Names for Standardized Floating-Point Formats

Abstract
I lack the courage to impose names upon floating-point formats of diverse widths, precisions, ranges and radices, fearing the damage that can be done to one's progeny by saddling them with ill-chosen names. Still, we need at least a list of the things we must name; otherwise how can we discuss them without talking at cross-purposes? These things include ...

Wordsize, *Width*, *Precision*, *Range*, *Radix*, …

all of which must be both declarable by a computer programmer, and ascertainable by a program through *Environmental Inquiries*. And precision must be subject to truncation by *Coercions*.

Conventional Floating-Point Formats

Among the names I have seen in use for floating-point formats are …

`float`, `double`, `long double`, `real`,
REAL*…, SINGLE PRECISION, DOUBLE PRECISION,
double extended, doubled double, TEMPREAL, quad, … .

Of these floating-point formats the conventional ones include finite numbers x all of the form

$$x = (-1)^s \cdot m \cdot ß^{n+1-P}$$

in which s, m, ß, n and P are integers with the following significance:

s is the *Sign bit*, 0 or 1 according as $x \geq 0$ or $x \leq 0$.

ß is the *Radix* or *Base*, two for *Binary* and ten for *Decimal*. (I exclude all others).

P is the *Precision* measured in *Significant Digits* of the given radix.

m is the *Significand* or *Coefficient* or (wrongly) *Mantissa* of $|x|$ running in the range

$$0 \leq m \leq ß^P - 1 .$$

n is the *Exponent*, perhaps *Biased*, confined to some interval $N_{min} \leq n \leq N_{max}$.

Other terms are in use. Precision can be measured in "Sig. Dec." (but only approximately when $ß \neq 10$). A *Normalized* nonzero significand can be represented as a *Fraction* $f = m/ß^P$ in the range $1/ß \leq f < 1$, or in *Scientific Notation* as a number ßf in $1 \leq ßf < ß$; in binary this Scientific notation looks like $1.f$ for some fraction f . Decimal Scientific notation is what the 1PEw.d FORMAT mask in Fortran displays, and then the exponent displayed is just n , as in

`6.0225E+23` which is easier to read than `0.60225E+24`

for Avogadro's number $60225 \cdot 10^{19}$ to 5 sig. dec. of precision.

What most distinguishes the floating-point formats of IEEE Standards 754/854 from all previous formats is the simplicity with which *all* of a format's finite floating-point numbers x can be characterized. That set of available finite numbers x is determined completely by just three positive integers N_{max}, P and either $ß = 2$ or $ß = 10$. A fourth integer $N_{min} < 0$ is determined by the requirement of an exponent range so *balanced* that $ß^{Nmax+Nmin+1} \geq 4$ as barely as possible; this means $N_{min} := -\text{floor}(N_{max} + 1 - \log_ß 4)$. Then *every* number of the form

$$x = (-1)^s \cdot m \cdot ß^{n+1-P} \quad \text{with } s^2 = s , \quad 0 \leq m \leq ß^P - 1 \quad \text{and} \quad N_{min} \leq n \leq N_{max}$$

is a floating-point number in that format, and it has no others besides $\pm\infty$ and some *NaN*s .

Previous floating-point formats allowed only *Normalized* numbers x. These are $x = 0$ and $x = \pm m \cdot ß^{n+1-P}$ with $ß^{P-1} \le m \le ß^P - 1$. (Some old machines "Normalized" slightly differently.) Allowing only normalized numbers compelled *Underflow* to flush to zero. IEEE Standards 754/854 underflow *Gradually* through the *Subnormal* numbers, namely the values x with $n = N_{min}$ and $0 < m < ß^{P-1}$; these are now 754/854's only unnormalized numbers. All told, each 754/854 format has $2ß^{P-1}(ß + (ß-1)(N_{max} - N_{min})) - 1$ finite floating-point numbers besides -0, plus $\pm\infty$ and some *NaN*s. ("NaN" means "Not a Number".)

Fully Specified Standard Binary Floating-Point Formats

Originally IEEE 754 specified two *Binary* floating-point formats so tightly that they could be used to share numerical data among diverse conforming computer systems. Now three binary ($ß = 2$) formats are so specified by their wordsizes in bytes, precisions P in sig. bits, and their $(k+1)$-bit exponent fields that keep exponents between $N_{max} = 2^k - 1$ and $N_{min} = 1 - N_{max}$:

Format Names (among others)	**Bytes** wide	N_{max}	**Precision** P sig. bits
Single Precision, `float`, REAL*4	4	$2^7 - 1$	24
Double Precision, `double`, REAL*8	8	$2^{10} - 1$	53
Quadruple Precision, `quad`, REAL*16	16	$2^{14} - 1$	113

Except for *Big-Endian vs. Little-Endian* variations, these formats are specified down to the location of every bit. Each floating-point word encodes a floating-point value $x = (-1)^s \cdot m \cdot 2^{n+1-P}$ in a word $X :=$ **[**s, Exp, SigBits**]** in which the leading bit s is x's sign bit, 0 for " + " or else 1 for " – "; the next $(k+1)$-bit field holds the *Biased Exponent* Exp; and the last field holds the *Significand*'s all-but-leading-bits. The fields of X after the sign bit s are decoded thus:

$1 \le Exp \le 2 \cdot N_{max}$ for *Normalized* x with $n = Exp - N_{max}$ and $m = SigBits + 2^{P-1}$; but

$Exp = 0$ for *Subnormal* x with $n = 1 - N_{max}$ and $m = SigBits$, so $m = 0$ for $x = \pm 0$.

$Exp = 2 \cdot N_{max} + 1 = 2^{k+1} - 1$ and $SigBits = 0$ for *Infinite* $x = \pm\infty$; but

$Exp = 2 \cdot N_{max} + 1$ and $SigBits > 0$ when x is NaN, a *Quiet NaN* if $SigBits \ge 2^{P-1}$.

Signaling NaNs with $0 < SigBits < 2^{P-1}$ trap immediately before arithmetic operations.

IEEE 754 specified also a family of *Extended* binary floating-point formats, but not so tightly that they could usefully be copied *verbatim* from one computer's memory to another's. The *Double-Extended* formats were required only to have $N_{max} = 2^k - 1$ for some $k \ge 14$, and $P \ge 64$, both chosen at the implementor's option. For example, Quadruple Precision listed above is a Double-Extended format with minimal $k = 14$ but $P = 113$. The minimal values of k and $P = 64$ are found in a *de facto* standard Double-Extended used by Intel *Pentium*s and their clones manufactured by others, by H-P/Intel's *Itanium*, and by the now almost extinct Motorola 88110 and 680x0. Correct use of this format is a nontrivial challenge to programmers for reasons varying from nonexistent support by compilers to a lack in standard languages of suitable *Environmental Inquiries* that properly written programs must invoke to discover an Extended format's range and precision at run-time. Such programs can run correctly though they deliver different accuracies on different machines with different Extended formats.

Families of Wider Standardized Floating-Point Formats

What follows expands upon the IEEE standards' presently allowed Double-Extended formats.

Conceivably occasions will arise for floating-point formats wider than the aforementioned ones, and somebody will have to give them names. For the present I propose to call them

FloatBin(k, p) and FloatDec(k, p)

for binary (ß = 2) and decimal (ß = 10) formats respectively with the indicated precisions of *at least* p digits of radix ß , and exponent range up to *at least* $N_{max} \geq ß^k - 1$. Actual precisions and ranges (and sometimes the radix ß too) are intended to be ascertained at run-time by apt Environmental Inquiries based upon the NextAfter function, about which more later.

This is not the place to explore memory-saving encodings of decimal floating-point formats beyond observing that in general, for both binary and decimal, a computer's floating-point registers may well contain information besides what is normally stored in memory packed into a standard format. What matters most to applications programmers using higher-level languages like C and Fortran is that they may specify what they regard as the least amount of precision and range adequate for their purposes. What matters to the numerical specialists who provide software support for applications programmers is that there be ascertainable parameters $P \geq p$ and $N_{max} \geq ß^k - 1$ by which the standardized inequalities

$$0 \leq m \leq ß^P - 1 \quad \text{and} \quad N_{min} := -\text{floor}(N_{max} + 1 - \log_ß 4) \leq n \leq N_{max} \quad \text{for} \quad x = \pm m \cdot ß^{n+1-P}$$

characterize *all* of a standard format's set of finite floating-point numbers x . Both kinds of programmers must be able to choose the standard format of the destination for every floating-point operation's result. We hope language conventions will help make that choice both apt and convenient; but for more than three formats, or if the widest runs too much slower than the others, these are tricky issues to be discussed another day.

Implementors will find some formats' choices of precision and range "natural" because they can be made to run at least as fast as other narrower formats. This slight anomaly, namely that range and/or precision slightly increased sometimes runs faster, is a consequence of processors' buss- and register-widths. The run-time library of mathematical functions also exhibits such anomalies because different formulas for a function, differing perhaps in the number of a series' terms summed, may provide different accuracies either rather less or rather more than the requested precision. Consequently an implementor must be allowed to exploit her processor architecture's features in ways that optimize the performance of her multi-word arithmetics only for certain discrete choices of precision and range. In particular, the *Width* of a standardized format need not be minimal for any particular choice of precision or range lest word alignment delays cripple performance. For instance, though Intel's Double-Extended fits in 10 bytes, Motorola spread it in 12-byte words, and 16-byte words for it are commonplace now that memory is so cheap but time is still dear. `Sizeof(…)` should reveal an implementation's width.

What remain to be discussed are the granularity of ranges and precisions that an implementor should support, and the Environmental Inquiries and Coercions that languages should provide.

A plausible if not foolproof strategy for an applications programmer who does not know in advance how much precision he needs is to repeat a computation using a sequence of ever wider precisions until the computed result settles down in as many leading digits as he desires.

"If at first you don't succeed, try, try, try again."

How hard?

This question happens to have a provably near-optimal answer:

Try again with about $\sqrt{2}$ times as many sig. digits as before.

Consequently the implementor of a family of arithmetics of ever higher precisions need not offer precisions closer than in a ratio of roughly $\sqrt{2} = 1.414\ldots$. For instance, after 16-byte `quad` the implementor may reasonably offer only formats 24, 32, 48, 64, 96, 128, … bytes wide in the event that these turn out to be substantially easier to support than a finer granularity of widths.

Environmental Inquiries

A program may have to discover the actual precision and range of the arithmetic it is executing in order to decide when an iteration has converged as well as can be expected, and when to scale variables to prevent over/underflow. For all floating-point arithmetics that conform to IEEE standards, if not for others, a recommended function NextAfter(x, y) can serve this purpose well. This function should be regarded as *Generic* in the Fortran sense, returning a value in the wider of its arguments' formats; the returned value is that format's next floating-point number after x in the direction towards y . In other words, this is x's neighbor on the same side as y unless y = x , in which case NextAfter(x, y) is y (to handle ±0 nicely).

If such generic functions are not natural to the programming language in use, names like
"sNextAfter", "dNextAfter", "eNextAfter", "qNextAfter", …, NextAfterß(k, p)
may become necessary. I prefer generic functions and will take them for granted henceforth.

The implementor of NextAfter for the IEEE Standard's three Fully Specified Binary formats, namely Single, Double and Quadruple precisions, can exploit their *Lexicographic Order.* This means that two words X and X' that encode respective floating-point values x and x' in the same format, but neither value NaN, share the same sign-magnitude ordering: X < X' if and only if x < x' . Therefore NextAfter can be implemented (not necessarily faster) using exclusively integer arithmetic operations including a few integer comparisons that are necessary to get the direction of incrementation right and to detect special cases like NaNs. If x' = NextAfter(x, y) ≠ x then the corresponding words X' and X differ by 1 as sign-magnitude integers.

Environmental Inquiries can be fashioned out of NextAfter plus a few arithmetic operations to reveal at run-time the radix ß , the precision P , the maximum exponent N_{max} and the minimum exponent N_{min} of standard floating-point formats and some others. Here is how:

The arithmetic's radix ß for variables of the same format as U can be determined by setting
U := 1.0 ; eps := NextAfter(U, +∞) – U ; ulp1 := U – NextAfter(U, –∞) ; ß := eps/ulp1 .

After that the arithmetic's actual precision P can be determined from
P := Round_to_Nearest_Integer($-\log_{ß}$(ulp1)) .

The arithmetic's overflow threshold is Ω := NextAfter(+∞·U, –∞) , and then the actual exponent range becomes apparent from N_{max} := Round_to_Nearest_Integer($\log_{ß}(Ω)$) – 1 and finally a standard format's N_{min} := –floor(N_{max} + 1 – $\log_{ß}$(4.0)) . (Other formats' N_{min} may differ.)

Alternatively, to avoid arithmetic with ∞ , compute the format's smallest positive number eta := NextAfter(0.0, U) . It is subnormal for a standard format, and then its arithmetic's normal underflow threshold is μ := eta/eps . (Most non-standard formats, like the DEC VAX's, flush underflow to zero; their eta is roughly our μ .) Now the standard extreme exponents are
N_{min} := Round_to_Nearest_Integer(($\log_{ß}(μ)$) ; N_{max} = ceil($\log_{ß}(4) - N_{min} - 1$) .

Computing $\log_{ß}(x) := \log(x)/\log(ß)$ is accurate enough for the foregoing purposes provided that $\log_2(4.0) = 2$ exactly.

In short, practical ways exist to determine at run-time the few integer constants that characterize any floating-point format conforming to IEEE 744/854 provided the programming language in use offers well-implemented versions of NextAfter, Round_to_Nearest_Integer and log .

WARNING: The foregoing environmental inquiries need not work for arithmetics that do not conform to IEEE 754/854, nor for conforming arithmetics accessed by languages that deny the programmer adequate control over the destinations of his floating-point operations. An instance of the last kind is a compiler that evaluates NextAfter in a precision wider than either of its operands. An instance of the former kind is Doubled-Double Precision, which represents each floating-point variable as an unevaluated sum of two Doubles that, if added and rounded to Double, would round to the bigger of the two. Alas, many algorithms successful in every arithmetic that conforms to an IEEE standard fail in Doubled-Double. For example its eps , if computed from the formula above, would turn out to be eta instead of a value like $|(4.0*U/3.0 – 1.0)*3.0 – 1.0|$ which gives a better indication of roundoff though still not quite right. Doubled-Double is probably satisfactory for matrix multiplication and power series, probably less satisfactory for solving differential equations by finite difference methods, and dangerous for some now widely used algorithms that exploit delicate relationships preserved by arithmetics conforming to IEEE standards 854/754.

Coercions and Conversions

Every implementation of radix ß floating-point arithmetic should offer a function Roundfß(j, x) that rounds its floating-point operand x to integer j sig. digits in radix ß . Usually $j \le P$. This is a special case, with the same format for both x and Roundfß, of conversion between different formats each with perhaps its own radix, as in Binary <—> Decimal conversion. Conversions and coercions between different formats with the same radix are normally accessed in a simpler way by an assignment statement like " x := y " or, amidst an arithmetic expression, a cast like " (float)x " when the variables x and y have data-types fully supported by the programming language. All such conversions and coercions, if inexact, should honor the directed rounding mode in effect at the time, and respond to Over/Underflow in the expected ways.

When conversion is intended to communicate numbers from one computer to another with possibly different arithmetics, the principal challenge is to choose the right names for source and destination formats. The names, like "FloatBin(k, p)" and "FloatDec(k, p)" , used by programmers to request adequate precision and range for would-be portable software are ill-suited to communication between computers with different arithmetics or even just different compilers for ostensibly the same language and arithmetic. The simplest way transmits decimal strings of ASCII characters long enough that any destination computer's Decimal —> FloatBin(k, p) conversion can reconstruct the intended value exactly, then coerce it into the computer's own format. But this simplest way can also be the slowest.

Faster ways require agreement upon parameterized names that describe how the destination computer encodes floating-point values as bit- or character-strings in memory. For Binary floating-point conforming to IEEE 754, the obvious parameters are field widths: P for the significand's precision, k+1 for the biased exponent's field. The unobvious parameters are a symbol "I" or "E" to indicate whether the significand's leading bit is implicit or explicit, and a number that tells how many bytes the computer addresses (±1 for byte-addressing, ±2 for the DEC VAX, …) and, by its sign, whether addressing is Big- or Little-Endian.

Computers with full hardware support for fast Decimal floating-point arithmetic pose additional challenges because they may pack, say, three decimal digits into ten bits in memory, unpacking the digits when a number is loaded into floating-point registers for arithmetic. They may also choose binary instead of decimal for the exponent field. This packing saves 16% of the memory space and transmission time taken by huge arrays of data. More important, packing puts more precision and range into conventional word sizes; *e.g.*, 4 bytes can hold 7 sig. dec. and a range from 10^{-113} to almost 10^{+114} . At this time the standardization of decimal floating-point formats seems premature.

Appendices for future exposition:

- How are Floating-Point and Fixed-Point Approximate Arithmetics Utterly Different?
- What Good are Extended Floating-Point Formats?
- What Good are Gradual Underflow and Subnormal Numbers?

How are Floating-Point and Fixed-Point Approximate Arithmetics Utterly Different?

Fixed-point arithmetic differs from floating-point in four ways:

		Fixed-point	**Floating-point**
i	Number spacing	Uniform	Roughly logarithmic
ii	Rounding procedures	Many	Few
iii	Destinations depend upon	Operation as well as …	Operands and ambiance
iv	WYSIWYG ?	Usually	Rarely

Each traditional fixed-point format consists of a set of uniformly spaced fixed-point numbers x characterized completely by three parameters, its *Low End* L , its *High End* H , and its *Quantum* q , so that all numbers afforded by the format have the form $x = m \cdot q$ in which integer m is bounded thus: $L \le m \le H$. In memory, a packed or encoded representation of the integer m is stored in place of x ; the programmer or compiler may associate x and all other variables of the same fixed-point format (or *Type*) with its parameters {L, H, q} stored elsewhere. The quantum q is usually (but not necessarily) the reciprocal of a positive integer power of a *Radix* like two for *Binary* or ten for *Decimal*; then the ends' magnitudes –L and H are often (but not necessarily) another integer power of that radix, or half of one, or 1 less. Here are examples each of which packs into four-byte words:

4-byte unsigned binary integers:	$L = 0$;	$H = 2^{32}-1$;	$q = 1$
4-byte twos'-complement fractions:	$L = -2^{31}$;	$H = 2^{31}-1$;	$q = 1/2^{31}$
15+16 bits sign-magnitude:	$L = 1-2^{31}$;	$H = 2^{31}-1$;	$q = 1/2^{16}$
7+2-dec. digits sign-magnitude:	$L = 1-10^9$;	$H = 10^9-1$;	$q = 1/100$
6+2-dec. digits tens'-complement:	L = -50000000 ;	H = 49999999;	q = 1/100

The last format packs into eight BCD digits, one nibble each, without an explicit sign digit; its hundred million numbers x consist of –500000**.**00 packed as "50000000", –499999**.**99 packed as "50000001", -499999**.**98 packed as "50000002", …, –0**.**01 packed as "99999999", 0 packed as "00000000", 0**.**01 packed as "00000001", …, 499999**.**98 packed as "49999998", and 499999**.**99 packed as "49999999". This format, a relic from the mechanical calculators' era, matches in electronic memory the way many people still visualize monetary numbers.

Fixed-point variables of a given format are often associated with a unit of measurement reflected in the quantum q . For instance, money in dollars and cents can have $q = 1/100$. Distances in kilometers measured to the nearest millimeter have $q = 1/10^6$. Weights in pounds to the nearest ounce have $q = 1/16$. Times in hours to the nearest second have $q = 1/3600$. The ends L and H need not correlate with a radix implied by the quantum q ; for instance, dollars and cents with $q = 1/100$ can also be stored in 4-byte twos'-complement integers with $L = -2^{31} = -1 - H$.

Addition/subtraction of fixed-point variables with the same quantum amounts to exact integer arithmetic provided the the sum/difference has the same quantum and does not overflow beyond its ends. Similarly for multiplication provided the product's quantum is the product of the factors'

quanta. All programming languages that support many fixed-point formats specify by default, for every arithmetic operation, a format for its result depending upon the operation as well as the operands' formats, but often leaving "undefined" any intermediate result that overflows beyond the result-format's ends L and H. A quotient's result-format may depend more upon the operand formats' ends than their quanta. Mathematically, what most distinguishes fixed- from floating-point arithmetic is the result-format's dependence upon the operation.

When fixed-point arithmetic cannot be exact, it must follow rounding procedures that depend upon the application's and programming language's conventions in ways too complicated and diverse (and sometimes bizarre) to discuss at length here. For example, some banks still round monetary quantum fractions bigger than 0.1 up or down in whichever direction favors the bank. Durations of telephone calls are commonly rounded up to minutes before charges are computed.

Ideally, programmers should be able to express, by some locution like "RoundΩ(1/q, Æ)", the manner (specified by Ω) in which an expression Æ should be rounded to an integer multiple of the quantum q, provided Æ involves at most one arithmetic operation. Ideally Æ should be computed exactly before being rounded according to procedure RoundΩ selected from a catalog or library supplied by the compiler and perhaps augmented by the programmer; otherwise "Æ" by itself (not an argument of RoundΩ) within a larger expression would be rounded according to whatever rounding procedure and quantum are the programming language's defaults for the operation and operands that appear in Æ. I know of no current language that behaves ideally.

Almost any rounding procedure could be simulated relatively easily and quickly on old 680x0-based Apple Macintoshes by using their SANE *comp* format, with $-L = H = 2^{64}-1$, as an intermediate for every fixed-point arithmetic operation. This *comp* format also afforded $\pm\infty$ and NaNs, missing from traditional fixed-point formats. Current Macintoshes, based upon IBM's PowerPC architecture, lack the *comp* format. It is unsupported by Microsoft's programming languages though latent in *Wintel* hardware's Double-Extended floating-point format. Deprived of so wide a fixed-point format, fixed-point programmers must resort to trickery, sometimes even to floating-point arithmetic, when trying to simulate rounding procedures foreign to their chosen programming language.

Hardware to compute RoundΩ(1/q, Æ) correctly does not need an infinitely wide register to compute Æ exactly before rounding. At least for rational operations and √, the register need only extend a few digits (called "guard, round and sticky") beyond the desired result's width to support the desired rounding operation. A more difficult challenge for hardware designers is the accommodation of quanta q and rounding procedures Ω more numerous for fixed- than needed for floating-point arithmetics. This implies that working fixed/floating-point registers must be somewhat wider than all values normally loaded from and stored to memory other than during register-dumps.

The next example illustrates the most pernicious difference between fixed- and floating-point arithmetics:

Assignment Statements	**Displayed Values**
x := 7.0/10.0	x = 0.7000…000
y := 4.0/10.0	y = 0.4000…000
z := 3.0/10.0	z = 0.3000…000
d := (x–y) – z	d = 0.0000…000

Is d really zero? Probably YES in fixed-point, because usually *What You See Is What You Get*. In binary floating-point no value displayed above can match the value stored in memory; the stored value of d can be about $-2.98/10^{8}$ or $-5.55/10^{17}$ or $-2.71/10^{20}$ depending upon the variables' precision (presumed all the same), respectively Single, Double, or Double-extended IEEE Standard binary floating-point.

Of course, displaying too few digits after the point can obscure fixed-point values; but common practice is to display enough digits to distinguish values that differ by one quantum. Floating-point values are often displayed in fixed-point fashion (like Fortran's F-format) and with fewer digits than suffice to distinguish adjacent values.

Many practitioners of fixed-point, and their clients, think of numbers as strings of digits that combine according to rules that schools and others propagate like Catechism:- articles of faith to be believed even if not understood. Some floating-point practitioners think that way too; but more often they think of noninteger numbers as slightly fuzzy things whose rightmost few digits are irrelevant and immaterial. Neither mode of thought is quite right, but that is also irrelevant here. The point here is that fixed-point practitioners have certain expectations, not the same for everyone, and are more likely to be surprised and confused by deviations from their expectations than are floating-point practitioners who tend to tolerate deviations even when they shouldn't.

What Good are Extended Floating-Point Formats?

This is more a marketing question than mathematical, and although the reasoning entails some mathematics beyond the acquaintance of most marketing personnel, they will all appreciate the conclusions, namely that …

- Were the wages of sin collected only by the sinners, extended formats could be optional.
- Extended formats are required more to support a mass market than for numerical experts.

A capacity for error-analysis is what most distinguishes a numerical expert from a numerically inexpert (but otherwise perhaps quite clever) programmer. The latter usually chooses formulas that would be correct in the absence of roundoff, and then executes them in floating-point expecting roundoff to contribute negligibly to the desired result. Usually roundoff is negligible. But otherwise, when roundoff degrades the result substantially, who suffers?

… to be continued …

Cambridge philosopher Ludwig Wittgenstein (1889-1951) ended his *Tractatus* (1921) with almost this tautology:

"What we cannot name we must pass over in silence."