RFC1832 Agosto 1995

(resumido por Juan A. Ternero)

XDR: Estándar de Representación Externa de Datos

1. INTRODUCCIÓN

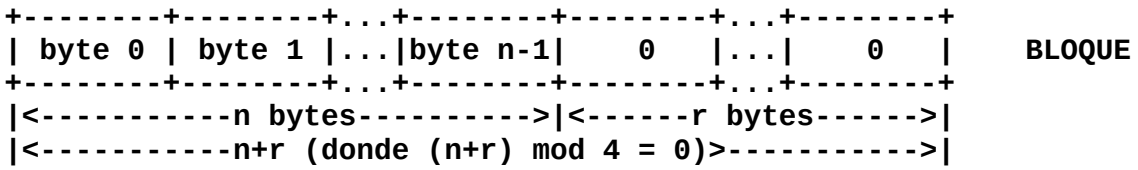
XDR es un estándar para la descripción y representación de datos.

XDR usa un lenguaje para describir los formatos de datos similar al lenguaje C.

2. TAMAÑO BÁSICO DE BLOQUE

En la representación, todos los elementos deben ser múltiplo de cuatro bytes (octetos).

Si es necesario, se añaden bytes (de 0 a 3) de valor cero.



3. TIPOS DE DATOS DE XDR

Declaración general:

- “menor que” y “mayor que” (< y >) denotan secuencias de datos de longitud variable
- corchetes ([y]) denotan secuencias de datos de longitud fija

3.1 Entero

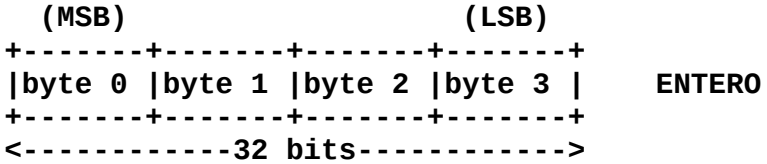
Dato de 32 bits en el rango [-2147483648,2147483647].

Declaración:

```
int identificador;
```

Representación:

Notación en complemento a dos



3.2. Entero sin signo

Dato de 32 bits en el rango [0,4294967295].

Declaración:

```
unsigned int identificador;
```

Representación:

```
 (MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |    ENTERO SIN SIGNO
+-------+-------+-------+-------+
<------------32 bits------------>
```

3.3 Enumerado

Adecuado para describir subconjuntos de enteros.

Declaración:

```
enum { nombre-identificador = constante, ... } identificador;
```

Representación:

Misma representación que enteros.

Ejemplo:

```
enum { ROJO = 2, AMARILLO = 3, AZUL = 5 } colores;
```

3.4 Lógico

Declaración:

```
bool identificador;
```

Esto es equivalente a:

```
enum { FALSE = 0, TRUE = 1 } identificador;
```

3.5 Hiper enteros (enteros de mayor rango, con y sin signo)

Números de 64 bits (8 bytes).

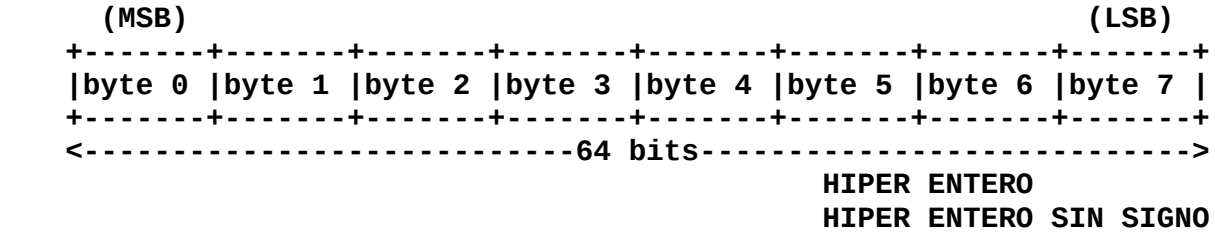
Declaraciones:

```
hyper identificador;
```

```
unsigned hyper identificador;
```

Representaciones:

Extensiones obvias de las definidas anteriormente para entero (con y sin signo).



3.6 Punto flotante

Tipo de dato en punto flotante "float" (32 bits).

Declaración:

```
float identificador;
```

Representación:

Estándar del IEEE para números normalizados en punto flotante de precisión simple.

Tres campos:

S: signo. 1 bit.

E: exponente. 8 bits.

F: mantisa. 23 bits

El número en punto flotante se describe por:

```
(-1)**S * 2**(E-127) * 1.F
```

3.7 Punto flotante de doble precisión

Tipo de dato en punto flotante de doble precisión "double" (64 bits).

Declaración:

```
double identificador;
```

Representación:

Una de las formas del Estándar del IEEE para números normalizados en punto flotante de precisión doble.

Tres campos:

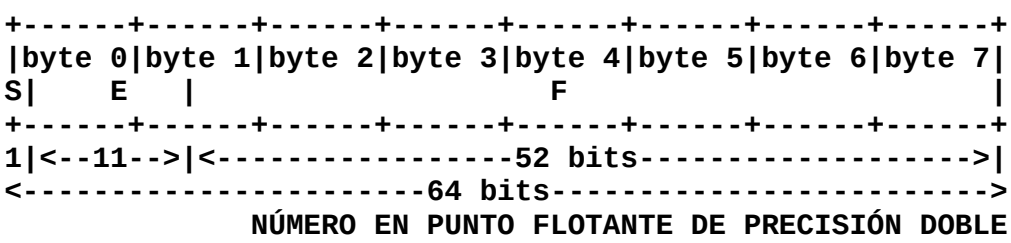
S: signo. 1 bit.

E: exponente. 11 bits.

F: mantisa. 52 bits

El número en punto flotante se describe por:

(-1)**S * 2**(E-1023) * 1.F



NÚMERO EN PUNTO FLOTANTE DE PRECISIÓN DOBLE

3.8 Punto flotante de cuádruple precisión

Tipo de dato en punto flotante de cuádruple precisión "quadruple" (128 bits).

Declaración:

```
quadruple identificador;
```

Representación:

Estándar del IEEE para números normalizados en punto flotante de precisión doble extendida.

Tres campos:

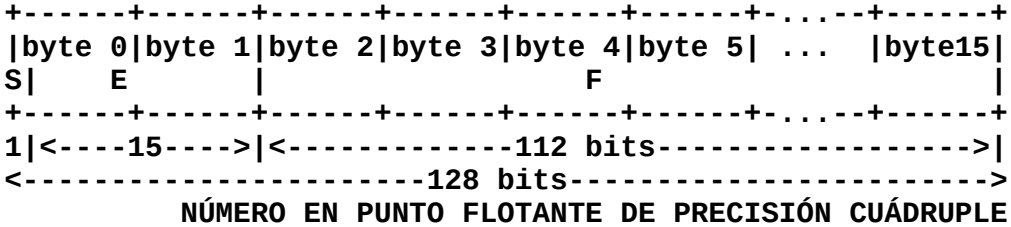
S: signo. 1 bit.

E: exponente. 15 bits.

F: mantisa. 112 bits

El número en punto flotante se describe por:

(-1)**S * 2**(E-16383) * 1.F



NÚMERO EN PUNTO FLOTANTE DE PRECISIÓN CUÁDRUPLE

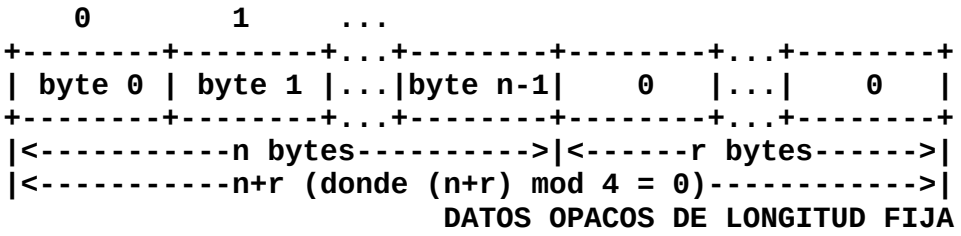
3.9 Datos opacos de longitud fija

Datos sin interpretar de longitud fija (n bytes).

Declaración:

```
opaque identificador[n];
```

Representación:



DATOS OPACOS DE LONGITUD FIJA

3.10 Datos opacos de longitud variable

Datos sin interpretar de longitud variable (contados).

Declaración:

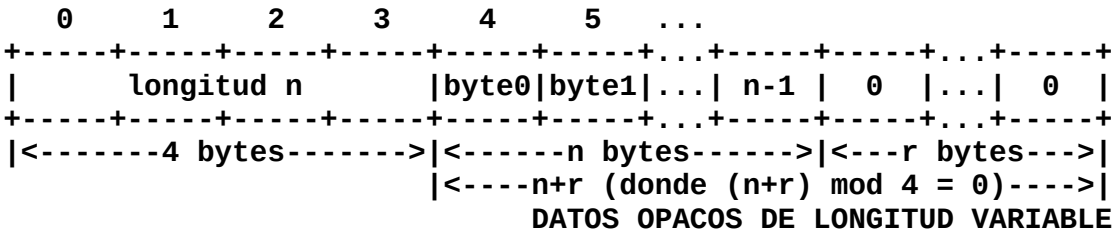
```
opaque identificador<m>;

opaque identificador<>;
```

La constante m es un límite superior de los bytes que la secuencia puede contener. Si no se especifica m, como en la segunda declaración, se supone que vale (2**32) - 1, la longitud máxima.

Ejemplo:

```
opaque datos_fichero<8192>;
```

Representación:



DATOS OPACOS DE LONGITUD VARIABLE

3.11 Cadena

Cadena de n bytes ASCII (numerados de 0 a n-1).

Declaración:

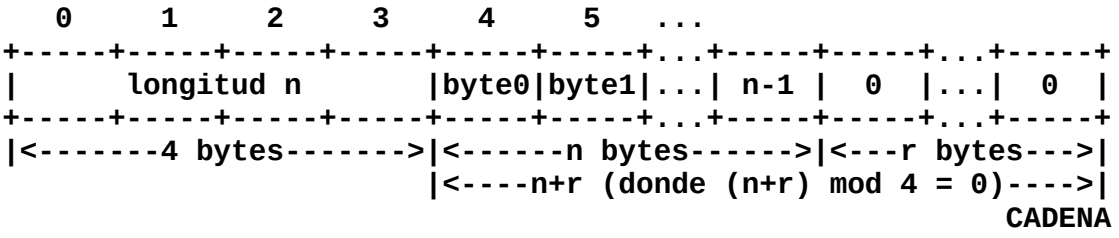
```
string object<m>;

string object<>;
```

La constante m es un límite superior de los bytes que la secuencia puede contener. Si no se especifica m, como en la segunda declaración, se supone que vale (2**32) - 1, la longitud máxima.

Ejemplo:

```
string nombre_fichero<255>;
```

Representación:



3.12 Tabla de longitud fija

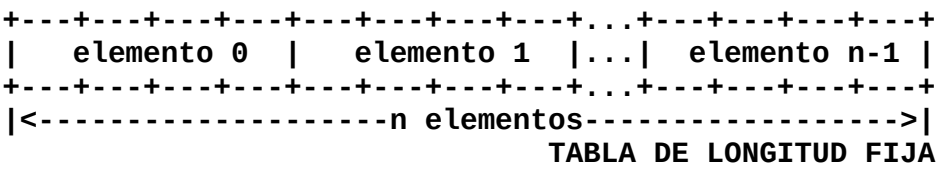
Tabla de longitud fija de elementos homogéneos.

Aunque todos los elementos son del mismo tipo, los elementos pueden tener diferentes tamaños.

Declaración:

```
nombre-tipo identificador[n];
```

Representación:



3.13 Tabla de longitud variable

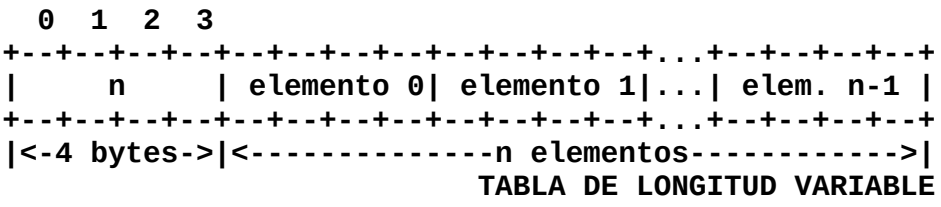
Tabla de longitud variable de elementos homogéneos.

Declaración:

```
nombre-tipo identificador<m>;

nombre-tipo identificador<>;
```

La constante m es un límite superior de los bytes que la secuencia puede contener. Si no se especifica m, como en la segunda declaración, se supone que vale (2**32) - 1, la longitud máxima.

Representación:



3.14 Estructura

Estructuras de diferentes tipos de datos.

Declaración:

```
struct {
  componente-declaración-A;
  componente-declaración-B;
...
} identificador;
```

Los componentes de la estructura son codificados en el orden de su declaración en la estructura. El tamaño de cada componente es múltiplo de cuatro bytes, aunque los componentes pueden tener diferentes tamaños.

Representación:

El mismo orden que en la declaración.

3.15 Unión discriminada

Tipo compuesto de:

- **discriminante**
- **UN tipo seleccionado de un conjunto de tipos preacordados**

El tipo del discriminante debe ser:

- **tipo entero ("int" o "unsigned int")**
- **tipo enumerado(incluyendo "bool")**

Los tipos del componente son llamados “brazos” de la unión, y están precedidos del valor del discriminante que implica su codificación.

Declaración:

```
union switch (declaración-discriminante) {
  case valor-A-discriminante:
    declaración-brazo-A;
  case valor-B-discriminante:
    declaración-brazo-B;
...

default: declaración-por-defecto;
} identificador;
```

Cada palabra clave "case" es seguida de un valor legal del discriminante.

El brazo default es opcional.

Representación:

```
  0   1   2   3
+---+---+---+---+---+---+---+---+
| discriminante |brazo implicado|    UNIÓN DISCRIMINADA
+---+---+---+---+---+---+---+---+
|<---4 bytes--->|
```

3.16 Void

0 bytes.

El tipo void es útil en uniones, donde algunos brazos implican que no se codifique nada (0 bytes).

Declaración:

```
  void;
```

Representación:

```
 ++
 ||    VOID
 ++
--><-- 0 bytes
```

3.17 Constante

La constante simbólica se usa para definir un nombre simbólico para una constante; no declara ningún dato.

Se puede usar igual que una constante normal.

Declaración:

```
  const nombre-identificador = n;
```

Representación:

No hay representación, ya que no se declara ningún dato.

Ejemplos:

```
  const DOCENA = 12;

  const TAM_MAX = 20;
```

3.18 Typedef

Sirve para definir nuevos identificadores para declarar datos.

Es similar al del lenguaje C.

Declaración:

```
  typedef declaración;
```

Representación:

No hay representación, ya que no se declara ningún dato.

Ejemplo 1:

```
typedef float real;

real v1;
float v2;  /* mismo tipo que v1 */
```

Ejemplo 2:

```
typedef int tabla_enteros[TAM_MAX];

tabla_enteros t1;
int t2 [TAM_MAX]; /* mismo tipo que t1 */
```

Cuando typedef se usa para una definición de enumerado, es equivalente (y preferido) eliminar el "typedef" y poner el identificador después de la palabra clave "enum".

Por ejemplo, véanse dos maneras de definir "bool":

```
typedef enum {    /* usando typedef */
  FALSE = 0,
  TRUE = 1
} bool;

enum bool {       /* alternativa preferida */
  FALSE = 0,
  TRUE = 1
};
```

Lo mismo se aplica a "struct" y a "union".

3.19 Datos opcionales

Es un tipo de unión.

Es muy útil para describir estructuras de datos recursivos como listas enlazadas y árboles.
Declaración:

```
nombre-tipo *identificador;
```

Esto es equivalente a la siguiente unión:

```
union switch (bool optado) {
  case TRUE:
    nombre-tipo elemento;
  case FALSE:
    void;
} identificador;
```

Esto es también equivalente a la siguiente tabla de longitud variable:

```
nombre-tipo identificador<1>;
```

Por ejemplo, la siguiente declaración define un tipo "lista_cadena" que codifica listas de cadenas de longitud arbitraria:

```
struct * lista_cadena {
  string item<>;
  lista_cadena siguiente;
};
```

Se podría haber declarado de forma equivalente como la siguiente unión:

```
union lista_cadena switch (bool optado) {
  case TRUE:
    struct {
      string item<>;
      lista_cadena siguiente;
    } elemento;
  case FALSE:
    void;
};
```

o como una tabla de longitud variable:

```
struct lista_cadena <1> {
  string item<>;
  lista_cadena siguiente;
};
```

Estas dos últimas declaraciones oscurecen la intención del tipo lista_cadena, por eso es preferible la declaración como datos opcionales.

Los datos opcionales tienen también una estrecha relación con la forma en que las estructuras de datos recursivos se representan en lenguajes de alto nivel como el Pascal o el C mediante el uso de punteros. De hecho, la sintaxis es la misma que la del lenguaje C para los punteros.