PROGRAMACIÓN AVANZADA (2)

Índice

REPASO BÁSICO

Como se ha visto en la primera parte de estos apuntes, con el autómata pude trabajarse con cuatro tipos de datos: BCD, INT, DINT y REAL.

Alguno de los formatos anteriores no permiten, por ejemplo, realizar operaciones aritméticas, este es el caso del formato BCD. Así, la única opción posible para realizar operaciones aritméticas sería convertir este tipo de datos en INT o DINT o REAL, según se defina la operación deseada.

Los códigos AWL para pasar de BCD a INT o DINT son:

BCD a INT → **BTI**
BCD a DINT→ **BTD**

Los códigos para pasar de INT o DINT a BCD son:

INT a BCD → **ITB**
DINT a BCD → **DTB**

También, con numeros enteros (INT o DINT) se pueden realizar operaciones definidas en el campo de los números reales. Para ello, habrá que convertir el número entero en doble entero y, posteriormente, el doble entero a real, es decir:

INT→ DINT→REAL

En este caso, los códigos para pasar de INT a DINT, y de DINT a REAL serán:

INT a DINT → **ITD**

DINT a REAL→ **DTR**

Con las siguientes sentencias se pueden truncar o redondear números reales con los siguientes códigos AWL:

REAL a DINT → **TRUNC**

REAL a DINT → **RND**

Por último indicar otras sentencias de conversión para RND que resultan útiles en la programación:

```
RND+    // redondea al DINT mayor siguiente
RND-    // redondea al DINT menor siguiente
```

Como ejemplo de operación de conversión supóngase que se quiere convertir el número entero almacenado en la marca MW120 en real, almacenándolo en la marca MW180. El código de programa sería:

```
L       MW      120
ITD
DTR
T       MD      240
```

Las operaciones lógicas con palabras en AWL se ejecutan bit a bit en cada una de las palabras y obedecen al código:

```
UW      // AND lógica
OW      // OR lógica
XOW     // XOR lógica
```

Ejemplo de operación lógica con palabras en AWL:

```
L       EW      0       // carga la palabra de la entrada del byte 0
L       W#16#12ED       // carga el número hexadecimal 12ED
UW/OW/XOW               // ejecuta la operación logíca bit a bit
T       MW      10      // transfiere el valor lógico de la operación
                           a la dirección de memoria del byte 10
```

Representación de los números reales:

32 bits

1º: **signo**
1 bit

$$num.real = S \cdot (1.f) \cdot 2^{(e-127)}$$

3º: mantisa
23 bits
desde 2^{-1} hasta 2^{-23}

2º: **exponente binario**
8 bits

Las funciones matemáticas más utilizadas en programación son:

```
SQR     // cuadrado de un número real
SQRT    // raíz cuadrada de un número real

EXP     // función exponencial de base e (2.718283)
LN      // logaritmo neperiano
```

Ejemplo de una aplicación de función matemática en AWL:

```
L     5.0          // carga el número real 5.0 en ACU1
SQRT               // calcula la raíz cuadrada de ACU1
T     MD    30     // transfiere el valor del cálculo a MD 30
```

También se utilizan las funciones trigonométricas utilizando como unidad angular el radián (SI):

SIN	// seno de un ángulo en rad
COS	// coseno de un ángulo
TAN	// tangente de un ángulo

y sus correspondientes funciones inversas:

ASIN	// arcoseno de un ángulo en rad
ACOS	// arcocoseno de un ángulo
ATAN	// arcotangente de un ángulo

Ejemplo de una aplicación de función trigonométrica en AWL:

```
L     5.0          // carga el ángulo 0.5 en ACU1
SIN                // calcula el seno de 0.5
T     MD    30     // transfiere el valor del cálculo a MD 30
```

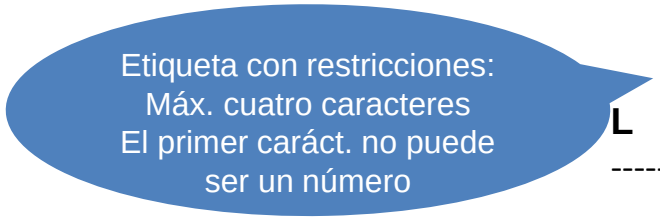
Los saltos son muy importantes en programación porque dotan de dependencia a operaciones concretas.

Por ejemplo, con las líneas de programa AWL:

```
U     E     0.0
L     MW    100
```

en ACU1 se cargará el valor de MW 100 independiente mente del valor del bit de entrada E 0.0.

Utilizando el ejemplo anterior, con un salto se podría condicionar la carga de MW 100:




```
----------------                    // código de programa
U           E       0.0             // RLO
INST        ETIQ
----------------                    // código de programa
----------------
ETIQ:
L   MW      100     // sentencias condicionadas
----------------
```

Los saltos son muy importantes en programación porque dotan de dependencia a operaciones concretas.

Los saltos pueden orientarse según una secuencia de programación de ***arriba*** hacia ***abajo***. La orientación contraria no es deseable ya que en el programa se podrían generar bucles.

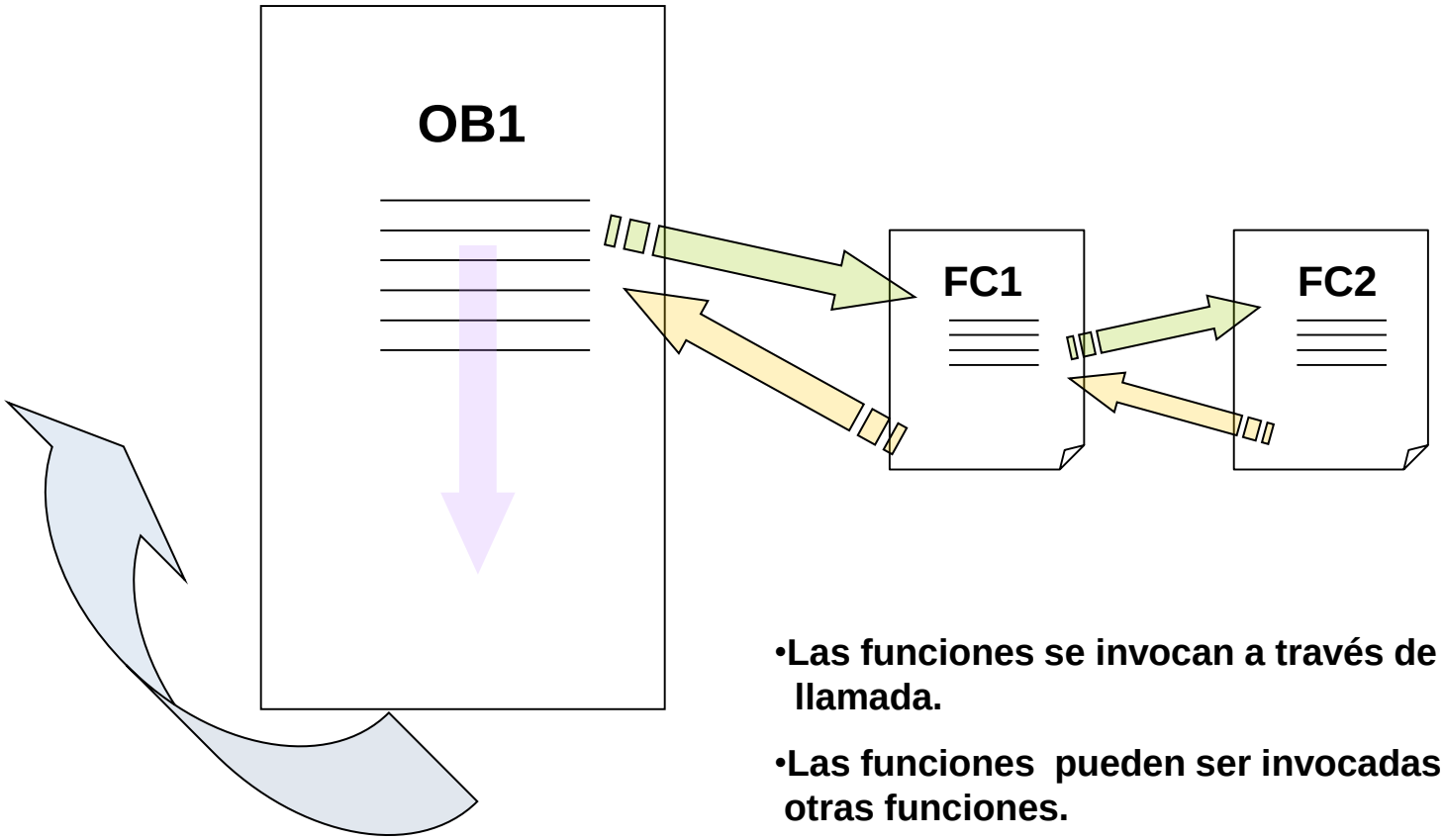
Existe un tipo de salto especial denominado **SALTO DE BUCLE**. La característica más importante de este tipo de salto es que en la secuencia de un programa, el salto se repetirá tantas veces como indique el acumulador ACU1.

```
ETIQ:
----------------    // sentencias condicionadas
----------------
LOOP ETIQ
```

Cada vez que se ejecuta LOOP se resta 1 al valor del acumulador ACU1. Después se compara el valor del acumulador con 0.

FUNCIONES Y BLOQUES

Puede definirse como un bloque FC como un conjunto de sentencias de programa que pueden ser llamadas desde OB1 u otro bloque FC de una forma repetitiva.



- **Las funciones se invocan a través de una llamada.**
- **Las funciones pueden ser invocadas desde otras funciones.**
- **Las llamadas se ejecutan a través de sentencias.**

Las tres sentencias que se utilizan a la hora de ejecutar una llamada a función son:

- **CALL FC1** (incondicional)
- **CC "FUNCION"** (condicional según expresión lógica)
- **UC "FUNCION"** (incondicional)

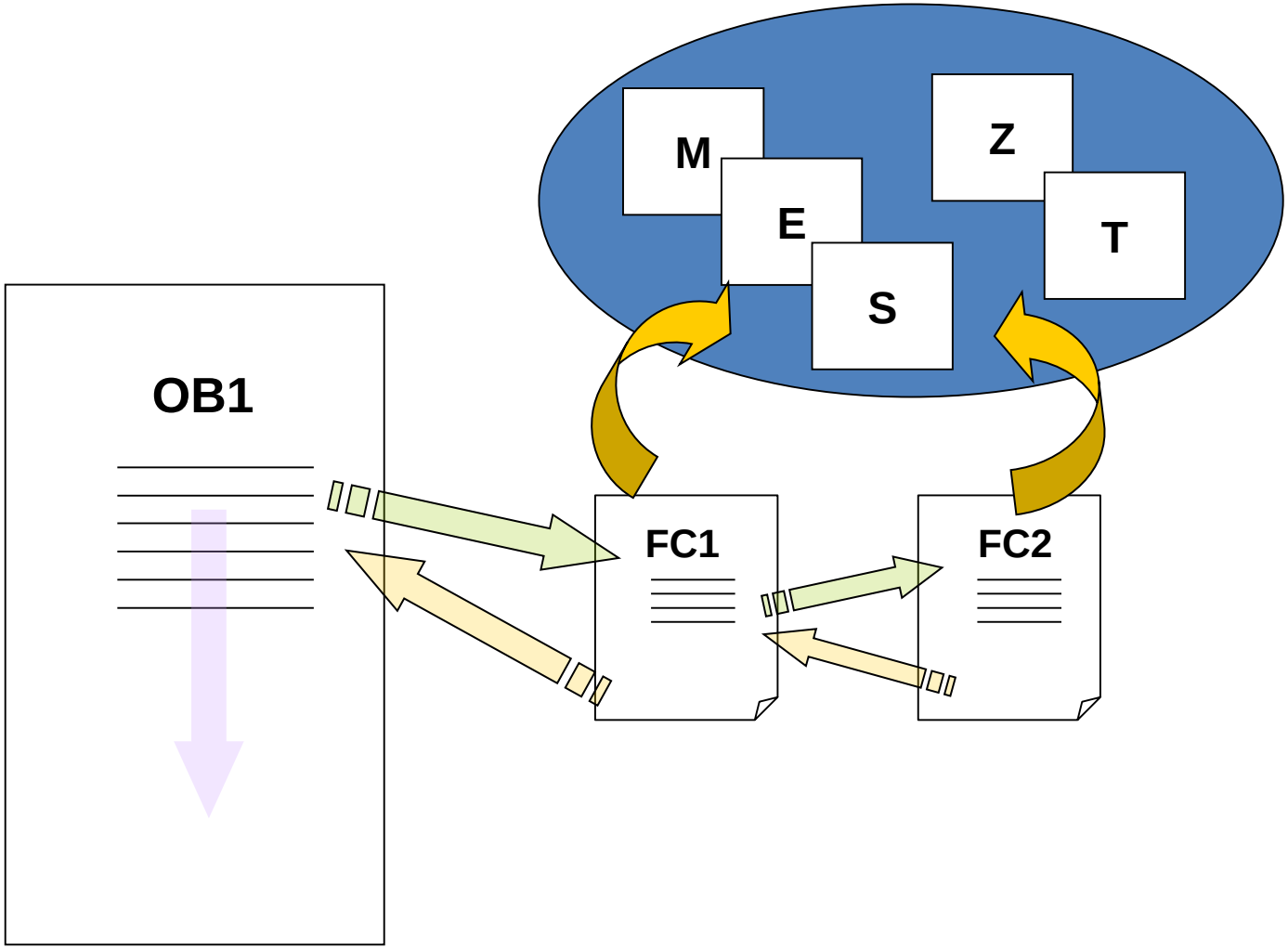
La diferencia entre CALL y UC es que CALL necesita ser introducida con parámetros y UC no necesita parámetros.

Los PLC's tienen limitado el número de funciones que pueden ser invocadas a la hora de ejecutar un programa, si bien, este número es muy elevado.

La ejecución de todas las FC's invocadas desde OB1 y las distintas funciones ha de ocupar un tiempo menor que el de **vigilancia**.

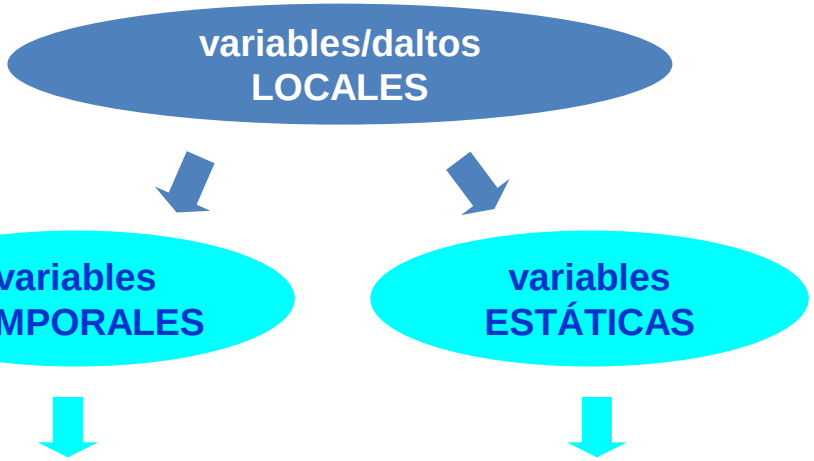
Al igual que en otros tipos de programación, para los PLC's es importante dotar a las FC's de independencia en cuanto al uso de variables.

En las FC's se pueden utilizar las mismas zonas de memoria que en el bloque OB1.



Tipos de variables

- **PAE / PAA**
- **E / A**
- **M / T / Z**
- **Áreas de DB's**



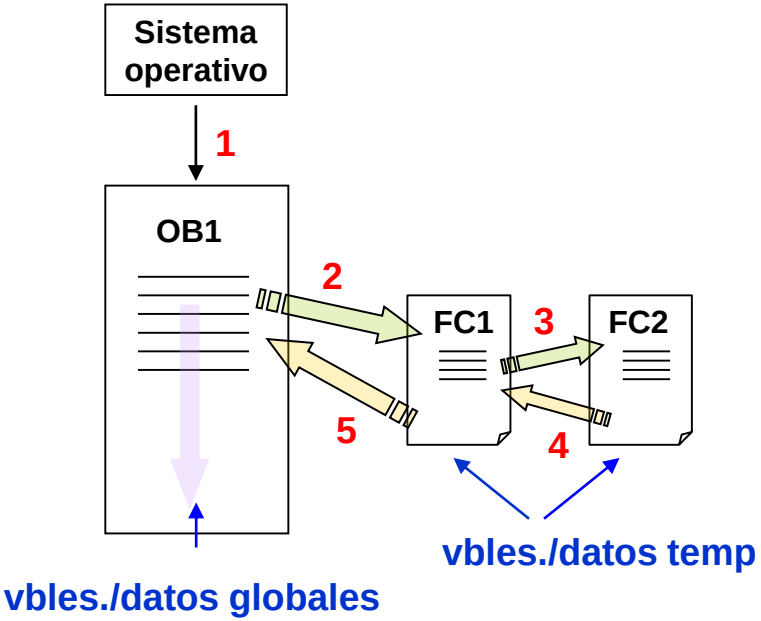
- **Se borran después de la ejecución de un bloque.**
- **Se almacenan de forma temporal en la *L. STACK*.**
- **Se utilizan en: OB's, FC's y FB's.**

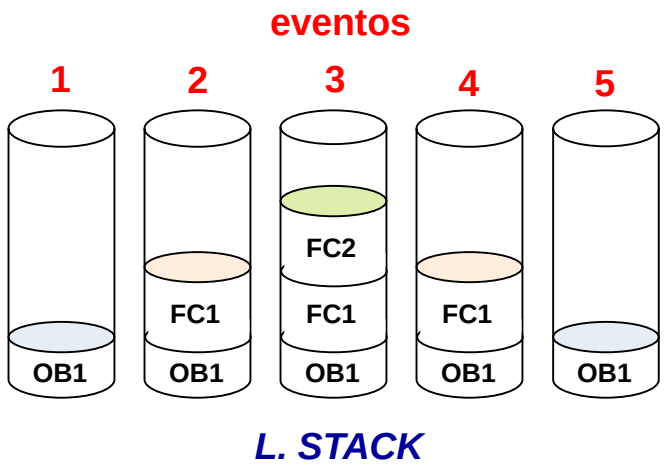
- **Se mantienen después de la ejecución de un bloque.**
- **Se almacenan de forma permanente en DB's.**
- **Se utilizan unicamente en FB's.**

Es importante el concepto de PILA DE DATOS LOCALES (***L. STACK***)

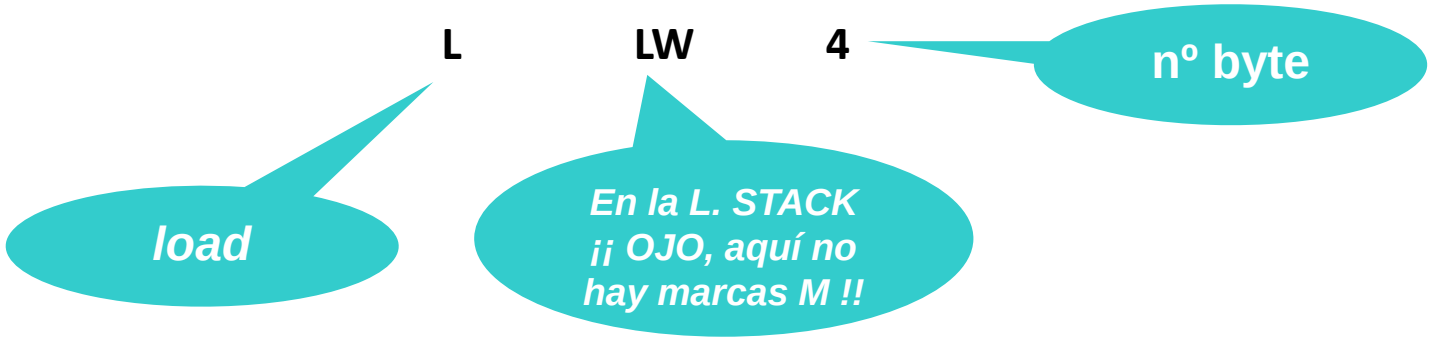
La *L.STACK* es una zona de memoria separada de la CPU que contiene las variables temporales de los bloques. Por lo tanto esta zona de memoria ha de ser gestionada por el sistema operativo.

El objetivo de la *L.STACK* es el de reservar una zona de memoria disponible para el uso de una función (ej.: FC1) de modo que dicha función pueda ser utilizada por otras OB1 sin que se generen conflictos.

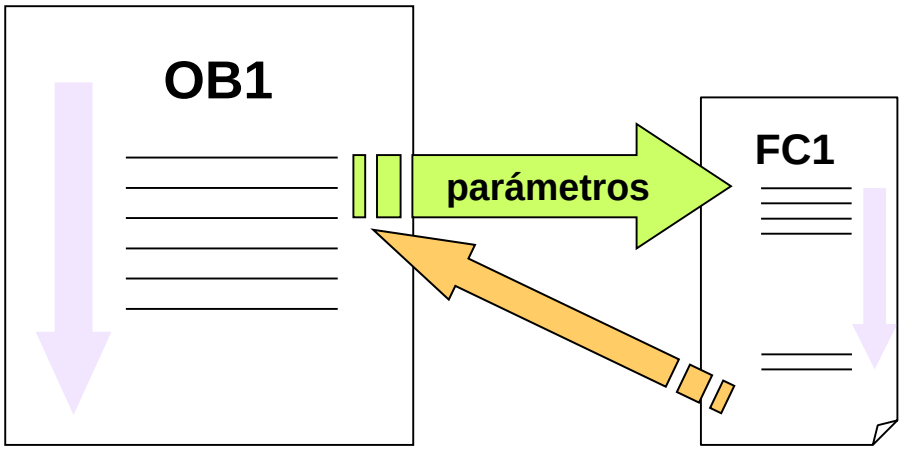




Para el direccionamiento de las zonas de memoria de una función (**FC**) se utiliza la marca **L** (*load*):



A las funciones (**FC's**) se les pueden pasar parámetros cuando se ejecuta una llamada:



Los parámetros se pasan a través de ***interfaces***. Los interfaces incorporan los siguientes tipos de parámetros:

interface {
- **IN (parámetros de entrada)**
- **OUT (parámetros de salida)**
- **IN.OUT (parámetros de entrada/salida)**
- **TEMP (van a la L. STACK)**
- **…**

Para insertar una función en OB1 se ejecutan los siguientes pasos:

1. Abrir el menú contextual en la carpeta BLOQUES e insertar un bloque del tipo FUNCIÓN
2. Dar un nombre a la función (ej.: FC10) y elegir el lenguaje en el que se va a programar.
3. Programar la interface teniendo mucho cuidado con las variables de entrada, salida y temporales, así como de su tipo (byte, int, etc.).

Cuando se abre una función (ejem. FC10) el editor para la función es el mismo que el utilizado en OB1 (no existe cambio alguno), la diferencia radica en el ***desplegable*** que aparece en la **ventana superior** en el que aparecen: IN, OUT, …, es decir en la **interface**.

Una vez declaradas las variables en la interface, se programa la función y se graba en disco.

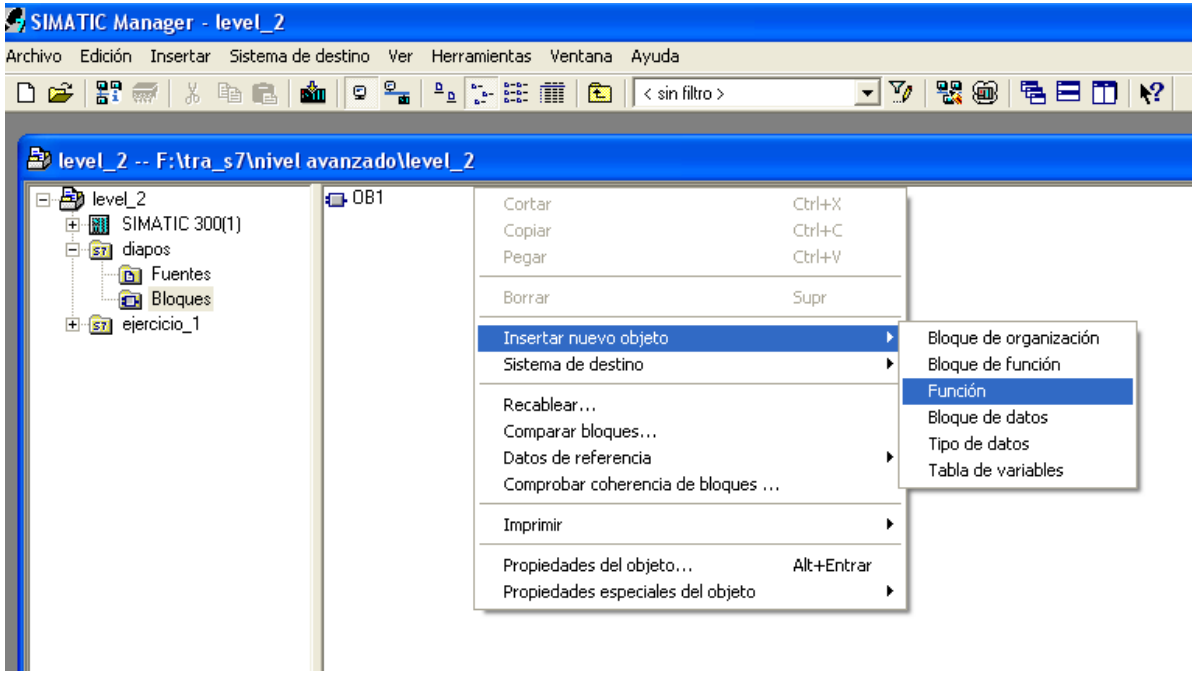
Desde OB1, dependiendo del tipo de llamada (CALL), al insertar el nombre de la función aparecerán automáticamente las variables declaradas en su interface y a dichas variables se les asignará un valor o parámetro, o también, el nombre de una dirección de memoria (esto es frecuente cuando se realizan operaciones y lo que se quiere obtener es un resultado introduciéndolo en una marca).

Se graba en disco OB1 y se cargan desde la ventana de bloques: **primero, la FUNCIÓN (FC10) y, después el bloque de organización OB1**. También, se pueden seleccionar juntos y se cargan a la vez.

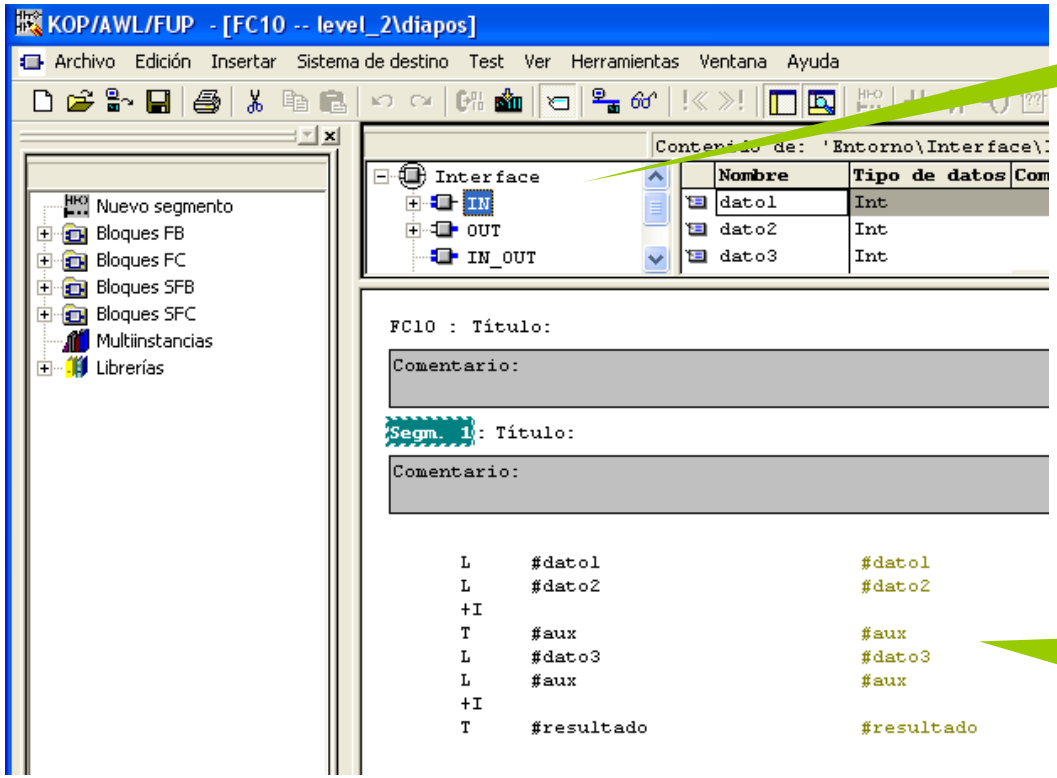
A partir de este momento ya se puede ejecutar el programa desde **OB1**.

EJEMPLO: Elaborar un programa en AWL en el que una función realice la suma de tres números enteros.

Primero, se crea la función con el menú contextual, que se denominará FC10:



Segundo, se abre la función con doble 'clic', se parametriza el interface y se insertan las sentencias de programa:

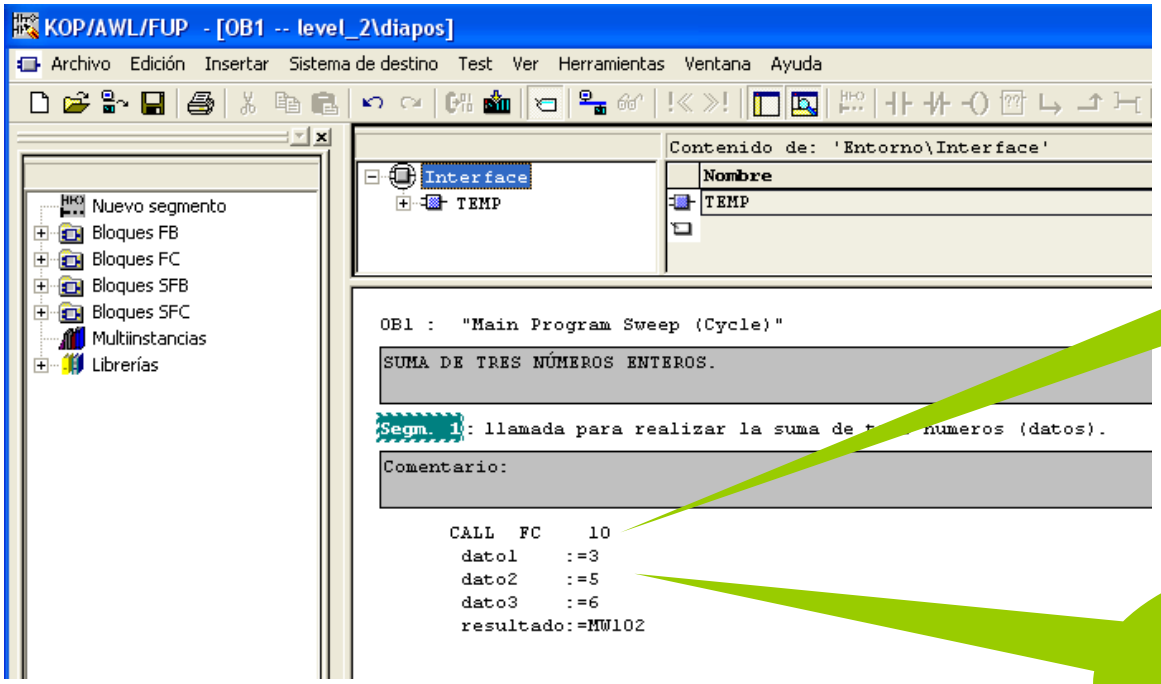


interface

Sentencias para sumar tres números

Tercero, se graba la función.

Cuarto, se escriben las sentencias de programa en OB1 invocando mediante una llamada de tipo CALL a la función FC10:

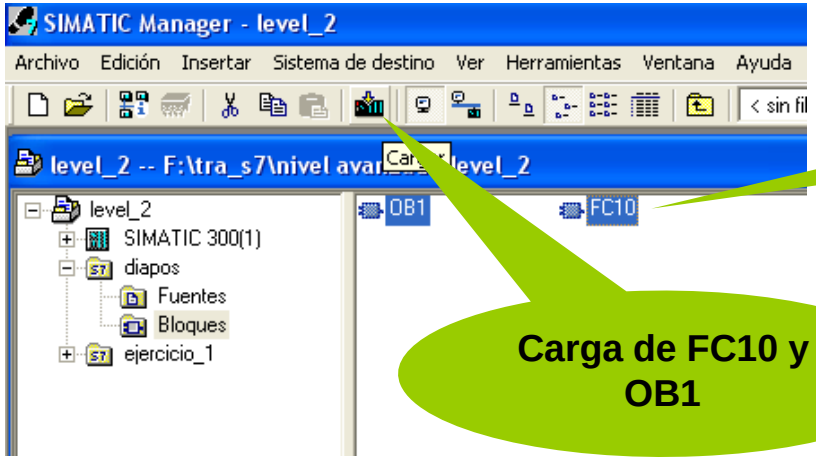


Llamada a FC10

Paso de parámetros para las variables de entrada y de salida.

Quinto, se graba OB1.

Sexto, se seleccionan FC10 y OB1 y se cargan en ese orden, o las dos a la vez:

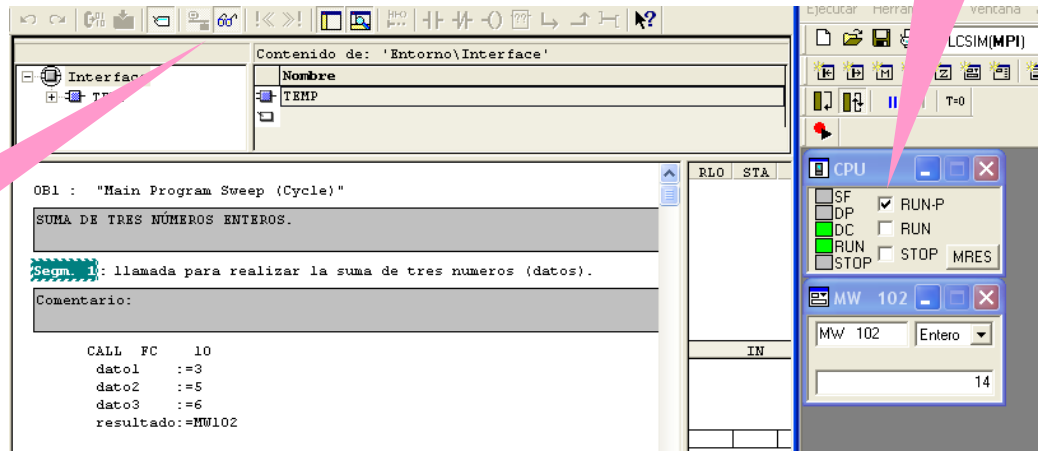


Selección de FC10 y OB1

Carga de FC10 y OB1

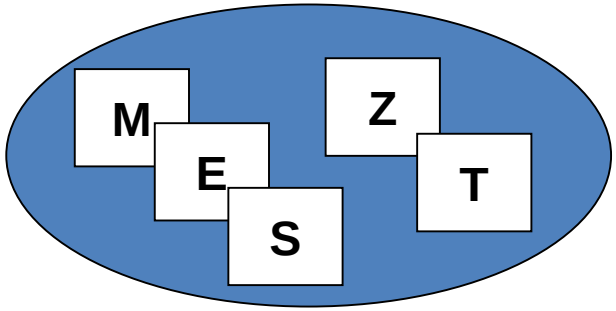
Ejecución del simulador

Séptimo, se ejecuta OB1:

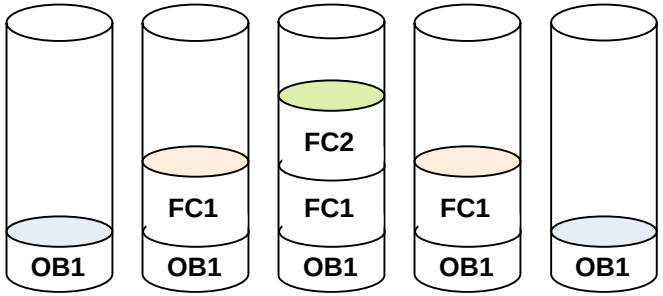


Ejecución de OB1

Hasta ahora se han tratado dos zonas de memoria que utizan las PLC's:



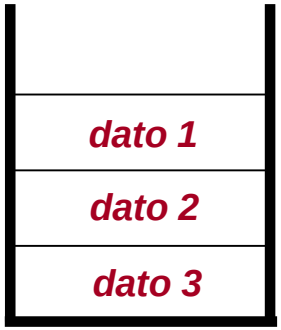
memoria FIJA



L-STACK

memoria DINÁMICA

Existe otra zona: ***LOS BLOQUES DE DATOS***

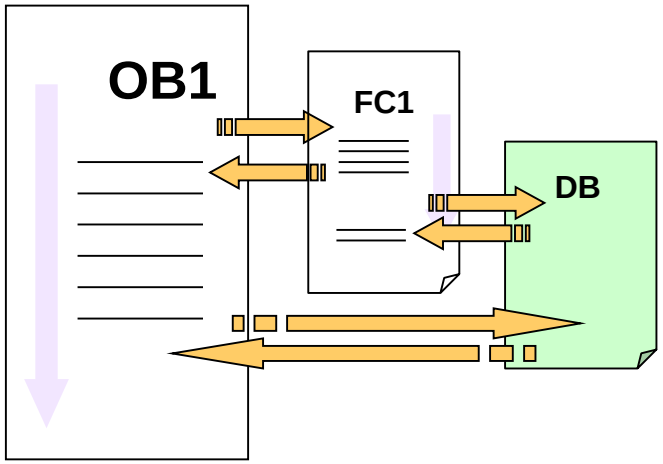


memoria DE USUARIO

El bloque de datos (DB), una vez creado, introduce los datos en la memoria.

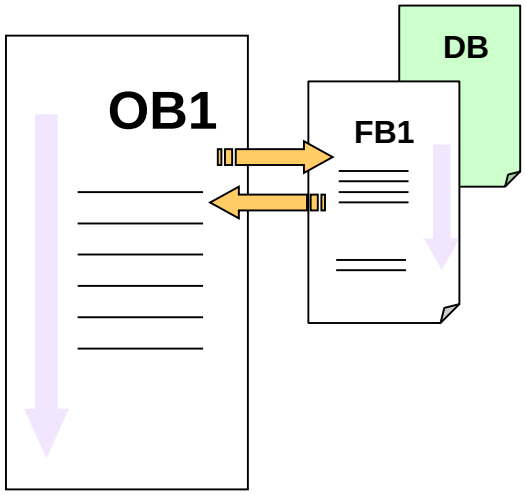
Existen dos tipos de bloques de datos:

DB's GLOBALES



- Intercambian y pasan datos a OB1 y FC's.

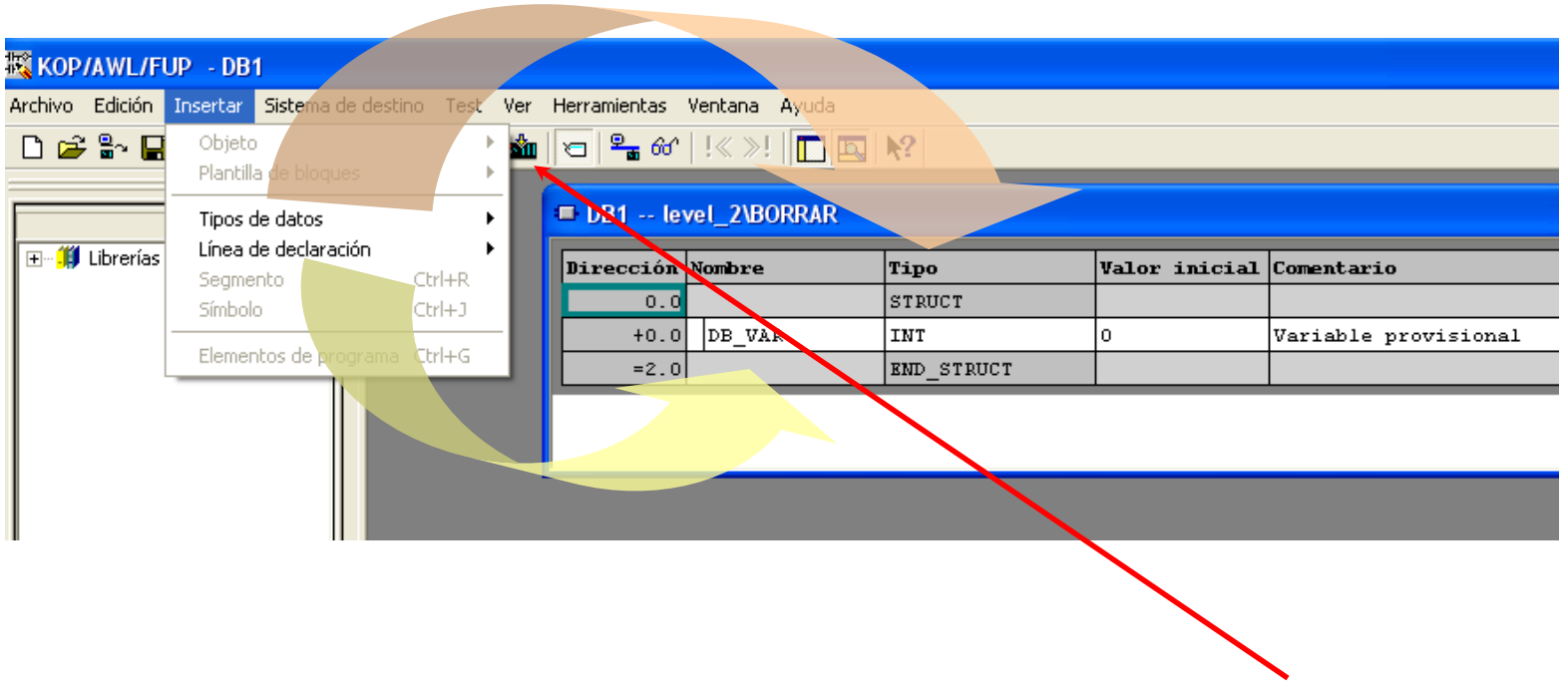
DB's DE INSTANCIA



- Van asociados a un BLOQUE DE FUNCIÓN (FB) (¡¡OJO, NO A UNA FUNCIÓN (FC) !

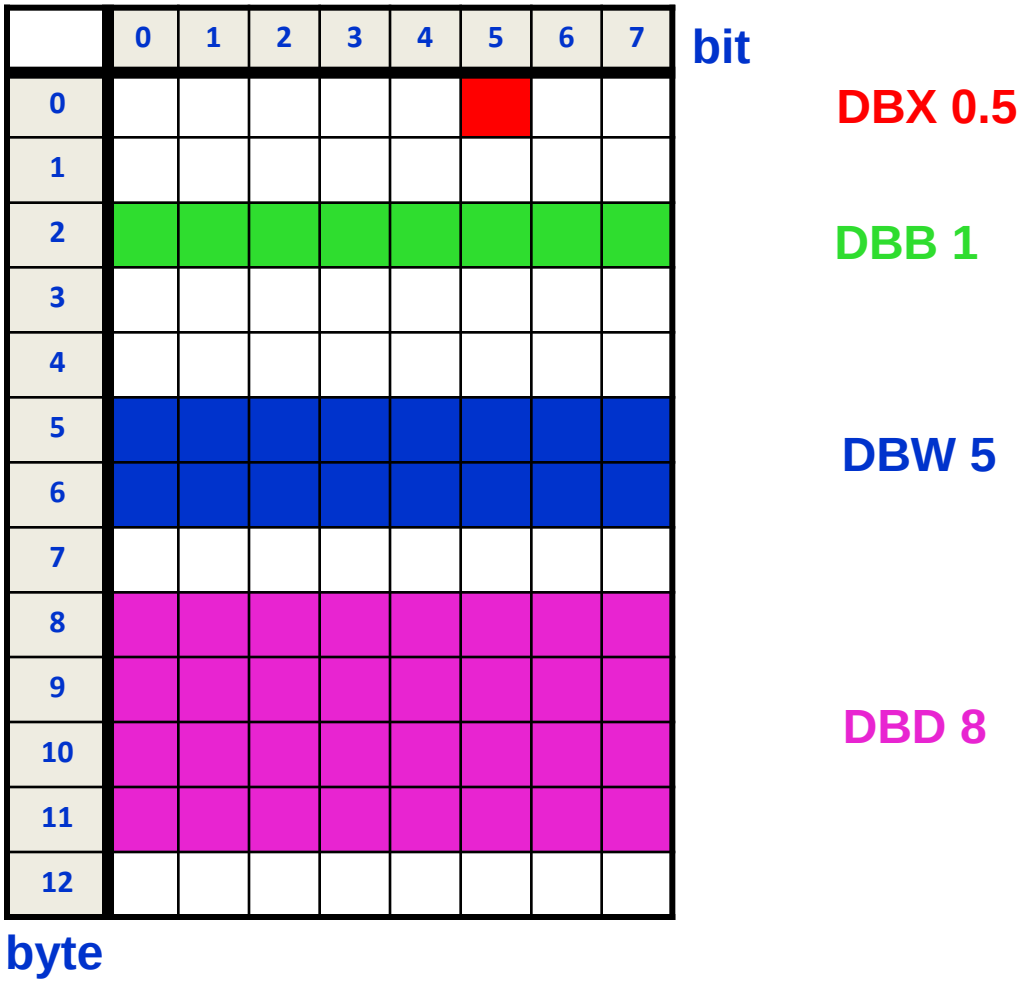
Dentro del editor de Bloques se dispone de dos vistas:

- **De declaración**
- **De datos**



¡¡ Una vez cargados los datos hay que cargarlos en el SIMULADOR (o PLC) !!

Antes de manejar los datos (leer, escribir, editar, etc.) es conveniente conocer cual es su direccionamiento en memoria:



Con el mapa de memoria se sabe si hay o no solapamientos y dónde se encuentran. En el mapa de la diapositiva anterior **no existen** solapamientos.

A la hora de leer datos existen tres opciones:

A)

```
AUF     DB1          // abre DB1
L       DBW    10    // carga el dato de los bytes 10 y 11
```

Esta opción es muy requerida cuando se utilizan punteros.

B)

```
L       DB1.DBW      10    // idem. anterior
```

Esta opción es muy utilizada por la sencillez de sintaxis.

C) Utilizando símbolos

```
L     "datos". Dato1     // carga el dato Dato 1 del bloque de
                         // datos DB al que se le ha asignado
                         // un símbolo denominado datos.
```

EJEMPLO: Crear dos ficheros de datos (DB's) con los siguientes elementos:

DB1: (1, 2, 3, 4, 5) pertenecientes a Z

DB2: (1.2, 1.3, 2.5, 5.4, 6.2) pertenecientes a R

Con estos datos se desea elaborar un programa en lenguaje AWL que realice las siguientes operaciones numéricas:

1. ***Sumar los cuatro primeros números de DB1 y el resultado se guarda en DB3.***
2. ***Multiplicar los cuatro primeros números de DB2 y guardar el resultado en la primera posición del bloque DB4.***
3. ***Sumar los dos resultados anteriores sitos en DB3 y DB4 y este resultado almacenarlo en DB5.***

En principio, los datos que puede manejar una PLC puedend dividirse en:

- Datos **SIMPLES (≤32 bits)**
- Datos **COMPUESTOS (>32 bits)**

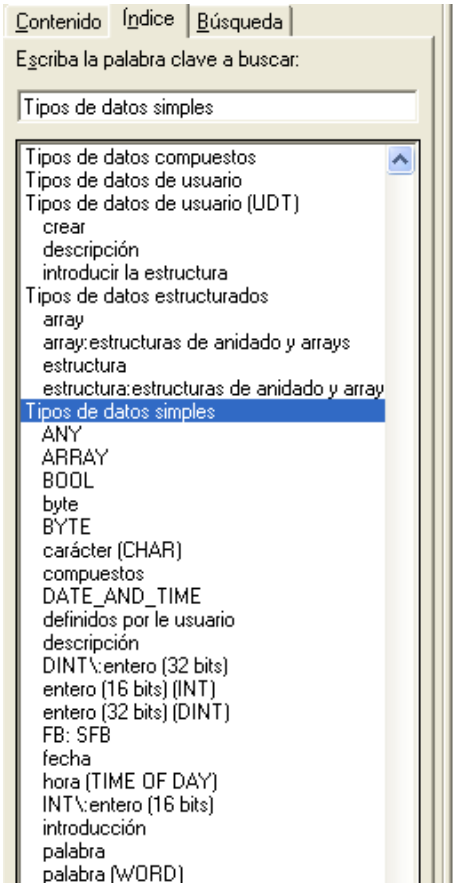
La diferencia entre estos dos tipos de datos es su ocupación en memoria.

DATOS **SIMPLES**

- **BOOL**
- **BYTE**
- **INT, DINT, REAL**
- **S5TIME** (son utilizados por los temporizadores de S7)
- **TIME** (es un dato de tiempo con diferente estructura que los anteriores; los datos TIME se utilizan cuando se trabaja en funciones; su formato es: **T# 0MS**; cuando se incrementa TIME se realiza de ms en ms).

- **DATE** (su formato es, p. ej.: D# 2012-7-14; se incrementa día a día)
- **TIME OF DAY** (su formato es: D# 0:0:0.000)
- **CHAR** (su formato es, p. ej.: 'A' -con comillas simples-)

Las características y representación de estos datos pueden verse con la herramienta de ayuda (***help***) del programa S7.



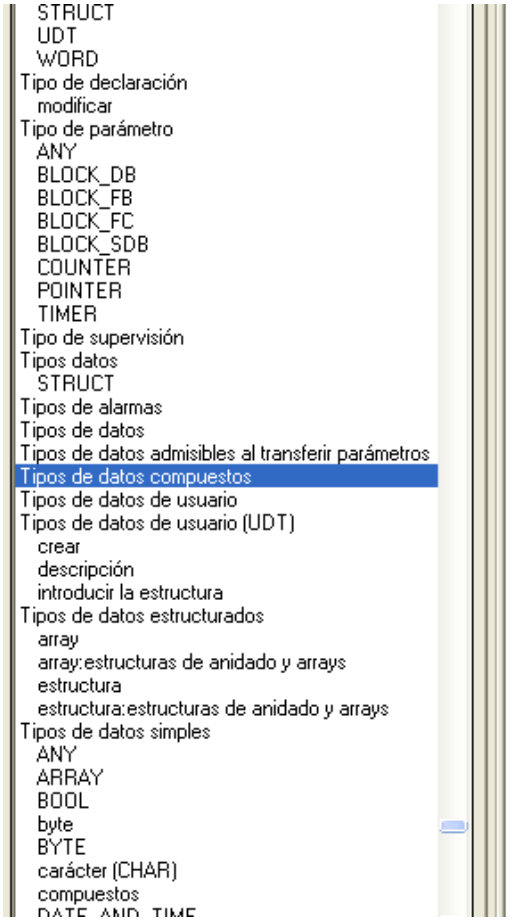
WORD (palabra)	16	Número binario Número hexadecimal BCD Número decimal sin signo	2#0 hasta 2#1111_1111_1111_1111 W#16#0 hasta W#16#FFFF C#0 hasta C#999 B#(0,0) hasta B#(255,255)	L 2#0001_0000_0000_0000 L W#16#1000 L word#16#1000 L C#998 L B#(10,20) L byte#(10,20)
DWORD (palabra doble)	32	Número binario Número hexadecimal Número decimal sin signo	2#0 hasta 2#1111_1111_1111_1111_1111_1111_1111_1111 DW#16#0000_0000 hasta DW#16#FFFF_FFFF B#(0,0,0,0) hasta B#(255,255,255,255)	2#1000_0001_0001_1000_1011_1011_0111_1111 L DW#16#00A2_1234 L dword#16#00A2_1234 L B#(1, 14, 100, 120) L byte#(1,14,100,120)
INT (número entero)	16	Número decimal con signo	-32768 hasta 32767	L 1
DINT (entero de 32 bits)	32	Número decimal con signo	L#-2147483648 hasta L#2147483647	L L#1
REAL (número en coma flotante)	32	IEEE Número en coma flotante	Límite superior: ±3.402823e+38 Límite inferior: ±1.175 495e-38	L 1.234567e+13

DATOS **COMPUESTOS**

- **DATE AND TIME** (ocupan 8 bytes y trabajan con funciones de librería: DT# *.FC.LIBRERIA; p. ej.: DT#1990-1-1-0:0:0.000)
- **ARRAY** (pueden comprender hasta 6 dimensiones; cuando se declara un array, el programa reserva un lugar de memoria para indicar el tipo de datos que se utilizará en el array ("la última casilla");EL ej. de sintaxis: COCHES[0..9] indica la declaración del array COCHES que contendrá 10 elementos); cuando se desea declarar un array bidimensional: MOTOS[0..9, 0..9] que significa que es un array de 10x10 elementos; para inicializarlo: COCHES[0..9]=4(0.0), 6(1.0), significa que los cuatro primeros elementos tienen el valor de 0.0 y los seis últimos 1.0).

  REAL
- **STRUCT** (son arrays que contienen diferentes tipos de datos; al igual que los arrays, las estructuras permiten introducir los tipos de datos en casillas de memoria reservadas)
- **STRING** (son cadenas de caracteres p. ej. STRING[254]; por defecto pueden albergar hasta 254 caracteres, pero el valor entre corchetes puede ser cualquiera inferior a ese)
- **UDT**
- **FB, SFB**

Del mismo modo que los datos SIMPLES, Las características y representación de este tipo de datos pueden verse con la herramienta de ayuda (***help***) del programa S7.



Tipo de datos	Descripción
DATE_AND_TIME DT	Define un área de 64 bits (8 bytes). Este tipo de datos memoriza en formato decimal codificado en binario:
STRING	Define un grupo de un máximo de 254 caracteres (tipo de datos CHAR). El área estándar reservada para una cadena de caracteres consta de 256 bytes. Este es el espacio requerido para memorizar 254 caracteres y un encabezamiento de 2 bytes. La capacidad de memoria requerida para una cadena de caracteres se puede reducir definiendo también la cantidad de caracteres a memorizar en dicha cadena (p.ej.: string[9] 'Siemens').
ARRAY	Define un agrupamiento multidimensional, similar a una matriz, de un tipo de datos (simple o compuesto). Por ejemplo: "ARRAY [1..2,1..3] OF INT" define un campo en formato de 2 x 3 números enteros. A los datos memorizados en un campo se accede a través del índice ("[2,2]"). En un campo se pueden definir hasta un máximo de 6 dimensiones. El índice puede ser un número entero discrecional (-32768 a 32767).
STRUCT	Define un agrupamiento de tipos de datos combinados discrecionalmente. Por ejemplo, se puede definir un campo compuesto de estructuras o una estructura compuesta de estructuras y campos.
UDT	Permite estructurar grandes cantidades de datos, simplificando así la entrada de tipos de datos al crear bloques de datos o al declarar las variables en la declaración correspondiente. STEP 7 permite combinar tipos de datos compuestos y simples, creando así un tipo de datos propio "de usuario" (UDT). UDTs tienen un nombre propio y, por consiguiente, pueden utilizarse varias veces.
FB, SFB	Determinan la estructura del bloque de datos de instancia asignado y permiten la transferencia de datos de instancia para varias llamadas de FB en un DB de instancia.

Para acceder a las variables que contengan estos tipos de datos hay que utilizar diferentes direccionamientos:

- **carga de un array:**

L "Datos". Dato5

- **carga del elemento [2, 2] de un array:**

L "Datos". Dato10[2, 2]

- **carga del elemento REAL de una estructura:**

L "Datos". Dato11.re

- **carga del primer carácter de una cadena de caracteres:**

L "Datos". rrr[1]

Las librerías son elementos de programación que se utilizan para gurardar componentes de programa reutilizables. Los componentes de programas pueden copiarse de los proyectos existentes a una librería o pueden generarse directamente en la librería, independientemente de los proyectos.

Las librerías no se pueden transferir directamente a la CPU y no pueden ser observadas.

Los bloques que han de usarse una vez tras otra pueden guardarse en librerías. Desde allí pueden copiarse al programa de usuario correspondiente y ser llamados por otros bloques.

Configuración de librerías

- **Una librería puede contener diferentes carpetas de programa.**

  Cada carpeta de programa contiene:
  - Las carpetas: *Bloques*, *Fuentes* y *Símbolos*.
  - La carpeta Esquemas (*Charts*) sólo para el software opcional S7-CFC.

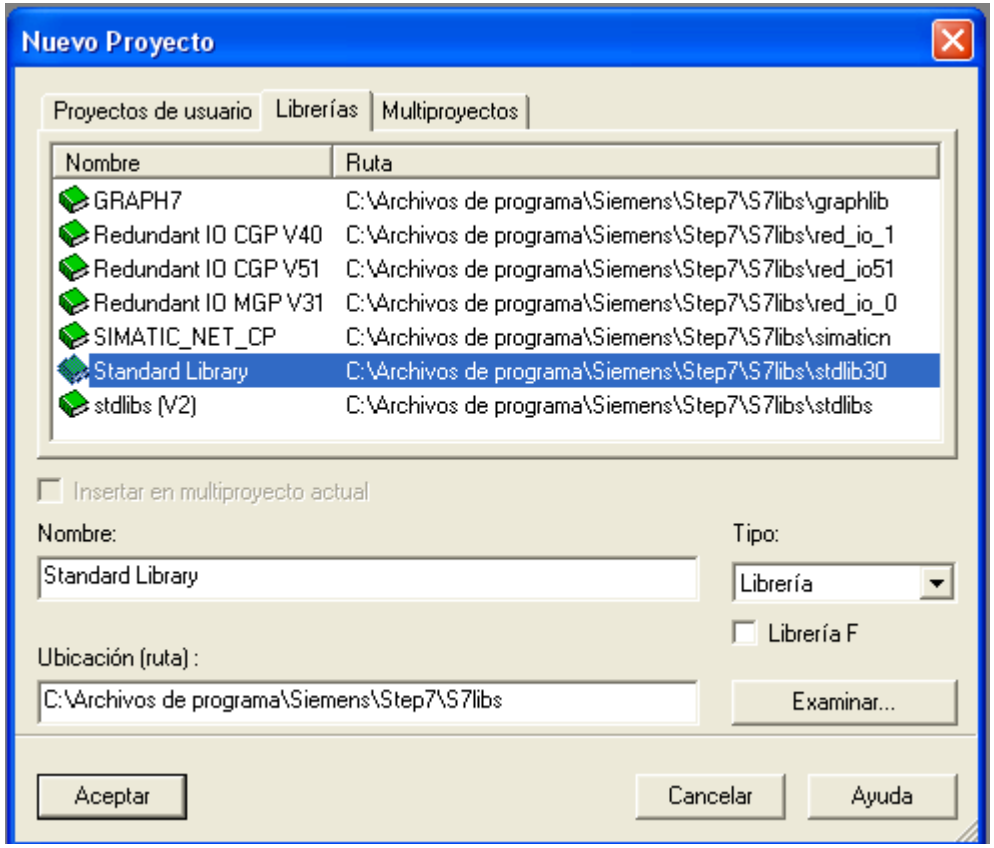
  . La carpeta bloques contiene los bloques de programa que se pueden cargar en la CPU.
  . Las tablas de variables (VAT's) y los tipos de datos definidos por el usuario y contenidos en ellas NO son cargados en la CPU.

- **Una librería NO puede contener ningún hardware.**

- **Una librería NO puede contener programas S7.**

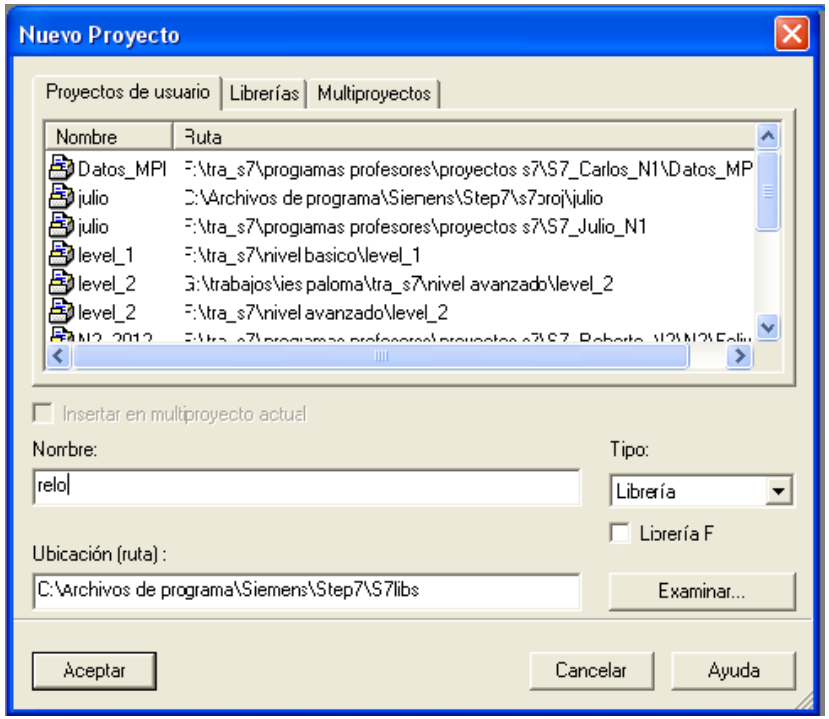
En SIMATIC existe una librería ya instalada por defecto o estándar:

standar library

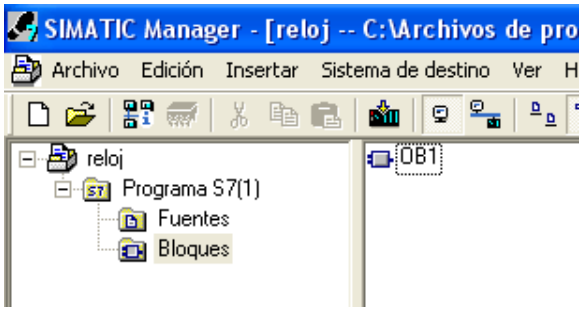


Para crear una librería se siguen los pasos que se indican a continuación:

- ***ARCHIVO***
  - ***NUEVO***
    - ***Proyecto de librería***
      - ***Introducir nombre***

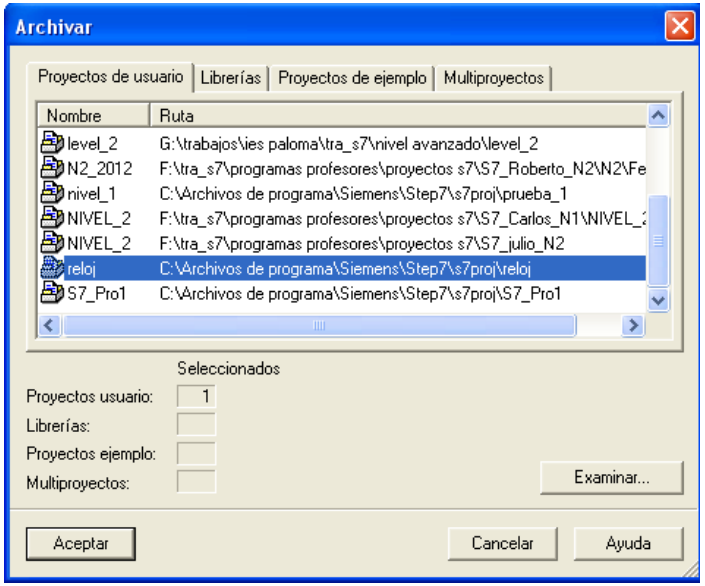


Una vez creada la librería (p. ej.:reloj) ya se pueden crear en ella **carpetas** y **bloques**:



Introducidos las carpetas y bloques, con click sobre la librería ya se puede guardar todo en la librería en el formato adecuado mediante:

- ***ARCHIVO***
- ***Archivar***



Ejemplo de utilización de librerías:

Mediante un pulsador (E 124.0) se actualizará la hora que marca la CPU mediante una variable introducida en un bloque de datos DB1.

Para la realización de este programa se seguiran los pasos y consejos que se indican a continuación:

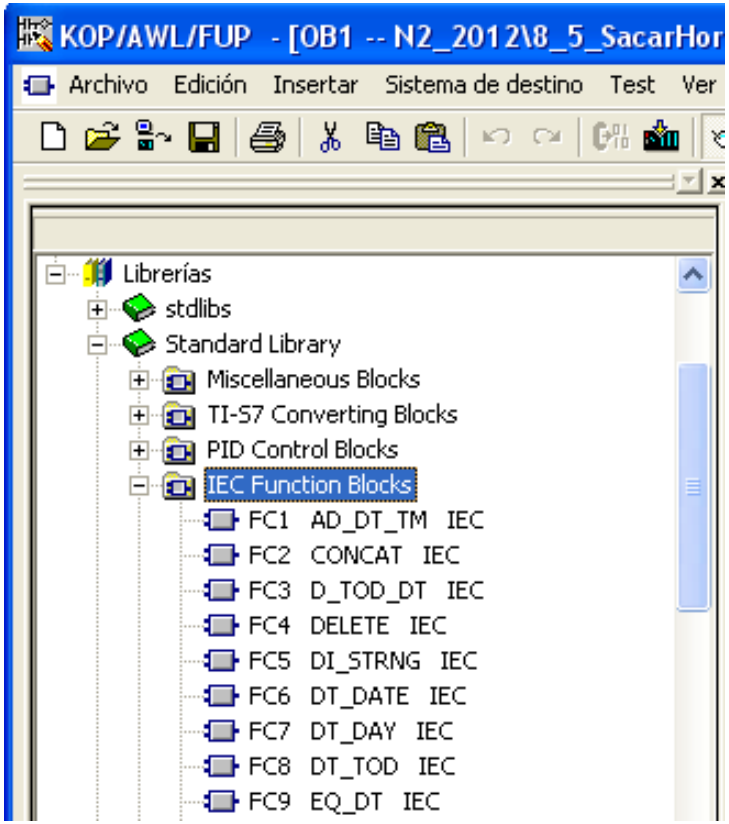
- Introducir en los bloques de programa la función protegida de librería SFC1: ***"READ_CLK"***

- SFC1 saca la hora de la PLC en formato compuesto (CDT) por lo que hay que buscar otra función que sepa leerlo para introducirlo en DB1 en formato de variable DATA AND TIME.

- Para la conversión de estos formatos existe la función FC8. Esta función toma el dato en formato DT#... y lo pasa a formato tipo TOD.

- La función FC8 se llama: ***"DT_TOD"***.

- Estas dos librerías, SFC1 y FC8, se encuentran en la ruta:



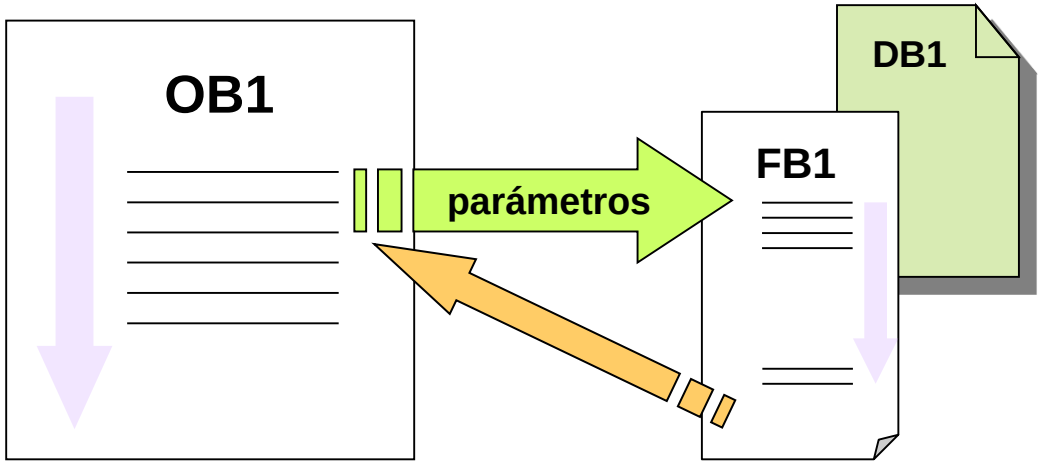
- El programa en OB1 podría ser:

```
OB1 :   "Main Program Sweep (Cycle)"
Comentario:

Segm. 1: Título:
Comentario:

      U     E    124.0
      SPB   hora
      BEA

hora: CALL   "READ_CLK"                  SFC1              -- Read System Clock
       RET_VAL:=MW0
       CDT    :="Datos".Hora_dia_PLC     P#DB1.DBX0.0      -- Variable provisional

      CALL   "DT_TOD"                    FC8               -- DT to TOD
       IN     :="Datos".Hora_dia_PLC     P#DB1.DBX0.0      -- Variable provisional
       RET_VAL:="Datos".Hora_PLC         DB1.DBD8
```

- Con DB1:

Dirección	Nombre	Tipo	Valor inicial	Comentario
0.0		STRUCT		
+0.0	Hora_dia_PLC	DATE_AND_TIME	DT#90-1-1-0:0:0.	Variable provisional
+8.0	Hora_PLC	TIME_OF_DAY	TOD#0:0:0.000	
=12.0		END_STRUCT		

En principio puede decirse que los Bloques de función (FB´s) son semejantes a las funciones (FC's), con la diferencia de que los primeros incorporan un soporte de datos locales denominado **bloque de datos (DB) de instancia**.



Cuando se llama a un bloque de función (FB) hay que especificar, además, el número del bloque de datos (DB) de instancia, el cual se abre automáticamente.

En los DB's de las FB's se encuentran **constantes** y **variables estáticas**.

Las variables estáticas sólo se pueden utilizar en el FB en cuya tabla de declaración estén así declaradas. Cuando se sale del bloque de función, el valor de las variables estáticas se mantiene.

Las funciones de bloque **pueden** o **no** tener parámetros (no es necesario introducir valores en los parámetros IN-OUT).

Cuando se llama a un FB los valores del parámetro real se almacenan en el bloque de datos de instancia (DB).

Si no se asignan parámetros reales a los parámetros formales en una llamada a un bloque, se usará el último valor almacenado en el DB de instancia para ese parámetro en la ejecución del programa.

Para cada llamada a un FB se pueden especificar diferentes parámetros reales.

Cuando se sale de un FB los datos del DB asociado se conservan.

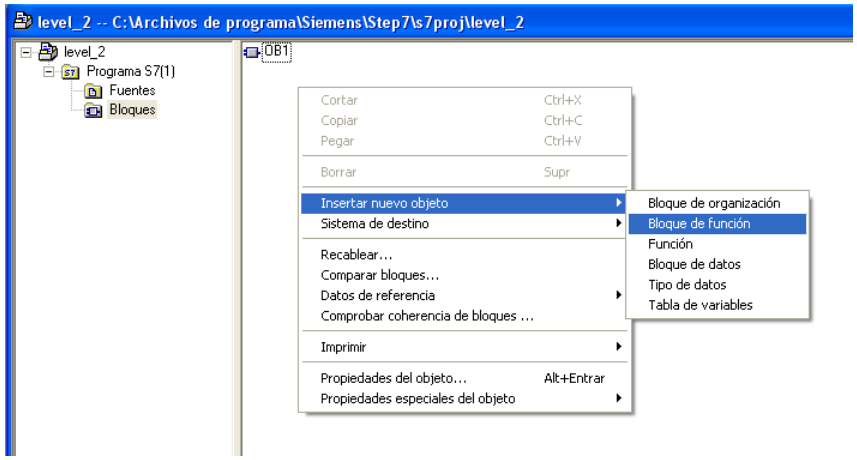
La llamada a una FB se ejecuta mediante la sentencia:

CALL FB1, DB1

La estructura de un DB puede calificarse de "rígida" y existen dos opciones para crear los bloques DB's:

- Desde la carpeta ***BLOQUES*** con ***INSERTAR***
- Desde la propia llamada

Con la primera opción:



Con la segunda opción (p. ejemplo):

CALL FB32, DB40

Aquí el bloque de datos de instancia DB40 es asociado a la función FB32.

Los parámetros que incorpora la interfase del bloque de función son:

- **IN**
- **OUT**
- **IN OUT**
- **STAT** **(es nuevo respecto a los otros DB's y mantienen su valor cuando el programa ejecuta la FB).**
- **TEMP** **(se crean en la *LSTACK*)**

TRATAMIENTO ANALÓGICO

TRATAMIENTO ANALÓGICO

La justificación del tratamiento de señales analógicas con PLC’s es la gran cantidad de instrumentos industriales que trabajan con señales continuas en el tiempo.

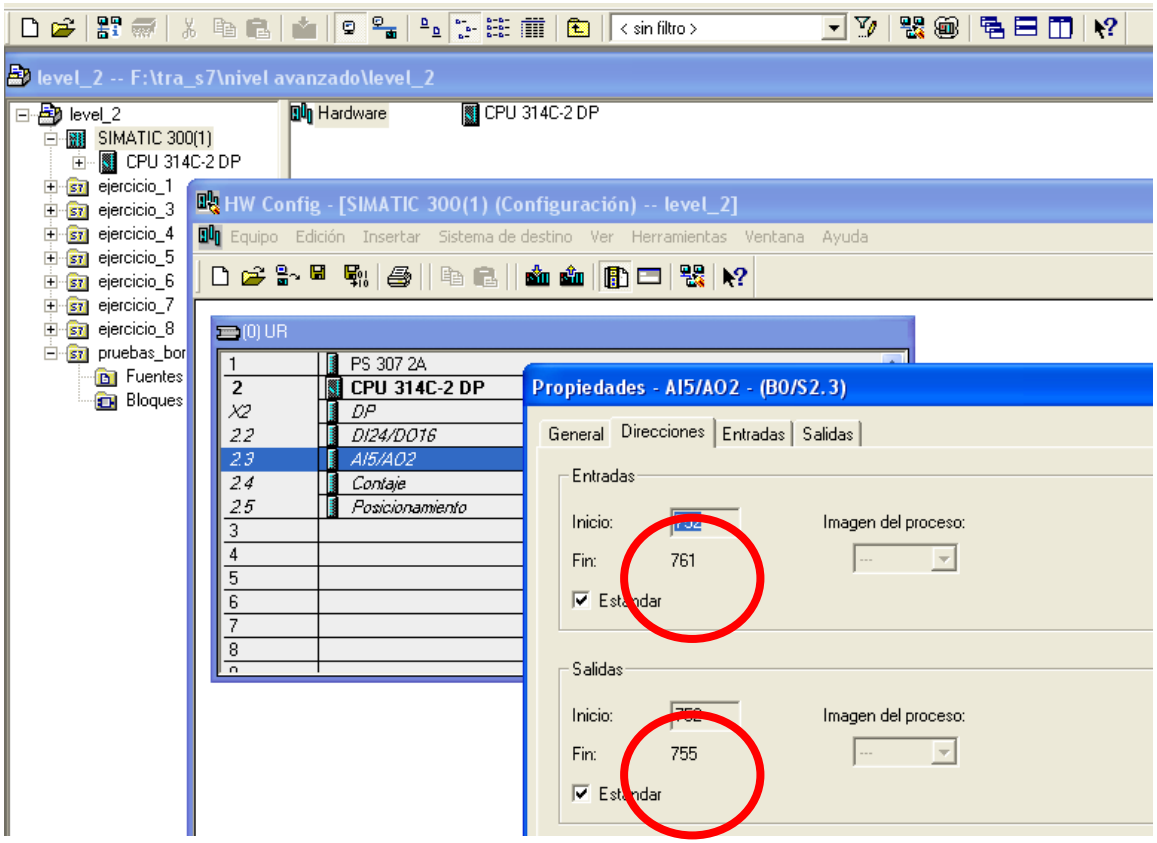
Generalmente, emiten tensiones comprendidas entre entre -10 V y +10 V, y hasta 20 mA de corriente.

Para poder trabajar con señales analógicas, los autómatas (PLC’s) han de incorporar módulos de entrada/salida (E/S) analógicos.

La justificación del tratamiento de señales analógicas con PLC’s es la gran cantidad de instrumentos industriales que trabajan con señales continuas en el tiempo.

TRATAMIENTO ANALÓGICO

El primer paso a seguir en el tratamiento analógico de señales es configurar la tarjeta (hardware). Para ello hay que ver las direcciones de E/S



acceso a la periferia

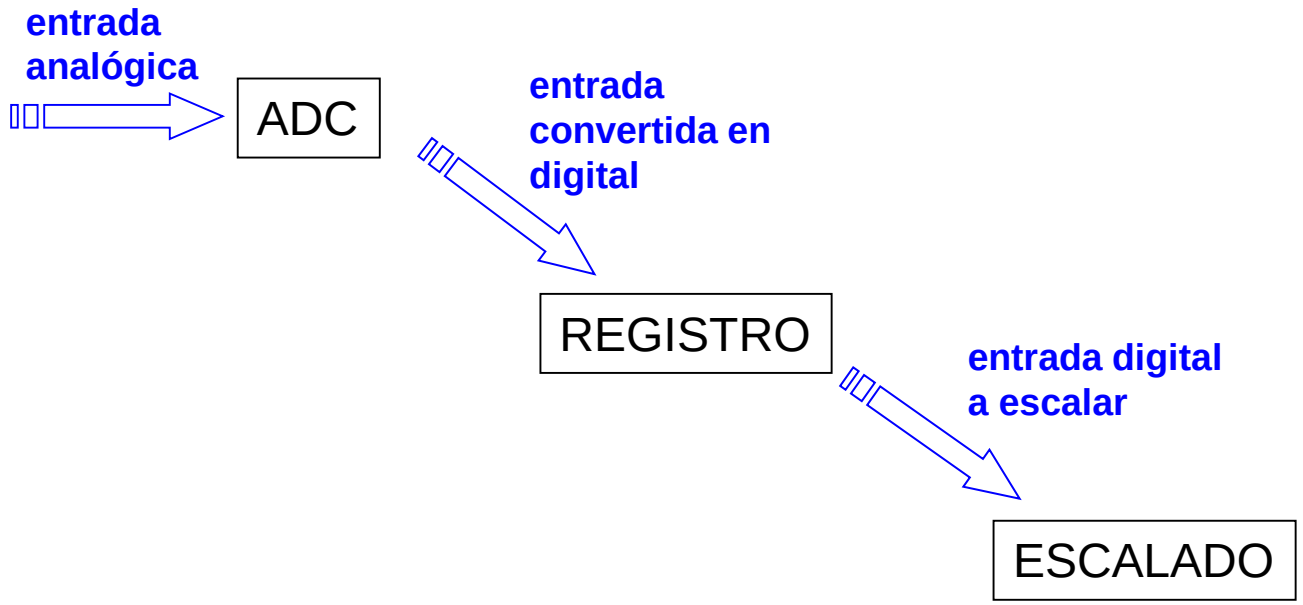
PEW 752 - 761

PAW 752 - 761

En la tarjeta de hardware se pueden modificar las tensiones y corrientes para poder adaptarla a la señal de los sensores.

TRATAMIENTO ANALÓGICO

El segundo paso consiste en manejar los valores analógicos mediante el programa (software). Para ello hay que completar las siguientes etapas:



ESCALADO (codificación en decimal):

0 mA → 0 decimal
20 mA → 27648 decimal

TRATAMIENTO ANALÓGICO

Para direccionar las señales analógicas se utiliza la línea de programa (p. ej.):

```
L     PEW   754
T     PAW   754
```

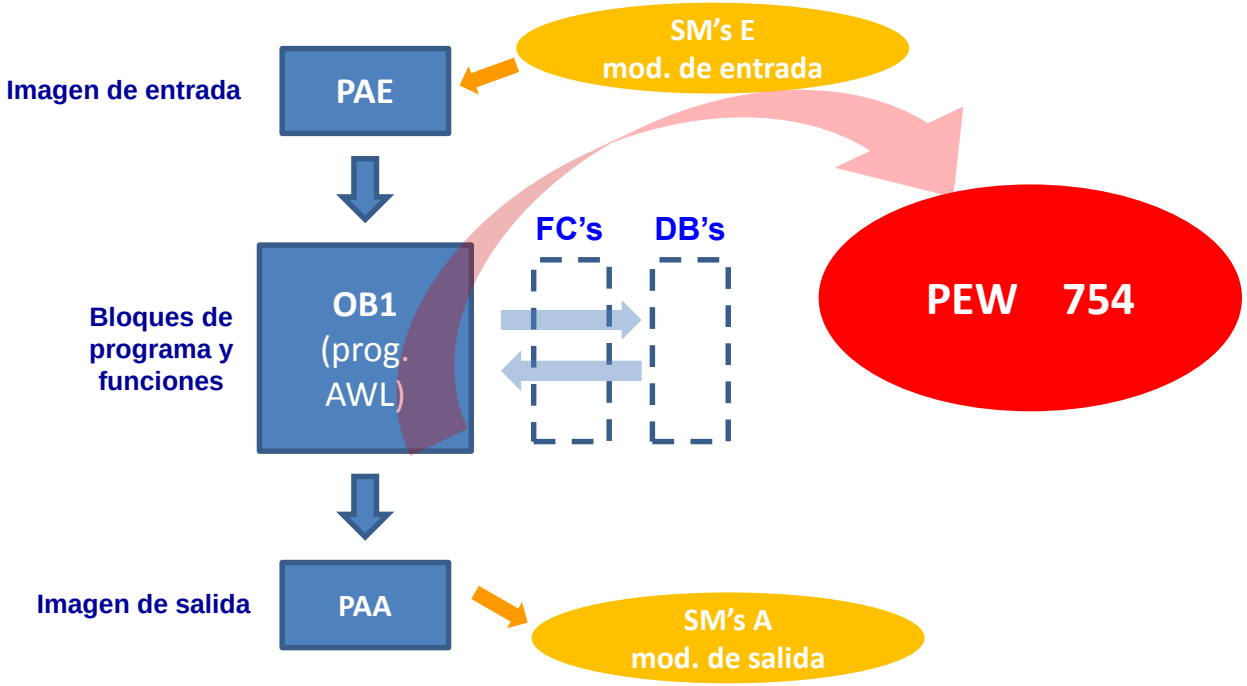
carga /transf.

acceso a la periferia

palabra de entrada / salida

dirección del byte

El acceso a la periferia se ejecuta del siguiente modo:



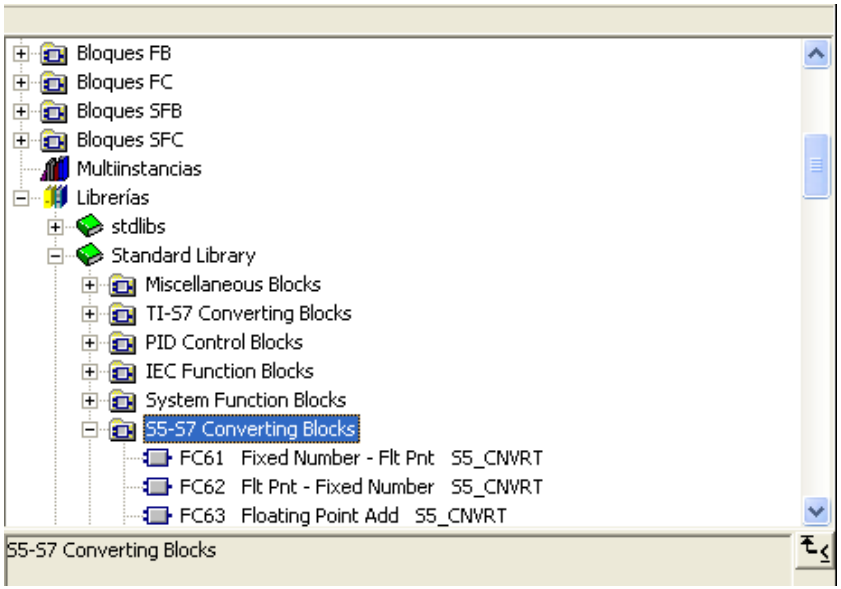
Cuando se ejecuta L PEW 754 El valor DECIMAL de la entrada analógica en el byte 754 se carga en ACU1

TRATAMIENTO ANALÓGICO

El tratamiento de los valores analógicos se realiza con la ayuda de las funciones de librería:

FC 105 (escalado)
y
FC 106 (desescalado)

Estas dos funciones se encuentran en la ruta:



TRATAMIENTO ANALÓGICO

Los parámetros de la función **FC105** son:

IN := (valor que se desea escalar –el de la PEW-)

HI_LIM := (valor límite sup. de la vble. de salida, 5 p. ej.)
LO_LIM := (valor límite inf. de la vble. de salida, 0 p. ej.)

BIPOLAR := (si es + o -)

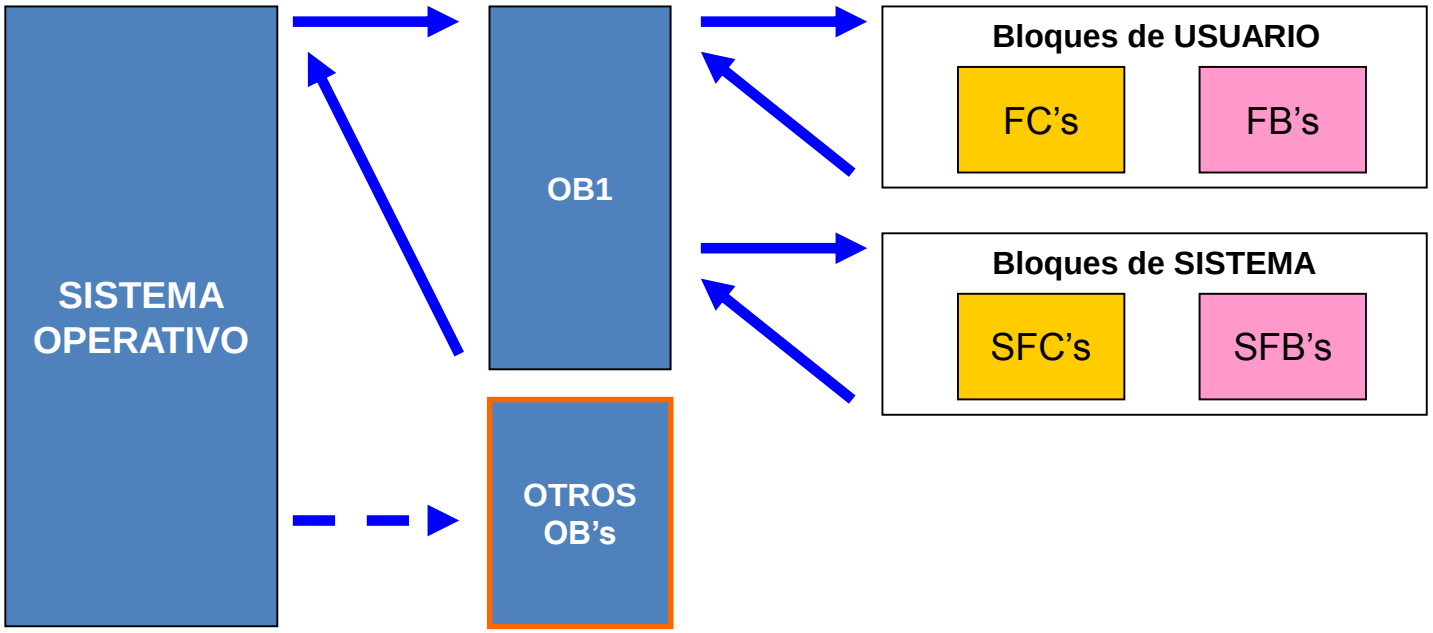
OUT := (valor real de salida)

Los parámetros de la función **FC106** hacen lo contrario: en la salida dará un valor entero y los límites serán los de entrada (0 y 5 p. ej.). La entrada tendrá formato real.

BLOQUES DE ORGANIZACIÓN

BLOQUES DE ORGANIZACIÓN

Son los denominados **OB's**. Su peculiaridad es que NO se pueden ejecutar desde dentro del programa y sólo pueden ser gestionados por el sistema operativo.

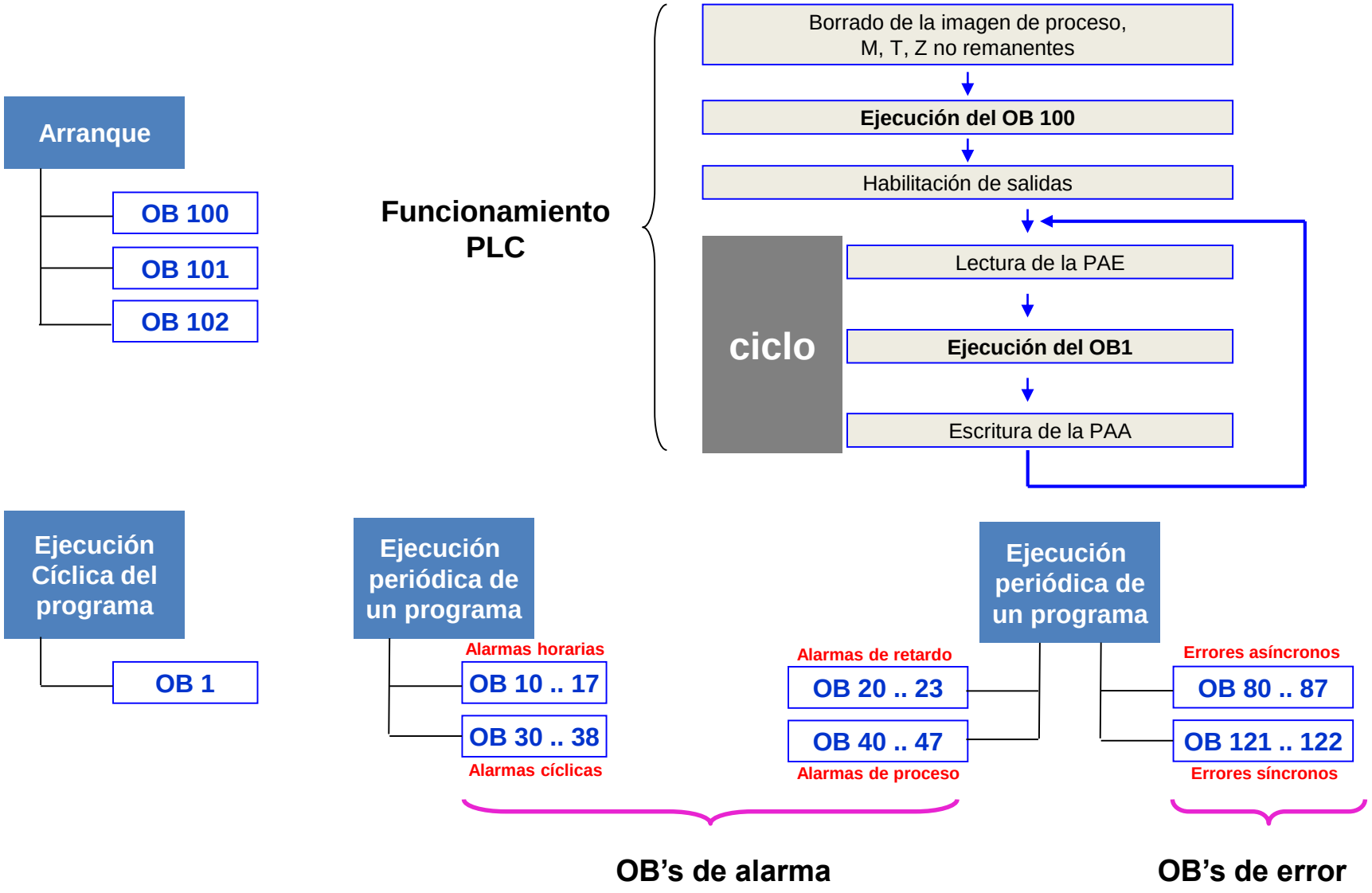


BLOQUES DE ORGANIZACIÓN

Las funciones de estos bloques son:

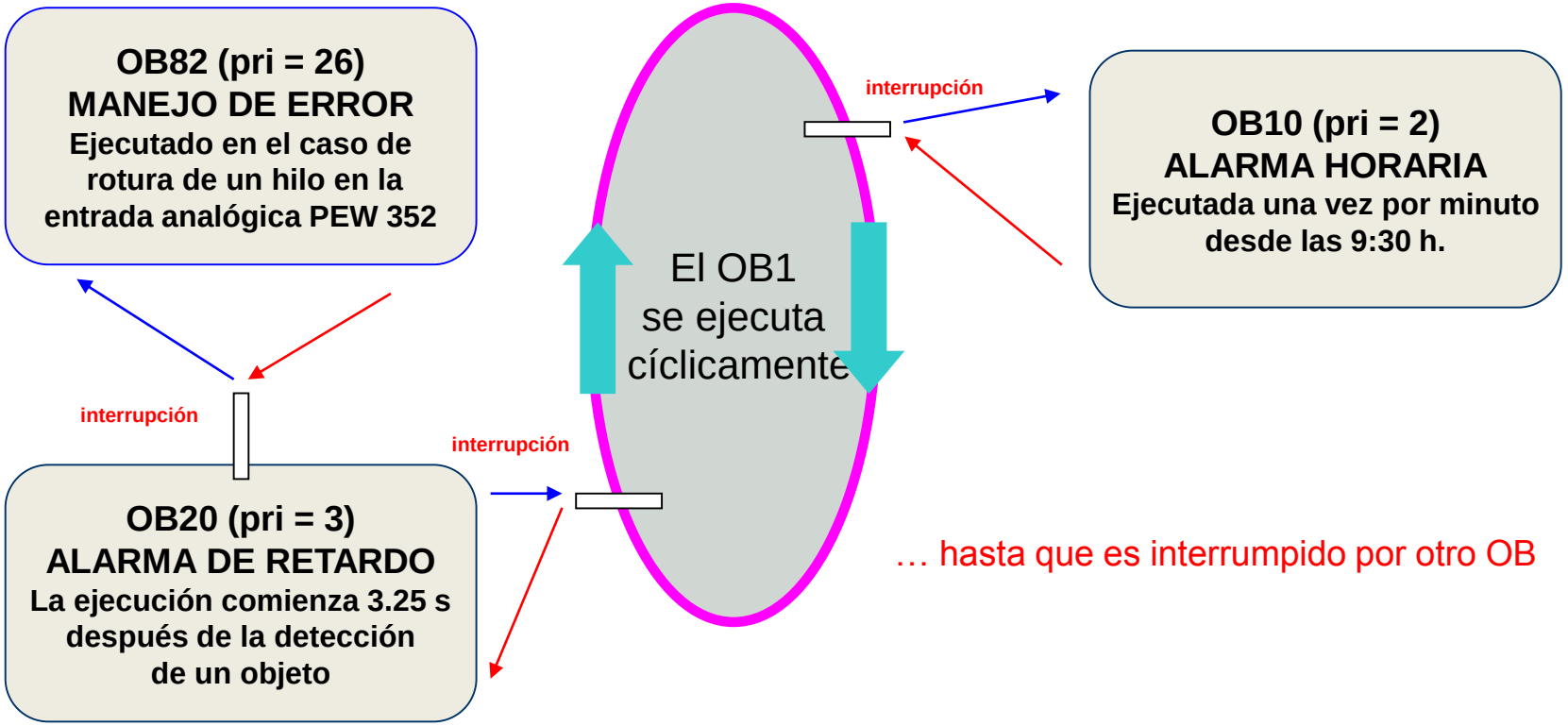
- **Arranque**.- Antes de la ejecución cíclica del programa, se ejecuta una secuencia de arranque en el caso de una recuperación de la alimentación (por previa caída) o un cambio en el modo de funcionamiento (cambio selección del modo en CPU o PG).
- **Ejecución cíclica del programa**.- En el bloque OB1 se almacena el programa que será ejecutado de modo cíclico a través de la CPU. El tiempo de actualización de imágenes de proceso y ejecución del ciclo es lo que determina el **ciclo de *scan***.
- **Ejecución periódica de un programa**.- Con la ejecución periódica de un programa se puede interrumpir la ejecución cíclica de un programa a intervalos fijos de tiempo. Por ejemplo, mediante OB's destinados a alarmas horarias se podrían hacer copias de seguriadad a una hora concreta todos los días.
- **Ejecución del programa sujeta a eventos**.- Las interrupciones del proceso pueden usarse para dar una rápida respuesta a eventos del proceso. Después de que tenga lugar un evento, el ciclo se interrumpe inmediatamente y se ejecuta un programa de alarma.

BLOQUES DE ORGANIZACIÓN



BLOQUES DE ORGANIZACIÓN

Los bloques de organización se ejecutan mediante interrupciones y un orden de prioridad asignado (1 = prioridad más baja; 29 = prioridad más alta)

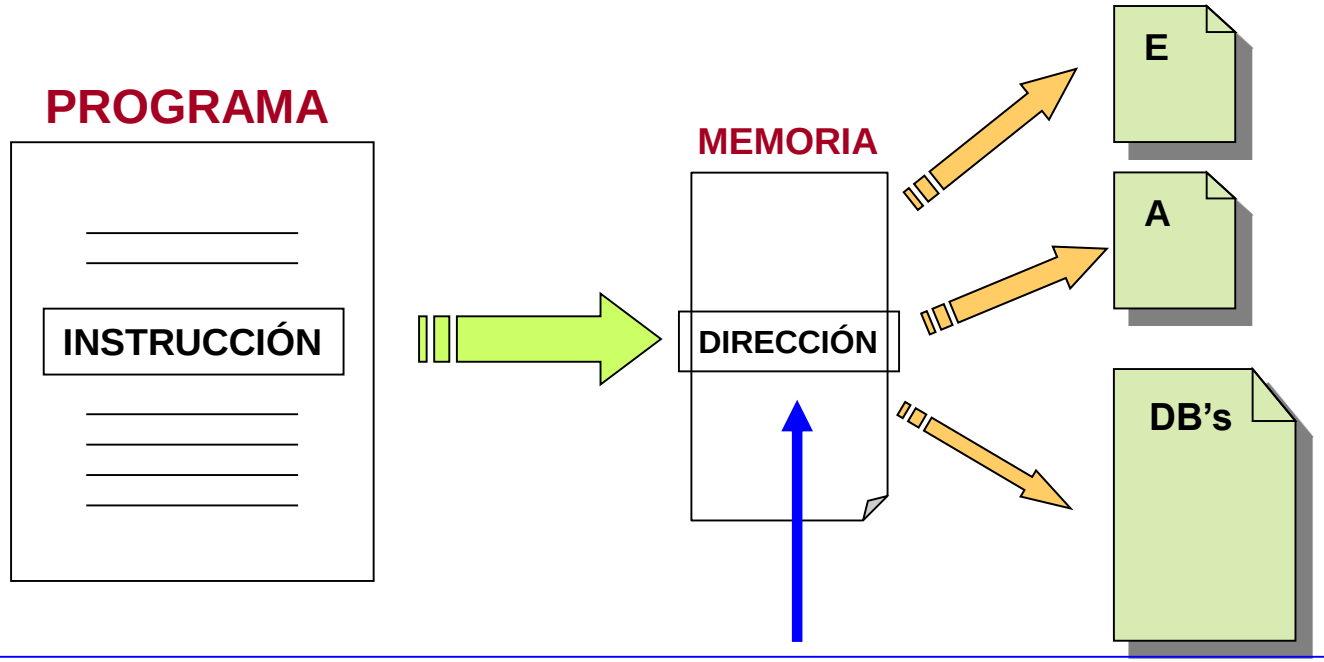


DIRECCIONAMIENTO INDIRECTO

DIRECCIONAMIENTO INDIRECTO

Hasta ahora se ha estado direccionando ya que se ha accedido a memoria mediante entradas (E), salidas (A), marcas (M), bloques de datos (DB's), etc. A este tipo de direccionamiento se le denomina **DIRECTO (el área de memoria se codifica en la instrucción: el identificacor del operando especifica la dirección del valor que va a procesar)**.

Existe otro tipo de direccionamiento a través del cual, a la hora de obtener información de la memoria, hay que introducir un paso intermedio. Este es el denominado **DIRECCIONAMIENTO INDIRECTO**. Con el direccionamiento indirecto se pueden direccionar identificadores de operandos cuya dirección se determina solamente en la ejecución del programa.



Con el uso de punteros se puede acceder mediante un parámetro a E/A /M, etc.

DIRECCIONAMIENTO INDIRECTO

Existen dos tipos de direccionamiento indirecto (DI):

- **DI por MEMORIA**
  - **de 16 bits**
  - **de 32 bits**

  Un puntero a la dirección apuntada se encuentra en una celda de la memoria de usuario (MD 30, p. ej.)

DI por REGISTRO

- **Intra-área**
- **Inter-área**

El puntero a la dirección apuntada se guarda en uno de los dos registros de instrucciones (AR1 o AR2) del procesador S7, antes de acceder a ella

DIRECCIONAMIENTO INDIRECTO

Con el direccionamiento indirecto por memoria, la dirección de la variable a la que se debe acceder se encuentra en una dirección (posición de memoria).

Las sentencias de programa que usan direccionamiento indirecto por memoria contienen:

- **Una operación** (p. ej.: AUF, U, L, etc.)
- **Un identificador del operando** (p. ej.: DB, T, Z, E, AW, MD, etc.)
- **Una variable [ ]** (desplazamiento) que se debe indicar entre corchetes. La variable contiene la dirección (puntero) del operando al cual accede la operación.

Dependiendo del identificador del operando utilizado, la operación interpretará el dato almacenado en las [variables] especificadas, como un puntero en formato palabra o doble palabra.

DIRECCIONAMIENTO INDIRECTO por MEMORIA

- **DI por MEMORIA (16 bits)**

Se usan punteros de 16 bits para direccionar:

- **Temporizadores**
- **Contadores**
- **Módulos DB, FC, FB**

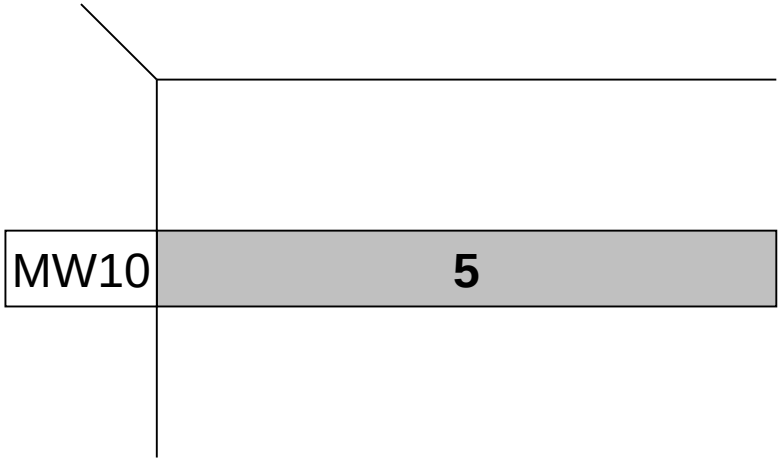
Supóngase como ejemplo las siguientes líneas de programa en código AWL para direccionar con 16 bits (puntero 16 bits formato palabra):

```
L     5              // carga el valor entero 5
T     MW    10       // lo transfiere a la marca MW 10
L     5ST#3S         // carga 3 s en formato de tiempo
SV    T[MW 10]       // carga el temporizador 5 de tipo SV
```

SV T5

DIRECCIONAMIENTO INDIRECTO por MEMORIA

El valor 5 será transferido indirectamente a través de la dirección MW 10



Otro ejemplo sería:

```
L     11
T     MW    60

AUF   DB[MW 60]   ➡   AUF   DB 11
```

DIRECCIONAMIENTO INDIRECTO por MEMORIA

- **DI por MEMORIA (32 bits)**

Se puede acceder a las siguientes direcciones con la ayuda del direccionamiento indirecto por memoria usando punteros de 32 bits:

- **Bits direccionados por operaciones lógicas con bits**.

  Se pueden usar E, A, M, L, DIX o DBX como identificadores del operando.

- **Bytes, palabras y dobles palabras direccionadas mediante operaciones de carga y transferencia**.

  Se pueden usar EB, EW, ED, DBB, DBW, DBD, DIB, DIW, DID, PEB, PEW, PED, como identificadores del operando.

Si se desea acceder a una dirección por medio del direccionamiento indirecto por memoria usando operaciones de carga o transferencia, debe asegurarse que la dirección del bit del puntero es “0”.

DIRECCIONAMIENTO INDIRECTO por MEMORIA

ÁREAS DE DIRECCIÓN PARA PUNTEROS

E/A/M (BITS)

```
L     p#4.0
T     MD    100

U     E     [MD 100]
=     A     [MD 100]
```

E/A/M (BYTES)

```
L     p#4.0
T     MD    100

L     EB    [MD 100]
T     AB    [MD 100]
```

Siempre que el DI sea a byte el puntero ha de estar direccionado al elemento 0 del byte.
Por ejemplo: **p#6.0**

ÁREAS DE DIRECCIÓN PARA PUNTEROS

DB's

```
L       p#0.0
T       MD      100

L       1
T       MW      10

AUF     DB      [MW10]
L       DBW     0
```



```
L       DB1.DBW0
```

~~L DB[MW10].DBW[MD100]~~

Como los punteros son variables, también pueden ser tratados mediante operaciones aritméticas:


```
L       p#4.3
L       p#2.0
+ D  - - - - - - - - - - →  p#6.3
```

EJEMPLO DIRECCIONAMIENTO INDIRECTO POR MEMORIA

Inicializar las entradas de un módulo de datos con el valor 0.

```
// ABRIR CON DI

L       #numerodb
T       MW      100
AUF     DB[MW100]

// BUCLE DE BORRADO

L       p#18.0
T       MD      40
L       10
next: T MB      50
      L 0
      T DBW     [MD 40 ]
      L MD      40
      L p#2.0
      - D
      T MD      40
      L MB      50
      LOOP next
```

// Cargar el número del DB en MW100 (param. de entrada
// Transfiere a marca
// Abre la base de datos (DB)

// Guarda dirección final como puntero …
// … en MD 40
// Preajusta el contador de bucle a 10
// Comienza bucle y transfiere valor de ACU1 a MB 50
// Carga valor de inicialización …
// … y lo transfiere al DB
// Carga el puntero
// Carga puntero para decrementar en 2 bytes
// Decrementa
// Transfiere a MD 40
// Carga contador de bucle
// Decrementa y si es necesario salta

DIRECCIONAMIENTO INDIRECTO por REGISTRO

En la memoria se dispone de dos registros de dirección, **AR1** y **AR2**:

AR1 → **p#4.0**
AR2 → **p#9.0**

Existen tres formas diferentes de introducir el puntero en los registros:

1ª)

```
L      p#4.0
LAR1
```

va escrito todo junto

2ª)

```
LAR1   p#4.0
```

Ahorramos una línea respecto al modo anterior

3ª)

```
L      p#4.0
T      MD      100
LAR1   MD100
```

Resulta un poco largo

DIRECCIONAMIENTO INDIRECTO por REGISTRO

Una vez cargado el puntero en el registro lo importante es saber cómo se direcciona. Existen dos formas:

- Intraárea
- Interárea

Intraárea.- La dirección (posición de memoria) del operando al que se va acceder se encuentra en uno de los dos registros de dirección AR1 ó AR2. En este tipo de direccionamiento:

- El contenido del registro de direcciones es un puntero de área interna.
- Con este tipo de direccionamiento hay que especificar un *offset* que se suma al registro de direcciones.
- La suma se realiza cuando la operación es ejecutada sin modificar el contenido del registro de direcciones.
- El offset tiene el formato de puntero a área.
- En el direccionamiento indirecto de direcciones digitales, el *offset* ha de tener como dirección el bit “0”.
- El máximo valor es: P#8191.7

DIRECCIONAMIENTO INDIRECTO por REGISTRO (intraárea)

Sintaxis

```
LAR1 P#10.0              // carga puntero dir. Reg1.
…
T MW [AR1, P#4.0]        // asigna dirreción digital incrementada
…
U E [AR1, P#2.1]         // asigna dirección binaria
```

Ejemplo:

```
LAR1  P#0.0              // carga puntero en registro
L  8                     // carga entero 8
lazo: T  MW  20          // abre lazo trasladando 8 a marca MW 20
      U E [AR1, P#0.0]   // toma lo que hay en E0.0 …
      =  A [AR1, P#4.0]  // lo pasa a A 4.0
      +  AR1 P#0.1       // incrementa el puntero en un bit
      L MW 20            // carga el valor de M 20 para el control del lazo
      LOOP lazo          // cierra bucle de control
```

DIRECCIONAMIENTO INDIRECTO por REGISTRO (interárea)

Interárea.- El identificador de área (E, M, A, etc.) y la dirección (posición de memoria del byte.bit) del operando al que se va a acceder se encuentra se encuentra como puntero intraárea en uno de los dos registros de dirección. En este tipo de direccionamiento:

- La dirección general está definida en uno de los dos registros de direcciones (AR1 o AR2).
- El contenido del registro de direcciones es un puntero en general.
- Con el direccionamiento general se escribe el área de direcciones junto con el puntero a área dentro del registro de direcciones.
- Con direccionamiento indirecto sólo se especifica un identificador para la dirección: B (para byte); W (para palabra); D (para doble palabra).
- Se especifica un *offset* con dirección bit.

Sintaxis

```
LAR1 P#M12.0          // carga puntero dir. Reg1.
…
L B [AR1, P#4.0]      // asigna dirección digital incrementada
…
= [AR1, P#0.7]        // asigna dirección binaria
```

DIRECCIONAMIENTO INDIRECTO por REGISTRO (interárea)

Ejemplo (equivale al indicado en direccionamiento intraárea):

```
LAR1  P#E0.0          // carga puntero en registro AR1
LAR2  P#A4.0          // carga puntero en registro AR2
L  8                  // carga entero 8
lazo: T  MW  20       // abre lazo trasladando 8 a marca MW 20
      U [AR1, P#0.0]  // toma lo que hay en E0.0 …
      =  [AR2, P#0.0] // lo pasa a A 4.0
      +  AR1 P#0.1    // incrementa el puntero en un bit
      +  AR2 P#0.1    // incrementa el puntero en un bit
      L MW 20         // carga el valor de M 20 para el control del lazo
      LOOP lazo       // cierra bucle de control
```