multiply.
- Divide.
- Square root.
- Remainder.
- Round to integral value.

All the basic arithmetic operations produce the best possible result: The mathematically exact result coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Section 8). See Sections 9 and 10 for auxiliary operations and higher-level functions supported by SANE.

3.1 Remainder

Generally, remainder (and mod) functions are defined by the expression

```
x rem y = x - y * n
```

where n is some integral approximation to the quotient x/y. This expression can be found even in the conventional integer-division algorithm:

```
                  _n_______   (integral quotient approximation)
(divisor)      y) x           (dividend)
                      y * n
                  ---------
                  x - y * n   (remainder)
```

SANE supports the remainder function specified in the IEEE Standard:

When y ≠ 0, the remainder r = x rem y is defined regardless of the rounding direction by the mathematical relation r = x - y * n, where n is the integral value nearest the exact value x/y; whenever |n - x/y| = 1/2, n is even. The remainder is always exact. If r = 0, its sign is that of x.

Example 1

Find 5 rem 3 . Here x = 5 and y = 3. Since 1 < 5/3 < 2 and since 5/3 = 1.66666... is closer to 2 than to 1, n is taken to be 2, so

```
   5 rem 3  =  r  =  5 - 3 * 2  =  -1
```

Example 2

Find 7.0 rem 0.4 . Since 17 < 7.0/0.4 < 18 and since 7.0/0.4 = 17.5 is equally close to both 17 and 18, n is taken to be the even quotient, 18. Hence,

```
   7.0 rem 0.4  =  r  =  7.0 - 0.4 * 18  =  -0.2
```

The IEEE remainder function differs from other commonly used remainder and mod functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. All the other remainder functions can be constructed from the IEEE remainder.

3.2 Round to Integral Value

An input argument is rounded according to the current rounding direction to an integral value and delivered to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0. (The rounding direction, which can be set by the user, is explained fully in Section 8.)

Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, numbers whose magnitudes are at least 223 are integral.

4 Conversions

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation will provide conversions between extended and those numeric types supported in its particular larger environment. For example, a Pascal implementation will have conversions between extended and the Pascal integer type.

```
--------                    ----------------
|single|________----------______|system-specific|
|double|        |extended|      |    integral    |
|comp  |        ----------      |     types      |
--------                    ----------------
```

SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may be included too.

```
                                ----------------
----------______________________|decimal string|
|single  |                      ----------------
|double  |                             |
|comp    |                             |
|extended|______________________----------------
----------                      |decimal record|
                                ----------------
```

4.1 Conversions between Extended and Single or Double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Section 8 for a discussion of exception flags.)

4.2 Conversions to Comp and Other Integral Formats

Conversions to integral formats are done by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an infinity, or out-of-range for the comp format, then the result is the comp NaN and for infinities and values out-of-range, the invalid exception is signaled. If the source operand of a conversion to a system-specific integer type is a NaN, infinity, or out-of-range for that format, then invalid is signaled (unless the type has an appropriate representation for the exceptional result). NaNs, infinities, and out-of-range values are stored in a two's-complement integer format as the extreme negative value (for example, in the 16-bit integer format, as -32768).

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Section 8.)

4.3 Conversions between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10}, or $1/2_{10}$, can be represented exactly as 0.1_2. On the other hand, 0.1_{10}, or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100...._2$. Its closest representation in single is $0.00011001100110011001100110 1_2$, which is closer to 0.10000000149_{10} than to 0.10000000000_{10}.

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversions should obtain this best value to reduce accumulated errors. Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that although in extreme cases the conversions do not deliver the correctly rounded result, the result delivered is very nearly as good as the correctly rounded result. (See the IEEE Standard [10] for a more detailed description of error bounds.)

4.3.1 Conversions from Decimal Strings to SANE Types

Routines may be provided to convert numeric decimal strings to the SANE data types. These routines are provided for the convenience of those who do not wish to write their own parsers and scanners. Examples of acceptable input are

```
123      123.4E-12      -123.     .456     3e9     -0

-INF     Inf      NAN(12)         -NaN()          nan
```

The 12 in NAN(12) is a NaN code (see Section 8).

The accepted syntax is formally defined, using Backus-Naur form, in Table 3-1:

Table 4-1.
Syntax for String Conversions

`<decimal number>`	`::=`	`[{space | tab}] <left decimal>`
`<left decimal>`	`::=`	`[+|-] <unsigned decimal>`
`<unsigned decimal>`	`::=`	`<finite number> | <infinity> | <NAN>`
`<finite number>`	`::=`	`<significand> [<exponent>]`
`<significand>`	`::=`	`<integer> | <mixed>`
`<integer>`	`::=`	`<digits> [.]`
`<digits>`	`::=`	`{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}`
`<mixed>`	`::=`	`[<digits>] . <digits>`
`<exponent>`	`::=`	`E [+|-] <digits>`
`<infinity>`	`::=`	`INF`
`<NAN>`	`::=`	`NAN[([<digits>])]`

Note: Square brackets enclose optional items, curly brackets enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the *E* marking the exponent field, may be either upper or lower case.

4.3.2 Decform Records and Conversions from SANE types to Decimal Strings

Each conversion to a decimal string is controlled by a decform record, which contains two fields:

```
style  -- 16-bit integer (0 or 1)
digits -- 16-bit integer
```

Style equals 0 for floating and 1 for fixed. Digits gives the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style (digits may be negative if the style is fixed). Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. Further formatting details are implementation dependent.

4.3.3 The Decimal Record Type

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

```
sgn -- 16-bit integer (0 or 1)
exp -- 16-bit integer
sig -- string (maximum length is implementation-dependent)
```

The value represented is

$$(-1)^{sgn} * sig * 10^{exp}$$

when the length of sig is 18 or less. (Some implementations allow additional information in characters past the eighteenth.) Sig contains the integral decimal significand: the initial byte of sig (sig[0]) is the length byte, which gives the length of the ASCII string that is left-justified in the remaining bytes. Sgn is 0 for + and 1 for –. For example, if sgn = 1, exp = -3, and sig = '85' (sig[0] = 2, not shown), then the number represented is -0.085.

4.3.4 Conversions from Decimal Records to SANE Types

Conversions from the decimal record type handle any sig digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If sig[1] = '0' (zero), the decimal record is converted to zero. For example, a decimal record with sig = '0913' is converted to zero.
- If sig[1] = 'N', the decimal record is converted to a NaN. Except when the destination is of type comp (which has a unique NaN), the succeeding characters of sig are interpreted as a hex representation of the result significand: if fewer than 4 characters follow N then they are right justified in the high-order 15 bits of the field f illustrated under Formats in Section 2; if 4 or more characters follow N then they are left justified in the result's significand; if no characters, or only 0's, follow N, then the result NaN code is set to nanzero = 15 (hex).
- If sig[1] = 'I' and the destination is not of comp type, the decimal record is converted to an infinity. If the destination is of comp type, the decimal record is converted to a NaN and invalid is signaled.
- Other special cases produce undefined results.

4.3.5 Conversions from SANE Types to Decimal Records

Each conversion to a decimal record is controlled by a decform record (see above). All implementations allow at least 18 digits to be returned in sig. The implied decimal point is at the right end of sig, with exp set accordingly.

Zeroes, infinities, and NaNs are converted to decimal records with sig parts 0 (zero), I, and strings beginning with N, while exp is undefined. For NaNs, N may be followed by a hex representation of the input significand. The third and forth hex digits following N give the NaN code. For example, 'N002100000000000' has NaN code 21 (hex).

When the number of digits specified in a decform record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for sig is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than 2 digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in sig and set exp so that the decimal record contains a valid floating-style representation of the number; alternatively, an implementation may simply set the string sig to '?'. In any implementation, the test

```
(-exp <> decform digits) or (sig[1] = '?')
```

determines whether a nonzero finite number is too large for the chosen fixed style.

4.4 Conversions between Decimal Formats

SANE implementations may provide conversions between decimal strings and decimal records.

4.4.1 Conversion from Decimal Strings to Decimal Records

This conversion routine is intended as an aid to programmers doing their own scanning. The routine is designed for use either with fixed strings or with strings being received (interactively) character by character. An integer argument on input gives the starting index into the string and on output is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed; if no numeric substring is recognized, then the index remains unchanged. Also, a Boolean argument is returned indicating that the input string, beginning at the input index, is a valid numeric string or a valid prefix of a numeric string. The accepted input for this conversion is the same as for conversions from decimal strings to SANE types (see above). Output is the same as for conversions from SANE types to decimal records (also above).

Examples

Input String	Index in	Index out	Output Value	Valid Prefix
12	1	3	12	TRUE
12E	1	3	12	TRUE
12E-	1	3	12	TRUE
12E-3	1	6	12E-3	TRUE
12E-x	1	3	12	FALSE
12E-3x	1	6	12E-3	FALSE
x12E-3	2	7	12E-3	TRUE
IN	1	1	UNDEFINED	TRUE
INF	1	4	INF	TRUE

4.4.2 Conversion from Decimal Records to Decimal Strings

This conversion is controlled by the style field of a decform record (the digits field is ignored). Input is the same as for conversions from decimal records to SANE types, and output formatting is the same as for conversions from SANE types to decimal strings. This conversion, actually a formatting operation, is exact and signals no exception.

5 Expression Evaluation

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

5.1 Using Extended Temporaries

A programmer may use extended temporaries deliberately to reduce the effects of round-off error, overflow, and underflow on the final result.

Example 1

To compute the single-precision sum

```
S  =  X[1]*Y[1] + X[2]*Y[2] + ... + X[N]*Y[N]
```

where X and Y are arrays of type single, declare an extended variable XS and compute

```
XS := 0;
FOR I := 1 TO N DO
  XS := XS + X[I]*Y[I];        {extended-precision arithmetic  }
S := XS;                       {deliver final result to single.}
```

Even when input and output values have only single precision, it may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see Example 2.

5.2 Extended-Precision Expression Evaluation

High-level languages that support SANE evaluate all non-integer numeric expressions to extended precision, regardless of the types of the operands.

Example 2

If C is of type comp and MAXCOMP is the largest comp value, then the right-hand side of

```
   C := (MAXCOMP + MAXCOMP) / 2
```

would be evaluated in extended to the exact result C = MAXCOMP, even though the intermediate result MAXCOMP + MAXCOMP exceeds the largest possible comp value.

5.3 Extended-Precision Expression Evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. Extended evaluation will on rare occasions produce results slightly different from those produced by other IEEE implementations that lack extended evaluation. Thus in a single-only IEEE implementation,

```
   z := x + y
```

with x, y, and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition x + y is performed in extended, then the result is coerced to the single precision of z, with at most two rounding errors. Both implementations conform to the standard.

The effect of a single- or double-only IEEE implementation can be obtained under SANE with rounding precision control, as described in Section 8.

6 Comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

SANE comparisons handle NaNs and infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b, exactly one of the following is true:

a < b
a > b
a = b
a and b are unordered

Determination is made by the rule:

If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that +0 = -0 = real 0, and $-\infty$ < each real number < $+\infty$.

(Note that a NaN always compares unordered--even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal or BASIC expression x <= y is true if x is less than y or if x equal y, and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered--even if not-equal is written <>, as in Pascal and BASIC. High-level languages supporting SANE supplement the usual comparison operators with a function that takes two numeric arguments and returns the appropriate relation (less, equal, greater, or unordered). This function can be used to determine whether two numeric representations satisfy any combination of less, equal, greater, and unordered.

A high-level language comparison that involves a relational operator containing less or greater, but not unordered, signals invalid if the operands are unordered (that is, if either operand is a NaN). For example, in Pascal or BASIC if x or y is a quiet NaN then x < y, x <= y, x >= y, and x > y signal invalid, but x = y and x <> y (recall that <> contains unordered) do not. If a comparison operand is a signaling NaN, then invalid is always signaled, just as in arithmetic operations.

7 Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic also supports infinities, NaNs, and denormalized numbers.

7.1 Infinities

An *infinity* is a special bit pattern that can arise in one of two ways:

- When a SANE operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity bit pattern.
- When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may (depending on the current rounding direction) be an infinity bit pattern.

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive (+INF) and one negative (-INF) , generally behave as suggested by the theory of limits. For example, 1 added to +INF yields +INF; -1 divided by +0 yields -INF; and 1 divided by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for +INF and -INF. The comp type has no representations for infinities. (An infinity moved to the comp type becomes the comp NaN.)

7.2 NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a *NaN* (Not-a-Number). For example, 0 divided by 0, +INF added to -INF, and sqrt(-1) yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (that is, has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: *quiet NaNs*, the usual kind produced by floating-point operations; and *signaling NaNs*. When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and, if no halt occurs, a quiet NaN is the delivered result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations. The most significant bit of the field f illustrated under Formats in Section 2 is clear for quiet NaNs and set for signaling NaNs. The unique comp NaN generally behaves like a quiet NaN.

A NaN in a floating-point format has an associated NaN code that indicates the NaN's origin. (These are listed in Table 7-1). The NaN code is the 8th through 15th most significant bits of the field f illustrated in Section 2. The comp NaN is unique and has no NaN code.

Table 7-1.
SANE NaN Codes

Name	Dec	Hex	Meaning
NANSQRT	1	$01	Invalid square root, such as sqrt(-1)
NANADD	2	$02	Invalid addition, such as (+INF) - (+INF)
NANDIV	4	$04	Invalid division, such as 0/0
NANMUL	8	$08	Invalid multiplication, such as 0 * INF
NANREM	9	$09	Invalid remainder or mod such as x rem 0
NANASCBIN	17	$11	Attempt to convert invalid ASCII string
NANCOMP	20	$14	Result of converting comp NaN to floating
NANZERO	21	$15	Attempt to create a NaN with a zero code
NANTRIG	33	$21	Invalid argument to trig routine
NANINVTRIG	34	$22	Invalid argument to inverse trig routine
NANLOG	36	$24	Invalid argument to log routine
NANPOWER	37	$25	Invalid argument to xi or xy routine
NANFINAN	38	$26	Invalid argument to financial function
NANINIT	255	$FF	Uninitialized storage (signaling NaN)

7.3 Denormalized Numbers

Whenever possible, floating-point numbers are *normalized* to keep the leading significand bit 1; this maximizes the resolution of the storage type. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a *denormalized* representation. A denormalized number is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type.

Example

The sequence below shows how a single-precision value becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called *gradual underflow*.

$$A_0 \quad = 1.100\ 1100\ 1100\ 1100\ 1100\ 1101 * 2^{-126} \approx 0.1_{10} * 2^{-122}$$

$$A_1 = A_0/2 = 0.110\ 0110\ 0110\ 0110\ 0110\ 0110 * 2^{-126} \text{ (underflow)}$$

$$A_2 = A_1/2 = 0.011\ 0011\ 0011\ 0011\ 0011\ 0011 * 2^{-126}$$

$$A_3 = A_2/2 = 0.001\ 1001\ 1001\ 1001\ 1001\ 1010 * 2^{-126} \text{ (underflow)}$$

. . . .

$A_{22} = A_{21}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0011 * 2^{-126}$

$A_{23} = A_{22}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0010 * 2^{-126}$ (underflow)

$A_{24} = A_{23}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 * 2^{-126}$

$A_{25} = A_{24}/2 = 0.0$ (underflow)

$A_1 \ldots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number in single type.

7.3.1 Why Denormalized Numbers?

The use of denormalized numbers makes statements like the following true for all real numbers:

$x - y = 0$ if and only if $x = y$

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For these systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result. For such *flush-to-zero* systems, if $x \neq y$ but $x - y$ is smaller than the smallest normalized number, then $x - y = 0$. IEEE systems do not have this defect, as $x - y$, although denormalized, is not zero.

(A few old programs that rely on premature flushing to zero may require modification to work properly under IEEE arithmetic. For example, some programs may test $x - y = 0$ to determine whether x is very near y.)

7.4 Inquiries: Class and Sign

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- Signaling NaN.
- Quiet NaN.
- Infinite.
- Zero.
- Normalized.
- Denormalized.

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

8 Environmental Control

8.1 Rounding Direction

The available rounding directions are:

- To-nearest.
- Upward.
- Downward.
- Toward-zero.

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. Except for conversions between binary and decimal (described in Section 4), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type, and when the round-to-integer operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integer operation.

8.2 Rounding Precision

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or double. If the SANE user sets rounding precision to single (or double) then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or double does not significantly enhance performance, and in some SANE implementations may hinder performance.

8.3 Exception Flags and Halts

SANE supports five exception flags with corresponding halt settings:

- Invalid-operation (or invalid, for short).
- Underflow.
- Overflow.
- Divide-by-zero.
- Inexact.

These exceptions are signaled when detected; and, if the corresponding halt is enabled, the SANE engine will jump to a user-specified location. (A high-level language need not pass on to its user the facility to set this location, but may halt the user's program). The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings). Further details of the halt (trap) mechanism are SANE implementation specific.

8.3.1 Exceptions

The *invalid-operation* exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions are these:

- (addition or subtraction) magnitude subtraction of infinities, for example, (+INF) + (-INF).
- (multiplication) 0 * INF.
- (division) 0/0 or INF/INF.
- (remainder) x rem y, where y is zero or x is infinite.
- (square root) if the operand is less than zero.
- (conversion) to the comp format or to a system-specific integer format when excessive magnitude, infinity, or NaN precludes a faithful representation in that format (see Section 4 for details).
- (comparison) via predicates involving < or >, but not "unordered," when at least one operand is a NaN.
- Any operation on a signaling NaN except sign manipulations (negate, absolute-value, and copy-sign) and class and sign inquiries.

The *underflow* exception is signaled when a floating-point result is both tiny and inexact (and therefore, perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The *divide-by-zero* exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, logb (0) returns -INF and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

The *overflow* exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

The *inexact* exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, inexact is always signaled in conjunction with overflow or underflow.

Valid operations on infinities are always exact and therefore signal no exceptions. Invalid operations on infinities are described above.

8.4 Managing Environmental Settings

The environmental settings in SANE are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. Often special precautions must be taken because a routine requires certain environment settings, or because a routine's settings are not intended to propagate outside the routine.

Example 1

The subroutine below uses to-nearest rounding while not affecting its caller's rounding direction. (Examples in this section use Pascal syntax. SANE implementations in other languages have operations with equivalent functionality.)

```
                  - - -
var r: RoundDir;        { local storage for rounding direction }
                  - - -
begin
    r := GetRound;          { save caller's rounding direction }
    SetRound (TONEAREST);   {            set to-nearest rounding }
                  - - -
    SetRound (r)        {  restore caller's rounding direction }
end;
```

Note that, if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient functions for managing the environment as a whole: procedure-entry and procedure-exit.

The procedure-entry function returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example 2

The following subroutine runs under the default environment while not affecting its caller's environment.

```
              - - -
   var e: Environment;            { local storage for environment }
              - - -
   begin
       ProcEntry (e);             { save caller's environment and }
                                  { set default environment       }
              - - -
         SetEnvironment (e)         { restore caller's environment  }
end;
```

The procedure-exit function facilitates writing subroutines which appear to their callers to be atomic operations (such as addition, sqrt, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. Procedure-exit, which takes a saved environment as arguments, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (If enabled, halts could occur at this step.)

Thus exceptions signaled between procedure-entry and procedure-exit are hidden from the calling program unless the exceptions remain raised when the procedure-exit function is called.

Example 3

The following function signals underflow if its result is denormal, and overflow if its result is infinite, but hides spurious exceptions occurring from internal computations.

```
function compres: double;
          - - -
var e: Environment;            { local storage for environment }
    c: NumClass;               { for class inquiry             }
          - - -
begin {compres}
   ProcEntry (e);             { save caller's environment and }
                              { set default environment -     }
                              { now halts disabled            }
         - - -
   compres := result;                 { result to be returned }
   c := ClassD (result);              { class inquiry         }
   ClearXcps;            { clear possibly spurious exceptions }

 { now raise specified exception flags:                        }
   if c = INFINITE then SetException (OVERFLOW, TRUE)
   else if c = DENORMALNUM then SetException (UNDERFLOW, TRUE);
    ProcExit (e)             { restore caller's environment,   }
                             { including any halt enables, and }
                             { then signal exceptions from     }
                             { subroutine                      }
end {compres} ;
```

9 Auxiliary Procedures

SANE includes a set of special routines--

negate,
absolute value,
copy-sign,
next-after,
scalb,
logb,

--which are recommended in an appendix to the IEEE Standard as aids to programming.

9.1 Sign Manipulation

The sign manipulation operations change only the sign of their argument. Negate reverses the sign of its argument. Absolute-value makes the sign of its argument positive. Copy-sign takes two arguments and copies the sign of one of its arguments onto the sign of its other argument.

These operations are treated as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

9.2 Next-After Functions

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The next-after functions (one for each of these formats) generate the next representable neighbor in the proper format, given an initial value x and another value y indicating a direction from the initial value.

Each of the next-after functions takes two arguments, x and y:

nextsingle(x,y)	(x and y are single)
nextdouble(x,y)	(x and y are double)
nextextended(x,y)	(x and y are extended)

As elsewhere, the names of the functions may vary with the implementation.

9.2.1 Special Cases for Next-After Functions

If the initial value and the direction value are equal, then the result is the initial value.

If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between -M and +M, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then underflow and inexact are signaled.

9.3 Binary Scale and Log Functions

The scalb and logb functions are provided for manipulating binary exponents.

Scalb efficiently scales a given number (x) by a given integer power (n) of 2, returning $x * 2^n$.

Logb returns the binary exponent of its input argument as a signed integral value. When the input argument is denormalized, the exponent is determined as if the input argument had first been normalized.

9.3.1 Special Cases for Logb

If x is infinite, logb(x) returns +INF.

If x = 0, logb(x) returns -INF and signals divide-by-zero.

10 The Elementary Functions

SANE provides a number of basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator. These functions are computed using the basic SANE arithmetic heretofore described.

All of the elementary functions, except the random number generator, handle NaNs, overflow, and underflow appropriately. All signal inexact appropriately, except that the general exponential and the financial functions may conservatively signal inexact when determining exactness would be too costly.

10.1 Logarithm Functions

SANE provides three logarithm functions.

– base-2 logarithm	:	$\log_2(x)$
– base-e or natural logarithm	:	ln(x)
– base-e logarithm of 1 plus argument	:	ln1(x)

Ln1(x) accurately computes ln(1 + x). If the input argument x is small, such as an interest rate, the computation of ln1(x) is more accurate than the straightforward computation of ln(1 + x) by adding x to 1 and taking the natural logarithm of the result.

10.1.1 Special Cases for Logarithm Functions

If x = +INF, then $\log_2(x)$, ln(x), and ln1(x) return +INF. No exception is signaled.

If x = 0, then $\log_2(x)$ and ln(x) return -INF and signal divide-by-zero. Similarly, if x = -1, then ln1(x) returns -INF and signals divide-by-zero.

If x < 0, then $\log_2(x)$ and ln(x) return a NaN and signal invalid. Similarly, if x < -1, then ln1(x) returns a NaN and signals invalid.

10.2 Exponential Functions

SANE provides five exponential functions.

– base-2 exponential	:	2^x
– base-e or natural exponential	:	e^x
– base-e exponential		

minus 1	:	exp1(x)	
– integer exponential	:	x^i	(i of integer type)
– general exponential	:	x^y	

Exp1(x) accurately computes $e^x - 1$. If the input argument x is small, such as an interest rate, then the computation of exp1(x) is more accurate than the straightforward computation of $e^x - 1$ by exponentiation and subtraction.

10.2.1 Special Cases for 2^x, e^x, exp1(x)

If x = +INF, then 2^x, e^x, and exp1(x) return +INF. No exception is signaled.

If x = -INF, then 2^x and e^x return 0; and exp1(x) returns -1. No exception is signaled.

10.2.2 Special Cases for x^i

If the integer exponent i equals 0 and x is not a NaN, then x^i returns 1. Note that with the integer exponential, $x^0 = 1$ even if x is zero or infinite.

If x is +0 and i is negative, then x^i returns +INF and signals divide-by-zero.

If x is -0 and i is negative, then x^i returns +INF if i is even, or -INF if i is odd: both cases signal divide-by-zero.

10.2.3 Special Cases for x^y

If x is +0 and y is negative, then the general exponential xy returns +INF and signals divide-by-zero.

If x is -0 and y is integral and negative, then x^y returns +INF if y is even, or -INF if y is odd; both cases signal divide-by-zero.

The general exponential x^y returns a NaN and signals invalid if

both x and y equal 0;

x is infinite and y equals 0;

x = 1 and y is infinite; or

x is -0 or less than 0 and y is nonintegral.

10.3 Financial Functions

SANE provides two functions, compound and annuity, that can be used to solve various financial, or time-value-of-money, problems.

10.3.1 Compound

The compound function computes

$$\text{compound}(r, n) = (1 + r)^n$$

where r is the interest rate and n is the number (perhaps nonintegral) of periods. When the rate r is small, compound gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

Compound is directly applicable to computation of present and future values:

$$PV = FV * (1 + r)^{(-n)} = \frac{PV}{\text{compound}(r, n)}$$

$$FV = PV * (1 + r)^n = PV * \text{compound}(r, n)$$

10.3.2 Special Cases for Compound(r,n)

If r = 0 and n is infinite, or if r = -1, then compound(r,n) returns a NaN and signals invalid.

If r = -1 and n < 0, then compound(r,n) returns +INF and signals divide-by-zero.

10.3.3 Annuity

The annuity function computes

$$\text{annuity}(r, n) = \frac{1 - (1 + r)^{(-n)}}{r}$$

where r is the interest rate and n is the number of periods. Annuity is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The annuity function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV = PMT * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{annuity}(r, n)$$

$$FV = PMT * \frac{(1 + r)^n - 1}{r}$$

$$= PMT * (1 + r)n * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * \text{compound}(r, n) * \text{annuity}(r, n)$$

where PMT is the amount of one periodic payment.

10.3.4 Special Cases for Annuity(r,n)

If r = 0, then annuity(r,n) computes the sum of 1 + 1 + ... + 1 over n periods, and therefore returns the value n and signals no exceptions (the value n corresponds to the limit as r approaches 0).

If r < -1, then annuity(r,n) returns a NaN and signals invalid.

If r = -1 and n > 0, then annuity(r,n) returns -INF and signals divide-by-zero.

10.4 Trigonometric Functions

SANE provides the basic trigonometric functions:

cosine	:	cos(x)
sine	:	sin(x)
tangent	:	tan(x)
arctangent	:	arctan(x)

The arguments for cosine, sine, and tangent and the results of arctangent are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Section 3) and the nearest extended-precision approximation of pi/2. Thus the cosine, sine, and tangent functions have periods slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become large. Number results from arctangent lie between -pi/2 and pi/2.

The remaining trigonometric functions can be easily and efficiently computed from these four (see Appendix C).

10.4.1 Special Cases for sin(x), cos(x):

If x is infinite, then cos(x) and sin(x) return a NaN and signal invalid.

10.4.2 Special Cases for tan(x):

If x is the nearest extended approximation to ±pi/2, then tan(x) returns ±INF.

If x is infinite, then tan(x) returns a NaN and signals invalid.

10.4.3 Special Case for arctan(x):

If x = ±INF, then arctan(x) returns the nearest extended approximation to ±pi/2.

10.5 Random Number Generator

SANE provides a pseudorandom number generator, random. Random has one argument, passed by address. A sequence of (pseudo)random integral values r in the range

$$1 \leq r \leq 2^{31} - 2$$

can be generated by initializing an extended variable r to an integral value (the seed) in the above range and making repeated calls random (r); each call delivers in r the next random number in the sequence.

If seed values of r are nonintegral or outside the range

$$1 \leq r \leq 2^{31} - 2$$

then results are unspecified.

A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from random by

$$2^{31} - 1 = \text{scalb}(31, 1) - 1 .$$

Appendix A Bibliography

1. Apple Computer, Inc. "Appendix A: The Transcend and Realmodes Units" and "Appendix E: Floating-Point Arithmetic," *Apple III Pascal Programmer's Manual,* Volume 2, pp. 2-9, 56-85.

   These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

2. Apple Computer, Inc. *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units.*

   This manual describes the Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple's first full implementation of IEEE arithmetic.

3. Apple Computer, Inc. *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units.*

   This manual, generalized from the Apple III manual (number 2 above), describes the Apple II and Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units.

4. Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 63-68.

   This paper compares the several contending proposals presented to the Working Group.

5. Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer,* Vol. 13, No. 1 January 1980.

   This paper is a forerunner to the work on the draft Standard.

6. Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 75-87.

7. Coonen, Jerome T. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in *ACM Transactions on Mathematical Software.*

8. Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in *SIAM Journal on Scientific and Statistical Computing*.

   These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

9. Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, April 1982, pp. 239-257.

   This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

10. Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

    The implementation of SANE is based upon Draft 10.0 of this Standard.

11. Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 51-62.

    This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

12. Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 70-74.

    This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

13. Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard,"*Interval Mathematics 1980* (ed. K.E.L. Nickel). New York: Academic Press, New York, 1980, pp. 99-128.

    This paper shows how the proposed Standard facilitates interval arithmetic.

Appendix B
Glossary

application type: A data type used to store data for applications.

arithmetic type: A data type used to hold results of calculations inside the computer. The SANE arithmetic type, extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

binary floating-point number: A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

comp type: A 64-bit application data type for storing integral values of up to 18- or 19-decimal-digit precision. It is used for accounting applications, among others.

denormalized number, or **denorm:** A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

double type: A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

environmental settings: The rounding direction and rounding precision, plus the exception flags and their respective halts.

exceptions: Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

exception flag: Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

exponent: The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

extended type: An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

halt: Each exception has a halt-enable that can be set or cleared. When an exception is signaled and the corresponding halt is enabled, the SANE engine will transfer control to the address in a halt vector. A high-level language need not pass on to its user the facility to get the halt vector, but may halt the user's program. Halts remain set until explicitly cleared.

infinity: A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

integer types: System types for integral values. Integer types typically use 16- or 32-bit two's complement integers. Integer types are not SANE types but are available to SANE users.

integral value: A value in a SANE type that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not a Number): A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

normalized number: A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

quiet NaN: A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

rounding direction: When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: to-nearest (the default), upward, downward, and toward-zero.

sign bit: The bit of a single, double, comp, or extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

signaling NaN: A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

significand: The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

single type: A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Appendix C
Other Elementary Functions

High quality transcendental functions which are not part of the Standard Apple Numeric Environment (SANE) can be constructed from the functions which SANE provides. Some common functions are provided below in pseudo-code. It should be relatively easy to adapt them for your use.

These functions are based on algorithms developed by Professor William Kahan, University of California at Berkeley. They are robust and accurate. The constant C is 2^{-33} = scalb (-33,1). It is chosen to be nearly the largest value for which $(1 - C^2)$ rounds to 1. All variables are extended.

Exception Handling

Unlike the SANE elementary functions, these functions do not provide complete handling of special-cases and exceptions. The most troublesome exceptions can be correctly handled if you:

- Begin each function with a call to procedure-entry.
- Clear the spurious exceptions indicated.
- End each function with a call to procedure-exit (see Section 8).

Functions

Secant

```
sec(x) <--- 1 / cos(x)
```

CoSecant

```
csc(x) <--- 1 / sin(x)
```

CoTangent

```
cot(x) <--- 1 / tan(x)
```

ArcSine

```
y <--- |x|
If y ≥ 0.3 then begin
                    y <--- Atan (x/sqrt ((1-x)*(1+x)))
                    spurious divide-by-zero may arise
                  end
else if y ≥ C then y <--- Atan (x / (sqrt (1 - x^2))
                else y <--- x
arcsin(x) <--- y
```

ArcCosine

```
arccos(x) <--- 2 * Atan (sqrt ((1-x)/(1+x)) )
spurious divide-by-zero may arise
```

Sinh

```
y <--- |x|
If y ≥ C then begin
                y <--- expl(y)
                y <--- 0.5 * (y + y/(1+y))
              end
copy the sign of x onto y
sinh(x) <--- y
```

Cosh

```
y <--- exp(|x|)
cosh(x) <--- 0.5 * y  +  0.25 / (0.5 * y)
```

Tanh

```
y <--- |x|
If y ≥ C then begin
                y <--- expl(-2*y)
                y <--- -y/(2 + y)
              end
copy the sign of x onto y
tanh(x) <--- y
```

ArcSinh

```
    y <--- |x|
    If y ≥ C then begin
                    y <--- ln1 (y + y / (1/y + sqrt(1 + (1/y)^2) ))
                    spurious underflow may arise
                  end
    copy the sign of x onto y
    asinh(x) <--- y
```

ArcCosh

```
    y <--- |x|
    acosh(x) <--- ln1 ( (sqrt (y-1)) * (sqrt (y-1) + sqrt (y+1)) )
```

ArcTanh

```
    y <--- |x|
    If y ≥ C then y <--- ln1 (2*y/(1 - y)) / 2
    copy the sign of x onto y
    atanh(x) <--- y
```

The 68000 Assembly-Language SANE Engine

Contents

The 68000
Assembly-Language SANE Engine

1 Introduction

The purpose of the software package described in this manual is to provide the features of the Standard Apple Numeric Environment (SANE) to assembly-language programmers on Apple's 68000-based systems. SANE--described in detail in *The Standard Apple Numeric Environment* in this binder--fully supports the IEEE Standard (754) for Binary Floating-Point Arithmetic; it augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

A functionally equivalent 6502 assembly-language SANE engine is available for Apple's 6502-based systems. Thus numerical algorithms coded in assembly language for an Apple 68000-based system can be readily recoded for an Apple 6502-based system. Suggested macros for accessing the 6502 and 68000 engines have been chosen to further facilitate algorithm portability.

This manual describes the use of the 68000 Assembly-Language SANE engine, but does not describe SANE itself. For example, this manual explains how to call the SANE remainder function from 68000 assembly language but does not discuss what this function does. See *The Standard Apple Numeric Environment* T for information about the semantics of SANE.

See Appendix A for information about accessing the 68000 SANE engine from the Apple 68000-based systems.

2 Basics

The following code illustrates a typical invocation of the SANE engine, FP68K.

```
PEA     A_ADR   ; Push address of A (single format)
PEA     B_ADR   ; Push address of B (extended format)
FSUBS           ; Floating-point SUBtract Single: B <— B - A
```

FSUBS is an assembly-language macro taken from the file listed in Appendix B. The form of the operation in the example (B <-- B - A, where A is a numeric type and B is extended) is similar to the forms for most FP68K operations. Also, this example is typical of SANE engine calls because operands are passed to FP68K by pushing the addresses of the operands onto the stack prior to the call. Details of SANE engine access are given later in this section.

The SANE elementary functions are provided in Elems68K. Access to Elems68K is similar to access to FP86K; details are given in Section 9.

2.1 Operation Forms

The example above illustrates the form of an FP68K binary operation. Forms for other FP68K operations are described in this section. Examples and further details are given in subsequent sections.

2.1.1 Arithmetic and Auxiliary Operations

Most numeric operations are either unary (one operand), like square root and negation, or binary (two operands), like addition and multiplication.

The 68000 assembly-language SANE engine, FP68K, provides unary operations in a one-address form:

```
DST <-- <op> DST          ... for example, B <-- sqrt(B)
```

The operation <op> is applied to (or operates on) the operand DST and the result is returned to DST, overwriting the previous value. DST is called the destination operand.

FP68K provides binary operations in a two-address form:

```
DST <-- DST <op> SRC      ... for example, B <-- B / A
```

The operation <op> is applied to the operands DST and SRC and the result is returned to DST, overwriting the previous value. SRC is called the source operand.

In order to store the result of an operation (unary or binary), the location of the operand DST must be known to FP68K, so DST is passed by address to FP68K. In general all operands, source and destination, are passed by address to FP68K.

For most operations the storage format for a source operand (SRC) can be one of the SANE numeric formats (single, double, extended, or comp). To support the extended-based SANE arithmetic, a destination operand (DST) must be in the extended format.

The forms for the copy-sign next-after functions are unusual and will be discussed in Section 4.

2.1.2 Conversions

FP68K provides conversions between the extended format and other SANE formats, between extended and 16- or 32-bit integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, and integer) and conversions from decimal to binary have the form

```
DST <-- SRC
```

Conversions from binary to decimal have the form

```
DST <-- SRC according to SRC2
```

where SRC2 is a DecForm record specifying the decimal format for the conversion of SRC to DST.

2.1.3 Comparisons

Comparisons have the form

```
<relation> <-- SRC, DST
```

where DST is extended and SRC is single, double, comp, or extended, and where <relation> is less, equal, greater, or unordered according as

```
DST <relation> SRC
```

Here the result <relation> is indicated by setting the 68000 CCR flags.

2.1.4 Other Operations

FP68K provides inquiries for determining the class and sign of an operand and operations for accessing the floating-point environment word and the halt address. Forms for these operations vary and will be given as the operations are introduced.

2.2 External Access

The SANE engine, FP68K, is reentrant, position-independent code, which may be shared in multi-process environments. It is accessed through one entry point, labeled FP68K. Each user process has a static state area consisting of one word of mode bits and error flags, and a two-word halt vector. The package allows for different access to the state word in single and multi-process environments.

The package preserves all 68000 registers across invocations, except that REMAINDER modifies D0. The package modifies the 68000 CCR flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit 68000 registers. Since the binary-decimal conversions themselves call the package (to perform multiplies and divides), they use about twice the stack space of the regular operations.

The access constraints described in this section also apply to Elems68K.

2.3 Calling Sequence

A typical invocation of the engine consists of a sequence of PEA's to push operand addresses followed by one of the Appendix B macros:

```
PEA     <source address>
PEA     <destination address>
<FOPMACRO>
```

PEA's for source operands always precede those for destination operands.

<FOPMACRO> represents a typical operation macro defined as

```
MOVE.W  <opword>,-(SP)          ; Push op code.
JSRFP
```

The macro JSRFP in turn generates a call to FP68K; for Macintosh, it expands to an A-line trap, while for Lisa it expands to an intrinsic unit subroutine call

```
JSR     FP68K
```

2.3.1 The Opword

The opword is the logical OR of a operand format code and an operation code.

The operand format code specifies the format (extended, double, single, integer, or comp) of one of the operands. The operand format code typically gives the format for the source operand (SRC). At most one operand format need be specified, since other operands' formats are implied.

The operation code specifies the operation to be performed by FP68K.

Opwords are listed in Appendix C; operand format codes and operation codes are listed in Appendix B.

Example

The format code for single is 0200 (hex). The operation code for divide is 0006 (hex). Hence the opword 0206 (hex) indicates divide by a value of type single.

2.3.2 Assembly-Language Macros

The macro file in Appendix B provides macros for

```
MOVE.W  <opword>,-(SP)
JSRFP
```

for most common <opword> calls to FP68K.

Example 1

To add a single-format operand A to an extended-format operand B, simply write:

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FADDS           ; Floating-point ADD Single: B <— B + A
```

Example 2

Compute B <-- sqrt(A), where A and B are extended. The value of A should be preserved.

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FX2X            ; Floating-point eXtended to eXtended: B <— A
PEA     B_ADR   ; Push address of B
FSQRTX          ; Floating SQuare RooT eXtended: B <— sqrt(B)
```

Example 3

Compute C <-- A - B, where A, B, and C are in the double format. Since destinations are extended, a temporary extended variable T is required.

```
PEA     A_ADR   ; Push address of A
PEA     T_ADR   ; Push address of 10-byte temporary variable
FD2X            ; Fl-pt convert Double to eXtended: T <— A
PEA     B_ADR   ; Push address of B
PEA     T_ADR   ; Push address of temporary
FSUBD           ; Fl-pt SUBtract Double: T <— T - B
PEA     T_ADR   ; Push address of temporary
PEA     C_ADR   ; Push address of C
FX2D            ; Fl-pt convert eXtended to Double: C <—
```

2.4 Arithmetic Abuse

FP68K is designed to be as robust as possible, but it is not bulletproof. Passing the wrong number of operands to the engine will damage the stack. Using UNDEFINED opword parameters or passing incorrect addresses will produce undefined results.

3 Data Types

FP68K fully supports the SANE data types

```
single    -- 32-bit floating-point
double    -- 64-bit floating-point
comp      -- 64-bit integer
extended  -- 80-bit floating-point
```

and the 68000-specific types

```
integer   -- 16-bit two's complement integer
longint   -- 32-bit two's complement integer
```

The 68000 engine uses the convention that least-significant bytes are stored in high memory. For example, let us take a variable of type single with bits

```
s              -- sign
e0 ... e7      -- exponent (msb...lsb)
f0 ... f22     -- significand fraction (msb...lsb)
```

The logical structure of this four-byte variable is shown below:

```
  msb          lsb msb                                         lsb  order
|---------------|---------------|---------------|---------------|
|s|e| | | | | | |e|f| | | | | | | | | | | | | | | | | | | | | |f|
| |0| | | | | | |0|0| | | | | | | | | | | | | | | | | | | | | |2|
| |0| | | | | | |7|0| | | | | | | | | | | | | | | | | | | | | |2|
|---------------|---------------|---------------|---------------|
   1000            1001            1002            1003
```

If this variable is assigned the address 1000, then its bits are distributed to the locations 1000 to 1003 as shown. The other SANE formats (see Section 2 in *The Standard Apple Numeric Environment*) are represented in memory in similar fashion.

4 Arithmetic Operations and Auxiliary Routines

The operations covered in this section follow the access schemes described in Section 2.

```
unary operations:  DST <-- <op> DST          (one-address form)

    PEA      <DST address>
    <FOPMACRO>

binary operations:  DST <-- DST <op> SRC     (two-address form)

    PEA      <SRC address>
    PEA      <DST address>
    <FOPMACRO>
```

The destination operand (DST) for these operations is passed by address and is generally in the extended format. The source operand (SRC) is also passed by address and may be single, double, comp, or extended. Some operations are distinguished by requiring some specific type for SRC, by using a nonextended destination, or by returning auxiliary information in the D0 register and in the processor CCR status bits. In this section, operations so distinguished are noted. The examples employ the macros in Appendix B.

4.1 Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

B <-- B / A , where A is double and B is extended:

```
PEA      A_ADR    ; push address of A
PEA      B_ADR    ; push address of B
FDIVD             ; divide with source operand of type double
```

4.2 Square Root

This is a unary operation and follows the one-address form.

Example

B <-- sqrt(B) , where B is extended.

```
PEA      B_ADR    ; push address of B
FSQRTX            ; square root (operand is always extended)
```

4.3 Round-to-Integer, Truncate-to-Integer

These are unary operations and follow the one-address form.

Round-to-integer rounds (according to the current rounding direction) to an integral value in the extended format. Truncate-to-integer rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operators, illustrated above for square root.

4.4 Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information: the low-order integer quotient (between -127 and +127) in D0.W. The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction--the principal use of the remainder function. The state of D0 after an invalid remainder is undefined.

Example

B <-- B rem A , where A is single and B is extended.

```
PEA      A_ADR   ; push address of A
PEA      B_ADR   ; push address of B
FREMS            ; remainder with source operand of type single
```

4.5 Logb, Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its source operand is a 16-bit integer.

Example

B <-- B * 2^I, where B is extended.

```
PEA      I_ADR   ; push address of I
PEA      B_ADR   ; push address of B
FSCALBX          ; scalb
```

4.6 Negate, Absolute Value, Copy-Sign

Negate and absolute value are unary operations and follow the one-address form.

Copy-sign uses the calling sequence

```
PEA      <SRC address>
PEA      <DST address>
FCPYSGNX
```

to copy the sign of DST onto the sign of SRC. Note that copy-sign differs from most two-address operations in that it changes the SRC value rather than the DST value. The formats of the operands for FCPYSGNX can be single, double, or extended. (For efficiency, the 68000 assembly-language programmer should copy signs directly rather than call FP68K.)

Example

Copy the sign of B (single, double, or extended) into the sign of A (single, double, or extended).

```
PEA      A_ADR   ; push address of A
PEA      B_ADR   ; push address of B
FCPYSGNX         ; copy-sign
```

4.7 Next-After

The next-after operations use the calling sequence

```
PEA      <SRC address>
PEA      <DST address>
<next-after macro>
```

to effect SRC <-- next value, in the format indicated by the macro, after SRC in the dirction of DST. Next-after operations differ from most two-address operations in that they change SRC values rather than DST values. Both source and destination operands must be of the same floating-point type (single, double, or extended).

Example

A <-- next-after(A) in the direction of B, where A and B are double (so *next-after* means *next-double-after*).

```
PEA      A_ADR   ; push address of A
PEA      B_ADR   ; push address of B
FNEXTD           ; next-after in double format
```

5 Conversions

This section discusses conversions between binary formats and conversions between binary and decimal formats.

5.1 Conversions Between Binary Formats

FP68K provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16- and 32-bit integer types.

5.1.1 Conversions to Extended

FP68K provides conversions of a source, of type single, double, comp, extended, or integer, to an extended destination.

```
                          single
                          double
    extended      <--     comp
                          extended
                          integer
```

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B, where A is of type comp and B is extended.

```
    PEA     A_ADR   ; push address of A
    PEA     B_ADR   ; push address of B
    FC2X            ; convert comp to extended
```

5.1.2 Conversions from Extended

FP68K provides conversions of an extended source to a destination of type single, double, comp, extended, or integer.

```
    single
    double
    comp          <--     extended
    extended
    integer
```

(Conversion to a narrower format may alter values.) Contrary to the usual scheme the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B where A is extended and B is double.

```
PEA      A_ADR    ; push address of A
PEA      B_ADR    ; push address of B
FX2D              ; convert extended to double
```

5.2 Binary-Decimal Conversions

FP68K provides conversions between the binary types (single, double, comp, extended, and integer) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Section 4 of *The Standard Apple Numeric Environment*. For FP68K, the maximum length of the sig digits field of a decimal record is 20. (The value 20 is specific to this implementation: algorithms intended to port to other SANE implementations should use no more than 18 digits in sig.)

5.2.1 Binary to Decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination. The maximum number of significant digits that will be returned is 19.

Example

Convert a comp-format value A to a decimal record D according to the decform record F.

```
PEA      F_ADR    ; push address of F
PEA      A_ADR    ; push address of A
PEA      D_ADR    ; push address of D
FC2DEC            ; convert comp to decimal
```

Fixed-Format "Overflow"

If a number is too large for a chosen fixed style, then FP68K returns the string '?' in the sig field of the decimal record.

5.2.2 Decimal to Binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

The maximum number of digits in sig is 19. If the length of sig is 20, then sig represents its first 19 digits plus one or more additional nonzero digits after the 19th. The exponent corresponds to the 19-digit integer represented by the first 19 digits of sig.

Example

Convert the decimal record D to a double-format value B.

```
PEA     D_ADR   ; push address of D
PEA     B_ADR   ; push address of B
FDEC2D          ; convert decimal to double
```

Techniques for Extreme Accuracy

The following techniques apply to FP68K; other SANE implementations require other techniques.

For maximum accuracy, insert or delete trailing zeros for the sig field of a decimal record in order to minimize the magnitude of the exp field. For example, for 1.0E60 set sig to '100000000000000000000000000' (17 zeros) and exp to 43, and for 300E-43 set sig to '3' and exp to -41.

If you are writing a parser and must handle a number with more than 19 significant digits, follow these rules:

- Place the implicit decimal point to the right of the 19 most significant digits.
- If any of the discarded digits to the right of the implicit decimal point are nonzero, then concatenate the digit '1' to sig.

6 Comparisons and Inquiries

6.1 Comparisons

FP68K offers two comparison operations: FCPX (which signals invalid if its operands compare unordered) and FCMP (which does not). Each compares a source operand (which may be single, double, extended, or comp) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

```
DST <relation> SRC
```

is true. The result is delivered in the X, N, Z, V, and C status bits:

<relation>	Status bits X N Z V C
greater	0 0 0 0 0
less	1 1 0 0 1
equal	0 0 1 0 0
unordered	0 0 0 1 0

These status bit encodings reflect that floating-point comparisons have four possible results, unlike the more familiar integer comparisons with three possible results. It's not necessary to learn these encodings, however; simply use the FBxxx series of macros for branching after FCMP and FCPX.

FCMP and FCPX are both provided to facilitate implementation of relational operators defined by higher level languages that do not contemplate unordered comparisons. The IEEE standard specifies that the invalid exception shall be signalled whenever necessary to alert users of such languages that an unordered comparison may have adversely affected their program's logic.

Example 1

Test B <= A, where A is single and B is extended; if TRUE branch to LOC; signal if unordered.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FCPXS           ; compare using source of type single,
                ; signal invalid if unordered
FBLE    LOC     ; branch if B <= A
```

Example 2

Test B not-equal A, where A is double and B is extended; if TRUE branch to LOC. (Note that not-equal is equivalent to less, greater, or unordered, so invalid should not be signaled on unordered.)

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FCMPD           ; compare using source of type double,
                ; do not signal invalid if unordered
FBNE    LOC     ; branch if B not-equal A
```

6.2 Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, or extended), which is passed by address, and places the result in a 16-bit integer destination.

The sign of the result is the sign of the source; the magnitude of the result is

1	signaling NaN
2	quiet NaN
3	infinite
4	zero
5	normal
6	denormal

Example

Set C to sign and class of A.

```
PEA     A_ADR   ; push address of A
PEA     C_ADR   ; push address of result
FCLASSS         ; classify single
```

7 Environmental Control

7.1 The Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:

```
msb                                                          lsb
|-------------------------------|-------------------------------|
| - | r | r | x | d | o | u | i | - | R | R | X | D | O | U | I |
|-------------------------------|-------------------------------|
rounding        exception       rounding           halt
direction         flags         precision          enables
```

rounding direction, bits 6000		rr
0000	-- to-nearest	
2000	-- upward	
4000	-- downward	
6000	-- toward-zero	
exception flags, bits 1F00		
0100	-- invalid	i
0200	-- underflow	u
0400	-- overflow	o
0800	-- division-by-zero	d
1000	-- inexact	x
rounding precision, bits 0060		RR
0000	-- extended	
0020	-- double	
0040	-- single	
0060	-- UNDEFINED	
halt enabled, bits 001F		
0001	-- invalid	I
0002	-- underflow	U
0004	-- overflow	O
0008	-- division-by-zero	D
0010	-- inexact	X

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the environment is 72 and the least significant byte is 0D.

Access to the environment is via the operations get-environment, set-environment, test-exception, set-exception, procedure-entry, and procedure-exit.

7.2 Get-Environment and Set-Environment

Get-Environment takes one input operand: the address of a 16-bit integer destination. The environment word is returned in the destination.

Set-Environment has one input operand: the address of a 16-bit integer, which is to be interpreted as an environment word.

Example

Set rounding direction to toward-zero.

```
PEA     A_ADR
FGETENV
MOVE.W  (A0),D0         ; D0 gets environment
OR.W    #$6000,D0       ; set rounding toward-zero
MOVE.W  D0,(A0)         ; restore A
PEA     A_ADR
FSETENV
```

7.3 Test-Exception and Set-Exception

Test-exception has one integer destination operand, which contains the hex values

```
01 -- invalid
02 -- underflow
04 -- overflow
08 -- divide-by-zero
 10 -- inexact
```

If the exception flag is set for the corresponding bit in the operand, then test-exception sets the destination to $100, otherwise, to zero.

Set-exception takes one integer source operand, which encodes an exception in the manner described above for test-exception. Set-exception stimulates the exception indicated in the operand.

7.4 Procedure-Entry and Procedure-Exit

Procedure-entry saves the current floating-point environment (16-bit integer) at the address passed as the sole operand, and sets the operative environment to the default state.

Procedure-exit saves (temporarily) the exception flags, sets the environment passed as the sole operand, and then stimulates the saved exceptions.

Example

Here is a procedure that appears to its callers as an atomic operation.

```
   ATOMICPROC
            PEA     E_ADR    ; push address to store environment
            FPPROCENTRY      ; procedure entry

            ...body of routine...

            PEA     E_ADR    ; push address of environment
            FPPROCEXIT       ; procedure exit
   RTS
```

8 Halts

FP68K provides the facility to transfer program control when selected floating-point exceptions occur. Since this facility will be used to implement halts in high-level languages, we refer to it as a halting mechanism. The assembly-language programmer can write a 'halt handler' routine to cause special actions for floating-point exceptions. The FP68K halting mechanism differs from the traps that are an optional part of the IEEE Standard.

8.1 Conditions for a Halt

Any floating-point exception can, under the appropriate conditions, trigger a halt. The halt for a particular exception is enabled when the user has set the halt-enable bit corresponding to that exception.

8.2 The Halt Mechanism

If the halt for a given exception is enabled, FP68K does these things when that exception occurs:

1. FP68K returns the same result to the destination address that it would return if the halt were not enabled.
2. It sets up the following stack frame:

top-of-stack -->	☐	A word containing the opcode.
	☐	A long word containing DST address.
	☐	A long word containing SRC address.
	☐	A long word containing SRC2 address.
	☐	A long word pointing to MISC.

MISC is a record consisting of:

MISC:	☐	A word containing halt exceptions.
	☐	A word containing pending CCR.
	☐	A long word containing pending D0.

The first word of MISC contains in its five low-order bits the AND of the halt-enable bits with the exceptions that occurred in the operation just completing. If halts were not enabled, then (upon return from FP68K) CCR and D0 would have the values given in MISC.

3. It passes control by JSR through the halt vector previously set by FSETHV, pushing another long word containing a return address in FP68K. If execution is to continue, the halt procedure must clear eighteen bytes from the stack to remove the opword and the DST, SRC, SRC2, and MISC addresses.

Set-halt-vector has one input operand: the address of a 32-bit integer, which is interpreted as the halt vector (that is, the address to jump to in case a halt occurs).

Get-halt-vector has one input operand: the address of a 32-bit integer, which receives the halt vector.

8.3 Using the Halt Mechanism

This example illustrates the use of the halting mechanism. The user must set the halt vector to the starting address of a halt handler routine. This particular halt handler returns control to FP68K which will continue as if no halt had occurred, returning to the next instruction in the user's program.

```
          LEA       HROUTINE,A0      ; A0 gets address of halt routine
          MOVE.L    A0,H_ADR         ; H_ADR gets same
          PEA       H_ADR            ;
          FSETHV                     ; set halt vector to HROUTINE
          ...
          PEA                        ; floating-point operand here
          <FOPMACRO>                 ; a floating-point call here
          . . .
HROUTINE                             ; called by FP68K
          MOVE.L    (SP)+,A0         ; A0 saves return address in FP68K
          ADD.L     #18,SP           ; increment stack past arguments
          JMP       (A0)             ; return to FP68K
```

The FP68K halt mechanism is designed so that a halt procedure may be written in Lisa Pascal. This is the form of a Pascal equivalent to HROUTINE:

```
type    miscrec = record
                halterrors : integer ;
                ccrpending : integer ;
                DOpending : longint ;
        end {record} ;

procedure haltroutine
        ( var misc : miscrec ;
          src2, src, dst : longint ;
          opcode : integer ) ;

begin {haltroutine}
end {haltroutine} ;
```

Like HROUTINE, haltroutine merely continues execution as if no halt had occurred.

Example

B <-- B^K, where the type of B is extended.

```
   PEA     K_ADR   ; push address of K
   PEA     B_ADR   ; push address of B
   FXPWRI          ; integer exponentiation
```

9.3 Three-Argument Functions

Compound and annuity use the calling sequence

```
   PEA     SRC2 address    ; push address of rate first
   PEA     SRC  address    ; push address of number of periods second
   PEA     DST  address    ; push address of destination third
   <EOPMACRO>
```

to effect

DST <-- <op> (SRC2, SRC)

where <op> is compound or annuity, SRC2 is the rate, and SRC is the number of periods. All arguments SRC2, SRC, and DST must be of the extended type.

Example

C <-- $(1 + R)^N$, where C, R, and N are of type extended.

```
   PEA     R_ADR   ; push address of R
   PEA     N_ADR   ; push address of N
   PEA     C_ADR   ; push address of C
   FCOMPOUND       ; compound
```

Appendix A
68000 SANE Access

In your assemblies include the file TLASM/SANEMACS.TEXT, which contains the macros mentioned in this manual. The standard version is for Macintosh. For programs that will run on Lisa, redefine the symbol FPBYTRAP as follows:

```
   FPBYTRAP .EQU   0
```

On Macintosh, the object code for FP68K and ELEMS68K is automatically loaded as needed by the Package Manager. On Lisa, it suffices to link your assembled code with the intrinsic unit file IOSFPLIB.OBJ.

Appendix B
68000 SANE Macros

```
;-----------------------------------------------------------
;
; FILE: SANEMACS.TEXT
;
;  These macros and equates give assembly language access to
;  the 68K floating-point arithmetic routines.
;-----------------------------------------------------------

;-----------------------------------------------------------
; WARNING: set FPBYTRAP for your system.
;-----------------------------------------------------------
FPBYTRAP        .EQU    1       ;0 for Lisa, 1 for Macintosh

        .MACRO  JSRFP
            .IF     FPBYTRAP
                _FP68K          ;defined in TOOLMACS
            .ELSE
                .REF    FP68K
                JSR     FP68K
            .ENDC
        .ENDM

        .MACRO  JSRELEMS
            .IF     FPBYTRAP
                _ELEMS68K       ;defined in TOOLMACS
            .ELSE
                .REF    ELEMS68K
                JSR     ELEMS68K
            .ENDC
        .ENDM

;-----------------------------------------------------------
; Operation code masks.
;-----------------------------------------------------------
FOADD           .EQU    $0000   ; add
FOSUB           .EQU    $0002   ; subtract
FOMUL           .EQU    $0004   ; multiply
FODIV           .EQU    $0006   ; divide
FOCMP           .EQU    $0008   ; compare, no exception from unordered
FOCPX           .EQU    $000A   ; compare, signal invalid if unordered
```

```
FOREM           .EQU    $000C   ; remainder
FOZ2X           .EQU    $000E   ; convert to extended
FOX2Z           .EQU    $0010   ; convert from extended
FOSQRT          .EQU    $0012   ; square root
FORTI           .EQU    $0014   ; round to integral value
FOTTI           .EQU    $0016   ; truncate to integral value
FOSCALB         .EQU    $0018   ; binary scale
FOLOGB          .EQU    $001A   ; binary log
FOCLASS         .EQU    $001C   ; classify
; UNDEFINED     .EQU    $001E

FOSETENV        .EQU    $0001   ; set environment
FOGETENV        .EQU    $0003   ; get environment
FOSETHV         .EQU    $0005   ; set halt vector
FOGETHV         .EQU    $0007   ; get halt vector
FOD2B           .EQU    $0009   ; convert decimal to binary
FOB2D           .EQU    $000B   ; convert binary to decimal
FONEG           .EQU    $000D   ; negate
FOABS           .EQU    $000F   ; absolute
FOCPYSGNX       .EQU    $0011   ; copy sign
FONEXT          .EQU    $0013   ; next-after
FOSETXCP        .EQU    $0015   ; set exception
FOPROCENTRY     .EQU    $0017   ; procedure entry
FOPROCEXIT      .EQU    $0019   ; procedure exit
FOTESTXCP       .EQU    $001B   ; test exception
; UNDEFINED     .EQU    $001D
; UNDEFINED     .EQU    $001F


;----------------------------------------------------------
; Operand format masks.
;----------------------------------------------------------
FFEXT           .EQU    $0000   ; extended -- 80-bit float
FFDBL           .EQU    $0800   ; double   -- 64-bit float
FFSGL           .EQU    $1000   ; single   -- 32-bit float
FFINT           .EQU    $2000   ; integer  -- 16-bit integer
FFLNG           .EQU    $2800   ; long int -- 32-bit integer
FFCOMP          .EQU    $3000   ; comp     -- 64-bit integer

;----------------------------------------------------------
; Precision code masks: forces a floating point output
; value to be coerced to the range and precision specified.
;----------------------------------------------------------
FCEXT           .EQU    $0000   ; extended
FCDBL           .EQU    $4000   ; double
FCSGL           .EQU    $8000   ; single
```

```
;-----------------------------------------------------------
; Operation macros: operand addresses should already be on
; the stack, with the destination address on top.  The
; suffix X, D, S, C, I, or L  determines the format of the
; source operand -- extended, double, single, comp,
; integer, or long integer, respectively; the destination
; operand is always extended.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Addition.
;-----------------------------------------------------------
        .MACRO  FADDX
        MOVE.W  #FFEXT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDD
        MOVE.W  #FFDBL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDS
        MOVE.W  #FFSGL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDC
        MOVE.W  #FFCOMP+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDI
        MOVE.W  #FFINT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDL
        MOVE.W  #FFLNG+FOADD,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Subtraction.
;-----------------------------------------------------------
        .MACRO  FSUBX
```

```
            MOVE.W   #FFEXT+FOSUB,-(SP)
            JSRFP
            .ENDM

            .MACRO   FSUBD
            MOVE.W   #FFDBL+FOSUB,-(SP)
            JSRFP
            .ENDM
            .MACRO   FSUBS
            MOVE.W   #FFSGL+FOSUB,-(SP)
            JSRFP
            .ENDM

            .MACRO   FSUBC
            MOVE.W   #FFCOMP+FOSUB,-(SP)
            JSRFP
            .ENDM

            .MACRO   FSUBI
            MOVE.W   #FFINT+FOSUB,-(SP)
            JSRFP
            .ENDM

            .MACRO   FSUBL
            MOVE.W   #FFLNG+FOSUB,-(SP)
            JSRFP
            .ENDM

;-----------------------------------------------------------
; Multiplication.
;-----------------------------------------------------------
            .MACRO   FMULX
            MOVE.W   #FFEXT+FOMUL,-(SP)
            JSRFP
            .ENDM

            .MACRO   FMULD
            MOVE.W   #FFDBL+FOMUL,-(SP)
            JSRFP
            .ENDM

            .MACRO   FMULS
            MOVE.W   #FFSGL+FOMUL,-(SP)
            JSRFP
            .ENDM

            .MACRO   FMULC
```

```
        MOVE.W  #FFCOMP+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULI
        MOVE.W  #FFINT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULL
        MOVE.W  #FFLNG+FOMUL,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Division.
;-----------------------------------------------------------
        .MACRO  FDIVX
        MOVE.W  #FFEXT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVD
        MOVE.W  #FFDBL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVS
        MOVE.W  #FFSGL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVC
        MOVE.W  #FFCOMP+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVI
        MOVE.W  #FFINT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVL
        MOVE.W  #FFLNG+FODIV,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Square root.
;-----------------------------------------------------------
        .MACRO   FSQRTX
        MOVE.W   #FOSQRT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Round to integer, according to the current rounding mode.
;-----------------------------------------------------------
        .MACRO   FRINTX
        MOVE.W   #FORTI,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Truncate to integer, using round toward zero.
;-----------------------------------------------------------
        .MACRO   FTINTX
        MOVE.W   #FOTTI,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Remainder.
;-----------------------------------------------------------
        .MACRO   FREMX
        MOVE.W   #FFEXT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMD
        MOVE.W   #FFDBL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMS
        MOVE.W   #FFSGL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO   FREMC
        MOVE.W   #FFCOMP+FOREM,-(SP)
        JSRFP
        .ENDM
        .MACRO  FREMI
        MOVE.W  #FFINT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREML
        MOVE.W  #FFLNG+FOREM,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Logb.
;-----------------------------------------------------------
        .MACRO  FLOGBX
        MOVE.W  #FOLOGB,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Scalb.
;-----------------------------------------------------------
        .MACRO  FSCALBX
        MOVE.W  #FFINT+FOSCALB,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Copy-sign.
;-----------------------------------------------------------
        .MACRO  FCPYSGNX
        MOVE.W  #FOCPYSGN,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Negate.
;-----------------------------------------------------------
        .MACRO  FNEGX
        MOVE.W  #FONEG,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Absolute value.
;-----------------------------------------------------------
        .MACRO  FABSX
        MOVE.W  #FOABS,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Next-after.  NOTE: both operands are of the same
; format, as specified by the usual suffix.
;-----------------------------------------------------------
        .MACRO  FNEXTS
        MOVE.W  #FFSGL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTD
        MOVE.W  #FFDBL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTX
        MOVE.W  #FFEXT+FONEXT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Conversion to extended.
;-----------------------------------------------------------
        .MACRO  FX2X
        MOVE.W  #FFEXT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2X
        MOVE.W  #FFDBL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2X
        MOVE.W  #FFSGL+FOZ2X,-(SP)
        JSRFP
        .ENDM
        .MACRO  FI2X
        MOVE.W  #FFINT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2X
        MOVE.W  #FFLNG+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2X
        MOVE.W  #FFCOMP+FOZ2X,-(SP)
        JSRFP
        .ENDM

;------------------------------------------------------------
; Conversion from extended.
;------------------------------------------------------------
        .MACRO  FX2D
        MOVE.W  #FFDBL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2S
        MOVE.W  #FFSGL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2I
        MOVE.W  #FFINT+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2L
        MOVE.W  #FFLNG+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2C
        MOVE.W  #FFCOMP+FOX2Z,-(SP)
        JSRFP
        .ENDM
```

```
;-------------------------------------------------------------
; Binary to decimal conversion.
;-------------------------------------------------------------
        .MACRO  FX2DEC
        MOVE.W  #FFEXT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2DEC
        MOVE.W  #FFDBL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2DEC
        MOVE.W  #FFSGL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2DEC
        MOVE.W  #FFCOMP+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FI2DEC
        MOVE.W  #FFINT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2DEC
        MOVE.W  #FFLNG+FOB2D,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Decimal to binary conversion.
;-------------------------------------------------------------
        .MACRO  FDEC2X
        MOVE.W  #FFEXT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2D
        MOVE.W  #FFDBL+FOD2B,-(SP)
        JSRFP
        .ENDM
        .MACRO  FDEC2S
        MOVE.W  #FFSGL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2C
        MOVE.W  #FFCOMP+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2I
        MOVE.W  #FFINT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2L
        MOVE.W  #FFLNG+FOD2B,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Compare, not signaling invalid on unordered.
;-----------------------------------------------------------
        .MACRO  FCMPX
        MOVE.W  #FFEXT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPD
        MOVE.W  #FFDBL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPS
        MOVE.W  #FFSGL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPC
        MOVE.W  #FFCOMP+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPI
        MOVE.W  #FFINT+FOCMP,-(SP)
        JSRFP
        .ENDM
        .MACRO  FCMPL
        MOVE.W  #FFLNG+FOCMP,-(SP)
        JSRFP
        .ENDM

;----------------------------------------------------------
; Compare, signaling invalid on unordered.
;----------------------------------------------------------
        .MACRO  FCPXX
        MOVE.W  #FFEXT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXD
        MOVE.W  #FFDBL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXS
        MOVE.W  #FFSGL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXC
        MOVE.W  #FFCOMP+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXI
        MOVE.W  #FFINT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXL
        MOVE.W  #FFLNG+FOCPX,-(SP)
        JSRFP
        .ENDM

;----------------------------------------------------------
; The following macros define a set of so-called floating
; branches.  They presume that the appropriate compare
; operation, macro FCMPz or FCPXz, precedes.
;----------------------------------------------------------
        .MACRO  FBEQ
        BEQ     %1
        .ENDM
```

```
 .MACRO  FBLT
BCS      %1
 .ENDM

 .MACRO  FBLE
BLS      %1
 .ENDM

 .MACRO  FBGT
BGT      %1
 .ENDM

 .MACRO  FBGE
BGE      %1
 .ENDM

 .MACRO  FBULT
BLT      %1
 .ENDM

 .MACRO  FBULE
BLE      %1
 .ENDM

 .MACRO  FBUGT
BHI      %1
 .ENDM

 .MACRO  FBUGE
BCC      %1
 .ENDM

 .MACRO  FBU
BVS      %1
 .ENDM

 .MACRO  FBO
BVC      %1
 .ENDM

 .MACRO  FBNE
BNE      %1
 .ENDM
```

```
                .MACRO  FBUE
                BEQ     %1
                BVS     %1
                .ENDM

                .MACRO  FBLG
                BNE     %1
                BVC     %1
                .ENDM

;------------------------------------------------------------
; Short branch versions.
;------------------------------------------------------------
                .MACRO  FBEQS
                BEQ.S   %1
                .ENDM

                .MACRO  FBLTS
                BCS.S     %1
                .ENDM

                .MACRO  FBLES
                BLS.S     %1
                .ENDM

                .MACRO  FBGTS
                BGT.S     %1
                .ENDM

                .MACRO  FBGES
                BGE.S     %1
                .ENDM

                .MACRO  FBULTS
                BLT.S     %1
                .ENDM

                .MACRO  FBULES
                BLE.S     %1
                .ENDM

                .MACRO  FBUGTS
                BHI.S     %1
                .ENDM
```

```
        .MACRO  FBUGES
        BCC.S    %1
        .ENDM

        .MACRO  FBUS
        BVS.S    %1
        .ENDM

        .MACRO  FBOS
        BVC.S    %1
        .ENDM

        .MACRO  FBNES
        BNE.S    %1
        .ENDM

        .MACRO  FBUES
        BEQ.S    %1
        BVS.S    %1
        .ENDM

        .MACRO  FBLGS
        BNE.S    %1
        BVC.S    %1
        .ENDM

;-----------------------------------------------------------
; Class and sign inquiries.
;-----------------------------------------------------------
FCSNAN          .EQU    1       ; signaling NAN
FCQNAN          .EQU    2       ; quiet NAN
FCINF           .EQU    3       ; infinity
FCZERO          .EQU    4       ; zero
FCNORM          .EQU    5       ; normal number
FCDENORM        .EQU    6       ; denormal number

        .MACRO  FCLASSS
        MOVE.W  #FFSGL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSD
        MOVE.W  #FFDBL+FOCLASS,-(SP)
        JSRFP
        .ENDM
        .MACRO  FCLASSX
        MOVE.W  #FFEXT+FOCLASS,-(SP)
        JSRFP
        .ENDM

;---------------------------------------------------------
; Bit indexes for bytes of floating point environment word.
;---------------------------------------------------------
FBINVALID       .EQU    0       ; invalid operation
FBUFLOW         .EQU    1       ; underflow
FBOFLOW         .EQU    2       ; overflow
FBDIVZER        .EQU    3       ; division by zero
FBINEXACT       .EQU    4       ; inexact
FBRNDLO         .EQU    5       ; low bit of rounding mode
FBRNDHI         .EQU    6       ; high bit of rounding mode
FBLSTRND        .EQU    7       ; last round result bit
FBDBL           .EQU    5       ; double precision control
FBSGL           .EQU    6       ; single precision control

;---------------------------------------------------------
; Get and set environment.
;---------------------------------------------------------
        .MACRO  FGETENV
        MOVE.W  #FOGETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETENV
        MOVE.W  #FOSETENV,-(SP)
        JSRFP
        .ENDM

;---------------------------------------------------------
; Test and set exception.
;---------------------------------------------------------
        .MACRO  FTESTXCP
        MOVE.W  #FOTESTXCP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETXCP
        MOVE.W  #FOSETXCP,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Procedure entry and exit.
;-----------------------------------------------------------
        .MACRO  FPROCENTRY
        MOVE.W  #FOPROCENTRY,-(SP)
        JSRFP
        .ENDM

        .MACRO  FPROCEXIT
        MOVE.W  #FOPROCEXIT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Get and set halt vector.
;-----------------------------------------------------------
        .MACRO  FGETHV
        MOVE.W  #FOGETHV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETHV
        MOVE.W  #FOSETHV,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Elementary function operation code masks.
;-----------------------------------------------------------
FOLNX           .EQU    $0000   ; base-e log
FOLOG2X         .EQU    $0002   ; base-2 log
FOLN1X          .EQU    $0004   ; ln (1 + x)
FOLOG21X        .EQU    $0006   ; log2 (1 + x)

FOEXPX          .EQU    $0008   ; base-e exponential
FOEXP2X         .EQU    $000A   ; base-2 exponential
FOEXP1X         .EQU    $000C   ; exp (x) - 1
FOEXP21X        .EQU    $000E   ; exp2 (x) - 1

FOXPWRI         .EQU    $8010   ; integer exponentiation
FOXPWRY         .EQU    $8012   ; general exponentiation
FOCOMPOUNDX     .EQU    $C014   ; compound
FOANNUITYX      .EQU    $C016   ; annuity

FOSINX          .EQU    $0018   ; sine
FOCOSX          .EQU    $001A   ; cosine
```

```
FOTANX          .EQU    $001C   ; tangent
FOATANX         .EQU    $001E   ; arctangent
FORANDOMX       .EQU    $0020   ; random


;-----------------------------------------------------------
; Elementary function macros.
;-----------------------------------------------------------
        .MACRO  FLNX            ; base-e log
        MOVE.W  #FOLNX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG2X          ; base-2 log
        MOVE.W  #FOLOG2X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLN1X           ; ln (1 + x)
        MOVE.W  #FOLN1X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG21X         ; log2 (1 + x)
        MOVE.W  #FOLOG21X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXPX           ; base-e exponential
        MOVE.W  #FOEXPX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP2X          ; base-2 exponential
        MOVE.W  #FOEXP2X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP1X          ; exp (x) - 1
        MOVE.W  #FOEXP1X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP21X         ; exp2 (x) - 1
        MOVE.W  #FOEXP21X,-(SP)
        JSRELEMS
        .ENDM
```

```
 .MACRO  FXPWRI            ; integer exponential
MOVE.W  #FOXPWRI,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FXPWRY            ; general exponential
MOVE.W  #FOXPWRY,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FCOMPOUNDX        ; compound
MOVE.W  #FOCOMPOUNDX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FANNUITYX         ; annuity
MOVE.W  #FOANNUITYX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FSINX             ; sine
MOVE.W  #FOSINX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FCOSX             ; cosine
MOVE.W  #FOCOSX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FTANX             ; tangent
MOVE.W  #FOTANX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FATANX            ; arctangent
MOVE.W  #FOATANX,-(SP)
JSRELEMS
 .ENDM

 .MACRO  FRANDOMX          ; random number generator
MOVE.W  #FORANDOMX,-(SP)
JSRELEMS
 .ENDM
```

```
;-----------------------------------------------------------
; NaN codes.
;-----------------------------------------------------------
NANSQRT .EQU     1    ; Invalid square root such as sqrt(-1).
NANADD  .EQU     2    ; Invalid addition such as +INF - +INF.
NANDIV  .EQU     4    ; Invalid division such as 0/0.
NANMUL  .EQU     8    ; Invalid multiply such as 0 * INF.
NANREM  .EQU     9    ; Invalid remainder or mod such as x REM 0.
NANASCBIN .EQU  17    ; Attempt to convert invalid ASCII string.
NANCOMP .EQU    20    ; Result of converting comp NaN to floating.
NANZERO .EQU    21    ; Attempt to create a NaN with a zero code.
NANTRIG .EQU    33    ; Invalid argument to trig routine.
NANINVTRIG .EQU 34    ; Invalid argument to inverse trig routine.
NANLOG  .EQU    36    ; Invalid argument to log routine.
NANPOWER .EQU   37    ; Invalid argument to x^i or x^y routine.
NANFINAN .EQU   38    ; Invalid argument to financial function.
NANINIT .EQU   255    ; Uninitialized storage.
;-----------------------------------------------------------
```

68000 SANE Quick Reference Guide

This Guide contains diagrams of the SANE data formats and the 68K SANE operations and environment word.

C.1 Data Formats

Each of the diagrams below is followed by the rules for evaluating the number v.

In each field of each diagram, the leftmost bit is the msb and the rightmost is the lsb.

Format Diagram Symbols

v	value of number
s	sign bit
e	biased exponent
i	explicit one's-bit (extended type only)
f	fraction

Single: 32 Bits

```
 1      8                        23                        widths
-----------------------------------------------------
|s|     e       |                  f                 |
-----------------------------------------------------

if 0 < e < 255,       then v = (-1)s * 2(e-127) * (1.f);
if e =   0 and f ≠ 0, then v = (-1)s * 2(-126) * (0.f);
if e =   0 and f = 0, then v = (-1)s * 0;
if e = 255 and f = 0, then v = (-1)s * oo;
if e = 255 and f ≠ 0, then v is a NaN.
```

Double: 64 Bits

```
 1     11                           52                     widths
---------------------------------------------------------
|s|     e      |                     f                   |
---------------------------------------------------------
```

```
    if 0 < e < 2047,       then v = (-1)s * 2(e-1023) * (1.f);
    if e =    0 and f =/ 0, then v = (-1)s * 2(-1022) * (0.f);
    if e =    0 and f = 0, then v = (-1)s * 0;
    if e = 2047 and f = 0, then v = (-1)s * oo;
    if e = 2047 and f =/ 0, then v is a NaN.
```

Comp: 64 Bits

```
 1             63                  widths
---------------------------------
|s|             d               |
---------------------------------
```

```
    if s = 1 and d = 0, then v is the unique comp NaN;
    otherwise, v is the two's-complement value of the
    64-bit representation.
```

Extended: 80 Bits

```
 1     15        1                  63                      widths
-----------------------------------------------------------
|s|     e      |i|                   f                    |
-----------------------------------------------------------
```

```
    if 0 <= e < 32767,       then v = (-1)s * 2(e-16383) * (i.f);
    if e = 32767 and f = 0, then v = (-1)s * oo, regardless of i;
    if e = 32767 and f =/ 0, then v is a NaN, regardless of i.
```

C.2 Operations

In the operations below, the operation's mnemonic is followed by the opword in parentheses: the first byte is the operation code; the second is the operand format code. For some operations, the first byte of the opword (xx) is ignored.

C.2.1 Abbreviations and Symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has various meanings, depending on context.

Operands

DST	destination operand (passed by address)
SRC	source operand (passed by address), pushed before DST
SRC2	second source operand (passed by address), pushed before SRC

Data Types

X	extended (80 bits)
D	double (64 bits)
S	single (32 bits)
I	integer (16 bits)
L	longint (32 bits)
C	comp (64 bits)
Dec	decimal Record
Decform	decform Record

68000 Processor Registers

D0	data register 0
X	extend bit of processor status register
N	negative bit of processor status register
Z	zero bit of processor status register
V	overflow bit of processor status register
C	carry bit of processor status register

Exceptions

I	invalid operation
U	underflow
O	overflow
D	divide-by-zero
X	inexact

For each operation, an exception marked with x indicates that the operation will signal the exception for some input.

Environment and Halts

EnWrd	SANE environment word (16-bit integer)
HltVctr	SANE halt vector (32-bit longint)

C.2.2 Arithmetic Operations and Auxiliary Routines (Entry Point FP68K)

Operation	Operands and Data Types				Exceptions
ADD	DST	<--	DST +	SRC	I U O D X
FADDX (0000)	X		X	X	x - x - x
FADDD (0800)	X		X	D	x - x - x
FADDS (1000)	X		X	S	x - x - x
FADDC (3000)	X		X	C	x - x - x
FADDI (2000)	X		X	I	x - x - x
FADDL (2800)	X		X	L	x - x - x
SUBTRACT	DST	<--	DST -	SRC	I U O D X
FSUBX (0002)	X		X	X	x - x - x
FSUBD (0802)	X		X	D	x - x - x
FSUBS (1002)	X		X	S	x - x - x
FSUBC (3002)	X		X	C	x - x - x
FSUBI (2002)	X		X	I	x - x - x
FSUBL (2802)	X		X	L	x - x - x
MULTIPLY	DST	<--	DST *	SRC	I U O D X
FMULX (0004)	X		X	X	x x x - x
FMULD (0804)	X		X	D	x x x - x
FMULS (1004)	X		X	S	x x x - x
FMULC (3004)	X		X	C	x - x - x
FMULI (2004)	X		X	I	x - x - x
FMULL (2804)	X		X	L	x - x - x
DIVIDE	DST	<--	DST /	SRC	I U O D X
FDIVX (0006)	X		X	X	x x x x x
FDIVD (0806)	X		X	D	x x x x x
FDIVS (1006)	X		X	S	x x x x x
FDIVC (3006)	X		X	C	x x - x x
FDIVI (2006)	X		X	I	x x - x x
FDIVL (2806)	X		X	L	x x - x x

```
SQUARE ROOT      DST   <--  sqrt(DST)                    I U O D X
FSQRTX (0012)     X               X                      x - - - x

ROUND TO INT     DST   <--  rnd(DST)                     I U O D X
FRINTX (0014)     X              X                       x - - - x

TRUNC TO INT     DST   <--  chop(DST)                    I U O D X
FTINTX (0016)     X               X                      x - - - x

REMAINDER        DST   <--  DST REM SRC                  I U O D X
FREMX (000C)      X          X       X                   x - - - -
FREMD (080C)      X          X       D                   x - - - -
FREMS (100C)      X          X       S                   x - - - -
FREMC (300C)      X          X       C                   x - - - -
FREMI (200C)      X          X       I                   x - - - -
FREML (280C)      X          X       L                   x - - - -

                 D0    <--   integer quotient DST/SRC,
                                 between -127 and +127

LOG BINARY       DST   <--  logb(DST)                    I U O D X
FLOGBX (001A)     X               X                      x - - x -

SCALE BINARY     DST   <--  DST * 2^SRC                  I U O D X
FSCALBX (0018)    X          X       I                   x x x - x

NEGATE           DST   <--  -DST                         I U O D X
FNEGX (000D)      X           X                          - - - - -

ABSOLUTE VALUE   DST   <--  |DST|                        I U O D X
FABSX (000F)      X           X                          - - - - -

COPY-SIGN        SRC   <--  SRC with DST's sign          I U O D X
FCPYSGNX (0011) XDorS       XDorS    XDorS               - - - - -

NEXT-AFTER       SRC   <--  next after SRC toward DST    I U O D X
FNEXTX (0013)     X                     X          X     x x x - x
FNEXTD (0813)     D                     D          D     x x x - x
FNEXTS (1013)     S                     S          S     x x x - x
```

C.2.3 Conversions (Entry Point FP68K)

<u>Operation</u>	<u>Operands and Data Types</u>				<u>Exceptions</u>				
CONVERT									
Bin to Bin	DST	<--	SRC		I	U	O	D	X
FX2X (0010)	X		X		x	-	-	-	-
FX2D (0810)	D		X		x	x	x	-	x
FX2S (1010)	S		X		x	x	x	-	x
FX2C (3010)	C		X		x	-	-	-	x
FX2I (2010)	I		X		x	-	-	-	x
FX2L (2810)	L		X		x	-	-	-	x
FD2X (080E)	X		D		x	-	-	-	-
FS2X (100E)	X		S		x	-	-	-	-
FC2X (300E)	X		C		-	-	-	-	-
FI2X (200E)	X		I		-	-	-	-	-
FL2X (280E)	X		L		-	-	-	-	-

Bin to Dec	DST	<--	SRC according to	SRC2	I	U	O	D	X
FX2DEC (000B)	Dec		X	Decform	x	-	-	-	x
FD2DEC (080B)	Dec		D	Decform	x	-	-	-	x
FS2DEC (100B)	Dec		S	Decform	x	-	-	-	x
FC2DEC (300B)	Dec		C	Decform	-	-	-	-	x
FI2DEC (200B)	Dec		I	Decform	-	-	-	-	x
FL2DEC (280B)	Dec		L	Decform	-	-	-	-	x

(First SRC2 is pushed, then SRC, then DST.)

				I	U	O	D	X
Dec to Bin	DST	<--	SRC	I	U	O	D	X
FDEC2X (0009)	X		Dec	-	x	x	-	x
FDEC2D (0809)	D		Dec	-	x	x	-	x
FDEC2S (1009)	S		Dec	-	x	x	-	x
FDEC2C (3009)	C		Dec	x	-	-	-	x
FDEC2I (2009)	I		Dec	x	-	-	-	x
FDEC2L (2809)	L		Dec	x	-	-	-	x

C.2.4 Compare and Classify (Entry Point FP68K)

Operation	Operands and Data Types		Exceptions
COMPARE			
No invalid for unordered	Status Bits <-- <relation> where DST <relation> SRC		I U O D X
FCMPX (0008)	X	X	x - - - -
FCMPD (0808)	X	D	x - - - -
FCMPS (1008)	X	S	x - - - -
FCMPC (3008)	X	C	x - - - -
FCMPI (2008)	X	I	x - - - -
FCMPL (2808)	X	L	x - - - -

(invalid only for signaling NaN inputs)

Signal invalid if unordered	Status Bits <-- <relation> where DST <relation> SRC		I U O D X
FCPXX (000A)	X	X	x - - - -
FCPXD (080A)	X	D	x - - - -
FCPXS (100A)	X	S	x - - - -
FCPXC (300A)	X	C	x - - - -
FCPXI (200A)	X	I	x - - - -
FCPXL (280A)	X	L	x - - - -

<relation>	Status Bits X	N	Z	V	C
DST > SRC	0	0	0	0	0
DST < SRC	1	1	0	0	1
DST = SRC	0	0	1	0	0
DST & SRC unordered	0	0	0	1	0

CLASSIFY

```
<class> <-- class of SRC
<sign>  <-- sign  of SRC
   DST  <-- (-1)^<sign> * <class>
```

			I U O D X
FCLASSX (001C)	I	X	- - - - -
FCLASSD (081C)	I	D	- - - - -
FCLASSS (101C)	I	S	- - - - -

SRC	‹class›	SRC	‹sign›
signaling NaN	1	positive	0
quiet NaN	2	negative	1
infinite	3		
zero	4		
normalized	5		
denormalized	6		

C.2.5 Environmental Control (Entry Point FP68K)

Operation	Operands and Data Types	Exceptions
GET ENVIRONMENT	DST <-- EnvWrd	I U O D X
FGETENV (0003)	I	- - - - -
SET ENVIRONMENT	EnvWrd <-- SRC	I U O D X
FSETENV (0001)	I	x x x x x

(exceptions set by set-environment cannot cause halts)

Operation	Operands and Data Types	Exceptions
TEST EXCEPTION	Zbit <-- SRC Xcps clear	I U O D X
FTESTXCP (001B)	I	- - - - -
SET EXCEPTION	EnvWrd <-- EnvWrd AND SRC	I U O D X
FSETXCP (0015)	I	x x x x x
PROCEDURE ENTRY	DST <-- EnvWrd, EnvWrd <-- 0	I U O D X
FPROCENTRY (0017)	I	x x x x x
PROCEDURE EXIT	EnvWrd <-- SRC AND current Xcps	I U O D X
FPROCEXIT (0019)	I	x x x x x

C.2.6 Halt Control (Entry Point FP68K)

SET HALT VECTOR FSETHV (xx05)	HltVctr <-- SRC L	I U O D X - - - - -
GET HALT VECTOR FGETHV (0007)	DST <-- HltVctr L	I U O D X - - - - -

C.2.7 Elementary Functions (Entry Point ELEMS68K)

Operation	Operands and Data Types	Exceptions
BASE-E LOGARITHM FLNX (0000)	DST <-- ln(DST) X X	I U O D X x - - x x
BASE-2 LOGARITHM FLOG2X (0002)	DST <-- log2(DST) X X	I U O D X x - - x x
BASE-E LOG1 (LN1) FLN1X (0004)	DST <-- ln(1+DST) X X	I U O D X x x - x x
BASE-2 LOG1 FLOG21X (0006)	DST <-- log2(1+DST) X X	I U O D X x x - x x
BASE-E EXPONENTIAL FEXPX (0008)	DST <-- e^DST X X	I U O D X x x x - x
BASE-2 EXPONENTIAL FEXP2X (000A)	DST <-- 2^DST X X	I U O D X x x x - x
BASE-E EXP1 FEXP1X (000C)	DST <-- e^DST - 1 X X	I U O D X x x x - x
BASE-2 EXP1 FEXP21X (000E)	DST <-- 2^DST - 1 X X	I U O D X x x x - x

```
INTEGER EXPONENTIATION  DST  <--  DST^SRC               I U O D X
FXPWRI (8010)            X         X   I                x x x x x


GENERAL EXPONENTIATION  DST  <--  DST^SRC               I U O D X
FXPWRY (8012)            X         X   X                x x x x x


COMPOUND INTEREST       DST  <--  compound(SRC2,SRC)    I U O D X
FCOMPOUND (C014)         X                  X    X      x x x x x

(SRC2 is the rate; SRC is the number of periods.)


ANNUITY FACTOR          DST  <--  annuity(SRC2,SRC)     I U O D X
FANNUITY (C016)          X                 X    X       x x x x x

(SRC2 is the rate; SRC is the number of periods.)


SINE                    DST  <--  sin(DST)              I U O D X
FSINX (0018)             X             X                x x - - x


COSINE                  DST  <--  cos(DST)              I U O D X
FCOSX (001A)             X             X                x x - - x


TANGENT                 DST  <--  tan(DST)              I U O D X
FTANX (001C)             X             X                x x - x x


ARCTANGENT              DST  <--  atan(DST)             I U O D X
FATANX (001E)            X              X               x x - - x


RANDOM                  DST  <--  random(DST)           I U O D X
FRANDX (0020)            X                X             x x x - x
```

C.3 Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:

```
msb                                                           lsb
|-------------------------------|-------------------------------|
| - | r | r | x | d | o | u | i| - | R | R | X | D| O | U |I |
|-------------------------------|-------------------------------|
rounding        exception       rounding           halt
direction         flags         precision          enables
```

```
        rounding direction, bits 6000          rr
                0000 -- to-nearest
                2000 -- upward
                4000 -- downward
                6000 -- toward-zero

        exception flags, bits 1F00
                0100 -- invalid                i
                0200 -- underflow              u
                0400 -- overflow               o
                0800 -- division-by-zero       d
                1000 -- inexact                x

        rounding precision, bits 0060          RR
                0000 -- extended
                0020 -- double
                0040 -- single
                0060 -- UNDEFINED

        halt enabled, bits 001F
                0001 -- invalid                I
                0002 -- underflow              U
                0004 -- overflow               O
                0008 -- division-by-zero       D
                0010 -- inexact                X
```

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

The StdUnit

Contents

The StdUnit Unit

1 Introduction

StdUnit is the "Standard Unit," an intrinsic unit that provides a number of standard functions. It contains functions dealing with:

- Character and string manipulartion.
- File name manipulation.
- Prompting.
- Error messages.
- Special Workshop features.
- Conversions.

Workshop tools should use the unit wherever possible, especially for prompting and Operating System error reporting, to make the Workshop interface consistent.

Note: All names in StdUnit begin with the letters SU. This avoids name conflicts when incorporating the unit into your code and identifies where things come from.

2 Functional Areas

2.1 Initialization

StdUnit needs to be initialized before it can be used. Using the unit without initializing it will often result in an address or bus error.

2.2 String and Character Manipulation

StdUnit provides a standard string type, SUStr; a type for sets of characters; definitions for several standard characters (such as CR and BS); and procedures for case conversion, trimming blanks, and appending strings and characters.

2.3 File Name Manipulation

File name functions let you determine if a pathname is a volume or device name only; add extensions (such as .TEXT) to the file names (the procedure knows the conventions about when extensions should and should not be added); splitting a pathname into its three basic components--the device, volume, or catalog component, the file name component, and the extension component; putting the components back together into a file name; and modifying a file name given optional defaults for missing volume, file or extension components.

Note: Several of the procedures return overflow flags for identifying when a file name component has exceeded its character limit. You may choose to ignore the overflow condition, particularly if you think it likely to occur only in perverse circumstances.

Note: The string parameters to these procedures are typed differently, sometimes SUStr's, or VAR SUStr's, or SUStrP's (pointers to SUStr's). This is to avoid problems with Pascal string typing when using the procedures with strings that are not SUStr's (e.g., PathName's), and to take into account the cases in which the parameters are likely to be string constants.

2.4 Prompting

StdUnit provides a number of procedures to get characters, strings, file names, integers, yes/no responses, etc., from the console, providing for default values where appropriate.

Most of the prompting procedures return a PrompState indicating whether an escape [CLEAR] was typed, whether the default was taken, or whether there was a request for options with ?. The states returned are given for each procedure. You can ignore the prompt states you are not interested in. For example, if you don't want to treat ? as an option request, you can ignore the SUOptions state and not treat the ? returned as a special character.

2.5 Error Text Retrieval

StdUnit provides a mechanism to retrieve single-line error messages from specially formatted error files. Error messages can be looked up by number in one or more error files.

You can use the OS error file OSErrs.ERR to return a real message when an OS error occurs (see Example 2, below). Note that OS errors are also returned via Pascal's IORESULT.

The ErrTool program lets you make your own compacted message files. Using this error mechanism, you can add and modify messages without recompiling your program. ErrTool is described in the *Workshop User's Guide*, Chapter 11, The Utilities.

A call to retrieve a message opens the error file, searches the directory for the error number, finds location of the message, and returns the text.

A program can use StdUnit to access more than one error file simultaneously. For example, your program can access different files for OS error messages and your own messages.

2.6 Workshop Support

Special Workshop functions let you:

- Stop the execution of an EXEC file in progress.
- Find out the name of the boot and current prefix volumes (SysVols).
- Use a super-RESET that will try to open a file first on the prefix volume, then on the boot volume, then on the current process volume.

2.7 Conversions

Conversion procedures let you convert from integers and longints to strings, and from strings to integers and longints.

3 Examples

Example 1

Assume we are going to prompt for an output file name (OutFName) and that we already have the input file name (InFName). We will use SUSplitFN to split the input file name into its various components. Then we will prompt for the output file name (with SUGetFN) using the volume and file name components of the input file name as defaults but with a .ERR extension. We then do a CASE on the prompt state (PState) returned by SUGetFN. The will terminate if the file specification is an escape [CLEAR]; say that no option are available if **?** is typed as an option request; prompt again if no file is specified, since we want to require an output file; and fall through if the default is accepted or some other file is specified. Note that we only have to check for the prompt states we are interested in for special handling.

```
9999:
     WRITE ('Name of Error Output File ');
     SUSplitFN (@InFName, @VolN, @FN, @Ext);
     SUGetFN (@OutFName, PState, VolN, FN, '.ERR');
     CASE PState OF
       SUEscape:  EXIT (ErrFileP);  {exit from program}
       SUOptions: BEGIN
                    WRITELN ('No options are available. ');
                    GOTO 9999;
                  END;
       SUNone:    GOTO 9999;
     END;  {CASE}
```

Example 2

Suppose we have just made a Pascal I/O call and want to report an error (along with the OS message text) if we receive a nonzero IORESULT. Note that we copy IORESULT into our IOStatus variable so that the subsequent WRITELN will not reset the value of IORESUILT before we get a chance to use it. (EMsg should be a SUStr.)

```
IF IORESULT <> 0 THEN
  BEGIN
    IOStatus := IORESULT;
    WRITELN ('Error opening input file.');
    SUErrText ('OsErrs.ERR', IOStatus, @EMsg);
    WRITELN (EMsg);
  END;
```

4 Interface

```
{------------------------------- SU:StdUnit --------------------------------}
{ Copyright 1983, 1984, Apple Computer, Inc.                                }
{                                                                           }
{ This unit provides a number of standard type definitions and a collection }
{ of procedures which perform a variety of common functions.  The areas     }
{ covered are:                                                              }
{   (1) String and Character manipulation                                   }
{   (2) File Name Manipulation                                              }
{   (3) Prompting                                                           }
{   (4) Retrieval of messages from disk                                     }
{   (5) Development System Support                                          }
{   (6) Conversions                                                         }
{                                                                           }
{ Fred Forsman 4-25-84                                                      }
{---------------------------------------------------------------------------}

{$SETC ForOS11orHigher := TRUE}

{$R-} { make it fast, no range checking }
{$S SULib }

UNIT StdUnit;
  INTRINSIC;

  INTERFACE

    USES
      {$U libOS/SysCall.obj    } SysCall, { for definition of PathName, etc. }
      {$U libPL/PasLibCall.obj } PasLibCall,
      {$U libPL/PPasLibC.obj   } PPasLibC;

    CONST
      SUMaxStrLeng = 255;
      SUNullStr    = '';
      SUSpace      = ' ';
      SUOrdCR      = 13;
      SUMaxPNLeng  = 66;    { max length of path name }
      SUMaxVNLeng  = 33;    { max length of volume name, includes leading '-' }
      SUMaxFNLeng  = 32;    { maximum length of file name }
      SUVolSuffix  = '-';   { suffix or end of device or volume name }

    TYPE
      SUSetOfChar  = SET OF CHAR;
      SUStrP       = ^SUStr;
      SUStrP        = ^SUStr;
      SUStr         = STRING[255];
      SUVolName     = STRING[SUMaxVNLeng];
      SUFile        = FILE;
      SUFileP       = ^SUFile;
      PromptState = (SUDefault,  { the default (if any) was chosen) }
                     SUEscape,   { the "Clear" key was pressed }
                     SUNone,     { nothing specified in response to prompt }
                     SUOptions,  { "?" was entered--ie, an option query }
                     SUValid,    { valid reponse }
                     SUInvalid   { invalid reponse--eg, non-number to SUGetInt}
                     );
      ErrTextRet  = (SUOk,           { successful }
                     SUBadEFOpen,    { could not open error file }
                     SUBadEFRead,    { error reading error file }
                     SUErrNNotFound  { error number not found }
                     );
      ConvNState  = (SUValidN,       { valid number }
                     SUNoN,          { no number -- nothing specified }
                     SUBadN,         { invalid number }
                     SUNOverFlow     { overflow -- number too big }
                     );

    VAR
      SUOsBootV   : SUVolName;    { The volume the OS was booted from }
      SUMyProcV   : SUVolName;    { The volume MyProcess was started from }
      SUBell, SUBackSpace, SUCr, SUTab, SUEsc,
        SUDle, SUNul : CHAR;      { predefined ch vars } {ff 1/23/84}
      SUNullS     : SUStr;        { predefined str var }
      SUKeyBoard : INTERACTIVE; { non-echoing console, used by SUGetCh }
                                                             {ff 2/29/84}

{============================== INIT AND DONE ===============================}

    PROCEDURE SUInit;
      { Should be called before using rest of unit.  On the OS this opens
        "-KeyBoard".  It also initializes the standard character variables. }

    PROCEDURE SUDone;
      { Can be called when done using unit (although this is not strictly
        necessary.  On the OS this closes "-KeyBoard". }

{============================ STRINGS AND CHARS =============================}

    FUNCTION SUUpCh (Ch : CHAR) : CHAR;
      { SUUpCh returns the ch that was passed, uppercased if it was lower
        case. }
```

```
FUNCTION SULowCh (Ch : CHAR) : CHAR;
  { SULowCh returns the ch that was passed, lowercased if it was upper
    case. }

PROCEDURE SUUpStr (S: SUStrP);
  { SUUpStr uppercases the string that is passed. }

PROCEDURE SULowStr (S: SUStrP);
  { SULowStr lowercases the string that is passed. }

FUNCTION SUEqStr (S1: SUStrP; S2: SUStrP) : BOOLEAN;          {ff 2/29/84}
  { SUEqStr returns TRUE if the two strings are equal (ignoring case). }

FUNCTION SUEq2Str (S1: SUStrP; S2: SUStr) : BOOLEAN;          {ff 3/7/84}
  { SUEq2Str returns TRUE if the two strings are equal (ignoring case).
    This variant of SUEqStr allows the second parameter to be a constant.}

PROCEDURE SUTrimLeading (S: SUStrP);                          {ff 2/29/84}
  { SUTrimLeading removes the leading blanks and tabs in the passed
    string. }

PROCEDURE SUTrimTrailing (S: SUStrP);                         {ff 2/29/84}
  { SUTrimTrailing removes the trailing blanks and tabs in the passed
    string. }

PROCEDURE SUTrimBlanks (S: SUStrP);
  { SUTrimBlanks removes leading and trailing blanks and tabs in the
    passed string. }

PROCEDURE SUAddCh (S: SUStrP; Ch : CHAR; MaxStrLeng : INTEGER;
                   VAR OverFlow : BOOLEAN);
  { SUAddCh appends the passed ch to the end of the passed string.
    OverFlow is set to TRUE if adding the ch will cause the string to be
    longer than MaxStrLeng. }

PROCEDURE SUConcat (S1: SUStrP; S2: SUStrP);
  { SUConcat appends the second passed str to the end of the first passed
    string.  It is assumed that the target string is of sufficient size to
    accomodate the new value. }

PROCEDURE SUAddStr (S1: SUStrP; S2: SUStrP; MaxStrLeng : INTEGER;
                    VAR OverFlow : BOOLEAN);
  { SUAddStr appends the second passed str to the end of the first passed
    string.  OverFlow is set to TRUE if adding the second string will
    cause the resulting string to be longer than MaxStrLeng. }
    PROCEDURE SUSetStr (Dest: SUStrP; Src: SUStrP);
      { SUSetStr sets the target string (Dest) to the given value (Src) by
        copying the value onto the target.  It is assumed that the target
        string is of sufficient size to accomodate the new value. }

    PROCEDURE SUCopyStr (Dest: SUStrP; Src: SUStrP; Start, Count: INTEGER);
      { SUCopyStr sets the destination string (Dest) to the specified
        substring of the source string (Src) by copying the appropriate part
        of the source to the destination.  It is assumed that the destination
        string is of sufficient size to accomodate the new value, and that the
        Start and Count values are reasonable. }

{=============================== FILE NAMES ================================}

    FUNCTION SUIsVolName (FN: SUStrP): BOOLEAN;
      { SUIsVolName returns a boolean indicating whether the passed file name,
        FN, is a volume or device name (i.e., not a full file name) }

    PROCEDURE SUVolPart (PathN: SUStrP; VolN: SUStrP);             {ff 2/29/84}
      { SUVolPart extracts the volume name part of a pathname (or catalog
        specification). }

    PROCEDURE SUAddExtension (FN: SUStrP; DefExt: SUStr;
                              MaxStrLeng: INTEGER; VAR OverFlow: BOOLEAN);
      { SUAddExtension will add the default extension, DefExt, to the end of
        the file name, S, if the extension is not already present.  If the
        file name ends with a dot, the dot will be removed and no extension
        will be added.  If the pathname is a device or volume name only no
        extension will be added.  OverFlow is set true if adding the extension
        will overflow the string (determined using MaxStrLeng). }

    PROCEDURE SUSplitFN (PathN: SUStrP; CatN: SUStrP; FN: SUStrP;
                         Ext: SUStrP);
      { SUSplitFN splits a PathName into its catalog, file name, and file
        name extension components. }

    PROCEDURE SUMakeFN (PathN: SUStrP; CatN: SUStrP; FN: SUStrP; Ext: SUStr;
                        VAR OverFlow: BOOLEAN);
      { SUMakeFN constructs a PathName from its catalog, file name, and
        file name extension components.  The OS CatN's are assumed to have a
        leading "-".  OverFlow is set if any of the file name components are
        too long.  This procedure will not create a file name over SUMaxPNLeng
        chars long.}

    PROCEDURE SUChkFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
                       DefFN: SUStr; DefExt: SUStr);
      { SUChkFN checks a file name specification, putting result type in
      PState.  If no file name is given, then DefFN is used.  If FN does not
      have DefExt  in it, then the extension is appended.  If no volume is
      specifed then  the DefVol is used.  PState is set appropriately:
        PState = SUOptions  if '?' is hit to ask for options
        PState = SUDefault  if nothing specified when a default is present
        PState = SUNone     if default overriden with '\' or if CR with no
                            default
        PState = SUInvalid  if one or more of the file name components
                            overflowed
        PState = SUValid    otherwise }

{=============================== PROMPTING ===============================}

    PROCEDURE SUGetCh (VAR Ch: CHAR);
      { SUGetCh reads a character from the console without echoing it and }
      { without interpreting <cr> as <sp>, as Read (Ch) does. }

    PROCEDURE SUGetLine (S: SUStrP; VAR PState: PromptState);
      { SUGetLine reads a line from the console a character at a time,
        performing its own line editing.  PState is set appropriately:
          PState = SUEscape  if <clear> was hit.
          PState = SUValid   otherwise. }

    PROCEDURE SUGetStr (S: SUStrP; VAR PState: PromptState; DefVal: SUStr);
      { SUGetStr reads a string from the console; it is like SUGetLine with
        the addition of defaults.  PState is set appropriately:
          PState = SUDefault if <cr> only was hit; S is set to DefVal.
          PState = SUEscape  if <clear> was the first character hit.
          PState = SUValid   otherwise. }

    PROCEDURE SUGetFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
      DefFN: SUStr; DefExt: SUStr);
        { SUGetFN reads a file name from the console, with result type in
          PState.  SUGetFN will print out any defaults in brackets (such as
          [FOO] [.TEXT]) before prompting for the file name.  If no file name
          is given, then DefFN is used.  If FN does not have DefExt in it,
          then the extension is appended.  If no volume is specifed then the
          DefVol is used.  PState is set appropriately:
            PState = SUEscape   if <clear> hit
            PState = SUOptions  if '?' is hit to ask for options
            PState = SUDefault  if nothing specified when a default is present
            PState = SUNone     if default overriden with '\' or if CR with no
                                default
            PState = SUInvalid  if one or more of the file name components
                                overflowed
            PState = SUValid    otherwise }
    PROCEDURE SUGetInt (VAR I: INTEGER; VAR PState: PromptState;
                        DefVal: INTEGER);
      { SUGetInt reads an INTEGER from the console, with PState set as in
        SUGetStr, except that PState = SUInvalid when a non-numeric is input.}

    PROCEDURE SUWaitEscOrSp (VAR PState: PromptState);
      { SUWaitEscOrSp prints a message 'Type <space> to continue, <clear> to
        exit.' & waits for the user to hit a <sp> or <clear>, setting PState
        appropriately:
          PState = SUEscape if <clear> was hit
          PState = SUValid  if <sp> was hit }

    PROCEDURE SUWaitSp;
      { SUWaitSp prints a message ('Type <space> to continue.') and waits for
        the user to hit a <sp>. }

    PROCEDURE SUGetChInSet (VAR Ch: CHAR; Chars: SUSetOfChar);
      { SUGetChInSet reads characters from the console (without echoing) until
        a character from the given set is typed.  The accepted character is
        echoed and an end-of-line is written.  The character matching ignores
        case. }

    FUNCTION SUGetYesNo : BOOLEAN;
      { SUGetYesNo prints the message "(Y or N)" and reads characters from the
        console (without echoing) until a 'y', 'Y', 'n', or 'N' is typed.  If
        a 'y' is typed "Yes" will be printed followed by an end-of-line; if
        'n' is typed "No" will be printed.  The appropriate boolean value is
        returned. }

    FUNCTION SUGetBool (Default: BOOLEAN): BOOLEAN;
      { SUGetBool prints the message "(Y or N)  [<default>]" and reads
        characters from the  console (without echoing) until a 'y', 'Y', 'n',
        'N', space or return is typed.  If a 'y' is typed "Yes" will be
        printed in the place of the default.  If 'n' is typed "No" will be
        printed.  If a space or return is typed the default is used.  The
        appropriate boolean value is returned. }

{========================== ERROR TEXT RETRIEVAL ==========================}

    PROCEDURE SUGetErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP;
                            VAR ErrRet: ErrTextRet);
      { SUGetErrText retrieves error message text, given an error number and
        and error file to look the error up in.  The error file should have
        been generated by the error file processor.  SUGetErrText uses
        SUSysReset to open the error file. }

    PROCEDURE SUErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP);
```

```
      { SUErrText retrieves error message text, just as does SUGetErrText;
        however, if the text is not obtainable due to a non-SUOk ErrRet value
        from SUErrText, SUErrText will return the string
          "Error message text not available." }

{========================== DEV. SYS. SUPPORT ============================}

    PROCEDURE SUStopExec (VAR ErrNum: INTEGER);
      { Should be called to stop the current exec file if an error occurs in a
        program running under an exec.  Returns any error conditions
        encountered in closing the exec file in the errnum var parameter.
        Informs the shell that the exec file was terminated due to an error. }

    PROCEDURE SUCloseExec (VAR ErrNum: INTEGER);   {ff 3/7/84}
      { Should be called to stop the current exec file only if you want to do
        so without informing the shell that the exec file was terminated due
        to an error.  You should probably use SUStopExec unless you have a
        good reason to use this alternate version. }

    PROCEDURE SUInitSysVols;
      { Initializes "SUMyProcV" and "SUOsBootV", the name of the volume on
        which my process was created and the name of the volume which the OS
        was booted off of.  A message may be printed if there is trouble
        getting this information from the OS.  This can be called more than
        once; it will only make the OS calls the first time. }

    PROCEDURE SUSysReset (F : SUFileP; FN : SUStr; VAR IOStatus : INTEGER);
      { SUSysReset is for opening system files, and will try the prefix, boot,
        and current process volumes (in that order) when trying to access a
        file.  SUSysReset assumes that the file name FN does not have a volume
        name.  SUSysReset may sometimes have to call SUInitSysVols. }

{============================ CONVERSIONS ================================}

    PROCEDURE SUIntToStr (N : INTEGER; S : SUStrP);
      { SUIntToStr converts an integer into its string form;  The string which
        S points to should be of length >= 6 (5 digits + sign). }

    PROCEDURE SULIntToStr (N : LONGINT; S : SUStrP);
      { SULIntToStr converts an longint into its string form;  The string
        which S points to should be of length >= 11 (10 digits + sign). }

    PROCEDURE SUStrToInt (NS : SUStrP; VAR N : INTEGER;
                          VAR CState : ConvNState);
      { SUStrToInt converts a string to an INTEGER.  Leading and trailing
        blanks and tabs are permitted.  A leading sign ['-', '+'] is
        permitted.  The CState variable (conversion state) will be set to
        indicate if the number was valid, if no number was present, if an
        invalid number was specified, or if the number overflowed. }

    PROCEDURE SUStrToLInt (NS : SUStrP; VAR N : LONGINT;
                           VAR CState : ConvNState);
      { SUStrToLInt converts a string to a LONGINT.  It behaves just like
        SUStrToInt otherwise. }
```

The ProgComm Unit

Contents

The ProgComm Unit

1 Introduction

ProgComm is an intrinsic unit in SULib that allows programs to communicate with the shell and with other programs. Three basic mechanisms are provided:

- *Set-Next-Run Command.* A program can tell the Workshop shell what to run next. The specified program will be run after the current program is done, taking precedence over even an exec file in progress.
- *The Program Return String.* The return string can be set by your program and accessed from the exec processor (via the RETSTR function). This allows exec scripts to be written that make choices based on program results.
- *The Communication Buffer.* The communication buffer is a 1K byte buffer global to the Workshop for communication between programs. A set of primitives supporting character- and line-oriented I/O to and from the buffer is provided.

These mechanisms can be used in conjunction with each other. For example, a program can write a series of invocation arguments to the communication buffer and then tell the shell which program to run next. This second program can check the communication buffer to find its arguments. Programs can be written so that, by convention, they first check the communication buffer for their arguments, and then prompt for input from the console only if the arguments are not found in the buffer.

2 ProgComm Routines

This section describes the ProgComm unit interface.

2.1 Initialization

The PCInit procedure initializes the ProgComm unit so that a program may use it.

Procedure PCInit;
PCInit should be called before using the ProgComm unit. The program's return string (RETSTR in the exec language) is initialized to the null string.

2.2 Set-Next-Run and the Return String

The PCSetRunCmd and PCSetRetStr procedures let a program set what program will run next and pass back a return string to the exec processor. The SUStr type comes from the Standard Unit (StdUnit in SULib), which provides a number of string-manipulation routines.

Procedure PCSetRunCmd (RC : SUStr);
PCSetRunCmd lets a program tell the shell what program or exec file to run after the current program terminates, allowing program chaining. RC, the run command passed to PCSetRunCmd, should be a string with the same program pathname or exec file invocation you would give to the Workshop Run command. The run command set in this way will take precedence over any keyboard type-ahead and over any pending exec file commands.

If you want to use PCSetRunCmd to run a Workshop tool normally invoked from the Workshop menu line, set RC to the two-character string consisting of an escape (CHR(27)) and the appropriate menu command letter. This is necessary because typing *E* to invoke the Editor is not always the same as saying Run Editor.OBJ. The Run command looks for Editor.OBJ on the three prefix volumes, while the E menu command looks on the Workshop boot volume first and then on the prefix volumes. (Note that only some items in the Workshop menu are actually separate tools that can be Run.)

Starting to run an exec file while you are already running another exec file causes the first one to be terminated so the second can run. This means that if exec file A runs program P, and P calls PCSetRunCmd to run exec file B, then, when program P terminates, exec file A will also be terminated so exec file B can run. Exec file A will not be resumed when exec file B has completed.

Procedure PCSetRetStr (RS : SUStr);
PCSetRetStr lets a program set a return string that can be accessed through the exec processor's RETSTR function. This lets exec files make choices based on information passed back to the shell by cooperating programs. How the return string is used and interpreted is up to you, and depends on what sort of information you want to pass back to the exec processor.

2.3 The Communication Buffer

The following procedures and functions operate on the *communication buffer,* a 1K byte buffer global to the Workshop shell (that is, it stays around between program invocations). The buffer can hold any type of information; a standard set of functions is provided for Pascallike character- or line-oriented access to the buffer.

Following are some constant, type, and variable declarations from the ProgComm interface which relate to the communication buffer.

```
CONST
  { communication buffer content types }
  PCNone    = -1;    { nothing in buffer }
  PCAny     = 0;     { for PCReset to match any content type  }
  PCText    = 1;     { text, as supported by PCGets & PCPuts  }
  PCBufrMax = 1023;  { max buffer index, ie, bufr is 1K bytes }
```

```
TYPE
  PCBufrP     = ^PCBufr; { pointer to bufr }
  PCBufr      = PACKED ARRAY [0..PCBufrMax] OF CHAR;
VAR
  PCBufrPtr   : PCBufrP; { points to bufr after successful open }
```

The communication buffer is given a *type* when it is opened for writing with PCReWrite. This type will be used to determine whether a potential reader trying to open the buffer with PCReset will be successful. The intent is to prevent reading of the buffer when the contents are not of the type expected by the reader. Three predefined constants are provided for buffer-typing: PCNone means that the buffer has no contents; PCText means that the buffer contains standard text with CR line delimiters; and PCAny matches any type, allowing a reader to override the typing mechanism. Other buffer content types (such as mouse events) may be defined by users, choosing a number to identify the new type that doesn't conflict with the predefined types. The only restriction is that communicating programs must have compatible conventions. To use the buffer for something other than text, use PCBufrPtr to access the buffer (using whatever means of interpretation of the buffer is desired).

The buffer also has an *access key*, which functions in much the same way as the content type (i.e., writers set it and readers must match it to gain access to the buffer). The intent of the access key is to prevent programs from reading the buffer when they are not the intended recipient. The access key should be established by agreement between communicating programs. If a buffer writer does not care about preventing unintended access to the buffer, the null string can be used for the access key. Note that the access key is case sensitive.

Following are the routines for opening and closing the communication buffer.

Procedure PCReWrite (WriteType: INTEGER; Key: SUStr);
PCReWrite opens the communication buffer for writing. The content type and access key are set. PCBufrPtr is set to point to start of the communication buffer. A PCReWrite will override any previous use of the buffer; that is, it will flush any previous buffer contents. WriteType should be an integer identifying the type of data you plan to write to the buffer. If you are planning to use the text-oriented primitives provided, WriteType should be PCText; otherwise, WriteType should be some integer established by agreement between the communicating programs. Key should be a string also established by agreement between the communicating programs. A useful form of key is one that identifies the intended recipient, so that contents left in the buffer are not read inadvertently by programs for which they were not intended.

Function PCReset (ReadType: INTEGER; Key: SUStr): BOOLEAN;
PCReset opens the buffer for reading. The boolean result will indicate whether the open was successful. The open will fail if ReadType does not match the type set by the last buffer writer or if Key does not match the key set by the last writer.

Function PCClose (KillBufr: BOOLEAN; Key: SUStr): BOOLEAN;
PCClose will close (or empty) the communication buffer. If KillBufr is true, the buffer will be emptied. In general, the buffer can be read more than once (by multiple readers) if desired. If a reader is finished with the buffer and knows that no one else should read the buffer, PCClose should be called with KillBufr set to true. The call to PCClose will fail if the access key does not match. PCClose may be used to flush buffers that were written by someone else, as long as you know the access key. PCClose may be called without calling PCReset or PCReWrite first.

2.4 Reading from and Writing to the Communication Buffer

The following functions provide a text-oriented buffer facility with Pascallike character- and line-oriented reads and writes.

Function PCPutCh (Ch: CHAR): BOOLEAN;
PCPutCh puts a character into the buffer. The boolean result indicates whether the operation was successful. It fails if the buffer is full or if the buffer was never opened successfully for writing. Note that PCPutCh(CR) is equivalent to PCPutLine('').

Function PCGetCh (VAR Ch: CHAR): BOOLEAN;
PCGetCh gets a character from the buffer. The boolean result indicates whether the operation was successful. It fails if the buffer is empty or if the buffer was never opened successfully for reading.

Function PCPutLine (L: SUStr): BOOLEAN;
PCPutLine puts a line into the buffer. A CR is put in the buffer following the string passed to PCPutLine. The boolean result indicates whether the operation was successful. It fails if the buffer is full or if the buffer was never opened successfully for writing.

Function PCGetLine (VAR L: SUStr): BOOLEAN;
PCGetLine gets a line from the buffer, where a line is the text from the current buffer pointer to the next CR or the end of file (whichever comes first). The boolean result indicates whether the operation was successful. It fails if the buffer is empty or if the buffer was never opened successfully for reading.

2.5 Internal Workshop Function

You will notice the following function in the ProgComm interface; it is used for special-purpose communication between the Workshop shell and various Workshop tools.

Function PCShellCmd (Cmd: INTEGER; P: SUStrP): BOOLEAN;
For internal use by Workshop tools only. Don't use this function.

3 Interface

```
INTERFACE

  USES
    {$U StdUnit    } StdUnit,
    {$U ShellComm } ShellComm;

  CONST
    { communication buffer content types for use with PCReset and PCReWrite }
    PCNone    = -1;       { nothing in buffer }
    PCAny     = 0;        { for PCReset to match any buffer content type }
    PCText    = 1;        { text, as supported by PCGet's and PCPut's below }

    PCBufrMax = 1023;     { max Bufr index, ie, comm bufr is 1K bytes }

    { command constants for PCShellCmd }
    PC_SetReallyStop   = 1;    { determines if SUStopExec really stops exec
                                  files }                          {ff 3/7/84}
    PC_GetReallyStop   = 2;
    PC_SetUnSavedEdits = 6960; { tells if unsaved edits are left in the
                                  editor     }                    {ff 3/12/84}
    PC_GetUnSavedEdits = 8751;

  TYPE
    PCBufrP   = ^PCBufr; { ptr to communication buffer }
    PCBufr    = PACKED ARRAY [0..PCBufrMax] OF CHAR;

  VAR
    PCBufrPtr : PCBufrP; { will point to PCBufr after successful PCReset or
                          PCReWrite }

  PROCEDURE PCInit;
    { PCInit should be called before using the ProgComm unit.  One effect of
note is that the program's return string (RetStr) is initialized to the null
string. }
  PROCEDURE PCSetRunCmd (RC : SUStr);
    { PCSetRunCmd enables a program to tell the shell what program (or exec
file) to run after the current program terminates, which allows program
"chaining".  The run command set in this way will take precedence over any
keyboard type-ahead and over any pending exec file commands. }
  PROCEDURE PCSetRetStr (RS : SUStr);
    { PCSetRetStr allows a program to set a return string which may be
accessed via the Exec Processor's RETSTR funciton.  This allows exec files to
make choices based on information passed back to the shell by cooperating
```

programs. How the return string should be used and interpreted is up to you, and will depend on what sort of information you want to pass back to the exec processor. (But in order to be a good citizen it is probably best to follow whatever system-wide conventions emerge and prevail.) }

{ The following procedures and function operate on the COMMUNICATION BUFFER, which is a 1K byte buffer which is global to the Workshop shell. The buffer can hold essentially any type of information, but a standard set of functions is provided for Pascal-like character or line-oriented access to the buffer.
The communication buffer is given a TYPE when it is opened for writing with PCReWrite. This type will be used to determine whether a potential reader trying to open the buffer with PCReset will be successful. The intent is to prevent reading of the buffer when the contents are not of the type expected by the reader. Three predefined constants are provided for buffer typing (PCNone which means the buffer has no contents; PCText which means that it has standard text with CR line delimiters; and PCAny which will match any type, allowing a reader to override the typing mechanism). Other buffer content types (such a mouse events) may be defined by users, choosing some number to identify the new type which does not conflict with the predefined types. We make no attempt here to provide a complete set of predefined types; the issue is simply one of having compatible conventions (agreement) between communicating programs. To use the buffer for something other than text, the variable PCBufrPtr may be used to access the buffer (using whatever means of interpretation is desired).
The buffer also has an ACCESS KEY, which functions in very much the same way as the content type (ie, writers set it and readers must match it to gain access to the buffer). The intent of the access key is to prevent programs from reading the buffer when they are not the intended recipient. The access key, again, is something that should be established by agreement between the communicating programs. If a buffer writer does not care about preventing unintended access to the buffer, the null string can be used for the access key. Note that the access key is case sensitive. }

```
  PROCEDURE PCReWrite (WriteType : INTEGER; Key : SUStr);
    { PCReWrite opens the buffer for writing. The contents type and access
key are set. PCBufrPtr is set to point to the communication buffer. }
  FUNCTION  PCReset (ReadType : INTEGER; Key : SUStr): BOOLEAN;
    { PCReset opens the buffer for reading. The boolean result will indicate
whether the open succeeded. The open will fail if contents type and access
key do not match the type and key set by the last buffer writer.}
  FUNCTION  PCClose (KillBufr : BOOLEAN; Key : SUStr): BOOLEAN;      {ff 2/2/84}
    { PCClose will close the buffer. If KillBufr is true the buffer will be
emptied. In general, the buffer can be read more than once (by multiple
readers) if desired. If a reader is finished with the buffer and knows that
no one else should read the buffer, PCClose should be called with KillBufr set
to true. The call to PCClose will fail if the access key does not match. }
```

```
  FUNCTION  PCPutCh (Ch : CHAR) : BOOLEAN;
    { PCPutCh will put a character into the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if the buffer is
full or if the buffer was never opened successfully for writing. }
  FUNCTION  PCGetCh (VAR Ch : CHAR) : BOOLEAN;
    { PCGetCh will get a character from the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if there is
nothing more to read or if the buffer was never opened successfully for
reading. }
  FUNCTION  PCPutLine (L : SUStr) : BOOLEAN;
    { PCPutLine will put a string into the buffer, followed by a CR.  The
boolean result will indicate whether the operation was successful.  It will
fail if the buffer is full or if the buffer was never opened successfully for
writing. }
  FUNCTION  PCGetLine (VAR L : SUStr) : BOOLEAN;
    { PCGetLine will get a line from the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if there is
nothing more to read or if the buffer was never opened successfully for
reading. }
  FUNCTION  PCShellCmd (Cmd : INTEGER; P : SUStrP): BOOLEAN;       {ff 3/7/84}
```

QuickPort Programmer's Guide

Contents

Preface

About This Manual

This manual describes QuickPort, a set of private and intrinsic units that facilitate porting Pascal programs to the Lisa desktop. This manual is written for experienced Lisa Pascal programmers who are already familiar with the Lisa Workshop and the Lisa Operating System and who understand the concepts and conventions used by the Lisa User Interface. In addition, those who intend to write terminal emulators are assumed to know Clascal.

For material not covered in this manual, refer to one of the listed documents for additional information:

- *Operating System Reference Manual for the Lisa.*
- *Workshop User's Guide for the Lisa.*
- *Lisa Internals Manual.*
- *Lisa User Interface Guidelines.*
- *An Introduction to Clascal.*

Chapter 1
Introduction

Introduction

1.1 What is QuickPort?

QuickPort is a set of private and intrinsic units that provide a fast and reliable way to run Pascal programs in the Lisa Office System. By using QuickPort, you can make a few changes in a typical Pascal program, and it will run on the Lisa desktop. Applications that use QuickPort are integrated so that you can cut and paste to and from other Lisa applications. QuickPort also provides standard menus for all applications that use it.

1.2 Types of QuickPort Applications

Before you can use QuickPort to port your application to the Desktop, your program must

- Run in the Lisa Workshop.
- Use only **readlns** and **writelns** for text input and output.

A Pascal program that runs in the Lisa Workshop and uses **readlns** and **writelns** for text input and output is called a "vanilla" Pascal program. Vanilla Pascal programs can be ported to the desktop with very few changes.

You can also use QuickDraw calls for graphics, use the mouse to get input, and use a subset of the Lisa Hardware Interface. However, the addition of a graphic panel and use of the hardware interface involves more coding to acheive the port than a vanilla Pascal program.

1.3 Additional Features

QuickPort also provides a set of additonal procedures for configuring the panels, text output, graphic output, and for applications that use the hardware interface. Using these features, you can increase the power of your application. The additional QuickPort features are described in Chapter 3.

Chapter 2
Using QuickPort

Using QuickPort

2.1 QuickPort Program Requirements

Vanilla Pascal programs need nothing but the addition of one or two list elements to the USEs statement in its main program. A vanilla Pascal program runs in the Lisa Workshop and uses only **readlns** and **writelns** for input and output. You can use QuickDraw, but there are some minor changes required. See Section 3.4.1.1, QuickDraw Requirements, in Chapter 3, for more information. If you use the Lisa Hardware Interface, you must modify your program and use the QuickPort Hardware Interface. The QuickPort Hardware Interface is a subset of the Lisa Hardware Interface; it is described in Section 3.11, Procedures for the QuickPort Hardware Interface, in Chapter 3.

If your program is a vanilla Pascal program, you can either enhance it using the QuickPort features described in Chapter Three, or port it directly to the Lisa Desktop. If you wish to port your program to the Lisa Desktop without using any of the additional QuickPort features, make sure your program works in the QuickPort execution environment described in Section 2.3, and then turn to Chapter Four: Bringing Your Application to the Lisa DeskTop.

2.2 Choices for QuickPort Applications

You can produce several different types of applications using QuickPort:

- Applications that produce text output only.
- Applications that use QuickDraw to produce graphic and/or text output.
- Graphic applications that use the QuickPort Hardware Interface to get mouse input in the graphic panel.

QuickPort provides three panels: the text panel, the input panel, and the graphic panel. The text panel saves all text output, unless the Don't Save Buffer command is chosen from the Edit menu. Any application that produces text output only gets a text panel automatically. The input panel displays text that has not been read by the program. You can choose to have the input panel or not; the default is no input panel. Any application that produces graphic output only gets a graphic panel. Such programs can use in addition, a text panel, and/or an input panel. The default is one panel.

The text and graphic panels can both be scrolled vertically and horizontally. The panels can be enlarged and shrunk to provide different views of the output. Both panels can be split vertically and horizontally, allowing the user to see different parts of the output at the same time.

2.3 The QuickPort Execution Environment

One of the most important things to remember when using QuickPort is that the Lisa Desktop is a multiprocessing intergrated environment and you can affect the state of other applications running on the desktop if you don't keep this in mind. Be particularly careful about using functions in the QuickPort Hardware Interface, because these functions change the state of hardware, thus affecting all applications (including the desktop).

QuickPort programs can be run in the background (inactive window) when they are not waiting for input. When a program running in the background needs input, it is suspended. Programs running in the background compete with the active window for CPU time. Programs with long CPU-bound loops should use either **Yield_CPU** or **QPYield_CPU** to yield the CPU to the active window.

User actions such as pulling down the menus and clicking the mouse are processed only when your program calls call screen I/O (WRITEs and **READs**, etc.). If you have a long CPU-bound loop, be sure to use either **Yield_CPU** or **QPYield_CPU**, so that your program will be more responsive to the user. If you have a tight loop, there is no way for the user to break out of the loop, unless the debugger is loaded and you can hit the NMI key to halt the process. Be sure to put **Yield_CPU**, **QPYield_CPU**, or **PAbortFlag** in any tight loops. Note that you must call **QPConfig** to pass an ⌘-period to your program if you need to call **PAbortFlag**. **QPConfig** is described in Section 3.6 of Chapter 3.

2.3.1 Using Operating System Calls

You can make any operating system calls, but remember that Lisa has a multiprocessing environment. Whenever a document is opened, a process may be created (tools that handle multiple documents create one process that handles one or more documents). If two documents are opened from the same tool, you have two processes running separate instances of the same program. This could result in inconsistent data if **Write_Datas** and **Read_Datas**, or **Rewrites** and RESETs are performed on the same file. If this is undesirable, you should add additional code to your application to check whether the file can be opened by more than one process.

2.3.1.1 Yield_CPU

Yield_CPU gives the CPU to any other ready process, but does not handle any user actions, such as pulling down menus, and moving windows. QuickPort provides an alternative procedure, **QPYield_CPU**, that allows the user to pull down menus and move the windows around.

2.3.1.2 Make_process

If you call **make_process** in a QuickPort application, the resulting processes cannot do any screen input and output.

2.3.1.3 LDSNs (Logical Data Segment Numbers)

You cannot use a logical data segment number less than 5, or larger than 11. Note that LDSN 5 is, by default, used by the Pascal heap. If you use a

Pascal heap larger than 128K bytes, LDSN 6 and up will be used for the heap. You can use **PLInitHeap** to change the Pascal heap to a different LDSN, but make sure you don't collide with the system LDSNs.

- LDSNs 1-4 -- QuickPort
- LDSN 5 -- Default Pascal heap
- LDSN 11 -- **OPEN** **'-printer'**, **RESET**, or **REWRITE** **'-printer'**
- LDSNs 12-16 -- LisaLibraries

2.3.1.4 Terminate_Process, Kill_Process

QuickPort programs should not call **Terminate_Process** or **Kill_Process**. These calls will terminate the program, leaving the user with no chance to do anything with the output. If you need to terminate program execution, use **halt** or drop through to the end statement of your program. ***PROGRAM TERMINATED*** will appear on the screen, and the user will be able to save and put away, copy, or print.

2.3.1.5 Terminating the Program Abnormally

TrmntExceptionHandler is the standard QuickPort exception handler for abnormal termination of a program. You can write your own terminate exception handler, but you must call **TrmntExceptionHandler** immediately in your exception handler. If this call is not made, the system will hang because QuickPort will not have a chance to clean up and transfer control to the desktop manager.

2.4 The QuickPort User Interface

QuickPort provides a standard user interface for its applications that is, with the exception of a few menu commands, the same as the standard Lisa user interface. Manipulating windows and using the mouse follow the standard Lisa user interface, as do opening and closing documents.

QuickPort provides some menu commands that are different from the standard Lisa menu commands. These commands allow the user to control program execution. A standard Lisa application continuously loops to get and process events. A QuickPort program, however, may run from beginning to end. When a QuickPort program reaches its end, it will not respond to input from the keyboard, and its window will remain open to allow the user to view the output. At this stage, the QuickPort application is idle, waiting for one of the following menu commands:

- Set Aside -- Places the document (without saving) in its icon on the desktop. If the document is reopened, the application will still be idle.
- Save & Put Away -- Saves the document. The process is then terminated. If this document is opened again, the program will not run

immediately -- it is waiting for the Restart command. If the user wants to browse through the document, it is not necessary to use the Restart command. Instead, use Save & Put Away, or Set Aside.

- Restart -- Restarts program execution.

QuickPort applications are started, from the desktop, by tearing off a document from the stationery pad and opening the document.

The QuickPort menus are discussed in Appendix A.

Chapter 3
Advanced QuickPort Features

Advanced QuickPort Features

3.1 Introduction to the Features

QuickPort provides a set of features that you can use to enhance your application. The additional procedures and functions are for

- Configuring the text and graphic panels.
- Controlling text output.
- Handling graphic output using the mouse for input.
- Providing printer support.
- Using the QuickPort hardware interface.
- Making use of the terminal emulators.

You can combine any of these procedures and functions within a QuickPort application.

You can also write your own terminal emulator. To do this you must know enough Clascal to understand subclasses, methods, and overriding methods. Read *An Introduction to Clascal* before attempting to write your own terminal emulator. See Appendix B, Writing Your Own Terminal Emulator for more information.

The logical device, '**-printer**', behaves in much the same way as it does in the Workshop, but also interacts with the Desktop's print manager. A section on printer support is included in this chapter.

3.2 Text Input and the Input Panel

QuickPort programs get input in two ways: from the keyboard, and from the clipboard. The input panel displays the text that has not yet been consumed by the program. Text in the input panel comes from two sources: "type ahead" text (text which is entered from the keyboard too quickly to be echoed immediately by the program), and text from the clipboard that will be "pasted" into the text window. The 'Read Input from Clipboard' command places the selected text in the input buffer. When the program does a **read**, the text in the input buffer is read firstd. If the input buffer is empty, the **read** waits for input from the keyboard or from a paste command.

3.3 Text Output and the Text Panel

The text output panel displays the **writeln** output from the program. The text panel corresponds to the Pascal device **output** and the logical device

'**-console**'. The text panel emulates a terminal display. The default size of the screen area is 24 lines by 80 columns. The width of the text panel can be changed either by the program, or by the user from the Setup menu. The Setup menu is described in Appendix A.

The text panel has a *buffer area* that saves text as it is scrolled above the screen area. The size of the buffer area is increased automatically as lines are saved. The size of the buffer is limited to the amount of memory available to increase the size of the buffer. When the buffer size reaches its limit, the lines scrolled off the top of the buffer area will not be saved. The limit is approximately 3500 80-character lines. The user can choose to save or not save scrolled output using the Setup menu. The Edit menu is described in Appendix A.

The screen area has a cursor that is affected by **readlns** and **writelns** from the program. The cursor position is always:

- Inside the screen area.
- Relative to the top left position of the screen area.

The cursor position is the insertion point for input. No menu commands change the logical cursor position; it is controlled solely by the program. The cursor position is always visible when there is a **read** from the program. In other words, if the panel has been scrolled so that the cursor position is hidden, QuickPort scrolls back to the cursor position when encountering a **read**. The cursor home position is the top left position of the screen area.

3.4 Graphic Output, the Graphic Panel, and Mouse Input

Graphics in QuickPort applications are created by QuickDraw. QuickPort provides an option that allows you to choose two panels, one for text output and one for graphic output, or one panel for both text and graphic output. The graphic panel corresponds to the Workshop screen. The screen size is 720 pixels wide and 364 pixels high. The entire graphic panel is equal to the screen area in the text panel. There is no buffer area in the graphic panel because graphic output will not be scrolled out of the graphic panel. All graphic objects created by the program are saved in the graphic panel using a QuickDraw picture.

In the text panel, the mouse is used to select text. In the graphic panel, mouse clicks are saved and passed to the program. Whenever the mouse button is pressed inside the graphic panel, a mouse event, **mouseDown,** with the mouse location is saved. When the mouse button is pressed while the mouse is moved, another mouse event, with different locations, is saved. When the mouse button is released, a **mouseUp** event is saved. To see if there are any mouse events in the queue, call **MouseEvent**. **MouseEvent** returns one event at a time, until there are no more mouse events in the queue. When **MouseEvent** is called, if the mouse button is down, control will not be returned to the caller until the button is released. For this reason, **VGetMouse** should not be used after a call to **MouseEvent**, because the mouse may be moved. Each mouse event stores a mouse location indicating where the mouse button was pressed. **VGetMouse** lets you track the mouse location when the mouse button is not down.

For more information on **MouseEvent**, Refer to Section 3.8.1.3.

3.4.1 QuickDraw Requirements

Pascal programs that run in the Lisa Workshop and use QuickDraw, call QDINIT and **OpenPort** (in the QD/Support unit). To use QuickDraw you must

- Remove the call to QDINIT and **OpenPort**. QuickPort initializes QuickDraw and opens a grafPort for drawing to the graphic panel.
- Not open a picture in this grafPort since QuickPort uses a picture to save the graphic output.
- Not customize low-level QuickDraw drawing routines in this grafPort.

If your program needs to use pictures, you can open a picture in another grafPort. If your program needs to redefine any of the QuickDraw low-level routines, you can do this in another grafPort. If your application uses multiple grafPorts, you must switch to the QuickPort grafPort whenever you want to draw to the screen.

If your application calls **DrawPicture**, you must call another QuickDraw drawing routine before calling **DrawPicture**. This is because QuickPort opens the picture when the first QuickDraw drawing routine is encountered. If **DrawPicture** is the first drawing routine encountered, QuickPort's picture will be opened incorrectly because QuickPort can handle only one picture at a time. Here is an example showing how to avoid such collisions:

```
GetPort (sysportptr); {saves system port}
OpenPort (@myPort);  {references alternate port}
myPicture := OpenPicture (thePort^.portRect);

. . . make  your QuickDraw calls here . . .

ClosePicture;
SetPort (sysportptr);    {switches to system grafPort}
EraseRect (thePort^.PortRect); {opens system picture
                                -- any drawing routing can
                                be used}
DrawPicture (myPicture, thePort^.PortRect);
```

If you call **OpenPicture** while the QuickPort grafPort is the current port, the following alert message appears on the screen and the program is aborted:

Your QuickPort tool has called another OpenPicture inside the QuickPort grafPort. This tool will be aborted.

The QuickDraw procedure **ScrollRect** is not supported by QuickPort. **ScrollRect** is not supported because QuickPort uses a picture to save the graphic output, and the effect of **ScrollRect** is not saved in a picture. This means that if the user scrolls the window, the picture is redrawn to the window as if **ScrollRect** had not been called.

The size limit for the QuickPort picture is 32K bytes. When the picture approaches this size, an alert is displayed. Subsequent graphic output is displayed on the screen, but is not saved in the picture. As the size of the picture increases, the redrawing that happens as the picture is scrolled or the window moved slows. You can find out the current picture size by calling **QPGrafPicSize**. Once the picture size reaches 32K bytes, the only way to save the remaining graphic output is to **EraseRect** the entire screen (**thePort^.PortRect**). The effect of this call is to delete the old picture and create a new picture.

You can draw bit images in the QuickPort grafPort. The entire graphic panel, including the bit images, can be printed. You can copy the bit images to a LisaWrite document, but you cannot copy bit images from a QuickPort application to a LisaDraw document.

3.5 Required Change to Your Program

Before you can call any of the additional QuickPort procedures, you must add **UQPortCall** to your USES list:

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortCall}  UQPortCall,
{$U QP/UQuickPort} UQuickPort; {or UQPortGraph,  or
                     UQPortVT100, or UQPortSoroc}
```

3.6 Procedures for all Applications

3.6.1 Configuring the Panels — QPConfig

You can choose several different ways to orient the panels in QuickPort applications. The procedure **QPConfig** lets you rearrange the panels and their orientations. Figure 1 shows some of the different layouts.

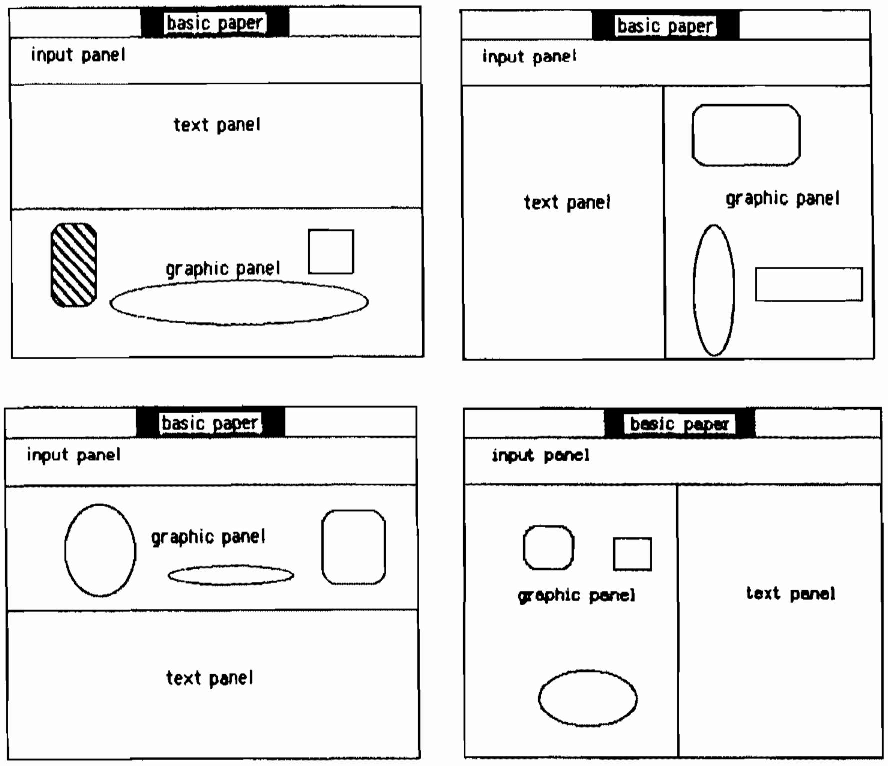


Figure 1.
QuickPort Window Layouts

Call the **QPConfig** procedure from your main program before any screen input and output is performed. You must set all the fields of a global variable of type **TQPConfigRec**.

```
 PROCEDURE QPConfig (config : TQPConfigRec);
```

where

```
        TQPConfigRec = RECORD
        tosaveBuffer     : BOOLEAN; {save lines in
                           buffer}
        passApplePeriod  : BOOLEAN; {pass apple '.' to
                           main program}
        showInputPanel   : BOOLEAN; {display input
                           panel}
        CASE twoPanels   : BOOLEAN OF {have both text
                           and graphic panels}
        TRUE : (vhs : VHSelect; {vertical or
                           horizontal split.  VHSelect
                           is defined in QuickDraw}
        grPanelSize : INTEGER); {initial width or
                           height in pixels, if < 0,
                           text panel is below or right
                           of the graph panel}
    END;
```

If **QPConfig** is not called, the default values are used. These defaults are in effect only if **QPConfig** is never called. If you call **QPConfig** you must set all fields, or else they will be undefined.. The default values are:

tosaveBuffer	false
passApplePeriod	false
showInputPanel	false
twoPanels	false

The graphic and text panels can be oriented in several different ways on the screen. To use **QPConfig** to set up the panels, you must first declare a variable of type **TQPConfigRec**. For example,

```
      .
      .
      .
  VAR
  MyConfig:          TQPConfigRec;
      .
      .
      .
  QPConfig(MyConfig);
      .
      .
      .
```

To have both a graphic and a text panel, **twoPanels** must be TRUE. You must initialize the **vhs** field if you set **twoPanels** to TRUE. Once you have two panels, you can choose to split the windows on the screen vertically or horizontally. Refer to Figure 1 to see what the screen looks like with vertical and horizontal splits between windows. Then you can set the **grPanelSize** field to the size you want the graphic panel when the document is first opened (the text panel will take up the remaining space in the window).

If **QPConfig** is not called, the default values are used. Programs that handle only text output have a default of one text panel. Programs that handle graphic output have a default of one graphic panel.

3.7 Procedures for Using the Text Panel

The procedures for QuickPort applications that produce text output allow you to:

- Change the terminal parameters.
- Get raw input from the console.
- Clear the screen.
- Control the cursor.
- Set and clear tabs.
- Control keyboard input.
- Change the character style.

3.7.1 Changing the Terminal Parameters -- SetupTermPara

SetupTermPara sets the terminal parameters for the screen area in the text panel. You can call **SetupTermPara** from your terminal emulator or from your main program, but the call must be made before any screen input or output is performed. If **SetupTermPara** is not called before performing screen input or output, the default parameters will not be changed. If you call **SetupTermPara** you must set all parameters.

```
PROCEDURE SetupTermPara (termpara : TTermPara);
```

where

```
maxPosLines    = 50; {max possible lines for any
                      terminal emulator}
maxPosColumns  = 132;

Tcursorshape = (blockshape, underscoreshape,
                invisibleshape);

TTermPara = RECORD
   rowsize            : 1..maxPosLines;
   columnsize         : 1..maxPosColumns;
   toWraparound       : BOOLEAN;
   keytoStopOutput    : CHAR;
   keytoStartOutput   : CHAR;
   tmcursorShape      : Tcursorshape;
END;
```

If **SetupTermPara** is not called, the default values are used:

rowsize	24 lines
columnsize	80 columns
toWraparound	TRUE
keytoStopOutput	⌘-S
keytoStartOutput	⌘-Q
tmcursorShape	Block

3.7.2 Getting Raw Input from the Console — Vread

You can use **Vread** instead of **read** to get keyboard input and the control keys. **Vread** does not echo characters as they are **read**.

```
PROCEDURE Vread (VAR ch: CHAR; VAR keycap : QPByte;
                        VAR applekey,  shiftkey,
                        optionkey: BOOLEAN);
```

The keycap is useful when you need to distinguish the numeric keypad from the main keyboard. Refer to Section 3.11.4 for the keycap definition. Note that the option key is typically used to generate extended Lisa characters. The extended Lisa characters are those characters in the range above ASCII 127. Try not to use the option key for other purposes to avoid confusing the users.

3.7.3 Clearing the Screen — ClearScreen

ClearScreen provides six different ways to clear all or part of the screen. The six ways are:

- Clear the whole screen.
- Clear from the cursor position to the end of the screen.
- Clear from the beginning of the screen to the cursor position.
- Clear the whole line.
- Clear from the cursor position to the end of line.
- Clear from the beginning of the line to the cursor position.

```
PROCEDURE ClearScreen (clearkind : INTEGER);


{clearkind definition for ClearScreen procedure}
  sclearScreen  = 1;     {clear the whole screen}
  sclearEScreen = 2      {clear to the end of the
                        screen}
```

```
sclearBScreen = 3      {clear from the beginning
                      of the screen to the cursor
                      position}
sclearLine    = 4      {clear the whole line}
sclearELine   = 5;     {clear to end of line}
sclearBLine   = 6;     {clear from the beginning
                      of the line to the cursor
                      position}
```

3.7.4 Controlling the Cursor -- VGotoxy and MoveCursor

3.7.4.1 VGotoxy

VGotoxy moves the cursor to a specified position in the window.

```
PROCEDURE VGotoxy (x, y : INTEGER);
```

VGotoxy is the same as the Pascal **gotoxy**, but faster.

3.7.4.2 MoveCursor

MoveCursor moves the cursor to a position in the window *relative to the current cursor position.* **MoveCursor** allows vertical scrolling only.

```
PROCEDURE MoveCursor (scroll : BOOLEAN; xdistance,
                      ydistance : INTEGER);
```

For the **xdistance, ydistance** parameters:

- A positive value moves the cursor to the right or down.
- A negative value moves the cursor to the left or up.

If the cursor is moved down, and **scroll** is **TRUE**, the output will be scrolled up.

3.7.5 Setting and Clearing Tabs -- SetTab and ClearTab

3.7.5.1 SetTab

SetTab sets a tab at a specified column, or at the current cursor position.

```
PROCEDURE SetTab (column : INTEGER);
```

SetTab sets tab at current cursor position if **column** <0.

3.7.5.2 ClearTab

ClearTab clears a tab at a specified column, or at the current cursor position.

```
PROCEDURE ClearTab (clearAll : BOOLEAN; column : INTEGER);
```

ClearTab clears tab at current cursor position if **column** <0.

3.7.6 Controlling Keyboard Input — StopInput and StartInput

3.7.6.1 StopInput

StopInput prevents recognition of keyboard input until **StartInput** is called.

```
PROCEDURE StopInput;
```

3.7.6.1 StartInput

StartInput allows recognition of keyboard input.

```
PROCEDURE StartInput;
```

3.7.7 Changing the Character Style — ChangeCharStyle

ChangeCharStyle changes the character attributes to any style combination defined by QuickDraw.

```
PROCEDURE ChangeCharStyle (newstyle : Style);
```

3.8 Procedures for Using the Graphic Panel

The procedures for QuickPort applications that produce graphic output allow you to use the mouse to get input. These procedures are:

- Get the current mouse location.
- Test to see if the mouse button is up or down.
- Get a mouse event.
- Get either mouse or keyboard input.

3.8.1 Mouse Routines

The mouse routines listed in this section should be used instead of the ones in the Lisa Hardware Interface.

MouseEvent is a polling function. Programs may loop on **MouseEvent** to wait for mouse input. This unnecessarily takes up CPU time. Also, if the application is run in the background, **MouseEvent** will force it to run periodically, just to find out there is no mouse input, and then control is returned to the active window. This slows down the execution and user response in the active window.

WaitMouseEvent is a blocking procedure. **WaitMouseEvent** will not return to the caller until there's a mouse event, allowing user actions to be processed immediately when there are no mouse events. When a program that uses **WaitMouseEvent** is in the background, it is suspended and consequently does not take CPU time from the active window.

3.8.1.1 VGetMouse

VGetMouse returns the current mouse location in the coordinates of the current grafPort.

```
PROCEDURE VGetMouse (VAR pt : Point);
```

Point is a type defined in QuickDraw. Refer to The Lisa Pascal Reference Manual, Appendix C, QuickDraw for the definition of **Point**.

3.8.12 MouseButton

MouseButton returns the current state of the mouse button.

```
FUNCTION MouseButton : BOOLEAN;
```

3.8.13 MouseEvent

MouseEvent returns a mouse event if there is one in the queue, and returns FALSE if there is not a mouse event in the queue. A mouse event is:

- A mouse buttondown (when the user presses the mouse button).
- Mouse motion while the button is pressed.
- A mouse buttonup (when the user releases the mouse button).

Moving the mouse without pressing the mouse button is not a mouse event. When **MouseEvent** is called, if the mouse button is down, control will not be returned to the caller until the button is released.

```
FUNCTION MouseEvent (VAR aMouseEvent : TMouseEvent)
                              : BOOLEAN;
```

where

```
     TMouseEvent = RECORD
     mouseLoc : Point;
     clicknum : INTEGER; {max 3 for triple clicks}
     mouseDown, meShift, meApple, meOption :
                              BOOLEAN;
END;
```

For each mouse down event (**mouseDown = TRUE**), several different **mouseLoc** events may be returned in subsequent calls. These **mouseLoc** events are always ended with a mouse up event (**mouseDown = FALSE**).

For a double click, **MouseEvent** returns events of down, up, down, up with the **clicknum** for the second mouse down event equal to two. If the mouse button is pressed twice, but the presses do not constitute a double click, the same sequence of events is returned, but with the **clicknum** for the second mouse down event equal to one.

For a triple click, **MouseEvent** returns events of down, up, down, up, down, up, with the **clicknum** for the third mouse down event equal to three.

If the **mouseDown** field is FALSE, all other fields are meaningless.

MeShift is TRUE if the mouse button and the Shift key are depressed.
MeApple is TRUE if the mouse button and the key are depressed.
MeOption is TRUE if the mouse button and the Option key are depressed.

3.8.1.5 WaitMouseEvent

WaitMouseEvent gets a mouse event. **WaitMouseEvent** blocks the caller until there is a mouse event in the queue.

You should use this call instead of **MouseEvent** to avoid polling and wasting CPU time. **WaitMouseEvent** also makes a program more responsive to user events such as pulling down menus, clicking in other windows, etc., when the program is waiting for mouse input.

```
  PROCEDURE WaitMouseEvent (VAR aMouseEvent :
                                 TMouseEvent);
where
      TMouseEvent = RECORD
      mouseLoc : Point;
      clicknum : INTEGER; {max 3 for triple clicks}
      mouseDown, meShift, meApple, meOption :
                              BOOLEAN;
END;
```

After **WaitMouseEvent** returns, a call to **MouseEvent** will get the rest of the mouse events.

3.8.1.6 WaitEvent

WaitEvent is a combination of **read** and **WaitMouseEvent**, blocking the caller until there is either keyboard or mouse input.

You should use this call instead of **MouseEvent** and **keypress** if you want both mouse and keyboard input. **WaitEvent** does not reurn input. You must call **read**, **Vread**, or **MouseEvent** depending on the value returned from the call.

```
  PROCEDURE WaitEvent (VAR fromKeyboard : BOOLEAN);
```

3.8.1.7 QPGrafPicSize

QPGrafPicSize returns the size of the picture in the system grafPort.

```
  FUNCTION QPGrafPicSize : INTEGER;
```

3.9 Printer Support

The printer is designated **-printer** by the Workshop. **-printer** is a logical device. To open the printer, use **reset** or **rewrite**, passing **-printer** as the file name. To send output to the printer, use **writeln** or **write**. Use **close** when you're finished sending information to the printer. **Close** lets the printshop manager know that the program is done with the printer and causes the last page to print out. If you do not call **close** after printing is finished, the printer is considered in use, and is unavailable to all other Lisa applications.

The printer is shared by all applications in the printshop. When you send something from a QuickPort application to the printer from QuickPort, you do not get immediate output. First the document is spooled to the printer queue by the printshop manager in the Lisa Office System. If there is nothing in the queue, the information comes out a page at a time. If there is something in the queue at the time of **reset** or **rewrite**, an error message is returned.

You can change the font the printer uses by calling **PrChangeFont**. The default font is 10-point, 10-pitch Century.

Paper size, printing orientation and print resolution can be changed using the Format for Printing command in the File/Print menu. Selections made using the Format for Printing command take effect only after a **reset** or **rewrite**.

The Print and Print As Is commands in the File Print menu print all the output in the selected panel.

3.10 The Terminal Emulators

QuickPort provides three terminal environments: the standard terminal, the VT100 terminal emulator, and the SOROC terminal emulator. This section summarizes the three emulators. If you want to write your own terminal emulator, go to Appendix B, Writing Your Own Terminal Emulator.

3.10.1 The Standard Terminal

The standard terminal is the terminal environment QuickPort uses unless you specify otherwise. The standard terminal provides a set of screen and cursor control functions. The standard terminal does not use escape sequences, but does interpret a set of standard control keys at output: BELL, backspace, horizontal tab, line feed, and carriage return (without line feed). Programs that use **reads** and **readlns** will have the backspace key processed automatically, i.e., the backspace key will not be passed to your program if you use **reads** and **readlns**. If your program needs to get the backspace key, use **vread** instead.

The standard Lisa applications use the ⌘-period combination to terminate long operations. QuickPort provides an option that suspends the program when the ⌘-period key combination is detected. The default is to detect the ⌘-period combination. This option is passed in **QPConfig**, which is described in Section 3.6. When a program is suspended, the user can select the Resume command to resume program execution, or the Save & Put Away command to terminate program execution.

The Setup menu (in all QuickPort applications) lets you select 80 or 132 columns per line, turn wraparound on or off, and set the tab positions.

3.10.2 The VT100 Terminal Emulator

The QuickPort VT100 terminal emulator interprets all VT100 and VT52 escape sequences, with the exception of escape sequences related to host communications. When you use the VT100 terminal emulator, the screen area in the text panel responds to VT100 and VT52 escape sequences from **writes** and **writelns**.

The character styles supported by the QuickPort VT100 terminal emulator are bold, underline, and highlight. Since highlighted text in Lisa applications traditionally means a selection, highlighted text in the VT100 screen area will be *shadowed*. Double-height and double-width characters are not supported.

To use the VT100 terminal emulator, add

{$U QP/UQPortVT100} UQPortVT100;

to the USES list at the beginning of your main program. For more information, refer to Section 4.1, Adding the USES List Elements, in Chapter 4.

3.10.3 The Soroc Terminal Emulator

Pascal programs that run in the Lisa Workshop, and on the Apple II or Apple III, use Soroc escape sequences for output display. QuickPort provides a Soroc-compatible terminal emulator to help port these applications to the Lisa desktop. The QuickPort Soroc terminal emulator interprets all Soroc escape sequences, with the exception of those escape sequences related to display protection.

To use the Soroc terminal emulator, add

{$U QP/UQPortSoroc} UQPortSoroc;

to the USES list at the beginning of your main program. For more information, refer to Section 4.1, Adding the USES List Elements, in Chapter 4.

3.11 Procedures for the QuickPort Hardware Interface

The QuickPort hardware interface is a subset of the Lisa hardware interface. These procedures are for the mouse, the screen, the speaker, the keyboard, the timers, and date and time.

To use the QuickPort hardware interface, you must add

{$U QP/Hardware} Hardware;

to the list elements in your program's USES statement. Refer to Chapter 4 for more information.

3.11.1 The Mouse

The mouse procedures let you

- Set the frequency at which the current mouse location is updated.
- Choose the relationship between physical and logical mouse movements.
- Count mouse movements.

3.11.1.1 Mouse Update Frequency

The mouse location is updated periodically, rather than continuously. The frequency of these updates can be set by calling **MouseUpdates**. The time between updates can range from 0 milliseconds (continuous updating) to 28 milliseconds, in intervals of 4 milliseconds. The initial setting is 16 milliseconds.

Procedure MouseUpdates (delay: MilliSeconds);

3.11.1.2 Mouse Scaling

MouseScaling enables and disables mouse scaling. **MouseThresh** sets the threshold between fine and coarse movements.

Procedure MouseScaling (scale:Boolean);

Procedure MouseThresh (threshold: Pixels);

The relationship between physical mouse movements and logical mouse movements is not necessarily a fixed linear mapping. Three alternatives are available: unscaled, scaled for fine movement and scaled for coarse movement. Initially mouse movements are unscaled.

When mouse movement is *unscaled*, a horizontal mouse movement of x units yields a change in the mouse X-coordinate of x pixels. Similiarly, a vertical movement of y units yields a change is the mouse Y-coordinate of y pixels. These rules apply irregardless of the speed of the mouse movement.

When mouse movement is *scaled*, horizontal movements are magnified by 3/2 relative to vertical movements. This is to compensate for the 2/3 aspect ratio of pixels on the screen. When scaling is in effect, a distinction is made between *fine* (small) movements and *coarse* (large) movements. Fine move- ments are slightly reduced, while coarse movements are magnified. For scaled fine movements, a horizontal mouse movement of x units yields a change in the X-coordinate of x pixels, but a vertical movement of y units yields a change of (2/3)*y pixels. For scaled coarse movements, a horizontal movement a x units yields a change of (3/2)*x pixels, while a vertical movements of y units yields a change of y pixels.

The distinction between fine movements and coarse movements is determined by the sum of the x and y movements each time the mouse location is updated. If this sum is at or below the *threshold*, the movement is considered to be a fine movement. Values of the threshold range from 0 (which yields all coarse movements) to 256 (which yields all fine movements). Given the default mouse updating frequency, a threshold of about 8 (**threshold**'s initial setting) gives a comfortable transition between fine and coarse movements.

3.11.1.3 Mouse Odometer

MouseOdometer returns the sum of the X and Y movements of the mouse since boot time. The value returned is in (unscaled) pixels. There are 180 pixels per inch of mouse movement.

Function MouseOdometer: ManyPixels;

3.11.2 The Screen

The screen procedures are used to

- Set the size of the display screen.
- Count the number of screen refreshes.
- Set the screen contrast, set automatic screen dimming.
- Set the fade delay.

3.6.2.1 Screen Size -- ScreenSize

The display screen is a *bit mapped display*. In other words, each pixel on the screen is controlled by a bit in main memory. The display has 720 pixels horizontally and 364 lines vertically, and therefore requires 32,760 bytes of main memory. The screen size may be determined by calling **ScreenSize**.

Procedure ScreenSize (var x: Pixels; var y: Pixels);

3.11.2.2 Screen Refresh Counter -- FrameCounter

The screen display is refreshed about 60 times per second. A *frame counter* is incremented between screen updates, at the vertical retrace interrupt. The frame counter is an unsigned 32-bit integer which is reset to 0 each time the machine is booted. **FrameCounter** returns this value. To minimize flickering, an application can synchronize with the vertical retraces by watching for changes in the value of this counter. The frame counter should *not* be used as a timer; use the millisecond and mircosecond timers instead.

Function FrameCounter: Frames;

3.11.2.3 Screen Contrast -- ScreenContrast, SetContrast and RampContrast

The screen's contrast level is under program control. Contrast values range from 0 to 255 ($FF), with 0 as maximum contrast and 255 as minimum. **ScreenContrast** returns the contrast setting; **SetContrast** sets the screen contrast. The low order two bits of the contrast value are ignored. The initial contrast value is 128 ($80).

Function Contrast: ScreenContrast;

Procedure SetContrast (contrast: ScreenContrast);

A sudden change in the contrast level can be jarring to the user. **RampContrast** gradually changes the contrast to the new setting over a period of about a second. **RampContrast** returns immediately, then ramps the contrast using interrupt driven processing.

Procedure RampContrast (contrast: ScreenContrast);

3.11.2.4 Automatic Screen Dimming -- DimContrast and SetDimContrast

The screen contrast level is automatically dimmed if no user activity is noted over a specified period (usually several minutes). The contrast level is dimmed to preserve the screen phospher. **DimContrast** returns the contrast value to which the screen is dimmed; **SetDimContrast** sets this value. The initial dim contrast setting is 176 ($B0).

Function DimContrast: ScreenContrast;

Procedure SetDimContrast (contrast: ScreenContrast);

3.11.2.5 Automatic Screen Fading — FadeDelay and SetFadeDelay

The delay between the last user activity and dimming of the screen is under software control. **FadeDelay** returns the fade delay; **SetFadeDelay** sets it. The actual delay will range from the specified delay to twice the specified delay. The initial delay period is five minutes.

Function FadeDelay: MilliSeconds;

Procedure SetFadeDelay (delay: MilliSeconds);

3.11.3 The Speaker

The speaker routines in this section provide square wave output from the Lisa speaker.

The speaker procedures let you

- Set the speaker volume.
- Use the speaker.

3.11.3.1 Speaker Volume -- Volume and SetVolume

The speaker volume can be set to values in the range 0 (soft) to 7 (loud). **Volume** reads the volume setting; **SetVolume** sets it. The initial volume setting is 4.

Function Volume: SpeakerVolume;

Procedure SetVolume (volume: SpeakerVolume);

3.11.3.2 Using the Speaker -- Noise, Silence and Beep

Noise and **Silence** are called in pairs to start and stop square wave output. **Beep** starts square wave output which will automatically stop after the specified period of time. The effects of **Noise, Silence** and **Beep** are overridden by subsequent calls.

Procedure Noise (waveLength: MicroSeconds);

Procedure Silence;

Procedure Beep (waveLength: MicroSeconds; duration:

Noise produces a square wave of approximately the specified wavelength. **Silence** shuts off the square wave. The minimum wavelength is about 8 microseconds, which corresponds to a frequency of 125,000 cycles per second, well above the audible range. The maximum wavelength is 8,191 micro-seconds, which corresponds to about 122 cycles per second.

3.11.4 The Keyboard

Three physical keyboard layouts are defined, the Old US Layout (with 73 keys on the main keyboard and numeric keypad), the Final US Layout (76 keys) and the European Layout (77 keys). Each key has been assigned a *keycode*, which uniquely identifies the key. Keycode values range from 0 to 127. Figure 2 defines the keycodes for the Final US Layout, using the legends from the US Keyboard. The Old US Layout has three fewer keys: |\, Alpha Enter, and Right Option are not on the old keyboard. The European Layout has one additional key, ><, with a keycode of $43.

Two keys on the Old US Layout generate keycodes different from the corresponding keys on the Final US Layout. To aid in compatibility, software changes the keycode for ~\` from $7C to $68, and the keycode for Right Option from $68 to $4E.

Figure 2
Keycodes for "Final US Layout"

HIGH→ LOW↓	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000 0	▨	▨	CLEAR	▨	_ -	(9	E	A
0001 1	DISK 1 INSERTED	▨	-	▨	+ =	) 0	^ 6	@ 2
0010 2	DISK 1 BUTTON	▨	+ ◀	▨	| \	U	& 7	# 3
0011 3	DISK 2 INSERTED	▨	* ▶	▨	▨	I	* 8	$ 4
0100 4	DISK 2 BUTTON	▨	7	▨	P	J	% 5	! 1
0101 5	PARALLEL PORT	▨	8	▨	BACKSPACE	K	R	Q
0110 6	MOUSE BUTTON	▨	9	▨	ALPHA ENTER	{ [	T	S
0111 7	MOUSE PLUG	▨	/ ▲	▨	▨	}]	Y	W
1000 8	POWER BUTTON	▨	4	▨	RETURN	M	~ `	TAB
1001 9	▨	▨	5	▨	0	L	F	Z
1010 A	▨	▨	6	▨	▨	: ;	G	X
1011 B	▨	▨	, ▼	▨	▨	" '	H	D
1100 C	▨	▨	.	▨	? /	SPACE	V	LEFT OPTION
1101 D	▨	▨	2	▨	1	< ,	C	CAPS LOCK
1110 E	▨	▨	3	▨	RIGHT OPTION	> .	B	SHIFT
1111 F	▨	▨	NUMERIC ENTER	▨	▨	O	N	`

The keyboard procedures allow you to

- Find out the keyboard identification number.
- Find out the state of keyboard.

3.11.4.1 Keyboard Identification -- Keyboard

The Lisa supports a host of different keyboards. Each keyboard has three major attributes: manufacturer, physical *layout*, and *legends*. The chart below describes how these three attributes are combined to form a keyboard identification number. The keyboards self-identify when the machine is turned on and when a new keyboard is attached. **Keyboard** returns the identification number of the keyboard currently attached.

Function Keyboard: KeybdId;

Function Legends: KeybdId;

Keyboard identification numbers:

7	6	5	4	3	2	1	0
Manufacturer		**Layout**		**Legends**			

Manufacturer:
00 -- APD (i.e., TKC)
01 --
10 -- Keytronics

Layout:
00 -- Old US (73 keys)
01 --
10 -- European (77 keys)
11 -- Final US (76 keys)

Layout/Legends
$0F -- Old US
$26 -- Swiss-German (proposed)
$27 -- Swiss-French (proposed)
$29 -- Portuguese (proposed)
$29 -- Spanish (proposed)
$2A -- Danish (proposed)
$2B -- Swedish
$2C -- Italian
$2D -- French
$2E -- German
$2F -- UK
$3C -- APL (proposed)
$3D -- Canadian (proposed)
$3E -- US-Dvorak
$3F -- Final US

3.11.4.2 Keyboard State -- KeyIsDown and KeyMap

Low level access to the keyboard is provided through a pollable keyboard state. This state information is based on the physical keycodes defined above. **KeyIsDown** returns the position of a single specified key. **KeyMap** returns a 128-bit map, one bit for each key.

Function KeyIsDown (key: KeyCap): Boolean;

Procedure KeyMap (var keys: KeyCapSet);

A zero indicates the key is up, a one indicates down. For the mouse plug, a zero indicates unplugged, a one indicates plugged in. Certain keys are not pollable; the corresponding bits will always be zero. These keys are the diskette insertion switches, parallel port, and power switch. (The parallel port and mouse plug keys are unreliable across reboots on older hardware.)

3.11.5 The Timers

The timer procedures let you use either the microsecond timer or the millisecond timer.

3.11.5.1 The Microsecond Timer -- MicroTimer

The **MicroTimer** function simulates a continuously running 32-bit counter which is incremented every microsecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 35 minutes, and rolls over about every 70 minutes.

Function MicroTimer: Microseconds;

The microsecond timer is designed for performance measurements. It has a resolution of 2 microseconds. Calling **MicroTimer** from Pascal takes about 135 microseconds. Note that interrupt processing will have a major effect on microsecond timings.

3.11.5.2 The Millisecond Timer -- Timer

The **Timer** function simulates a continuously running 32-bit counter which is incremented every millisecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 25 days, and rolls over about every 7 weeks.

Function Timer: Milliseconds;

The millisecond timer is designed for timing user interactions such as mouse clicks and repeat keys. It can also be used for performance measurements, assuming that millisecond resolution is sufficient.

3.11.6 Date and Time -- DateTime, SetDateTime and DateToTime

The date and time procedures let you

- Set the current date and time.
- Find out the date and time.

The current date and time are available as a set of 16-bit integers that represent the year, day, hour, minute and second, by calling **DateTime** and **SetDateTime**. The date and time are based on the hardware clock/calendar. This restricts dates to the years 1980-1995. The clock/calendar continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the date and time also sets the time stamp described below. **DateToTime** converts a date and time to a time stamp, defined in the next section.

Procedure DateTime (var date: DateArray);

Procedure SetDateTime (date: DateArray);

Procedure DateToTime (date: DateArray; var time: Seconds);

3.11.7 Time Stamp -- TimeStamp, SetTimeStamp and TimeToDate

The current date and time are also available as a 32-bit unsigned integer which represents the number of seconds since the midnight prior to 1 January 1901, by calling **TimeStamp** and **SetTimeStamp**. The time stamp will roll over once every 135 years. Beware--for dates beyond the mid 1960's, the sign bit is set. The time stamp is based on the hardware clock/calendar. This clock continues to operate during soft power off. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the time stamp also sets the date and time described above. Since the date and time is restricted to 1980-1995, the time stamp is also restricted to this range. **TimeToDate** converts a time stamp to the date and time format defined above.

The time stamp procedures let you

- Set the time stamp.
- Convert between standard date and time and the time stamp.

Function TimeStamp: Seconds;

Procedure SetTimeStamp (time: Seconds);

Procedure TimeToDate (time: Seconds; var date: DateArray);

The current date and time are available as a set of 16-bit integers that represent the year, day, hour, minute and second, by calling **DateTime** and **SetDateTime**. The date and time are based on the hardware clock/calendar. This restricts dates to the years 1980-1995. The clock/calendar continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the date and time also sets the time stamp described below. **DateToTime** converts a date and time to a time stamp, defined in the next section.

Procedure DateTime (var date: DateArray);

Procedure SetDateTime (date: DateArray);

Procedure DateToTime (date: DateArray; var time: Seconds);

3.11.7 Time Stamp -- TimeStamp, SetTimeStamp and TimeToDate

The current date and time are also available as a 32-bit unsigned integer which represents the number of seconds since the midnight prior to 1 January 1901, by calling **TimeStamp** and **SetTimeStamp**. The time stamp will roll over once every 135 years. Beware--for dates beyond the mid 1960's, the sign bit is set. The time stamp is based on the hardware clock/calendar. This clock continues to operate during soft power off. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the time stamp also sets the date and time described above. Since the date and time is restricted to 1980-1995, the time stamp is also restricted to this range. **TimeToDate** converts a time stamp to the date and time format defined above.

The time stamp procedures let you

- Set the time stamp.
- Convert between standard date and time and the time stamp.

Function TimeStamp: Seconds;

Procedure SetTimeStamp (time: Seconds);

Procedure TimeToDate (time: Seconds; var date: DateArray);

Chapter 4
Bringing Your Application to the Lisa Desktop

Bringing Your Application to the Lisa Desktop

4.1 Adding the USES List Elements

Before bringing your application to the Lisa desktop you must add the required USES list elements to your MAIN program and any of your units. Depending on what kind of application you are porting, you use different USES list elements.

1. For text output only

```
{$U QP/UQuickPort}  UQuickPort;
```

2. For graphic (QuickDraw) and/or text output

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortGraph} UQPortGraph;
```

3. If you need to use Graf3D (order of list elements important)

```
{$U QuickDraw}       QuickDraw,
{$U QP/Graf3D.OBJ}  Graf3D,
{$U QP/UQPortGraph} UQPortGraph;
```

4. For graphic (QuickDraw) and/or text output, and the hardware interface

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortGraph} UQPortGraph,
{$U QP/Hardware}    Hardware;
```

5. To use the VT100 terminal emulator

```
{$U QP/UQPortVT100}  UQPortVT100;
```

6. To use the Soroc terminal emulator

```
{$U QP/UQPortSoroc}  UQPortSoroc;
```

7. If you are calling the additional QuickPort procedures (order of list elements important)

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortCall}  UQPortCall,
{$U QP/UQuickPort}  UQuickPort; {or UQPortGraph,
                                  UQPortVT100,
                                  UQPortSoroc}
```

UQPortCall, unlike the other units, is only an interface and contains no code.

4.2 System Configuration

This section assumes that you are using a two-ProFile system to develop your QuickPort applications. The ProFile with the office system is called "office" in this discussion, and the ProFile with the Workshop is called "workshop." In the Workshop, set the prefix to the workshop volume. If you have a Lisa 2/10 you will not need to set the prefixes as described in this section because all development will be done on one volume.

There are two different environments to consider:

- The development environment. That is, the environment you use when developing a QuickPort application. The development environment is the Workshop.
- The run-time environment. This is the environment that the QuickPort application runs in. The run-time environment is the Office System.

4.2.1 The Development Environment

When developing, you must

- Boot from the Workshop.
- From the Workshop System Manager, set the prefix to the Workshop volume.
- Place all files listed in the USES statement on the prefix volume.

You must have the following files on your prefix volume:

- **QP/UQPortCall**
- **QP/UQPortGraph**
- **QP/UQPortSoroc**
- **QP/UQPortVT100**
- **QP/UQuickPort**
- **QP/Hardware**
- **QP/Graf3D**
- **QPLib.Obj**
- **TKLib.Obj**
- **TK2Lib.Obj**
- **QP/Phrase**

The QuickPort exec file, **qp/make**, must be on the workshop ProFile.

4.2.2 The Run-Time Environment

When running a QuickPort application, you must

- Boot from the office system.
- Have all the libraries your application needs on the office system volume.
- Have **TKLib.Obj**, **TK2Lib.Obj**, and **QPLib.Obj** on the office system volume.

4.3 Generating Your Tool

To generate your tool, you must run the QuickPort exec file, **qp/make**, or customize **qp/make** to compile, assemble, and link your tool. **Qp/make** assumes all source files are in Pascal. You can customize **Qp/make** to assemble your files. **Qp/make** forces recompilation of all your application's units, compiles your application's main program, and then links your application's units with the QuickPort intrinsic units. Then **qp/make** assigns the tool name and creates the phrase file using the tool number in the file name.

Qp/make renames the object code to a file name of the form:

{T##}obj

where **##** is the tool number you specified when **qp/make** was invoked. **Qp/make** copies the phrase file to a file name of the form:

{T##}PHRASE

If your application uses other support files, such as data files, rename the files using the **{T##}** tool number as the first part of the file name, e.g.,

{T###}support

Then, whenever a user selects the tool's icon from the desktop, all the files with the **{T##}** will be copied or deleted. **Qp/make** assumes that the source files and libraries are on the prefix volume. Refer to System Configuration above for more information.

Qp/make can be invoked in two ways, depending on how many units your application has, and depending on whether you need to specify additional object files that your application does not generate but needs to link to. If your application has four or fewer units and does not need to specify additional object files for linking, **qp/make** can be invoked as follows:

Run <qp/make (mainprogram, tool##, tool volume, unita, unitb, unitc, unitd)

where

mainprogram	is the filename of your application's main program.
tool ##	is the tool number you want used in your application's tool name. We recommend you use

	your Lisa's serial number plus an offset. Using the serial number plus an offset will prevent duplication of tool numbers among different software developers. For testing you can use any number greater than 20.
tool volume	is the office disk name. The tool will be copied to the office system.
unita, unitb unitc, unitd	Up to four units for your application. If you use more than four units, use the alternate way to invoke **qp/make** as described below.

If your application has more than four units, and/or needs additional units to link against, **qp/make** can be invoked as follows:

Run <qpmake (mainprogram, tool#, tool volume, <, UnitList, OtherObjList)

where

mainprogram, tool #, and **tool volume** are the same as above.

UnitList	is a file that contains the names of all your units. When you create your UnitList file, be sure to list the units in the order they should be compiled.
OtherObjList	is a file that lists any object files that your application links against but you don't generate.

Refer to some QuickPort examples programs (qp sample, note, text, and so forth) on the release diskette.

4.4 Installing Your Tool

After you run **qp/make** successfully, you must install the application on the Lisa desktop. This installation process creates a tool icon and stationery pad for your tool. To install a tool you run **InstallTool** from the Workshop. After **InstallTool** is finished, when you leave the Workshop and start the Office System, your tool and its stationery pad will be on the desktop.

To install a tool, run **InstallTool** from the Workshop with the tool number you specified in **qpmake**.

Run what Program? InstallTool

The **InstallTool** program will prompt you as follows:

Please enter the name of the device your tool is on. [PARAPORT]
This is the name of your Office System ProFile.

Please enter your tool id number
Enter the tool number you specified when you ran **qp/make**.
Remember, *every tool must have a unique number.*

Does your tool create documents? (Y or N) [YES]
If you answer no, a tool like the Calculator is created. In other

words, a tool that allows only one instance of itself at a time.

Can your tool handle more than one document at a time? If you don't know, press return. (Y or N) [NO]
Some tools, such as LisaWrite, create one process that controls multiple documents. You must answer no for QuickPort tools.

The stationery opening rectangle is defaulted to 10, 40, 640, 290
These values are always the same.

Do you wish to specify a different one? (Y or N) [NO]
If you answer yes, you are prompted for the values for the size of the rectangle when a document is opened. This rectangle will be used whenever a document is opened.

Please enter the name of your tool.
Every tool has a tool number and a tool name. When you enter a tool name, the install program places the tool name in the desktop names of the tool and its stationery.

"Tool name" has been sucessfully installed in the Office System and it will appear in the disk window associated with the device.

After you've finished running the **InstallTool** program, boot the Office System. Your application's tool and stationery pad should be on the desktop. You only need to run **InstallTool** once even if you regenerate your tool. If you do regenerate it, however, the tool name in the object file will be lost, and "Tool xx" will be listed in all the alerts. To get the tool name back in the alerts, you must run **InstallTool** again.

4.5 The Icon Editor

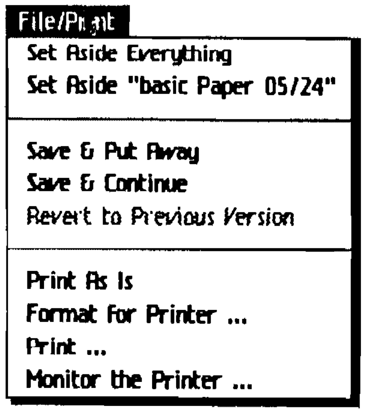
The icons created by the **InstallTool** program are blank (without pictures). If you want to design an icon for your application, contact Macintosh Technical Support. (Use program called "IconEdit" in Lisa Toolkit)

4.6 Shipping Your Application

Your application's phrase file, as well as the object file, must be shipped. The phrase file contains the standard QuickPort menus and alerts, and it must be shipped with your application.

Appendix A
The Standard QuickPort Menus

A.1 File/Print Menu



Set Aside Everything Returns all windows to their icons without saving the contents.

Set Aside "your document" Returns the current document to its icon without saving the contents.

Save & Put Away Saves the contents of the document, closes the window, terminates the program, and returns the icon to its original location.

Save & Continue Saves the contents of the document and leaves the window open.

Revert to Previous Version Always gray -- not supported by QuickPort.

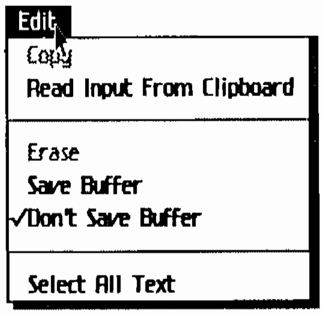
Print As Is Prints one copy of the document.

Format for Printer Sets formats in the document based on the printer that will be used.

Print Prints the document using the settings from the Format for Printer dialog box. You may choose to print multiple copies.

Monitor the Printer Shows the status of the document(s) being printed.

A.2 Edit Menu



Copy Copies the current selection onto the Clipboard. In the text panel the selection is done as in LisaWrite. In the graphic panel, the entire panel is copied. If there is a text panel, and a graphic panel, you must use Select All Graph to make the selection.

Read Input From Clipboard Places what is in the Clipboard into the input buffer.

Erase Erases the current selection.

Save Buffer Saves the lines that scroll off the top of the screen area. A check next to Save Buffer indicates that the lines will be saved.

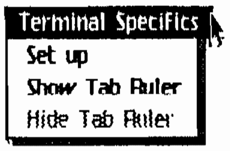
Don't Save Buffer Does not save the lines that scroll off the top of the screen area. A check next to Don't Save Buffer indicates that the lines will not be saved.

Select All Text Selects all the text in the text panel when there is a text panel.

Flush Input Clears the input panel. This command is shown only when the input panel is shown.

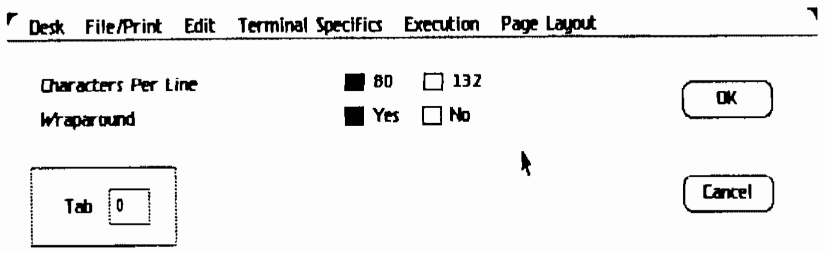
Select All Graph Selects the entire graphic panel when there is a graphic panel.

A.3 Terminal Specifics



Set up Allows you to select 80 or 132 characters per line, and line wraparound.

The following dialog box appears for you to fill in:



Characters Per Line ■ 80 □ 132

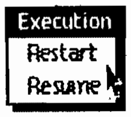
Wraparound ■ Yes □ No

Tab 0

Show Tab Ruler Displays the tab ruler.

Hide Tab Ruler Hides the tab ruler.

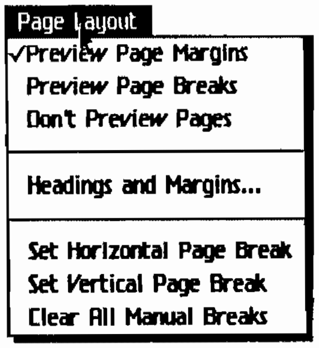
A.4 Execution



Restart Restarts program execution.

Resume Starts program execution at the point where it was suspended by an ⌘-period.

A.5 Page Layout



Preview Page Margins Shows the page margins. Note that the default page margins are such that the output in the text panel will not fit in the width of an 8" by 11" page. Before printing you should adjust the left and right margins so that each vertical page will fit in one 8" by 11" page.

Preview Page Breaks Shows the page breaks.

Don't Preview Pages Does not show the page boundaries.

Set Horizontal Page Break Sets a horizontal page break at the position of the last mouse click.

Set Vertical Page Break Sets a vertical page break at the position of the last mouse click.

Clear All Manual Breaks Clears all the page breaks set in the document.

Appendix B
Writing Your Own Terminal Emulator

Writing Your Own Terminal Emulator

B.1 Introduction

This appendix briefly discusses how to write your own terminal emulator, using the standard terminal as a template. To write a terminal emulator, you must understand Clascal. Specifically, you must understand how to extend a Clascal program by creating a subclass, overriding existing methods, and creating new methods. This section assumes you are comfortable with these basic Clascal concepts. If you don't understand Clascal, contact Macintosh Technical Support for a copy of *An Introduction to Clascal* before reading this section.

To write a terminal emulator, you create a subclass of **TStdTerm**. **TStdTerm** is the standard terminal provided by QuickkPort. The subclass you create defines the terminal emulator you want. This appendix discusses **TStdTerm**, the methods you *must* override in your subclass, and the methods used by **TStdTerm**. You can also add your own methods in your subclass.

B.2 TStdTerm

TStdTerm is the standard terminal that is used by QuickPort applications unless the VT100, Soroc, or any other terminal emulator is specified. The **TStdTerm** fields and methods are discussed in this section.

B.2.1 TStdTerm Fields

The fields you need to know about in **TStdTerm** are listed below. These fields explain how the standard terminal behaves. You may want to change some or all of this behavior in your terminal emulator.

maxLines	The maximum number of lines in the window.
maxColumns	The maximum number of columns in the window.
cursorshape	The shape of the cursor. The standard terminal uses a box cursor.
saveBuffer	To save lines as they scroll off the top of the screen into the buffer.
wraparound	BOOLEAN, whether wraparound is on or off.
stopOutputKey	Used to stop output.
startOutputKey	Used to start output.

You can only chage these fileds in your **CREATE** method.

B.2.2 TStdTerm Methods You Must Override

You must override three of these four methods in your subclass. You may want to override **CtrKeyWrite**.

B.2.2.1 CREATE

CREATE creates an object of class **TStdTerm**. You must override the **CREATE** method in your subclass.

```
FUNCTION {TStdTerm}CREATE (object: TObject; heap :
                           Theap) : TSTdTerm;
```

You must use **object** and **heap** as arguments in your **CREATE** method.

B.2.2.2 VWrite

VWrite is called by QuickPort when the program calls a **write**. You must override the **VWrite** method in your subclass to handle escape sequences that apply to your terminal.

```
PROCEDURE {TStdTerm}VWrite (VAR str : Tstr255);
```

B.2.2.3 Vread

Vread is called by QuickPort when the program calls a **read**. You must override the **Vread** method in your subclass to return any escape sequences generated from your terminal.

```
PROCEDURE {TStdTerm}Vread (VAR ach: char; VAR
                 keycap : Byte; VAR applekey,
                 shiftkey, optionkey ; BOOLEAN);
```

B.2.2.4 CtrKeyWrite

CtrKeyWrite handles the control keys for the terminal emulator. You should override this method in your subclass if you want to handle different control keys.

```
PROCEDURE {TStdTerm}CtrKeyWrite (ctrch: CHAR);
```

The control keys handled in the standard terminal are CR (no LF), LF, Bell, Backspace, Horizontal Tab.

B.3 Procedures Terminal Emulators Can Call

The procedures listed in this section can be called by any terminal emulators. Note that these are not methods and do not need to be overridden in your subclass.

B.3.1 Screen Control Procedures

These procedures use escape sequences.

B.3.1.1 Manipulating Lines -- VGetLine and VPutLine

VGetLine deletes the specified line. **VPutLine** inserts the line at the specified line number.

```
PROCEDURE VGetLine (lineNo : INTEGER; VAR line :
              Tstr255; delete : BOOLEAN);

PROCEDURE VPutLine (lineNo : INTEGER; VAR line :
             Tstr255; insert : BOOLEAN);
```

B.3.1.2 Redrawing — RedrawScreen and RedrawLine

RedrawScreen and **RedrawLine** are used after **VGetLine** and **VPutLine**. **RedrawScreen** repaints the entire screen after a change to the lines or a screen size change. **RedrawLine** repaints a line after its attributes have been changed.

```
PROCEDURE RedrawScreen;

PROCEDURE VPutLine (lineNo : INTEGER);
```

B.3.1.3 Scrolling -- VScrollLines

VScrollLines scrolls output on the screen without changing the data structure.

```
PROCEDURE VScrollLines (topRegion, bottomRegion :
              INTEGER; scrollhowmanylines :
              INTEGER);
```

A positive value for **scrollhowmanylines** scrolls down.

B.3.1.4 Changing the number of columns -- ChangeMaxColumns

ChangeMaxColumns changes the maximum number of columns per line to the specified number. When **ChangeMaxColumns** is called, the corresponding character font is used. If the columns per line is 80 or less, QuickPort uses a 12-pitch font, otherwise a 20-pitch font is used.

```
PROCEDURE ChangeMaxColumns (newColumns : INTEGER);
```

B.3.1.5 Changing Fonts — ChangeFont

ChangeFont changes to the specified font. Because of cursor positioning, QuickPort supports only fixed pitch fonts.

```
PROCEDURE ChangeFont (newFont : INTEGER);
```

B.2.4 VStrWrite

VStrWrite writes the string from the cursor position. This call is the one that does the actual display of output. Terminal emulators should call this after determining there is no escape sequence in the string. This call actually displays the output. No control functions are allowed in the string. This call handles wraparound.

```
PROCEDURE VStrWrite (VAR str : Tstr255);
```

FINIS