12 Apéndice C: Tipos de datos en IndraLogic

Durante la programación, el usuario puede utilizar tipos de datos estándar y tipos de datos definidos por él mismo. A cada identificador se le asigna un tipo de datos que determina cuánto espacio de memoria se reserva y qué valores se corresponden con el contenido de la memoria.

12.1 Tipos de datos estándar

BOOL

Las variables del tipo de datos **BOOL** pueden adoptar los valores lógicos TRUE y FALSE. Se reservan 8 bits de espacio de memoria.

Tipos de datos de números enteros

Entre los tipos de datos de números enteros se cuentan **BYTE**, **WORD**, **DWORD**, **SINT**, **USINT**, **INT**, **UINT**, **DINT**, **UDINT**.

Los distintos tipos de números cubren rangos numéricos distintos. Para los tipos de datos de números enteros rigen los siguientes rangos:

Tipo	Límite inferior	Límite superior	Capacidad de memoria
BYTE	0	255	8 bits
WORD	0	65535	16 bits
DWORD	0	4294967295	32 bits
SINT:	-128	127	8 bits
USINT:	0	255	8 bits
INT:	-32768	32767	16 bits
UINT:	0	65535	16 bits
DINT:	-2147483648	2147483647	32 bits
UDINT:	0	4294967295	32 bits

Fig. 12-1 : Tipos de datos de números enteros y sus límites de rango

Debido a esto, puede ocurrir que durante la conversión de tipos mayores a menores se pierda información.

REAL / LREAL

Los tipos de datos **REAL** y **LREAL** son los denominados tipos de coma flotante. Son necesarios cuando se utilizan números racionales. El espacio de memoria reservado es de 32 bits para REAL y 64 bits para LREAL

Valores admisibles para REAL: 1.175494351e-38F hasta 3.402823466e+38F

Valores admisibles para LREAL: 2.2250738585072014e-308 hasta 1.7976931348623158e+308

STRING

Una variable del tipo de datos **STRING** puede adoptar cualquier cadena de caracteres. La indicación de tamaño para la reserva de espacio de memoria durante la declaración se refiere a caracteres y puede realizarse en paréntesis o corchetes. Si no se especifica ningún tamaño, se adoptan por defecto 80 caracteres.

¡En principio, la longitud de string es ilimitada, pero las funciones de string sólo pueden procesar longitudes de 1-255!

```
str:STRING(35):='Esto es un string';
```

Fig. 12-2 : Ejemplo de una declaración de string con 35 caracteres

Tipos de datos de tiempo

Los tipos de datos **TIME**, **TIME_OF_DAY** (abreviado como **TOD**), **DATE** y **DATE_AND_TIME** (abreviado como **DT**) se tratan internamente como DWORD.
En TIME y TOD, el tiempo se indica en milisegundos, y en TOD se calcula a partir de las 00:00 horas.

En DATE y DT, el tiempo se indica en segundos, y se calcula a partir del 1 de enero de 1970 a las 00:00 horas.

A continuación se describen los formatos de datos de tiempo para la asignación (constantes de tiempo y de fecha):

Constantes de tiempo TIME:
Una constante TIME consiste siempre en una "t" o "T" inicial ("time" o "TIME" en la forma no abreviada) y una cruz doble "#".

A continuación viene la declaración de tiempo propiamente dicha, que puede consistir en días (indicados con "d"), horas (indicadas con "h"), minutos (indicados con "m"), segundos (indicados con "s") y milisegundos (indicados con "ms"). Se debe tener en cuenta que las indicaciones de tiempo deben estar ordenadas según su tamaño (d antes de h antes de m antes de s antes de m antes de ms), si bien no tienen por qué aparecer todos los tiempos.

Valor máximo: 49d17h2m47s295ms (4194967295 ms)

TIME1 := T#14ms;	
TIME1 := T#100S12ms;	(*Se permite la superación del límite en el componente más alto*)
TIME1 := t#12h34m15s;	

Fig. 12-3 : Ejemplos de constantes TIME correctas en una asignación ST

TIME1 := t#5m68s	(*Superación del límite en un componente más bajo*)
TIME1 := 15ms;	(*Falta T#*)
TIME1 := t#4ms13d;	(*Orden incorrecto de los datos de tiempo*)

Fig. 12-4 : Ejemplos de constantes TIME **incorrectas** en una asignación ST

Constantes DATE, para indicaciones de fecha:

Una constante DATE se declara mediante una "d", "D", "DATE" o "date" inicial, seguida de un símbolo "#". A continuación puede introducir una fecha cualquiera en el orden año-mes-día. Valores posibles: 1970-00-00 hasta 2106-02-06.

```
DATE#2005-05-06
d#1998-03-29
```

Fig. 12-5 : Ejemplos de constantes DATE

Constantes TIME_OF_DAY, para guardar horas:

Una declaración TIME_OF_DAY empieza por "tod#", "TOD#", "TIME_OF_DAY#" o "time_of_day#", y a continuación puede indicar una hora en la notación: hora:minuto:segundo. Los segundos pueden indicarse como números reales, y por lo tanto también se pueden especificar fracciones de segundo. Valores posibles: 00:00:00 hasta 23:59:59.999.

```
TIME_OF_DAY#15:36:30.123
tod#00:00:00
```

Fig. 12-6 : Ejemplos de constantes TIME_OF_DAY

Constantes DATE_AND_TIME, combinación de fecha y hora:

Las constantes DATE_AND_TIME empiezan por "dt#", "DT#", "DATE_AND_TIME#" o "date_and_time#". Tras la indicación de la fecha sigue un guión y a continuación la hora. Valores posibles: 1970-00-00-00:00:00 hasta 2106-02-06-06:28:15.

```
DATE_AND_TIME#2005-05-06-15:36:30
dt#1998-03-29-00:00:00
```

Fig. 12-7 : Ejemplos de constantes DATE_AND_TIME

12.2 Tipos de datos definidos

Array

Se soportan campos unidimensionales, bidimensionales y tridimensionales (arrays) de tipos de datos elementales. Los arrays pueden definirse en la parte de declaración de un componente y en las listas de variables globales. Mediante el encaje de arrays (ARRAY[0..2] OF ARRAY[0..3] OF …) pueden crearse un máximo de 9 dimensiones.

```
<Nombre de campo>:ARRAY
[<ug1>..<og1>,<ug2>..<og2>,<ug3>..<og3>] OF <tipo elem.>
```

Fig. 12-8 : Sintaxis para la definición de arrays

ug1, ug2, ug3 indican el límite inferior del área de campo, y og1, og2, og3 indican el límite superior. Los valores límite deben ser números enteros y seguir el intervalo de valores DINT.

```
Juego_de_cartas : ARRAY [1..13, 1..4] OF INT;
```

Fig. 12-9 : Ejemplo de definición de un array

Inicialización de arrays:

Ejemplos de inicialización completa de un array:arr1 : ARRAY [1..5] OF INT := 1,2,3,4,5;

```
arr2 : ARRAY [1..2,3..4] OF INT := 1,3(7);
       (* abreviatura de 1,7,7,7 *)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := 2(0),4(4),2,3;
       (* abreviatura de 0,0,4,4,4,4,2,3 *)
```

Fig. 12-10 : Ejemplo de la inicialización de arrays

Ejemplo de inicialización de un array de una estructura:

```
TYPE STRUCT1
STRUCT
      p1:int;
      p2:int;
      p3:dword;
END_STRUCT

ARRAY[1..3] OF STRUCT1:=
(p1:=1,p2:=10,p3:=4723),(p1:=2,p2:=0,p3:=299),
(p1:=14,p2:=5,p3:=112);
```

Fig. 12-11 : Ejemplo de inicialización de un array de una estructura

Ejemplo de inicialización parcial de un array:

```
arr1 : ARRAY [1..10] OF INT := 1,2;
```

Fig. 12-12 : Ejemplo de inicialización parcial de un array

Los elementos para los cuales no se especifica ningún valor se inicializan con el valor inicial predeterminado del tipo básico. Por lo tanto, en el ejemplo anterior los elementos anarray[6] hasta anarray[10] se inicializan con 0.

Acceso a componentes de array:

Para acceder a componentes de arrays en un campo bidimensional se utiliza la siguiente sintaxis:

```
<Nombre_de_campo>[Indice1,Indice2]
```

Fig. 12-13 : Sintaxis para el acceso a componentes de un array bidimensional

```
Juego_de_cartas[9,2]
```

Fig. 12-14 : Ejemplo de acceso a un componente de un array

Nota: ¡Si define en su proyecto una función con el nombre CheckBounds, con ella puede comprobar automáticamente superaciones del rango en arrays!

Función CheckBounds

¡Si define en su proyecto una función con el nombre **CheckBounds**, con ella puede comprobar automáticamente superaciones del rango en arrays! El nombre de la función es fijo y debe tener únicamente esta designación.

```
FUNCION CheckBounds : DINT
VAR_INPUT
 index, lower, upper: DINT;
END_VAR
IF  index < lower THEN
 CheckBounds := lower;
ELSIF  index > upper THEN
 CheckBounds := upper;
ELSE  CheckBounds := index;
END_IF
```

Fig. 12-15 : Ejemplo de la función CheckBounds

El siguiente programa ejemplar para la comprobación de la función CheckBounds excede los límites de un array definido. La función CheckBounds garantiza que el valor TRUE no sea asignado a la posición A[10], sino al límite de rango superior todavía válido A[7]. De este modo, mediante la función CheckBounds se pueden corregir accesos fuera de límites de array.

```
PROGRAM PLC_PRG
VAR
a : ARRAY [0.0,7] OF INT := 1,2;
b : INT := 10;
END_VAR
a[b]:=TRUE;
```

Fig. 12-16 : Programa de prueba para la función CheckBounds

Nota: ¡La función CheckBounds contenida en la CheckLib es una solución de ejemplo! Antes de utilizar la biblioteca, compruebe si la función trabaja tal como desea o implemente una función CheckBounds apropiada como componente directamente en su proyecto.

Pointer

En los pointers se guarda la dirección de variables o bloques de función mientras el programa está en ejecución.

```
<Denominador>: POINTER TO <Tipo de dato/bloque de
funcion>;
```

Fig. 12-17 : Sintaxis para la declaración de pointer

Un pointer puede apuntar a cualquier tipo de dato y bloque de función, incluso a los definidos por el propio usuario.

Mediante el operador de dirección ADR se asigna al pointer la dirección de una variable o un bloque de función.

La desreferenciación de un pointer se realiza insertando el operador de contenido "^" detrás del denominador del pointer.

Importante: ¡Un pointer se incrementa por bytes! Mediante la instrucción p=p+SIZEOF(p^); se puede alcanzar un incremento como en el Compilador C.

```
pt:POINTER TO INT;
var_int1:INT := 5;
var_int2:INT;
pt := ADR(var_int1);
var_int2:= pt^; (* var_int2 es ahora 5 *)
```

Fig. 12-18 : Ejemplo de pointer

Para comprobar si la dirección a la que accede el pointer se halla en el área de memoria válida, se puede implementar la función Check pertinente , que se llamará automáticamente antes de cada acceso al contenido de un pointer. La función debe llevar el nombre CheckPointer y estar disponible en el proyecto (directamente en el proyecto o mediante una biblioteca). Se pueden utilizar los siguientes parámetros de entrada:

```
FUNCTION CheckPointer : DWORD
VAR_INPUT
 dwAddress : DWORD;
 iSize : INT;
 bWrite: BOOL;
END_VAR
```

Fig. 12-19 : Función CheckPointer para sistemas con pointer de 32 bits

```
FUNCTION CheckPointer : WORD
VAR_INPUT
 dwAddress : WORD;
 iSize : INT;
 bWrite: BOOL;
END_VAR
```

Fig. 12-20 : Función CheckPointer para sistemas con pointer de 16 bits

La función devuelve la dirección que se utiliza para la desreferenciación del pointer, esto es, en caso de éxito, la dirección que se transmitió como parámetro de entrada *dwAddress*.

Nota: Cuando se utiliza Cambio Online, los contenidos de las direcciones pueden desplazarse. Tenga esto en cuenta al utilizar pointers en direcciones.

Funciones CheckPointer y CheckPointerAligned

Para comprobar los accesos de pointer durante el tiempo de ejecución, se pueden crear funciones Check con el nombre mencionado más abajo, que se llaman automáticamente antes de cada acceso al contenido de un pointer si están disponibles en el proyecto (directamente en el proyecto o mediante una biblioteca):

- Función **CheckPointer** para comprobar si la dirección a la que accede el pointer se halla en el área de memoria válida,
- Función **CheckPointerAligned**, la cual contiene las funciones de CheckPointer y además comprueba el alineamiento de la memoria.

Las funciones deben tener exactamente los nombres mencionados. Devuelven la dirección que se utiliza para la desreferenciación del pointer, esto es, en caso de éxito, la dirección que se transmitió como primer parámetro de entrada (dwAddress en el ejemplo mostrado más abajo).

Vea en el siguiente ejemplo de una función CheckPointerAligned qué parámetros de entrada pueden utilizar las funciones Check. Los nombres de parámetros también son ejemplos. Una función CheckPointer debería tener el mismo aspecto, con la salvedad de que desaparecería el parámetro para la granularidad del acceso.

FUNCTION CheckPointerAligned : DWORD	El tipo de dato de la función (valor de devolución) debe coincidir con el tipo de dato para el apuntador en el sistema de destino actualmente configurado, esto es, DWORD para sistemas que utilizan pointer de 32 bits, WORD para sistemas que utilizan pointer de 16 bits*)
VAR_INPUT	
dwAddress : DWORD;	(* Dirección de destino del pointer; el tipo de dato debe coincidir con el tipo de dato para el apuntador en el sistema de destino actualmente configurado, ver arriba valor de devolución de la función *)
iSize : DINT;	(* Tamaño del acceso; el tipo de dato debe ser compatible con enteros y cubrir el tamaño máximo concebible de los datos en la dirección de acceso *)
iGran : DINT;	(* **desaparece en las funciones CheckPointer**: Granularidad del acceso, p. ej. "2", si INT es el tipo de dato no estructurado más pequeño utilizado en la dirección; el tipo de dato debe ser compatible con enteros *)
bWrite: BOOL;	(* Acceso de lectura o de escritura; TRUE=de escritura; el tipo de dato debe ser BOOL *)
END_VAR	

Fig. 12-21 : Ejemplo de una función CheckPointerAligned

Si en el proyecto existen tanto una función CheckPointer como una función CheckPointerAligned, se llama la función CheckPointerAligned.

Tipo de enumeración, enumeración

Un tipo de enumeración es un tipo de dato definido por el usuario, compuesto por un conjunto de constantes de string. Estas constantes se denominan valores de enumeración.

Los valores de enumeración son conocidos en todo el proyecto, aunque hayan sido declarados localmente en un componente. Es recomendable crear sus tipos de enumeración como objetos en el Object Organizer en la pestaña **Tipos de datos**  . Empiezan con la palabra clave TYPE y terminan con END_TYPE.

```
TYPE <Denominador>:(<Enum_0> ,<Enum_1>, ...,<Enum_n>);
END_TYPE
```

Fig. 12-22 : Sintaxis para la declaración de tipos de enumeración

Una variable del tipo <Denominador> puede adoptar uno de los valores de enumeración y se inicializa con el primero. Los valores son compatibles con números enteros, es decir, con ellos se pueden realizar operaciones igual que con INT. Se puede asignar un número x a la variable. Si los valores de enumeración no están inicializados, el conteo empieza por 0. Al inicializar, asegúrese de que los valores iniciales sean ascendentes. La validez del número se comprueba en el momento de su ejecución.

```
TYPE SEMAFORO: (Rojo, Ambar, Verde:=10); (*Rojo tiene el
valor inicial 0, Ambar 1, Verde 10 *)
END_TYPE
SEMAFORO1 : SEMAFORO ;
SEMAFORO1:=0; (* Semaforo tiene el valor Rojo*)
FOR i:= Rojo TO Verde DO
 i := i + 1;
END_FOR;
```

Fig. 12-23 : Ejemplo de tipos de enumeración

El mismo valor de enumeración **no** puede utilizarse dos veces ni dentro de una enumeración ni en caso de utilización de diversas enumeraciones dentro del mismo componente.

```
SEMAFORO: (rojo, ambar, verde);
COLOR: (azul, blanco, rojo);
```

Fig. 12-24 : Ejemplo de una declaración **errónea** de tipos de enumeración

El ejemplo mostrado en Fig. 12-24 es inadmisible, dado que no se puede emplear rojo para SEMÁFORO y COLOR si éstos se utilizan en el mismo componente.

Estructuras

Las estructuras se guardan como objetos (tipos de datos) en el Object Organizer en la pestaña **Tipos de datos** . Empiezan con las palabras clave TYPE y STRUCT y terminan con END_STRUCT y END_TYPE.

```
TYPE <Nombre de estructura>:
STRUCT
 <Declaracion de variable 1>
 .
 .
 <Declaracion de variable n>
END_STRUCT
END_TYPE
```

Fig. 12-25 : Sintaxis para la declaración de una estructura

<Nombre de estructura> es ahora un tipo conocido en todo el proyecto y que puede utilizarse como un tipo de datos estándar.

Se admiten las estructuras encajadas. La única restricción es el hecho de que no se pueden situar variables en direcciones (¡no se admite la declaración AT!).

```
TYPE Trazo poligonal:
STRUCT
 Start:ARRAY [1..2] OF INT;
 Punto1:ARRAY [1..2] OF INT;
 Punto2:ARRAY [1..2] OF INT;
 Punto3:ARRAY [1..2] OF INT;
 Punto4:ARRAY [1..2] OF INT;
 Fin:ARRAY [1..2] OF INT;
END_STRUCT
END_TYPE
```

Fig. 12-26 : Ejemplo de una definición de estructura Trazo poligonal

```
Poly_1: Trazo poligonal:= ( Start:=3,3, Punto1:=5,2,
Punto2:=7,3, Punto3:=8,5, Punto4:=5,7, Fin := 3,5);
```

Fig. 12-27 : Ejemplo de inicialización de una estructura del tipo Trazo poligonal

No son posibles las inicializaciones con variables. Un ejemplo de inicialización de un array de una estructura se muestra en "Array" en la página 12-6.

Acceso a estructuras:

```
<Nombre_de_estructura>.<Nombre_de_componente>
```

Fig. 12-28 : Sintaxis para el acceso a componentes de estructuras

Para el ejemplo mostrado en **Fig. 12-26** de la estructura Trazo poligonal, el acceso al componente Start tiene lugar por lo tanto mediante `Poly_1.Start`

Referencias

El tipo de dato definido por el usuario Referencia sirve para crear un nombre alternativo (alias) para un tipo de datos o un bloque de función.

Cree sus referencias como objetos en el Object Organizer en la pestaña **Tipos de datos** . Empiezan con la palabra clave TYPE y terminan con END_TYPE.

```
TYPE <Denominador>: <Expresion de asignacion>;
END_TYPE
```

Fig. 12-29 : Sintaxis para la declaración de referencias

```
TYPE message:STRING[50];
END_TYPE;
```

Fig. 12-30 : Sintaxis para la declaración de una referencia

Tipos de subrangos

Un tipo de subrango es un tipo de dato cuyo rango de valores sólo abarca un subconjunto de un tipo básico. La declaración puede realizarse en la pestaña **Tipos de datos** , pero también se puede declarar una variable directamente con un tipo de subrango:

```
TYPE <Name> : <Inttype> (<ug>..<og>)
END_TYPE;
```

Fig. 12-31 : Sintaxis para la declaración de tipos de subrangos

<Name>	debe ser un denominador IEC válido,
<Inttype>	es uno de los tipos de datos SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD (LINT, ULINT, LWORD).
<ug>	es una constante que debe ser compatible con el tipo básico y que establece el límite inferior del tipo de rango. El límite inferior propiamente dicho pertenece a este rango.
<og>	es una constante que debe ser compatible con el tipo básico y que establece el límite superior del tipo de rango. El límite superior propiamente dicho pertenece a este tipo básico.

Fig. 12-32 : Explicaciones para Fig. 12-31

```
TYPE
 SubInt : INT (-4095..4095);
END_TYPE
```

Fig. 12-33 : Ejemplo de declaración de tipos de subrangos

Declaración directa de una variable con un tipo de subrango:

```
VAR
 i1 : INT (-4095..4095);
 i2: INT (5..10):=5;
 ui : UINT (0..10000);
END_VAR
```

Fig. 12-34 : Asegúrese de especificar correctamente un valor inicial si el subrango no contiene el '0'

Si se asigna una constante a un tipo de subrango (en la declaración o en la implementación) que no se halla en este rango (p. ej. i:=5000), se emite un mensaje de error.

Para comprobar el cumplimiento de los límites de rango durante la ejecución, se deben insertar las funciones **CheckRangeSigned** o **CheckRangeUnsigned**. En éstas se pueden capturar violaciones del rango de forma apropiada (p. ej. se puede cortar el valor o colocar un flag de error). Se llaman implícitamente al escribir en una variable que sea de un tipo de subrango formado a partir de un tipo con signo antepuesto o sin signo antepuesto.

Ejemplo:

En caso de una variable de un tipo de subrango con signo antepuesto (es decir, como i en el ejemplo superior), se llama la función CheckRangeSigned, la cual podría estar programada de la siguiente manera para recortar un valor al rango permitido:

```
FUNCTION CheckRangeSigned : DINT
VAR_INPUT
 value, lower, upper: DINT;
END_VAR
IF (value < lower) THEN
 CheckRangeSigned := lower;
ELSIF(value > upper) THEN
 CheckRangeSigned := upper;
ELSE
 CheckRangeSigned := value;
END_IF
```

Fig. 12-35 : Ejemplo de la función CheckRangeSigned

Para la llamada automática es imprescindible el nombre de función CheckRangeSigned y la configuración de la interfaz: valor de devolución y tres parámetros del tipo DINT.

Al ser llamada, la función se parametriza de la siguiente manera:

- value: recibe el valor que se debe asignar al tipo de rango
- lower: el límite inferior del rango
- upper: el límite superior del rango
- return value: el valor que se asigna realmente al tipo de rango

A partir de una asignación i := 10*y; en este ejemplo se genera implícitamente la siguiente:

```
i := CheckRangeSigned(10*y, -4095, 4095);
```

Por ejemplo, si y tiene el valor 1000, pese a ello después de esta asignación i tiene sólo el valor 4095.

Lo mismo se aplica a la función CheckRangeUnsigned: El nombre de la función y la interfaz deben ser correctos:

```
FUNCTION CheckRangeUnsigned : UDINT
VAR_INPUT
 value, lower, upper: UDINT;
END_VAR
```

Fig. 12-36 : Ejemplo de la función CheckRangeUnsigned

Nota: ¡Si no está presente ninguna de las funciones CheckRangeSigned y CheckRangeUnsigned, no tendrá lugar ninguna comprobación de los tipos de subrangos durante la ejecución! ¡En ese caso, la variable i podría adoptar perfectamente cualquier valor entre –32768 y 32767!

Si está implementada una función CheckRangeSigned o CheckRangeUnsigned tal como se muestra arriba, al utilizarse el tipo de subrango en un **bucle FOR** puede generarse un bucle infinito. ¡Esto ocurrirá si el rango especificado para el bucle FOR es exactamente igual o mayor que el del tipo de subrango!

¡La función CheckRangeSigned contenida en la biblioteca CheckLib es una solución de ejemplo! Antes de utilizar la biblioteca, compruebe si la función trabaja tal como desea o implemente una función CheckRange apropiada como componente directamente en su proyecto.

```
VAR
 ui : UINT (0..10000);
END_VAR
FOR ui:=0 TO 10000 DO
...
END_FOR
```

Fig. 12-37 : Ejemplo en el que no se sale del bucle FOR, dado que ui no puede llegar a ser mayor que 10000

Nota: ¡También se debe tener en cuenta el contenido de las funciones CheckRange al utilizar valores de incremento en el bucle FOR!

Para sus notas