PROGRAMACIÓN AVANZADA (1)

Índice

Índice

INTRODUCCIÓN AL S7

A la hora de configurar un proyecto de automatización desde S7 (SIEMENS SIMATIC STEP 7) hay que insertar en la configuración hardware del programa todos los equipos que se vayan a utilizar en el proyecto.

Por motivos de extensión y ajuste a los contenidos del módulo, sólo se insertará un equipo al proyecto. Concretamente, uno de los más utilizados en los procesos automáticos avanzados el SIEMENS S7 300.

El equipo (hardware) se compone de dos elementos: a) unidad de programación (PC), y b) de un autómata con una fuente de alimentación, una unidad central de proceso (CPU) y diferentes módulos de entrada y salida (E/A) analógicos y digitales.

Además de hardware, dentro del equipo se han de incluir los bloques de programación (software) que se vayan a utilizar.

Conocer las técnicas de programación avanzada para estos bloques es el objetivo principal para esta asignatura.

Hay que tener siempre presente que cada vez que se elabore un nuevo proyecto hay que definir un hardware nuevo.

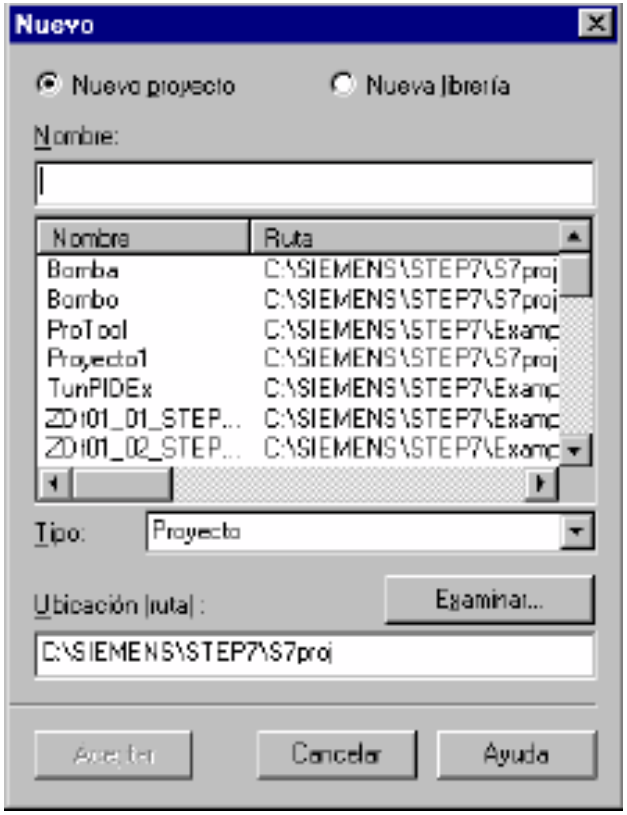
También, sobre un determinado proyecto con un hardware ya definido se pueden crear nuevos programas. Este será el procedimiento a seguir en el curso.

Para crear un nuevo proyecto hay que abrir el Administrador SIMATIC. En este administrador se encuentra disponible un ASISTENTE (para utilizarlo sólo hay que ir obedeciendo a lo que pide introducir).

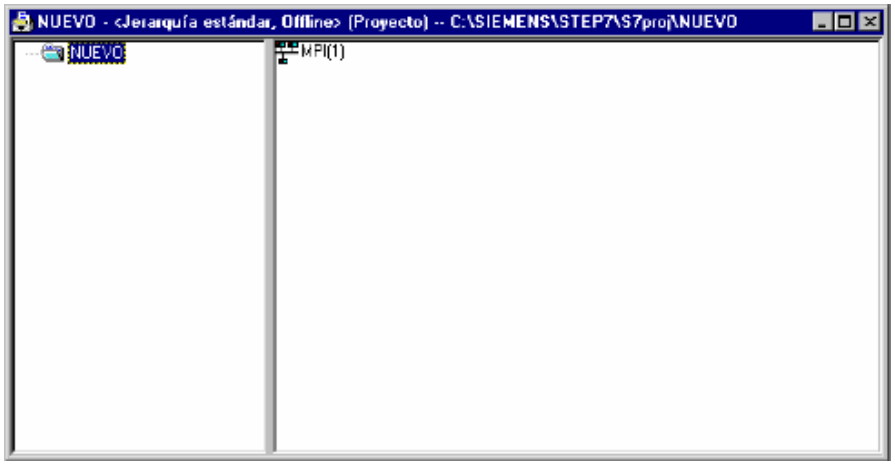
Generalmente, los programadores no utilizan el asistente y generan por sí mismos el nuevo proyecto.

Para comenzar hay que ir al menú: ***Archivo > nuevo***. También, desde la barra de herramientas hacer *clic* en el icono que representa una hoja en blanco.

Aparecerá una ventana de la forma:



Una vez que se la ha dado nombre al proyecto se pulsa **aceptar** y se observará la siguiente ventana:

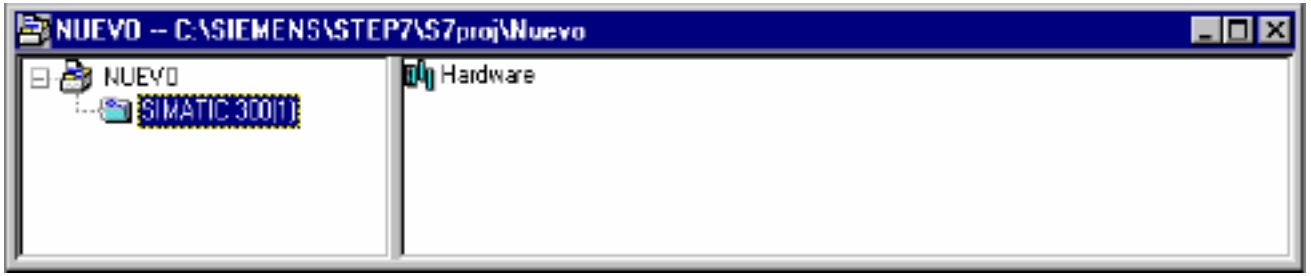


El **nombre** del proyecto aparece en la parte izquierda (**nuevo**) y la red MPI (*multi point interface*) en la parte derecha.

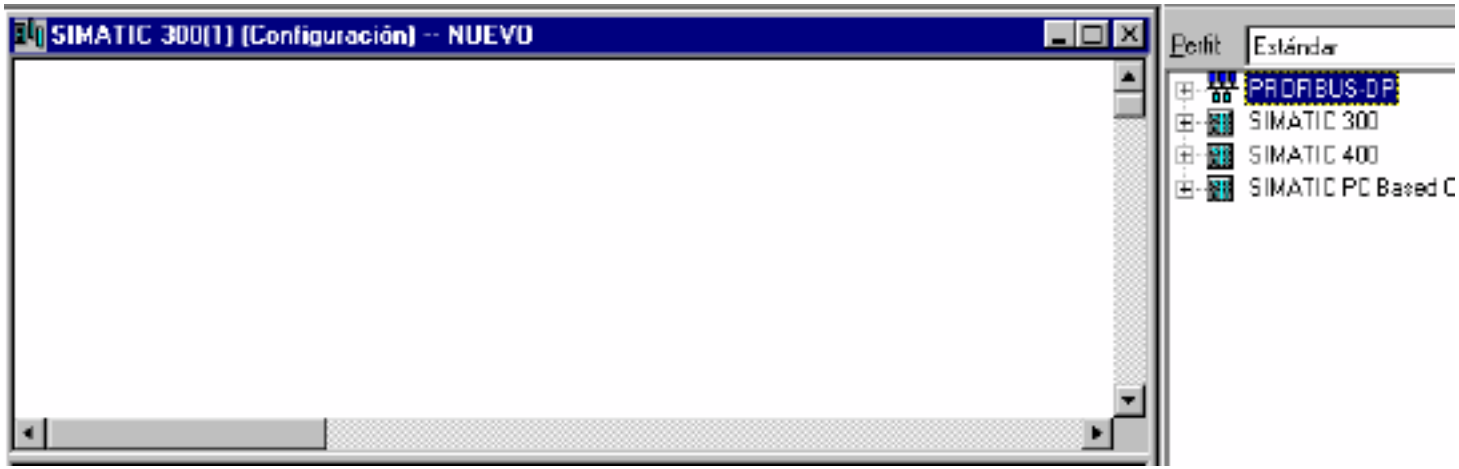
El icono de la red MPI siempre aparece por defecto. Esto se debe a que al menos hay que tener una red MPI en el equipo, ya que la programación de la CPU se hace a través su puerto MPI.

Ahora hay que insertar los equipos que se vayan a utilizar en el proyecto, que en este caso va a ser sólo uno. Para ello hay que **pinchar** sobre el **proyecto**, optar por **insertar** y elegir como **equipo** el **S7 300**.

Se observará que sobre el proyecto se ha creado un equipo. Haciendo *clic* sobre él, en la parte derecha de la ventana aparecerá un icono con el nombre **hardware**:



Clic sobre el icono de hardware y se entra en su configuración:



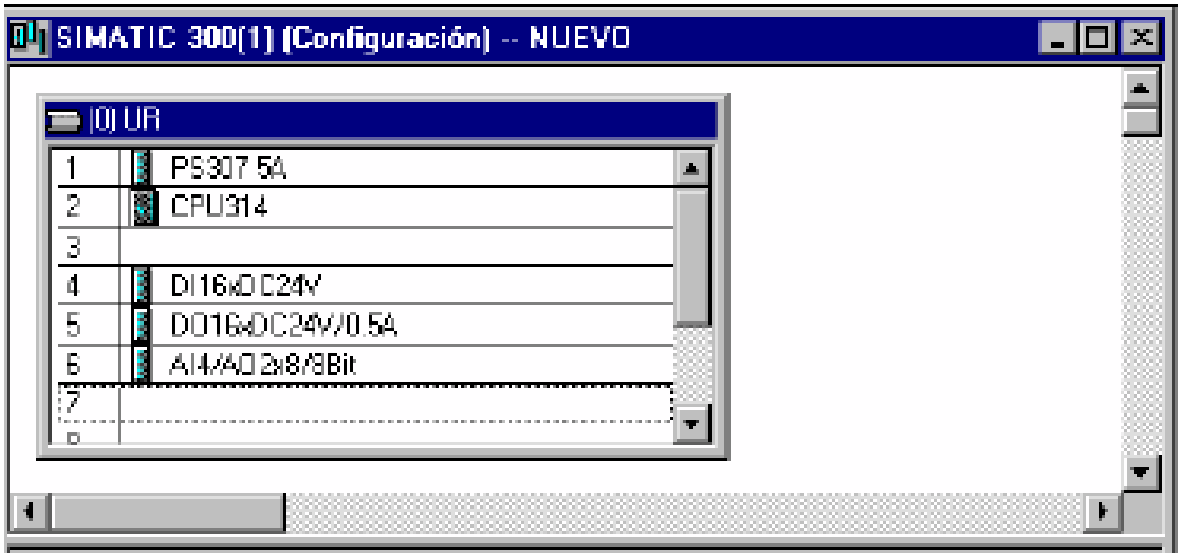
Para **insertar** los **módulos** que conforman el equipo, lo primero que hay que hacer es **abrir** el **catálogo** (suele estar abierto): ***ver > catálogo***.

Ahora hay que **desplegar** la cortina correspondiente al **SIMATIC 300** y:

1. **Insertar** el **bastidor**.
2. Situarse en la **posición 1** e insertar la **fuente de alimentación (PS)**.

3. Situarse en la **posición 2** e **insertar la CPU**.
4. En la **posición 3** no se puede insertar cualquier módulo ya que está reservada para los **módulos IM** (tarjetas IM). Las tarjetas IM sirven para configurar diversas líneas de bastidor. En el caso que nos ocupa solo hay una línea de bastidor por lo que habrá que dejar la posición 3 **libre**.
5. En la **posición 4** y en las siguientes posiciones, hay que insertar los **módulos** de **entrada/salida (E/A)**. Si en el catálogo se encuentran varios módulos del mismo modelo, habrá que comprobar la referencia de cada elemento.

El hardware quedará configurado de la siguiente manera:



En la pantalla principal y debajo de esta tabla, se va creando otra en la que pueden visualizarse todos los elementos que se han creado con sus respectivas referencias y direcciones.

Ahora hay que guardar la configuración. Como se está trabajando con dos CPU a la vez (programadora -PC- y la PLC), hay que guardar la información de la configuración en los dos sitios mediante el uso de los iconos:

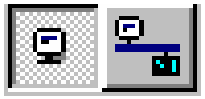
- El icono que representa un disquete permite guardar la información en el PC (programadora).
- El icono que representa un PLC y una flecha que entra permite guardar la información en el PLC.

Saliendo del editor de hardware y pasando a la **ventana del administrador** ya se puede observar que en la ventana de la izquierda aparece el **proyecto**, el **equipo**, la **CPU** y un **programa por defecto** (*Programa S7(1)*):

En la misma ventana aparecen dos carpetas de programa: **Fuentes** y **Bloques**.

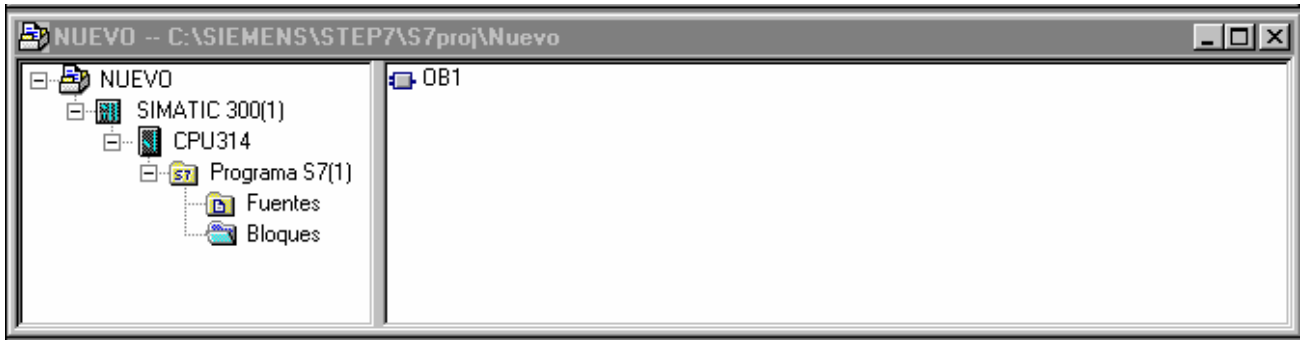
Pinchando sobre **bloques** se observa en la parte derecha de la ventana una carpeta que contendrá todos los **datos del sistema** y un fichero de bloques de programa denominado **OB1**.

En la barra de herramientas del administrador de SIMATIC hay dos iconos: uno seleccionado que representa un PC; y el otro, sin seleccionar que representa un PC conectado mediante un bus a un PC:



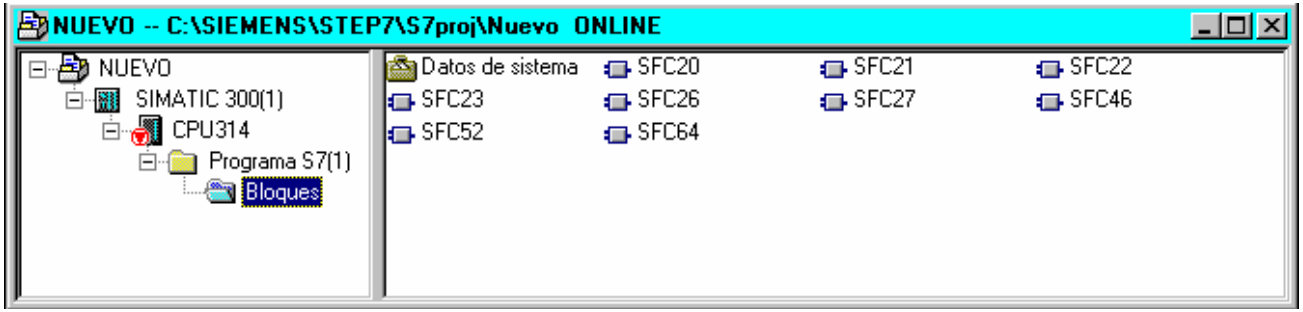
Con estos dos iconos se puede trabajar en modo ***OFF LINE*** y ***ON LINE***, respectivamente.

En modo ***OFF LINE*** se trabaja únicamente con la programadora (PC), y, como puede comprobarse, sólo se leerá información de su disco duro. En la carpeta de bloques sólo se dispone del bloque OB1, que es el que crea el proyecto por defecto:



En modo ***ON LINE*** siempre se estará trabajando directamente en el PLC. En este caso, se estará leyendo directamente la información que contenga el PLC. Así, se observan otros bloques: son los bloques que integra la CPU (están protegidos).

Dependiendo de la CPU aparecerán bloques difierentes que, al estar protegidos, no se pueden ver editar y borrar. Únicamente con la tecla de ayuda (**F1**) puede conecerse cuál es su función.



Cuando al trabajar ***ON LINE*** aparecen bloques que no son los propios del sistema (**SFC** o **SFB**) quiere decir que en el PLC se ha grabado algún programa. Siempre y cuando se desee trabajar con un programa nuevo sobre la PLC es necesario borrar todo lo que haya grabado en ella.

Para ello, se hace *clic* sobre el icono de la **CPU** en modo ***ON LINE***, se elige **Sistema Destino**, y, finalmente, **Borrado Total** (con esto se borrarán todos los bloques excepto los de sistema y la configuración de hardware).

Para comprobar si el borrado ha sido efectivo se vuelve a hacer *clic* sobre la carpeta **Bloques** y se verá que sólo quedan en ella los ficheros o bloques de sistema. Sobre los bloques de sistema se puede trabajar tanto en modo ***ON LINE*** como en modo ***OFF LINE***, por lo tanto, a la hora de guardar lo que se vaya programando podrá hacerse sobre la programadora (PC), o sobre la CPU.

Hay que tener en cuenta que, a la hora de trabajar con bloques, en un momento dado se podrían utilizar tres bloques con el mismo nombre simultáneamente.

Por ejemplo, si se está trabajando con un OB1, el bloque que se visualiza en la programadora, mientras no se guarde en ningún sitio, estará ubicado únicamente en su memoria RAM; al mismo tiempo, se puede dar el caso de que se disponga de un bloque con el mismo nombre en el PLC, y se disponga de otro bloque guardado en disco duro, en el que se acaba de hacer una modificación y se tenga en pantalla sin haber sido todavía transferida.

Es importante indicar que si con el bloque en la pantalla se selecciona el icono de **guardar** o el icono de **transferir al autómata**, se estará grabando en el disco duro o en el autómata lo que haya en la pantalla. Pero si volvemos a la pantalla principal (**Administrador de Simatic**) sin haber guardado previamente el bloque en disco duro, y transferimos algún bloque arrastrándolo con ayuda del ratón, vamos a transferir lo último que hubiésemos guardado en disco duro, y no las últimas midificaciones que hemos hecho en el bloque.

Con SIMATIC STEP S7 se pueden programar los siguientes tipos de bloques:

- **OB** Bloques de sistema
- **FC** Funciones
- **FB** Bloques de función
- **DB** Bloques de datos
- **UDT** Tipo de datos.

OB: Se denominan así a los **bloques de organización**. Existen diferentes bloques OB, cada uno de ellos realiza una determinada función. El **OB1** es el único bloque de **ejecución cíclica** y es ejecutado por la CPU sin que nadie le invoque. Los demás OB tienen una función determinada y se ejecutan cuando les corresponda, sin que nadie les llame desde ningún sitio del programa. Así, existen bloques OB asociados a diferente errores de la CPU, a alarmas, etc.

FC: Las FC son **funciones**. Son subprogramas (creadas por el programador o por otros programadores en librerias) que realizan una función determinada dentro del proyecto. Se ejecutan cuando son invocadas desde algún punto del programa (tienen la misma labor que las funciones en lenguaje C). Pueden ser parametrizables o no. Las FC de librería (creadas por otros programadores) no pueden ser leídas ni editadas.

FB: Se denominan así a los **bloques de función**. De forma general, puede decirse que estos bloques desempeñan trabajan igual que las FC, con la diferencia de que en las FB se guarda la **tabla de parámetros** en un módulo de datos. Esto tiene dos ventajas: una es que se posibilita el acceso a los parámetros desde cualquier punto del programa; y la otra es que cada vez que se llame a la FB no es necesario que se le den todos los parámetros . Los parámetros que no se introduzcan, tomarán los valores por defecto de la última vez que se introdujeron.

DB: Se denominan así a los **módulos de datos**. En este tipo de bloques no se desarrollan programas. Únicamente son tablas de datos en las que se puede leer y escribir (editar).

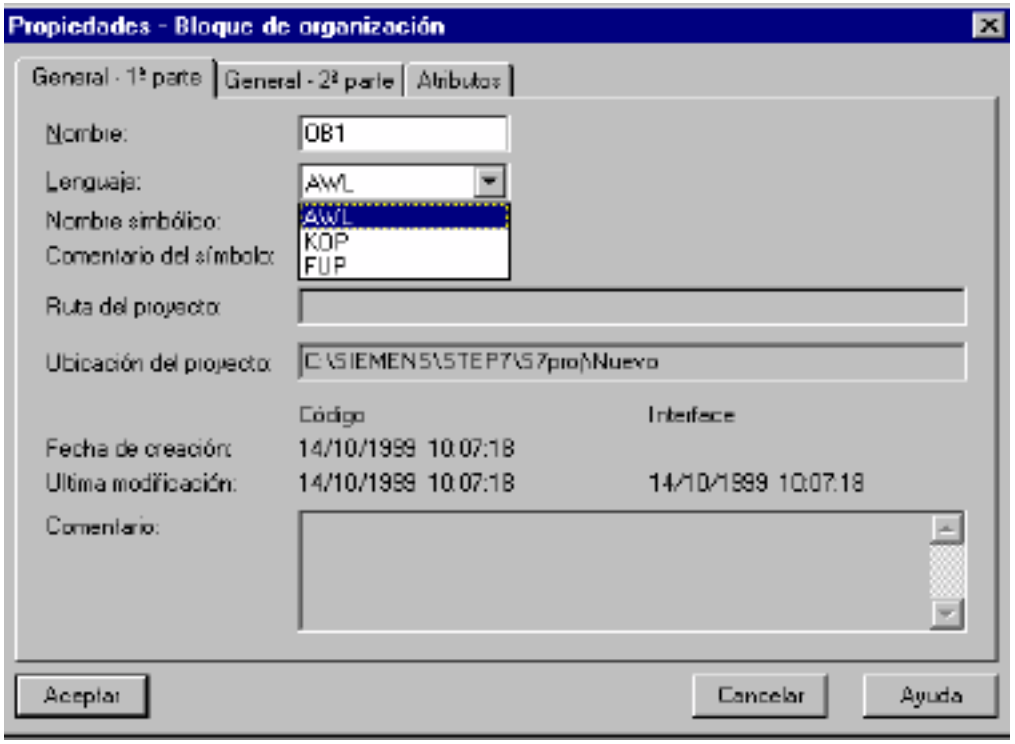
UDT: Son bloques en los que se definen el **tipo de datos** para poder utilizarlos en los bloques DB.

OPERACIONES CON BITS. INSTRUCCIONES BINARIAS

Programación de un OB1

Antes de comenzar a desarrollar un programa hay que crearlo dándole nombre, o utilizar el programa por defecto que se generó al configurar el hardware.

Abierta la carpeta del programa, ha de seleccionarse bloques y hacer *clic* sobre OB1. A partir de este momento aparecerá la siguiente pantalla:



Ahora sólo hay que elegir uno de los tres lenguajes de programación para bloques:

- **KOP**
- **FUP**
- **AWL**

El lenguaje **KOP** es muy utilizado porque se basa en símbolos relacionados con la lógica cableada (contactos abiertos, cerrados, relés, temporizadores, etc.).

El lenguaje **FUP** se utiliza en programación mediante funciones de lógica binaria.

El lenguaje **AWL** es un lenguaje de código y es muy potente. Puede decirse que es un lenguaje tipo código máquina. Este lenguaje consiste en una lista de instrucciones que se ejecutan al arrancar el programa. El lenguaje AWL permite la programación estructurada. A partir de este momento, toda la programación del curso se realizará en AWL.

El lenguaje **SCL** es un lenguaje de programación de alto nivel (muy parecido al PASCAL). Con este lenguaje se utilizarán sentecias que facilitan la labor de programación notablemente (ej. If..then, case …, for, etc.), principalmente en los procesos medida y tratamiento de datos.

Otra ventaja añadida es que estas sentencias ya son conocidas al haberlas utilizado en programación estructurada mediante lenguaje C.

Durante el seguimiento del curso, que se realizará en lenguaje AWL, paralelamente, se mostrarán las sentencias más basicas de programación en SCL, todo ello diferenciando las láminas de *power point* mediante un color gris de fondo.

La operación más sencilla con bits es la de utilizar un contacto (entrada **E**) para activar, por ejemplo, un relé (salida **A**).

Antes de elaborar un programa que realice esta operación hay que conocer cuáles son las entradas y las salidas disponibles en la memoria para poder utilizarlas.

En primer lugar hay que recurrir al hardware (pantalla de **Administrador Simatic,** y *clic* en la **CPU**) y se verá que pueden utilizarse como entradas los ocho bits de los bytes 124 y 125. Para este programa se elegirá, por ejemplo, el **E 124.0**; y para la salida el **A 124.0**.

En segundo lugar, hay que consultar la ocupación de memoria (**mapa de ocupación de memoria**) para comprobar que estos bits no están siendo ocupados por ningún otro tipo de memoria (byte, palabra, o doble palabra) en el programa.

En tercer lugar habrá que asignar un **símbolo** a la entrada y otro a la salida para que después el programa pueda ser leído más fácilmente.

Ya sólo queda que introducir los comentarios que se crean oportunos en la cabecera del programa y en los segmentos que se utilicen.

El código a introducir será:

```
U     E     124.0
=     A     124.0
```

Ahora hay que cargar el hardware en el SIMULADOR, transferir los datos del programa al simulador y, por último, ejecutar el programa.

Si se desea, en la parte inferior de la pantalla de programación se puede visualizar el comportamiento y valor de las variables (en este caso E y A) durante la ejecución del programa.

En SCL el programa anterior tendría la siguiente sintaxis:

```
IF("entrada"=1) THEN
  "salida":=1;
ELSE
  "salida":=0;
END_IF;
```

Para este sencillo programa se ha utilizado la sentencia de control:

IF-ELSE-END_IF

Con la siguiente asignación de variables

entrada	Bool	%E124.0
salida	Bool	%A124.0

Las operaciones binarias pueden reducirse a:

- **AND/AND NEGADA:**

```
U    E    124.0
UN   E    124.1
=    A    124.0
```

- **OR/OR NEGADA:**

```
O    E    124.0
ON   E    124.1
=    A    124.0
```

- **XOR:**

```
X    E    124.0
X    E    124.1
=    A    124.0
```

Para las operaciones lógicas habrá que utilizar la siguiente sintaxis:

Operación	Operador
Negación	NOT
Conjunción	AND
Disyunción exclusiva	XOR
Disyunción	OR

Estas operaciones se pueden utilizar para los siguientes tipos de datos:

BOOL, BYTE, WORD, DWORD

Programa ejemplo:

Simular el automatismo MARCHA-PARO para un MOTOR que pueda ser controlado desde dos lugares diferentes

```
IF(("marcha1"=1)OR ("marcha2"=1))THEN
   "motor":=1;
END_IF;
IF(("paro1"=1) OR ("paro2"=1)) THEN
   "motor":=0;
END_IF;
```

Con la siguiente asignación de variables

paro2	Bool	%E124.7
paro1	Bool	%E124.6
motor	Bool	%A124.0
marcha2	Bool	%E124.1
marcha1	Bool	%E124.0

Las marcas son lugares de memoria tipo bit que almacenan el valor de una determinada operación o valor lógico.

Por lo tanto, las marcas han de ir asociadas a una dirección de memoria de tipo bit, por ejemplo: **M 0.0**.

Las marcas pueden ser utilizadas y requeridas para utilizar su valor durante la ejecución del programa.

El programa anterior podría escribirse utilizando una marca de la forma:

```
U     E     124.0
=     M     0.0

U     M     0.0
=     A     124.0
```

En este programa, el resultado de activar **E 124.0** pasa a la marca **M 0.0**, y el valor de la marca **M 0.0** sirve para activar la salida **A 124.0**

A la hora de utilizar marcas es muy importante saber si su dirección de memoria está siendo ocupada por alguna variable a la que ha sido asignada la misma dirección de memoria.

Existe una zona de memoria en la PLC denominada **MARCA DE CICLO** que, una vez activada, ofrece la posibilidad de disponer de un reloj con diferentes frecuencias. Para activar el byte de la marca de ciclo, habrá que proceder en las propiedades de la CPU, y una vez activado el byte, de la CPU podemos contar con 8 frecuencias (una por bit) diferentes para poder operar con ellas.

En SCL a las marcas se les puede asociar el estado de una variable utilizando también la sentencia de control **IF-THEN-ELSE_IF**:

```
IF("marcha1"=1)THEN
  "Tag_1":=1;
END_IF;
IF("Tag_1"=1) THEN
  "motor":=1;
END_IF;
IF("paro1"=1) THEN
  "motor":=0;
END_IF;
```

Donde la etiqueta “Tag_1” está asociada a una determinada a la marca:

Tag_1	Bool	%M0.0

En SCL se puede utilizar la MARCA DE CICLO de la misma forma que se utiliza en lenguaje AWL.

Consideremos el siguiente ejemplo en el que queremos simular una salida cuando activemos una entrada de tipo bit, y que la salida deje de funcionar de forma intermitente cuando la entrada se desactive.

En primer lugar se procede a activar la marca de ciclo para el área de memoria M150 (por ejemplo) y de este byte se elegirá el bit 5 que se corresponde con una frecuencia de 1 Hz.
El programa será el siguiente:

```
IF "start" THEN
  "salida intermitente":="Tag_1";
ELSE
  "salida intermitente":=0;
  END_IF;
```

Con la asignación de variables:

start	Bool	%E124.0
salida intermitente	Bool	%A124.0
Tag_1	Bool	%M150.5

Se **detecta un flanco** (subida o bajada) cuando se produce un cambio en un resultado de una operación lógica (**RLO**).

Cuando el cambio de la operación lógica es de 0 a 1, el **flanco** es **positivo**, y la instrucción de consulta (FP) devuelve el valor 1 a una determinada marca o bit de datos.

Cuando el cambio de la operación lógica es de 1 a 0, el **flanco** es **negativo**, y la instrucción de consulta (FN) devuelve el valor 1 a una determinada marca o bit de datos.

El valor del flanco almacenado en la marca sólo dura el tiempo de un ciclo de programa.

Para conocer cuál es la sintaxis de la detección de flanco, supongamos una subrutina según la cual cuando se activen dos contactos de entrada se active también una salida:

```
U     E     124.0
U     E     124.1
FP    M     0.1
S     A     124.0
```

Esta secuencia de código se lee así:

Cuando los contactos **E 124.0** y **E 124.1** pasan de 0 a 1 (**RLO=1**), se **activa un flanco positivo** (FP), y su valor (**1**) **pasa a** la marca **M 0.1**, que, **durante ese ciclo de programa**, **activará** (**S, set**) para ese y los demás ciclos la salida **A 124.0**.

NOTA : En las líneas de código anterior es importante señalar que para activar un *set* (**S**) o un *reset* (**R**) **NO** hace falta un flanco.

En SCL las sentencias SET y RESET se implementan mediante sendas funciones denominadas con el mismo nombre.

En el siguiente ejemplo se muestra la sintaxis para dichas sentencias en lenguaje SCL:

```
IF "pulsa" THEN
 SET(S_BIT:="luz",
    N:=1);
END_IF;
IF "apaga" THEN
  RESET(S_BIT:="luz",
    N:=1);
END_IF;
```

Con la asignación de variables:

pulsa	**Bool**	**%E124.0**
apaga	**Bool**	**%E124.7**
luz	**Bool**	**%A124.0**

Es conveniente recordar que para ejecutar el programa, además de ***main*** hay que compilar y cargar los bloques **SET** y **RESET**.

De la colección de ejercicios propuestos uno de ellos trata sobre la simulacion de una función memoria (telerruptor) mediante la utilización de flancos. El siguiente ejemplo considera el mismo ejercicio pero utilizando el lenguaje SCL, sin la utilización de flancos. Como se va a ver inmediatamente, aquí utilizaremos el concepto de contador y convertidor para realizar el ejercicio.

Ejemplo: Activar una de salida mediante un telerruptor

```
"valor":= S_CU(C_NO:="Tag_1", CU:="pulsa", S:="set", PV:=0, R:="reset",  CV=>"memoria", Q=>"Tag_2");
"new_valor":= WORD_TO_INT("valor");     //convertidor de WORD a INT
IF ("new_valor" MOD 2 = 0) THEN         // si es par …
    "luz":=0;
ELSE                                    // si es impar …
     "luz":=1;
 END_IF;
```

Con la asignación de variables:

pulsa	Bool	%E124.0
luz	Bool	%A124.0
reset	Bool	%E124.7
valor	Word	%MW0
inicio	Word	%MW2
Tag_1	Counter	%Z0
memoria	Word	%MW4
set	Bool	%E124.6
Tag_2	Bool	%A124.7
new_valor	Int	%MW6

OPERACIONES DIGITALES

En este primer nivel de programación se va trabajar con cuatro elementos de ocupación de memoria, cuyos direccionamientos son los siguientes:

bit	**E 0.0**	**ej.: entrada**
byte	**MB 100**	**ej.: marca**
word	**MW 200**	**ej.: marca**
doble word	**MD 220**	**ej.: marca**

Existe otro tipo de ocupación de memoria como es el puntero y que se verá en el segundo nivel de programación. Con este elemento se puede ocupar la cantidad de memoria que se desee:

*** P # M 0.0 BYTE 100**

El puntero anterior indica que desde el **bit M 0.0** se ocupa una cantidad de memoria de **100 bytes**.

Es conveniente volver a insistir en el mapa de ocupación de memoria y es que hay que controlar en todo momento dónde se ubican **marcas**, **entradas** y **salidas,** ya que como la ocupación de estos elementos puede ser diferente puede darse el caso de que se “pisen” unos a otros.

Por ejemplo, supóngase que se tienen las marcas:

M 0.6
M B 0
MW 2
MD 3

Según el ejemplo anterior, la marca **MB 0** “pisaría” a la **M 0.6**, y la marca **MD 3** “pisaría” a la **MW 2**.

L y T obedecen a las palabras *load* y *transfer* que se utilizan para cargar información de una zona de memoria y transferirla a otra. Estas zonas bien podrían ser cargar de una entrada (E) y transferir a información a una salida (A).

Para ver su importancia, supóngase que se desea transferir a una salida de tipo *byte* la información recibida en una entrada de tipo *byte*.

Parecería que lo lógico, según se ha visto con la codificación AWL, sería escribir:

```
U     E W    124
=     AW     124
```

Según lo anterior, la información en la palabra de entrada ubicada en el *byte* de entrada 124 pasaría a la salida ubicada en el byte de salida *124*. Sin embargo, esto no es válido en programación puesto que los códigos anteriores sólo sirven para operaciones con *bits*.

Por lo anterior, para poder transferir la información desde una palabra de entrada a otra de salida habrá que utilizar el código:

```
L     E W    124
T     AW     124
```

La carga de información se lleva a cabo utilizando una nueva zona de memoria: los registros **ACU1** y **ACU2** (pueden interpretarse como acumulador 1 y 2). En el S7-300 sólo existen estos dos registros y en autómatas más avanzadas como el S7-400 existen tres registros.

Los registros son zonas de memoria de la CPU diferentes a las E, A, M, DB y L-STACK, y funcionan de la siguiente manera:

Cuando un dato se carga mediante L el dato pasa automáticamente a ACU1 permaneciendo ACU2 vacío de información. Si, seguidamente, se carga otro dato con L, la información que había en ACU1 pasa a ACU2 y el nuevo dato pasaría a ACU1.

La carga de un byte en un área de memoria es muy sencillo en SCL:

"byte de salida":="byte de entrada";

Con la asignación de variables:

byte de entrada	Byte	%EB0
byte de salida	Byte	%AB0

Según la sentencia anterior, cada vez que se lea desde OB1 al byte AB0 le será asignado el valor del byte EB0.

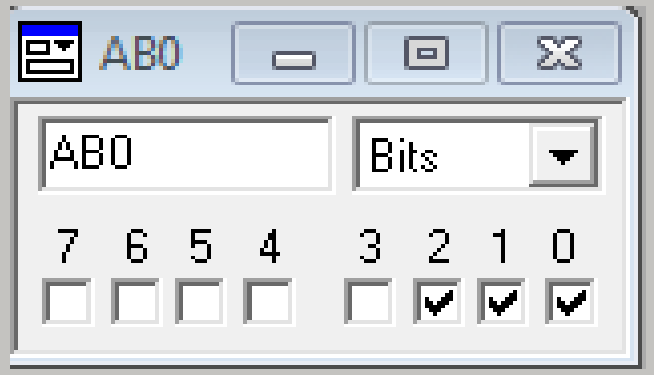
También resulta muy sencillo asignar a un byte un valor entero:

"byte de salida":=7;

Con la asignación de variables:

byte de salida	Byte	%AB0

Que desde el simulador de S7 podría verse como:



Siempre que se ejecute una transferencia mediante T, el dato transferido será el que ocupe en ese momento ACU1.

Para entender lo anterior tómense las siguientes líneas de código:

```
L     7
L     8
L     9
T     MW    0
```

La interpretación sería la siguiente:

- Se carga el valor entero **7 en ACU1**.
- El valor entero **7 pasa a ACU2** y **en ACU1 se carga** el valor entero **8**.
- El valor entero **8 pasa** a **ACU2** y **en ACU1 se carga** el valor entero **9**. El valor **7 se pierde**.
- El valor entero **9** (ocupa ACU1) **es transferido** a la **marca 0** de tipo **word** (un entero ocupa 2 *bytes*).

Los tipos de datos que se utilizan en progamación de PLC's se resumen en la tabla siguiente:

Tipo	Código Prog.	Rango	Ocup. mem.
BCD	16#F385	-999 +999	WORD (2 bytes)
INT	-385	-32768 +32767	WORD
DINT	L#100000	$-2000 \cdot 10^6$ $+2000 \cdot 10^6$	DWORD (4 bytes)
REAL	-385.00	$-1.17 \cdot 10^{-38}$ $3.4 \cdot 10^{38}$	DWORD (4 bytes)

Por lo que se ha comentado, las funciones de **L** (*load*) y **T** (*transfer*) no se utilizarán en SCL, pudiéndonos olvidar del control de los registros ACU1 y ACU2.

Por otro lado, lo que sí hay que controlar es la asignación de datos a los distintos tipos de variable ya que el área de memoria donde se va a guardar el dato ha de ser coherente con el tipo de variable utilizada.
Por ejemplo, si declaramos un área de memoria de tipo **BYTE** para la variable **x**, no podremos asignar a **x** un dato de tipo real.

La carga de los diferentes tipos de datos se lleva a cabo mediante las siguiente sentencias ejemplo:

```
L     +5                  // valor int. de 16 bit (INT)
L     L#523123            // valor int. de  32 bit (DINT)
L     B#16#EF             // byte en formato hexadecimal
L     W#16#12BE           // palabra en formato hexadecimal
L     2#0011011011110001  // valor binario de 16 bit
L     3.14159             // valor real
```

Respecto a la tabla anterior es necesario hacer las siguientes consideraciones:

- En la codificación BCD cada dígito de un numero decimal es codificado mediante 4 posiciones de *bit* (el número más alto decimal que es el 9 necesita cuatro *bits*) y el signo necesita también otros cuatro *bits*. Por lo tanto un número en BCD requiere de 16 *bits*.

- Cuando se quiera obtener una información en BCD, también hay que escribirla en BCD.

- Por ejemplo, para escribir el número decimal -385 en BCD habrá que poner: **16#F385**. En el ejemplo anterior, el **16** indica el número de *bits,* la letra **F** el signo (-), y **385** el número decimal. Si el número hubiese sido positivo, entonces no se introduciría la letra F.

- El tipo de dato INT también ocupa 16 *bits*, reservándose el *bit* 15 para el signo (0 si es positivo y 1 si es negativo).

- El tipo DINT, o **doble entero**, es el que se denomina en otros lenguajes como **entero largo**. Su ocupación en memoria es de 32 *bits*.

- El tipo REAL, también denominado de **coma flotante**, al igual que los DINT, ocupa 32 bits. Su *bit* más significativo indica el **signo** y los *bits* restantes la **mantisa** y el **exponente en base 2**. Este tipo de números se introduce separando la parte decimal con un punto. En la programadora (PG) estos números se representan mediante notación exponencial.

Con los tres tipos de datos anteriores se pueden realizar las siguientes operaciones aritméticas y lógicas:

INT:

+I, *I, /I, -I, ==I, <>I, <I, >I, <=I, >=I

DINT:

+D, *D, /D, -D, ==D, <>D, <D, >D, <=D, >=D, MOD

REAL:

+R, *R, /R, -R, ==R, <>R, <R, >R, <=R, >=R

Para entender un poco mejor como se operaría con números reales, las líneas de programa necesarias para sumar los números reales 4 y 5 serían:

```
L     4.0
L     5.0
+     R
T     MD    0
```

Que se interpretarían de la siguiente forma:

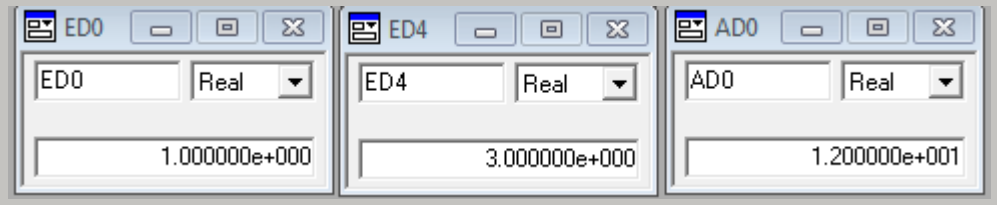
- Carga el número de coma flotante 4.0 en ACU1
- Pasa el valor de ACU1 a ACU2 y carga 5.0 en ACU1
- Suma los valores reales de ACU1 y ACU2
- Transfiere el valor de la suma a una zona de memoria M (marca) con ocupación D (doble palabra).

Al ser el lenguaje SCL un lenguaje de alto nivel, la programación de operaciones aritméticas es realmente sencilla.
Por ejemplo: imaginemos que queremos sumar 8 al valor de la suma de dos números reales desde las áreas de memoria ED0 y ED4 y el resultado total pasarlo al área AD0:

"resultado":=8+ "sumando1"+"sumando2";

Con la asignación de variables:

sumando1	Real	%ED0
sumando2	Real	%ED4
resultado	Real	%AD0



Los saltos son elementos de programación que tienen el efecto similiar a la instrucción **GO TO** en los lenguajes de programación avanzada.

Los saltos se ejecutan bajo determinadas condiciones: cuando se produce un resultado lógico, se cumple una condición lógica determinada, etc.

La sintaxis que se utiliza para los saltos en programacion AWL es la siguiente:

SPA (incondicional)
SPB (condicional con RLO=1)
SPBN (condicional con RLO=0)

Fin de ciclo
BEA (incondicional)
BEB (condicional)

Pongamos un ejemplo para ilustrar todo esto.

Ejemplo: Elaborar un programa en lenguaje AWL que realice la suma de los números enteros 2 y 5 cuando se active un interruptor dado, y el producto de esos dos números cuando se active otro diferente.

```
U       E       124.0
SPB     OP0
U       E       124.1
SPB     OP1
BEA
```

```
OP0:    L       2
        L       5
        +I
        T       MW      124
BEA
```

```
OP1:    L       2
        L       5
        *I
        T       MW      126
BEA
```

En el programa anterior existe un cuerpo de programa (definido mediante un recuadro en rojo) y dos subrutinas OP0 y OP1 recuadradas en color azul.

Las llamadas a las subrutinas (saltos) se realizan mediante las sentencias SPB y se ejecutarán siempre que el resultado lógico de la operación anterior (RLO) sea un 1.

Las sentencias BEA en cada una de las subrutinas indica que ha finalizado su ejecución y tiene el mismo efecto que la sentencia ***break*** en, por ejemplo, lenguaje C.

La sentencia BEA en el programa principal indica también el fin del programa.

En lenguaje SCL los saltos pueden ejecutarse mediante la sentencia:

goto

```
IF (E124.0) THEN
    GOTO ETIQ1;
ELSE
      A124.0:=0;
END_IF;
//.....................
ETIQ1 : A124.0:= 1;
```

En lenguaje SCL es aconsejable **NO** utilizar la sentencia GOTO ya que va en contra de los principios de la programación estructurada.

Por otro lado, GOTO no se debe utilizar en bucles.

CONTADORES

Los contadores son elementos de programación que se utilizan en el contaje de variables que intervienen en los procesos industriales.

Existen contadores incrementadores, decrementadores y un tipo mixto que realiza operaciones tanto de incremento como de decremento. Este último tipo es el que se va a utilizar en este curso de iniciación, y se denomina ***ZAEHLER*** (**Z**).

Los contadores pueden parametrizarse y para que realicen las operaciones de contaje deseadas (incremento o decremento) hay que introducir el valor lógico 1 en determinadas entradas. Los valores de salida (contaje) pueden ser cargados en memoria en formato DUAL o BCD.

CONTADORES

En lenguaje AWL la programación de un contador **Z0** sigue la sintaxis:

```
U     E     124.0   // incrementa
ZV    Z0
U     E     124.1   // decrementa
ZR    Z0
U     E     124.2   // reset
R     Z0
U     E     124.3   // intro valor precarga
L     C#10          // carga valor precarga
S     Z0
L     Z0            // carga salida DUAL
LC    Z0            // carga salida BCD
```

El valor de salida DUAL posee codificación hexadecimal pero luego puede ser tratada como un valor entero.

Los valores de salida han de ser transferidos a marcas de memoria.

En programación mediante SCL, al igual que en los otros lenguajes de programación, se pueden implementar tres tipos de funciones para contadores:

- **S_CU** (ascendente)
- **S_CD** (descendente)
- **S_CUD** (ascendente-descendente)

Los tipos de datos para los parámetros de los contadores ASCENDENTES-DESCENDENTES son:

Parte / Parámetro	Declaración	Tipo de datos	Descripción
CU	Input	BOOL	Entrada de contaje ascendente
CD	Input	BOOL	Entrada de contaje descendente
R	Input	BOOL	Entrada de reset
LD	Input	BOOL	Entrada de carga
PV	Input	Enteros	Valor con el que se activa la salida QU / QD.
QU	Output	BOOL	Estado del contador ascendente
QD	Output	BOOL	Estado del contador descendente
CV	Output	Enteros	Valor actual de contaje

En lenguaje SCL los contadores son muy sencillos de implementar. Con el siguiente ejemplo se pretende programar un contador integrador (S_CU), y que también se pueda inicializar a cero cada vez que así se desee. En el ejemplo consideraremos un bloque contador CTU (CEI).

```
"IEC_Counter_0_DB".CTU(CU:="incrementa",
                       R:="cero",
                       PV:=0,
                       Q=>A126.0,
                       CV=>"pantalla");
```

incrementa	Bool	%E124.0
cero	Bool	%E124.7
pantalla	Int	%AW124

Según el ejemplo anterior, es importante señalar, que cuando se hace una llamada a un contador hay que declarar una base de datos ("IEC_Counter_0_DB“) como zona de memoria donde se irá registrando el contaje. La base de datos hay que cargarla en la CPU para poder realizar el contaje.

TEMPORIZADORES

Los temporizadores (**T**) son elementos de programación cuya salida (0 o 1) está en función de la entrada y de la temporización preestablecida.

Existen cinco tipos de temporizadores distintos:

S_IMPULS	**SI**	// se activa durante un tiempo y no tiene memoria
S_VIMP	**SV**	// idem que SI pero con memoria
S_EVERZ	**SE**	// la salida se activa una vez transcurrido el tiempo // no tiene memoria
S_SEVERZ	**SS**	// idem que anterior pero con memoria
S_AVERZ	**SA**	// idem a SV pero se activa con flanco negativo (**FN**)

TEMPORIZADORES

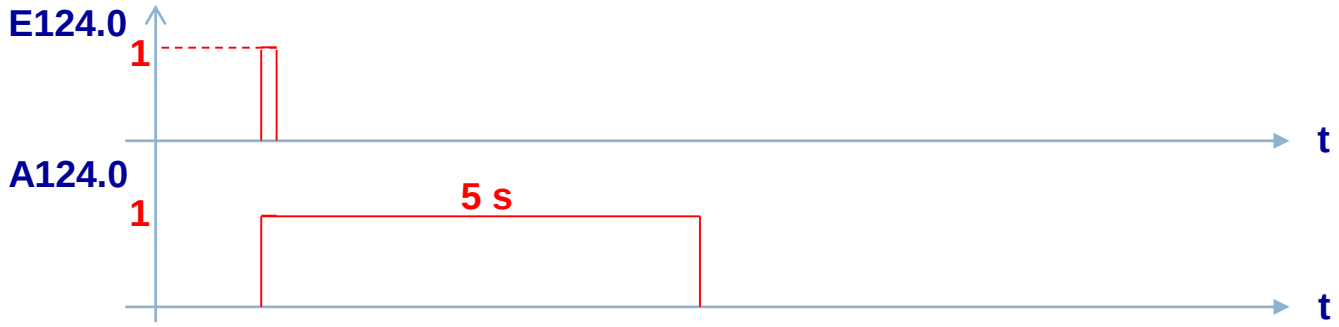
La sintaxis de los temporizadores en lenguaje AWL es la siguiente:

```
U       E       124.0   // cuando sea 1 la entrada E124.0
L       S5T#5S          // carga el tiempo: 5 s …
SV      T0              // … en el temporizador T0 de tipo SV

U       T0
=       A       124.0   // la salida de T0 se carga en A124.0

U       E       124.1   // cuando sea 1 la entrada E124.1
R       T0              // … reset T0
```

El temporizador anterior funcionaría de la siguiente forma:



A las funciones de temporización se llama de igual forma que a las funciones de contaje. La identificación de función puede utilizarse en cualquier expresión en lugar de un operando siempre que el tipo del resultado de la función sea compatible con el del operando sustituido.

Nombre de la función	Significado
S_PULSE	Impulso (Pulse)
S_PEXT	Impulso prolongado (Pulse Extended)
S_ODT	Retardo a la conexión (On Delay Time)
S_ODTS	Retardo a la conexión con memoria (Stored On Delay Time)
S_OFFDT	Retardo a la desconexión (Off Delay Time)

Los parámetros y sus tipos de dato que deben especificarse en las llamadas están explicados en la descripción de cada una de las funciones estándar de temporización. Son los siguientes:

Parámetro	Tipo de dato	Descripción
T_NO	TIMER	Identificador del temporizador; el área depende de la CPU
S	BOOL	Entrada inicial
TV	S5TIME	Predefinición del valor de temporización (formato BCD)
R	BOOL	Entrada de inicialización
Q	BOOL	Estado del temporizador
BI	WORD	Valor de temporización residual (binario)

A las funciones de temporización se las llama de igual forma que a las funciones de contaje. En el siguiente ejemplo utilizaremos un temporizador de tipo IMPULSO PROLONGADO mediante la función de temporización S_PEXT (SIMATIC). La función entregará un valor de tipo 5STIME a una variable de tipo WORD previamente definida (tiempo_actual):

```
"tiempo_actual":=S_PEXT(T_NO:="T0“,
                        S:="start",
                        TV:=S5T#5S,
                        R:=FALSE,
                        BI=>"valor binario“,
                        Q=>"salida temporizada");
```

Con la asignación de variables:

start	Bool	%E124.0
salida temporizada	Bool	%A124.0
T0	Timer	%T0
valor binario	Word	%AW200
tiempo_actual	S5Time	%AW202

Antes de finalizar las notas sobre temporizadores y contadores es necesario insistir en algunos detalles.

A la hora de programar contadores y temporizadores en TIA PORTAL disponemos de dos tipos de elementos de programación:

1. BLOQUES de datos previamente definidos: CTU, CTD y CTUD para contadores, y TP, TON, TOF para temporizadores, cuya denominación aparece en el programa como CEI. Estos bloques hay que compilarlos y cargarlos en la CPU a la hora de ejecutar el programa.
2. FUNCIONES definidas en el programa: S_CU, S_CD y S_CUD para contadores, y S_PULSE, S_PEXT, S_ODT, S_ODTS y S_OFFDT para temporizadores, todos ellas con denominación SIMATIC. El valor que entregan estas funciones hay que asignarle a una variable de usuario previamente definida y del tipo que requiera la función. Estas funciones se llaman desde MAIN y se compilan de forma automática al compilar el bloque OB1 (*main*).

COMPARADORES

Los comparadores son elementos de programación que ofrecen un resultado lógico (0 o 1) tras realizar una comparación entre dos valores.

Su sintaxis es muy sencilla y sólo necesitan de la carga de dos valores y una salida digital tipo ***bit*** donde cargar el valor lógico de la comparación:

```
L     MW    0      // carga el valor entero de la marca MW 0
L     MW    2      // carga el valor entero de la marca MW 2
> I                // compara los dos valores por orden
=     A     124.0  // el valor lógico de la comparación lo
                     introduce en la salida A124.0
```

SEÑALES ANALÓGICAS

En numerosas ocasiones los autómatas trabajan con señales analógicas provenientes de los diferentes sensores, y que hay que proporcionar a los actuadores, de que disponen los sistemas de control industriales modernos.

Por lo tanto, es necesario saber tratar estas señales y conocer los elementos de programación necesarios para poder trabajar con este tipo de información.

En primer lugar hay que decir que el autómata ha de incorporar los necesarios **módulos analógicos** de **entrada** y/o **salida** según requiera la aplicación y control a realizar. A los módulos ha de entrar una señal de tensión normalizada que será proporcional a la variable de campo que se desea medir.

Para poder medir de un sensor o para poder enviar a un actuador señales analógicas se utilizan, respectivamente, las sentencias:

SEÑALES ANALÓGICAS

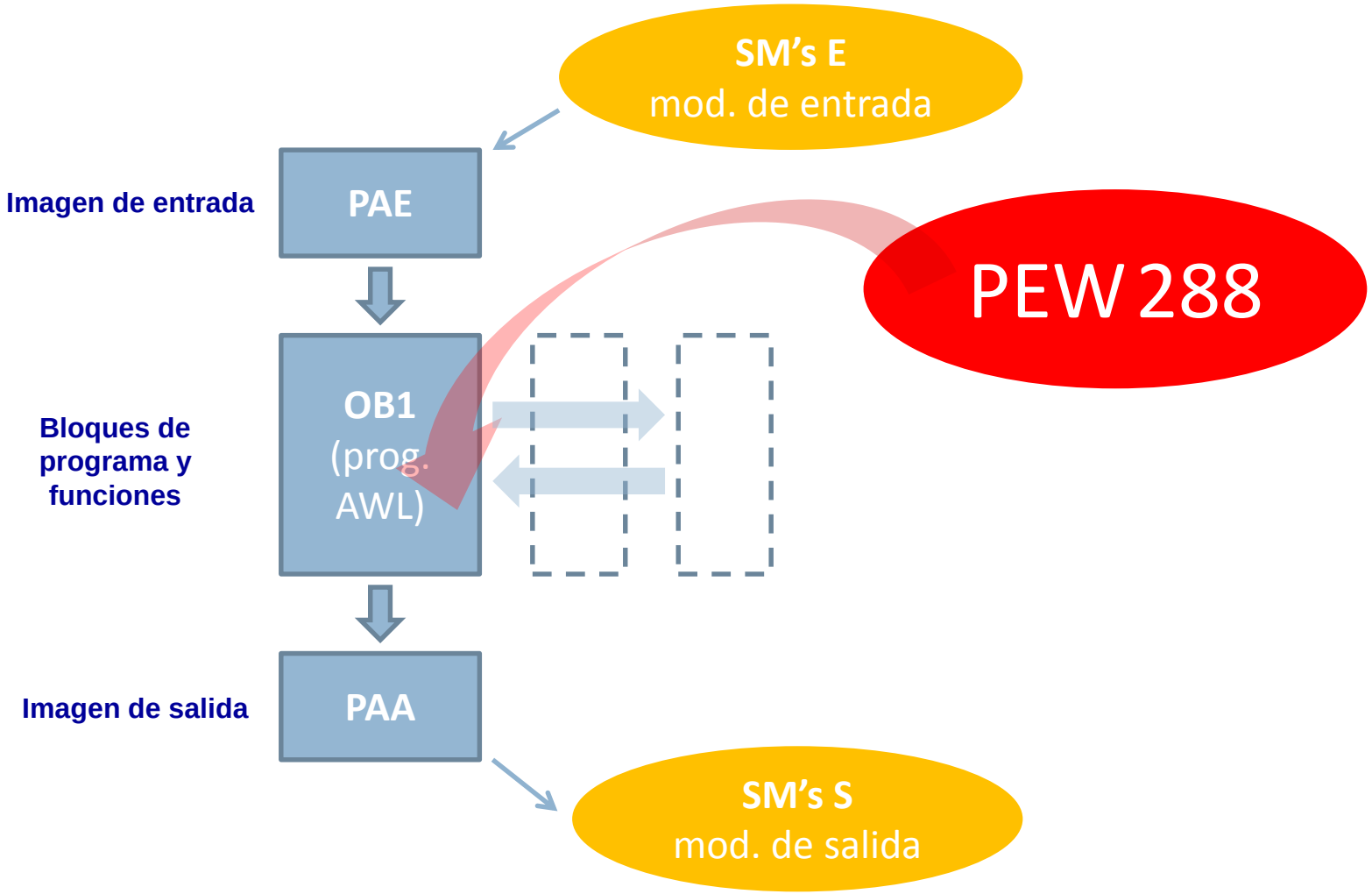
```
L     PEW   288      // carga el valor analógico de
                     // tensión de la entrada (tipo W
                     // , 2 bytes) del byte de
                     // direccionamiento 288
y

T     PAW   288      // transfiere el valor analógico de
                     // tensión a la salida (tipo W
                     // , 2 bytes) del byte de
                     // direccionamiento 288
```

La letra **P** de las siglas **PEW** y **PAW** significa **acceso a la periferia**, lo que quiere decir que cuando aparecen estas siglas en una línea de programa el automáta leerá directamente de la dirección de memoria indicada **SIN HACER CASO A LA IMAGEN DE ENTRADA**.

SEÑALES ANALÓGICAS

Para entender esto último más fácilmente es necesario recurrir al siguiente esquema ilustrativo:



SEÑALES ANALÓGICAS

Cuando se trabaja con señales analógicas es muy frecuente utilizar funciones de programa (**FC's**) como es el caso de la **FC105** (escalado) y la **FC106** (desescalado).

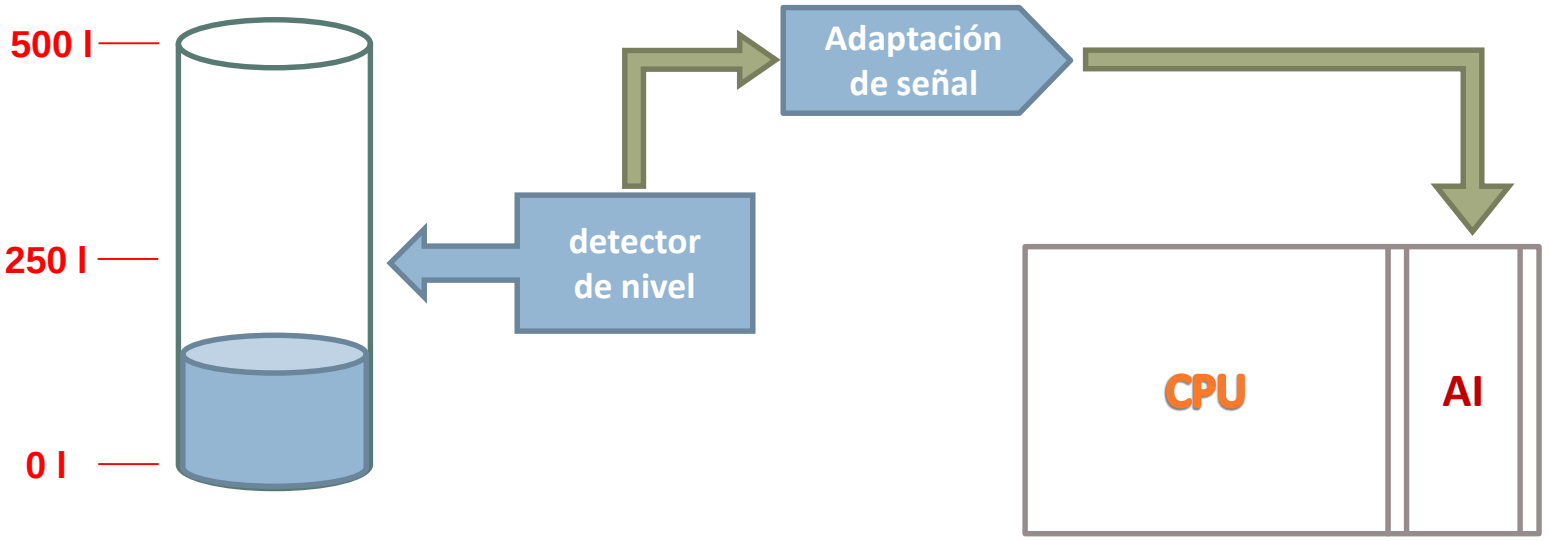
Para poder utilizar estas funciones cuando son invocadas desde **OB1**, dichas funciones han de ser cargadas en la carpeta de programa y posteriormente compiladas.

Mediante el ejemplo práctico de desarrollará un sencillo programa para esclarecer estos conceptos básicos sobre señales analógicas.

Programa ejemplo:

Se desea desarrollar un programa en lenguaje AWL para detectar el nivel medio de llenado de un depósito cilíndrico de 500 litros. La detección se realizará mediante una salida digital de tipo ***bit***.

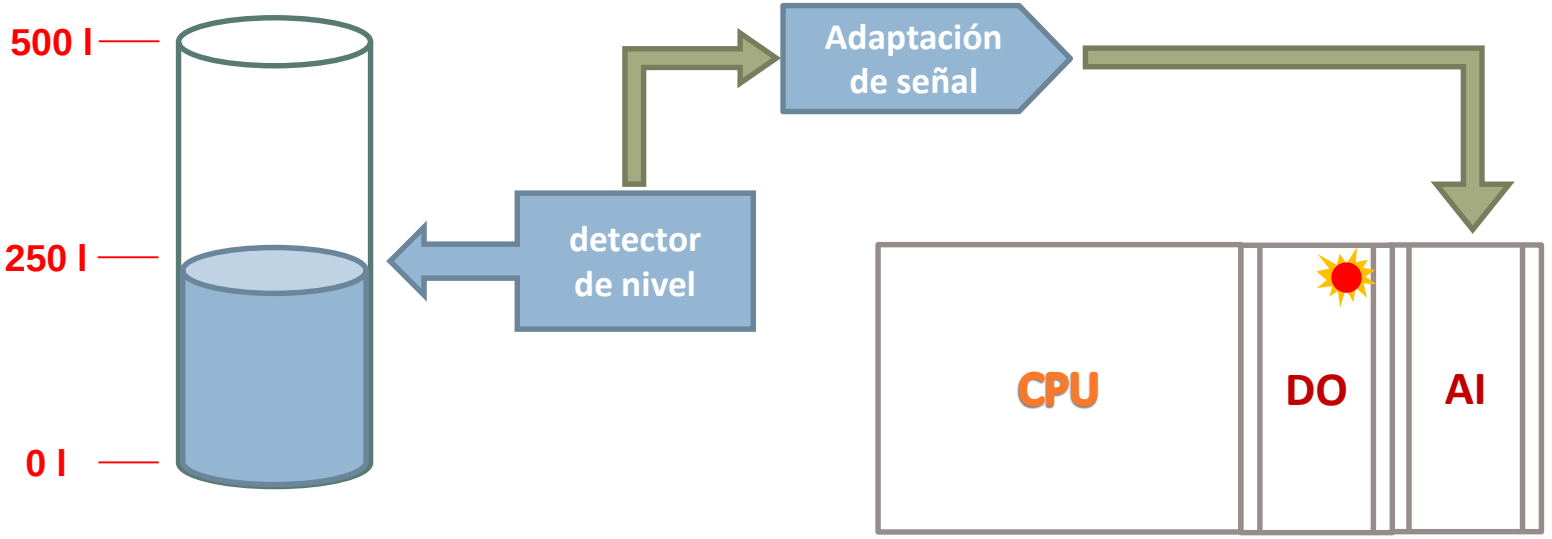
El montaje del sistema seria el siguiente:



SEÑALES ANALÓGICAS

y su funcionamiento:

Según el depósito se va llenando, el detector de nivel envía una señal en tensión proporcional a la altura del líquido. Esta señal ha de ser adaptada para que pueda ser convertida en digital en la PLC.
Cuando se alcanzan los 250 l una salida digital (*led* rojo) advertirá de que se ha llegado al nivel indicado.



SEÑALES ANALÓGICAS

Así pues, en la adaptación de señales y el escalado se tendrán en cuenta los siguientes valores:

Capacidad (m^3)	AI	CPU	FC105 *(scale)*
500	10	27648	500
250	5	13824	-
0	0	0	0

Bajo estas premisas, y cargada la función FC105 junto a OB1 (ambas en la carpeta de bloques) y compiladas, el programa en lenguaje AWL en OB1 sería el siguiente:

SEÑALES ANALÓGICAS

```
call      "SCALE"                 // llama a la función de escalado FC105
IN        :=      PEW288        // acceso a la perifieria
HI_LIM    :=      500.0         // valor máximo de escalado
LO_LIM    :=      0.0           // valor mímino de escalado
BIPOLAR:=         FALSE         // porque la tensión es postiva de 0 a 10 V
RET_VAL:=         MW0           // activa una zona de memoria
OUT       :=      MD2           // introduce salida  real escalada en MD2
```

```
L         250.0                 // carga en ACU1 el valor real 250.0
L         MD2                   // carga en ACU1 el valor real de MD2
                                // y pasa a ACU2 el valor 250.0
>=R                             // cuando MD2>=250.0 …
=         A       124.0         // … activa la salida A124.0
```

Para poder tratar las señales analógicas de entrada y salida en programación SCL también se utilizan las funciones ***SCALE*** y ***UNSCALE***. Sus sintanxis son las siguientes:

SCALE

```
RET_VAL := SCALE(IN:=_in_,
                 HI_LIM:=_in_,
                 LO_LIM:=_in_,
                 BIPOLAR:=_in_,
                 OUT=>_out_);
```

Donde:

Parámetro	Declaración	Tipo de datos	Área de memoria	Descripción
IN	Input	INT	I, Q, M, D, L, P o constante	Valor de entrada que se escala.
HI_LIM	Input	REAL	I, Q, M, D, L, P o constante	Límite superior
LO_LIM	Input	REAL	I, Q, M, D, L, P o constante	Límite inferior
BIPOLAR	Input	BOOL	I, Q, M, D, L	Indica si el valor del parámetro IN se interpreta como bipolar o unipolar. El parámetro puede adoptar los valores siguientes: 1: Bipolar 0: Unipolar
RET_VAL	Output	WORD	I, Q, M, D, L, P	Información de error
OUT	Output	REAL	I, Q, M, D, L, P	Resultado de la instrucción

UNSCALE

```
RET_VAL := UNSCALE(IN:=_in_,
                   HI_LIM:=_in_,
                   LO_LIM:=_in_,
                   BIPOLAR:=_in_,
                   OUT=>_out_);
```

Donde:

Parámetro	Declaración	Tipo de datos	Área de memoria	Descripción
IN	Input	REAL	I, Q, M, D, L, P o constante	Valor de entrada que se desescala en un valor entero.
HI_LIM	Input	REAL	I, Q, M, D, L, P o constante	Límite superior
LO_LIM	Input	REAL	I, Q, M, D, L, P o constante	Límite inferior
BIPOLAR	Input	BOOL	I, Q, M, D, L	Indica si el valor del parámetro IN se interpreta como bipolar o unipolar. El parámetro puede adoptar los valores siguientes: 1: Bipolar 0: Unipolar
OUT	Output	INT	I, Q, M, D, L, P	Resultado de la instrucción
RET_VAL	Output	WORD	I, Q, M, D, L, P	Información de error

Es conveniente volver a recordar que para ejecutar correctamente el programa en el que se llama a estas funciones es necesario compilar y cargar las funciones junto a ***main*** (OB1).