

**Formats:** 754 “basic formats” defined both coteries and encodings, and thus object sizes in bits, while 754 “extended formats” defined only coteries. This distinction becomes critical with decimal arithmetic that can be packed in different ways to optimize different properties. So in 754R, the former notion of a “format” is replaced by these notions:

- **coterie:** a set of Level 2 representable entities comprising numbers and symbols; no particular encoding is implied. A higher-level language application programmer usually thinks at this level when referring to the "type" of a variable. This standard defines several kinds of **standard coteries**: this standard also defines **cliche** encodings for **basic** and **storage coteries** **but not for non-interchange** and **variable-width coteries**.
  - **floating-point coteries** allow access to sign, exponent, and significand by the higher-level functions specified by this standard.
  - **[fixed-point coteries are not specified by this standard. 754R's native decimal arithmetic is designed to minimize roundoff. The quantum - radix point - specifies how many significand digits are to be thought of as to the right of the point. To use that arithmetic to implement a fixed-point coterie, follow each operation that might change the quantum - multiplication, fma, division, sqrt, etc - by a quantize operation. Two rounding errors might arise, one from the arithmetic operation and one from the quantize. Use the higher-level functions specified by this standard to determine the sign, quantum - radix point, and significand of a decimal coterie thought of as fixed-point. A language that supports fixed-point arithmetic fully should allow programmers to defend fixed-point coteries by declaring bounds and enabling bounds checking. Note that this document refers to fixed-point coteries and fixed-width coteries.]**
- **encoding** : Level 4 bit strings to represent a coterie. A lower-level system programmer usually thinks at this level when referring to the "type" of a variable. That's what's important for programming communications protocols, storage formats, cache optimizations, and the kind of assembly language, masquerading as higher-level, represented by equivalence/union optimizations. This standard specifies **cliches** – encodings for basic and storage coteries only. Cliches are used for bit-level external data representations e.g. RFC 1832 XDR. In contrast, this standard specifies no method for interchanging other standard coteries between platforms.  
[Some encodings might become popular enough to become de-facto cliches, like

IA32 x80.]

[Each of this standard's decimal cliches encode one floating-point and a number of fixed-point coterie, one for each exponent.]

[In decimal cliches, should an exception be signaled when:

Data overflows from an (unnormalized with a radix point position) fixed-point coterie to a (normalized with an exponent) floating-point coterie?

Cancellation changes computation on a normalized floating-point coterie to computation on an unnormalized fixed-point coterie?]

“Interchange” refers to interchange between different platforms. A platform is a combination of hardware and software (calling conventions, kernel, and shared libraries) that defines an Application Binary Interface, an **ABI**, which is the whole contract between the platform and applications that run upon it. Note that a platform might support one programming language or several cooperating ones.

The cliches not only govern how interchange coterie are communicated between different platforms, but also how interchange coterie are communicated within a platform by such means as separately compiled function calls. This standard defines methods for determining properties of coterie but not cliches. Unless a subprogram accesses encoding properties by means such as `sizeof()` or `equivalence/union`, within that subprogram a compiler might substitute a more efficient encoding for the cliché. It is generally better for the compiler to decide that than for the programmer to attempt to express that by declaring variables of different non-portable types. Such optimization might render data unrecognizable while debugging, just as optimization might render code unrecognizable while debugging.

"Implementing the 754R standard" in a particular radix means implementing either an interchange coterie (and its cliché) or a standard coterie that is a superset of an interchange coterie. Thus an implementation of b64 conforms to 754R in base two, because b64 is an interchange coterie. And an implementation of x80 conforms to 754R because x80 is a superset of an interchange coterie, namely b64.

[Why does a specialized application such as a spreadsheet need to implement an interchange coterie to be conforming? Does such an application need to be conforming? Or need to be called conforming for marketing reasons?]

This standard defines several kinds of standard coterie:

- a **variable-width coterie** is available on some platforms for arithmetic and storing data, but are not for interchanging data among platforms. The precision and range of a variable-width coterie might vary dynamically at run time, as might the size in storage. The standard does not require a variable-width coterie, and defines neither a name nor an encoding. Interchange among platforms is via character sequences. Section 3.5 describes variable-width coterie special properties and special operations. [\[What about expression widening for variable-width coterie? Widest needed including generics?\]](#)
- **fixed-width coterie**, of several varieties, are available on some platforms for storing data. The precision and range of a fixed-width coterie is determinable statically from the program text, [and the corresponding encoding is usually defined so that all members have the same size in storage.](#) [\[An exception might be very wide coterie that, to save space, are encoded to suppress leading or trailing zeros or otherwise compress the significands. All members have the same fixed logical width, but some members might be encoded into lesser storage widths. So, as with variable-width coterie, C's sizeof\(\) would not be a compile-time function for types employing such compressed encodings.\]](#)
  - **non-interchange coterie** ([x80, unpacked decimal](#)) are fixed-width coterie with no defined standard names or encodings. None are required by this standard. If implemented they are available for arithmetic, but not for interchanging data among platforms. Interchange among platforms is via character sequences.
  - **interchange coterie** are fixed-width coterie with standard names and cliches. They are widely available for storage and for data interchange among platforms. The coterie names used in this standard are not usually those used in programming environments (binary64 vs. double vs. REAL\*8).
    - **basic coterie** (XDR)(external)(basic) (portable) (b32, b64, b128, d64, d128) are interchange coterie, available for arithmetic [on systems that implement them](#) This standard requires that at least one be implemented for conformance in a particular radix.
    - **storage coterie** (b16,d32) are interchange coterie, [which this standard permits to be implemented by providing only conversion operations to storage coterie destinations; other computational](#)

operations with these coterie as destinations are not required .  
No storage coterie is required to be implemented by this standard.

**Types in languages:** Programming languages may provide multiple data types. A type corresponds to a coterie and perhaps an encoding, but one coterie might have different possible encodings denoted by different language types, or selected by an optimizing compiler: an interchange coterie might be encoded in its cliché or encoded in an unpacked form for use internally. A cliché and internal encoding may represent the same numbers, infinities, and NaNs, but the internal encoding might be faster. In languages like C, these internal encodings might be identified with type names like `float_t` and `double_t`. The difference between `double` and `double_t` may be coterie (due to widento) or encodings (due to unpacking).

One might want to declare various types:

- `double_xt` interchange
- `double_st` storage/memory (same coterie and size as interchange)
- `double_rt` register (same coterie as interchange)
- `double_et` evaluation (widento coterie)
- computational (internal unpacked form mentioned by John Harrison, not visible to programmer]

“Implementing an interchange coterie” means

- allowing variables and constants to be declared in that coterie
- providing a widento mode for that coterie
- providing conversions between all implemented coterie
- preferably providing conversions to narrower interchange coterie, even if unimplemented for arithmetic (because no corresponding widento mode is implemented)
- **preferably implementing some or all** interchange coterie that are subsets of the implemented interchange coterie

A specialized implementation like a spreadsheet might implement just one coterie such as `b64` or `d128`. Likewise scripting languages often don't have explicit variable declarations and so all variables are of just one type. A more general-purpose implementation might support several coterie including non-interchange coterie like `x80` or variable width.

“Implementing a standard non-interchange coterie” means

- allowing variables and constants to be declared in that coterie
- providing a widento mode for that coterie
- providing conversions between all implemented coterie
- preferably providing conversions to narrower interchange coterie, even if unimplemented for arithmetic (because no corresponding widento mode is

implemented)

- **implementing at least one, and preferably all**, interchange coterie that are subsets of the implemented non-interchange coterie

Thus conforming to 754R by implementing x80 implies providing computational operations that support at least x80 destinations, and conversions between x80 and b32 or b64 or preferably both.

Programming languages should provide programmers with locutions to:

- declare a variable or constant in a particular encoding (type)
- declare a variable or constant belonging to an encoding to be chosen by the compiler, with **at least** the amounts of range and precision specified by the programmer [typically provided by parameterized types or with macros. A compiler rejects as errors demands for more range and precision than it can provide.]
- determine the actual range and precision associated with any particular encoding, variable, or constant. These locutions are typically invocations of library functions.

Operations:

- Section 5 lists operations: explicit-width, implicit-width, and non-computational.
- Section W: **Widening modes** specify the coterie to which implicit-width operations are rounded.
- Section O: **Optimizations** that are licensable. Some examples proposed:
  - widen coterie across function calls
  - use internal encodings across function calls
  - use convert-contraction optimizations (these use the contraction operations producing narrower results than operands),
  - skip conversion to a narrower coterie when followed by conversion back to the wider coterie
- Section 6 discusses roundings to destination coterie.
- Section 7 discusses exceptions and default exception handling.
- Section 8 discusses alternate exception handling. Which of these frameworks should be added to programming environments to be used to define alternate exception handling? In all cases implementation could be by argument/result testing, flag testing, or by trapping.
  - one static mode declaration for arbitrary replacement handling; protected code is not resumable; undefined how much of it was executed: *{protected code}* but on exception *{arbitrary alternate code}*
  - several static mode declarations for specific alternate handling, protected code is not interrupted (so if implemented by trapping, traps must be resumable) *{protected code}* but on exception *action* *action* is from a predefined menu for each *exception*:
    - on underflow abrupt
    - on spill scale and count *invariable*
    - on invalid replace with *variable*
  - [REJECTED: too much programmer effort: several modeless new operations, such as:
    - abrupt multiply
    - scaled product
    - divide with presubstitution]

For required exceptions, each of these frameworks eliminates dynamic global flags

- Section 9: Fixed-point (unnormalized) arithmetic, operations, and exceptions (possible normalization exception indicating possibly accidental migration from fixed-point to floating-point realm?)
- Annex S: signaling NaNs

**Implementation mechanisms:** The following are described as implementation mechanisms in annexes rather than required programming interfaces:

- global dynamic mode bits. Required programming interface is static mode declarations. Dynamic mode bits are optional to support debugging.
- global dynamic flag bits. Required programming interface is static alternate exception handling mode declarations [[Alternate exception handling modes require their named exceptions.](#)] Dynamic flag bits are optional to support debugging [[of unrequited exceptions](#)][[Annex D](#)]
- global dynamic alternate exception handling mode bits. Required programming interface is static alternate exception handling mode declarations. Dynamic mode bits are optional to support debugging.
- rounding precision modes. Required programming interface is static wide to mode declarations. Dynamic mode bits are optional to support debugging.
- 754 section 8 traps. Required programming interface is static alternate exception handling mode declarations. Trapping interface is very platform-specific and not standardizable in a useful way.
- signaling NaNs (for debugging) and application+platform-specific extensions like missing values. Tightly related to trapping interface and not standardizable in a useful way.