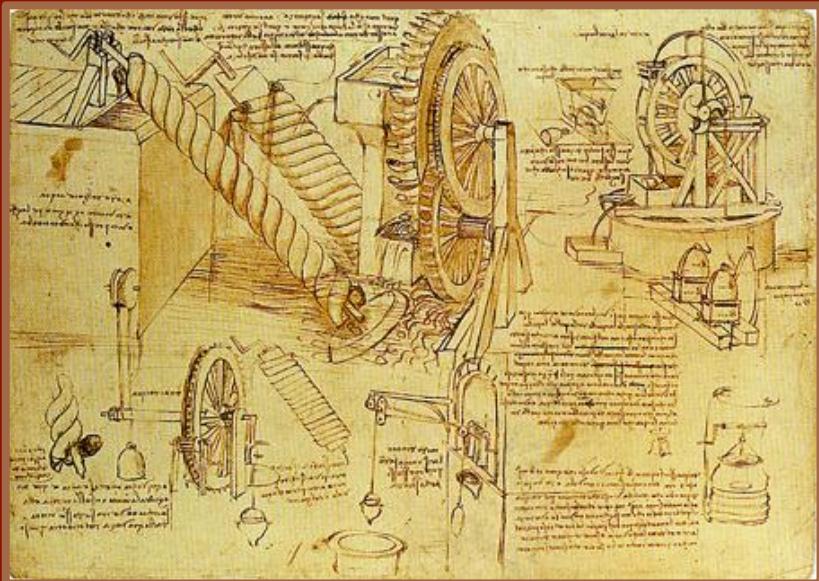


# Introducción a la Programación de Autómatas Programables usando CoDeSys

Universidad de Sevilla



Miguel Ángel Ridaó Carlini

Escuela Técnica Superior de Ingeniería  
Universidad de Sevilla









# Introducción a la Programación de Autómatas Programables usando CoDeSys

Miguel Ángel Ridao Carlini

© 2014 Miguel Ángel Ridao Carlini.

Todos los derechos reservados

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Departamento de Ingeniería de Sistemas y Automática

Camino de los descubrimientos s/n

41092 Sevilla

Spain

Imprimido en Sevilla, el X del X del 20XX

Registro de la propiedad intelectual *Poner número aquí*

Primera edición. Versión 1.0. Junio 2012. Compilado el 1 de octubre de 2012 a las 12:34 p.m.

**No está permitida la distribución de este texto, a terceras personas, por cualquier medio. El uso de este texto y formato, en fichero electrónico, está restringido al personal y alumnado de la Escuela Técnica Superior de Ingeniería.**

Diseño de portada: Fernando García García

Imagen central de portada: Sistema de riego de Leonardo da Vinci.

*A* ....



# Agradecimientos

---

Los estilos adoptados por nuestra Escuela y utilizada en este texto es una versión y adaptación a Word® del la versión L<sup>A</sup>T<sub>E</sub>X que el Prof. Payán realizó para un libro que desde hace tiempo viene escribiendo para su asignatura. Por otro lado, la adaptación se hizo sobre un formato que el prof. Aguilera arregló, basándose en su tesis doctoral. Su aportación ha sido muy relevante para que este formato vea la luz. Esta adaptación la llevamos a cabo el alumno Silvio Fernández, becario del Centro de Cálculo, y yo mismo, sobre un trabajo preliminar del alumno Julián José Pérez Arias.

A esta hoja de estilos se le incluyó unos nuevos diseños de portada. El diseño gráfico de las portadas para proyectos fin de grado, carrera y máster, está basado en el que el prof. Fernando García García, de la Facultad de Bellas Artes de nuestra Universidad, hiciera para los libros, o tesis, de la sección de publicación de nuestra Escuela. Nuestra Escuela le agradece que pusiera su arte y su trabajo a nuestra disposición.

Juan José Murillo Fuentes  
Subdirección de Comunicaciones y Recursos Comunes  
Sevilla, 2013



# **Prólogo**

---

En ....



# Índice

---

Agradecimientos	iii
Prólogo	v
Índice	vii
Índice de Figuras	xv
Índice de Tablas	xxi
<b>1. Autómatas Programables</b>	<b>23</b>
<i>1.1. Automatización y Control</i>	23
<i>1.2. Sensores y actuadores</i>	26
1.2.1. Sensores y otras entradas al sistema de control	26
1.2.2. Actuadores y otras salidas del sistema de control	28
<i>1.3. Implementación de Automatismos Lógicos</i>	30
<i>1.4. Definición de autómata programable</i>	31
<i>1.5. Estructura externa de un autómata programable</i>	32
<i>1.6. Estructura interna de un autómata programable</i>	33
1.6.1. Memoria	35
1.6.2. Módulos de Entrada/Salida	37
1.6.3. Entornos de programación	39
<i>1.7. Funcionamiento de un autómata programable</i>	39
1.7.1. Ciclo del autómata programable	40
1.7.2. Modos de funcionamiento del autómata programable	42
1.7.3. Modo periódico	43
1.7.4. Otros modos de funcionamiento	44
<b>2. Introducción a la Norma IEC-61131</b>	<b>47</b>
<i>2.1. Elementos básicos en la norma IEC-61131-3</i>	49
<i>2.2. Program Organization Units (POU)</i>	

<i>2.3. Variables</i>	50
<i>2.4. Lenguajes de programación</i>	52
<b>3. Primeros pasos con CoDeSys</b>	<b>55</b>
<i>3.1. Algunos conceptos básicos en CoDeSys</i>	55
<i>3.2. Descarga del software CoDeSys</i>	56
<i>3.3. Ejecución de CoDeSys</i>	57
<i>3.4. Pantalla principal</i>	59
<i>3.5. Primer programa en lógica de contactos</i>	60
<i>3.6. Ejemplo 2. Puesta en marcha y parada de un motor</i>	63
<i>3.7. Ejemplo 3. Cambio del sentido de giro de un motor</i>	65
<i>3.8. Ejemplo 4. Control de un cilindro neumático (I)</i>	69
<i>3.9. Ejemplo 5. Control de un cilindro neumático (II)</i>	71
<i>3.10. Ejemplo 6. Control del nivel de un depósito</i>	72
<b>4. Tipos de Datos y Variables</b>	<b>75</b>
<i>4.1. Tipos de datos</i>	75
4.1.1. Tipos de datos elementales	75
4.1.2. Representación de datos	76
4.1.3. Tipos de datos derivados	77
4.1.4. Tipos de datos genéricos	78
<i>4.2. Variables</i>	79
4.2.1. Atributos	80
4.2.2. Variables de Entrada/Salida	81
4.2.3. Valores iniciales	83
4.2.4. Matrices y estructuras	83
<i>4.3. Definición de tipos de datos derivados en CoDeSys</i>	85
<b>5. Unidades de Organización de Programas (POU)</b>	<b>87</b>
<i>5.1. Tipos de POUs</i>	87
<i>5.2. Declaración de variables en CoDeSys</i>	89
<i>5.3. Bloques funcionales</i>	90
5.3.1. Estructura de datos asociada a un bloque functional	93
5.3.2. Llamada al bloque functional	93
5.3.3. Restricciones en el uso de bloques funcionales	96
<i>5.4. Funciones</i>	96
5.4.1. Diferencia en el funcionamiento entre funciones y bloques funcionales	97

5.4.2.	Restricciones en el uso de funciones	98
5.5.	<i>Programas</i>	98
5.6.	<i>Creación de POU en CoDeSys</i>	98
5.7.	<i>Funciones y bloques funcionales estándar</i>	99
5.7.1.	Algunas funciones estándar	100
5.7.2.	Algunos bloques funcionales estándar	104
5.8.	<i>Ejercicios</i>	105
5.8.1.	Bloque funcional para Marcha-Parada de un motor	105
5.8.2.	Bloque funcional Contador	105
5.8.3.	Bloque funcional de detección de flancos	107
<b>6.</b>	<b>Programación en LD</b>	<b>109</b>
6.1.	<i>Contactos y bobinas</i>	110
6.1.1.	Ejemplos	112
6.2.	<i>Bloques funcionales</i>	113
6.2.1.	Ejemplos	114
6.3.	<i>Entrada EN en funciones y bloques funcionales</i>	115
6.3.1.	Ejemplos	116
6.4.	<i>Saltos</i>	118
6.4.1.	Ejemplo	118
6.5.	<i>Editor de LD en CoDeSys</i>	120
6.5.1.	Añadir nuevas redes	120
6.5.2.	Edición de contactos y bobinas	121
6.5.3.	Edición de bloques funcionales	122
6.5.4.	Edición de saltos en CoDeSys	123
6.6.	<i>Ejemplo</i>	123
6.7.	<i>Ejercicios</i>	127
6.7.1.	Ejercicio 1	127
6.7.2.	Ejercicio 2	127
6.7.3.	Ejercicio 3	127
<b>7.</b>	<b>Programación en IL</b>	<b>131</b>
7.1.	<i>Operadores básicos y llamadas a funciones</i>	133
7.2.	<i>Uso de funciones y bloques funcionales</i>	137
7.2.1.	Llamada a funciones	137
7.2.2.	Llamada a bloques funcionales	138
7.3.	<i>Saltos</i>	141
7.4.	<i>Ejercicios</i>	141

7.4.1.	Ejercicio 1	141
7.4.2.	Ejercicio 2	142
7.4.3.	Ejercicio 3	143
<b>8.</b>	<b>Temporizadores y Contadores</b>	<b>145</b>
<i>8.1.</i>	<i>Temporizadores</i>	<i>145</i>
8.1.1.	Temporizador de connexion (TON)	146
8.1.2.	Temporizador de Tiempo Fijo (TP)	147
8.1.3.	Temporizador de desconexión (TOF)	148
8.1.4.	Uso de los temporizadores	149
<i>8.2.</i>	<i>Contadores</i>	<i>150</i>
8.2.1.	Contador de incremento (CTU)	150
8.2.2.	Contador de decrement (CTD)	151
8.2.3.	Contador de incremento-decremento	152
8.2.4.	Uso de los contadores	153
<i>8.3.</i>	<i>Ejercicios</i>	<i>153</i>
8.3.1.	Ejercicio 1	153
8.3.2.	Ejercicio 2	154
8.3.3.	Ejercicio 3	154
8.3.4.	Ejercicio 4	155
8.3.5.	Ejercicio 5	155
8.3.6.	Ejercicio 6	157
8.3.7.	Ejercicio 7	158
8.3.8.	Ejercicio 8	160
<b>9.</b>	<b>Programación en ST</b>	<b>163</b>
<i>9.1.</i>	<i>Instrucciones en ST</i>	<i>163</i>
9.1.1.	Asignación	164
9.1.2.	Sentencia IF	164
9.1.3.	Sentencia CASE	164
9.1.4.	Sentencia FOR	165
9.1.5.	Sentencia WHILE	165
9.1.6.	Sentencia REPEAT	165
9.1.7.	Sentencia EXIT	166
<i>9.2.</i>	<i>Uso de funciones y bloques funcionales</i>	<i>166</i>
9.2.1.	Llamadas a funciones	166
9.2.2.	Llamada a bloques funcionales	167
9.2.3.	Sentencia RETURN	168

<b>9.3. Ejercicios</b>	<b>169</b>
9.3.1. Ejercicio 1	169
9.3.2. Ejercicio 2	170
9.3.3. Ejercicio 3	171
<b>10. Programación en FBD</b>	<b>173</b>
10.1. Cómo trabajar con bloques en CoDeSys	174
10.1.1. Posición del cursor gráfico	174
10.1.2. Comandos en FBD	176
10.2. Lenguaje CFC	178
10.3. Ejercicios	179
10.3.1. Ejercicio 1	179
10.3.2. Ejercicio 2	180
10.3.3. Ejercicio 3	182
<b>11. Programación en SFC</b>	<b>185</b>
11.1. Introducción a GRAFCET	186
11.1.1. Etapas	186
11.1.2. Transiciones	187
11.1.3. Arcos orientados	188
11.1.4. Reglas de evolución	189
11.2. Estructuras básicas en GRAFCET	192
11.2.1. Secuencia única	192
11.2.2. Secuencia simultánea	193
11.2.3. Selección de secuencia	193
11.2.4. Salto de etapa	195
11.2.5. Repetición de secuencia	196
11.3. Acciones en GRAFCET	197
11.4. GRAFCET en la norma IEC-1161-3	198
11.4.1. Etapas	199
11.4.2. Transiciones	200
11.4.3. Acciones	201
11.5. GRAFCET en CoDeSys	206
11.5.1. Edición de un GRAFCET	206
11.5.2. Comandos básicos para edición del GRAFCET	208
11.5.3. Edición de transiciones	210
11.5.4. Edición de acciones	212
11.6. Ejercicios	214

11.6.1.	Ejercicio 1	214
11.6.2.	Ejercicio 2	215
11.6.3.	Ejercicio 3	215
11.6.4.	Ejercicio 4	215
11.6.5.	Ejercicio 5	216
11.6.6.	Ejercicio 6	216
11.6.7.	Ejercicio 7	217
11.6.8.	Ejercicio 8	217
11.6.9.	Ejercicio 9	218
11.6.10.	Ejercicio 10	219
11.6.11.	Ejercicio 11	220
11.6.12.	Ejercicio 12	221
<b>12.</b>	<b>Usando Señales Analógicas</b>	<b>225</b>
<i>12.1.</i>	<i>Señales analógicas</i>	225
12.1.1.	Entradas analógicas: del proceso al autómata	226
12.1.2.	Salidas analógicas: del autómata al proceso	228
<i>12.2.</i>	<i>Tratamiento de señales analógicas en el PLC</i>	229
12.2.1.	Cambio de tipo de variable	229
12.2.2.	Escalado de señales analógicas	229
12.2.3.	Ejemplo	232
<i>12.3.</i>	<i>Control de sistemas continuos en un PLC</i>	233
12.3.1.	Controlador P	234
12.3.2.	Controlador PI	238
12.3.3.	Controlador PID	241
<i>12.4.</i>	<i>Bloques funcionales para el control de procesos</i>	242
12.4.1.	Generador de salida PWM	243
12.4.2.	Control de una válvula motorizada	244
<i>12.5.</i>	<i>Ejercicios</i>	248
12.5.1.	Ejercicio 1	248
12.5.2.	Ejercicio 2	250
12.5.3.	Ejercicio 3	251
<b>13.</b>	<b>Problemas</b>	<b>253</b>
<i>13.1.</i>	<i>Cruce con tranvía</i>	253
<i>13.2.</i>	<i>Cruce completo con tranvía</i>	254
<i>13.3.</i>	<i>Automatización con Guía GEMMA</i>	256
13.3.1.	Proceso a controlar	257

13.3.2.	Modos de funcionamiento	259
13.3.3.	Desarrollo del ejercicio	261
13.3.4.	Lista de señales	264
<i>13.4. Control de un ascensor</i>		265
13.4.1.	Lista de señales	266
<i>13.5. Control de dos ascensores simples</i>		267
13.5.1.	Lista de señales	268
<i>13.6. Control de dos ascensores completos</i>		269
13.6.1.	Lista de señales	270
<i>13.7. Control de una mezcladora</i>		271
13.7.1.	Lista de señales	274
<b>Apéndice A</b>	<b>Configuración de CoDeSys</b>	<b>277</b>
<i>A.1. Gestión de librerías</i>		277
<i>A.2. Configuración de tareas</i>		278
<b>Apéndice B</b>	<b>Visualizaciones en CoDeSys</b>	<b>283</b>
<i>B.1. Editor de Visualizaciones</i>		283
<i>B.2. Configuración básica de elementos</i>		284
B.2.1.	Text	286
B.2.2.	Text variables	287
B.2.3.	Line width	288
B.2.4.	Color	288
B.2.5.	Colorvariables	288
B.2.6.	Motion absolute	289
B.2.7.	Motion relative	289
B.2.8.	Variables	289
B.2.9.	Input	290
B.2.10.	Text for tooltip	290
<i>B.3. Configuración avanzada de elementos</i>		291
B.3.1.	Table	291
B.3.2.	Scrollbar	292
B.3.3.	Indicador de aguja	292
B.3.4.	Diagrama de barras	292
B.3.5.	Inclusión de archivos gráficos	292
B.3.6.	Graficas de variables frente al tiempo	292
<b>Referencias</b>		<b>295</b>



# ÍNDICE DE FIGURAS

---

Figura 1-1– Control en lazo abierto.....	24
Figura 1-2– Control en lazo cerrado.....	25
Figura 1-3– a) Sensor de proximidad inductivo AB Elektronik. b) Sensor fotoeléctrico de Siemens c) Interruptor de nivel de WIKA.....	27
Figura 1-4– Elementos de mando de IDEC .....	28
Figura 1-5– Actuadores. a) Válvula solenoide de Danfoss. b) Contactor Siemens. c) Cilindro neumático Festo. ....	29
Figura 1-6– PLC Compacto Mitsubishi FX3G.....	33
Figura 1-7 – PLC Modular Siemens Simatic S7-400.....	34
Figura 1-8 – Arquitectura de un PLC.....	34
Figura 1-9 – Mapa de memoria de un PLC .....	36
Figura 1-10 – Entrada analógica.....	38
Figura 1-11 – Ciclo de funcionamiento de un autómata programable. ....	40
Figura 1-12 – Modo de funcionamiento normal o cíclico.....	43
Figura 1-13 – Modo de funcionamiento periódico. .....	44
Figura 2-1. Ejemplo del lenguaje LD .....	52
Figura 2-2. Ejemplo del lenguaje FBD.....	53
Figura 3-1 – Ventana de selección de la configuración .....	57
Figura 3-2. Ventana de selección de los parámetros de la configuración .....	58
Figura 3-3 . Ventana de creación de una POU .....	58
Figura 3-4 – Pantalla principal en CoDeSys .....	59
Figura 3-5 – Visualización ejercicio <i>Cap3_1.pro</i> .....	60

Figura 3-6 – Ventana de edición del ejercicio <i>Cap3_1.pro</i> .....	61
Figura 3-7 – Botones de edición en lógica de contactos .....	62
Figura 3-8 – Solución del Ejercicio 2 (I) .....	64
Figura 3-9 – Ventana de declaración de variables.....	65
Figura 3-10 – Solución del Ejercicio 2 (II).....	66
Figura 3-11 – Conexionado para cambio de giro del motor .....	67
Figura 3-12 – Acceso a la ventana de variables globales.....	68
Figura 3-13 – Cilindro doble efecto y válvula 5/2 .....	70
Figura 3-14 – Sistema a controlar en Ejemplo 4. ....	71
Figura 3-15 – Sistema a controlar en Ejemplo 5. ....	72
Figura 3-16 – Sistema a controlar en Ejemplo 6. ....	73
Figura 4-1 – Nuevos tipos de datos en CoDeSys.....	86
Figura 4-2 – Ventana de edición de nuevos tipos de datos en CoDeSys. ....	86
Figura 5-1 – Estructura de un Programa Organization Unit (POU). ....	88
Figura 5-2 – Ventana de asistente de declaración de variables. ....	89
Figura 5-3 – Llamada a un bloque funcional en lenguaje FBD. ....	95
Figura 5-4 – Llamada a una función en lenguaje FBD. ....	97
Figura 5-5 – Añadir una nueva POU.....	99
Figura 6-1 – Lógica de contactos. Diagrama LD .....	109
Figura 6-2 – Ejemplo de contactos y bobinas.....	113
Figura 6-3 – Ejemplo de bloque funcional en LD .....	114
Figura 6-4 – Ejemplo de bloque funcional temporizador en LD.....	115
Figura 6-5 – Ejemplo de uso de la entrada EN con un temporizador.....	116
Figura 6-6 – Ejemplo de uso de la entrada EN con funciones .....	117
Figura 6-7 – Ejemplo del uso de saltos en LD .....	119

Figura 6-8 – Asignación incorrecta para activar <i>Luz</i> con <i>A</i> o con <i>B</i> .....	119
Figura 6-9 – Asignación correcta para activar <i>Luz</i> con <i>A</i> o con <i>B</i> .....	120
Figura 6-10 – Sistema a controlar .....	124
Figura 6-11 – Grafo de estados .....	125
Figura 6-12 – Programa en LD .....	126
Figura 6-13 – Visualización del ejercicio 3 .....	128
Figura 7-1 – Visualización del ejercicio 2 .....	142
Figura 7-2 – Visualización del ejercicio 3 .....	144
Figura 8-1 – Cronograma del temporizador TON .....	146
Figura 8-2 – Cronograma del temporizador TP .....	147
Figura 8-3 – Cronograma del temporizador TOF .....	148
Figura 8-4 – Uso de un temporizador TON en lenguaje LD .....	149
Figura 8-5 – Uso de un contador CTUD en lenguaje LD.....	153
Figura 8-6 – Temporización de los semáforos .....	157
Figura 8-7 – Visualización del ejercicio 6. ....	158
Figura 8-8 – Visualización del ejercicio 7. ....	159
Figura 8-9 – Visualización del ejercicio 8. ....	161
Figura 9-1 – Display de siete segmentos .....	169
Figura 9-2 – Visualización del ejercicio 1 .....	170
Figura 10-1 – Ejemplo de programa en FBD en CoDeSys.....	173
Figura 10-2 – Visualización de la válvula todo-nada .....	181
Figura 10-3 – Visualización de la válvula motorizada .....	183
Figura 11-1 – Ejemplo en GRAFCET.....	185
Figura 11-2 – Representación de una etapa .....	186
Figura 11-3 – Transiciones (I) .....	187

Figura 11-4 – Transiciones (II).....	188
Figura 11-5 – Reglas de evolución (I) .....	190
Figura 11-6 – Reglas de evolución (II) .....	191
Figura 11-7 – Reglas de evolución 5 .....	191
Figura 11-8 – Estructura secuencial .....	192
Figura 11-9 – Estructura de secuencia simultanea .....	193
Figura 11-10 – Estructura de selección de secuencia .....	194
Figura 11-11 – Selección exclusiva .....	195
Figura 11-12 – Estructura de salto de etapa .....	196
Figura 11-13 – Repetición de secuencia .....	196
Figura 11-14 – Representación de acciones en GRAFCET .....	197
Figura 11-15 – Representación de una etapa con acción.....	199
Figura 11-16 – Condición lógica escrita en ST en una transición.....	200
Figura 11-17 – Condición lógica asociada al identificador <i>Trans0</i> escrita en LD en una transición .....	201
Figura 11-18 – Acciones de tipo S y R .....	202
Figura 11-19 – La acción asociada es código en FBD que consiste en llamada a un contador .....	203
Figura 11-20 – La acción asociada es código en FBD que consiste en llamada a un temporizador .....	204
Figura 11-21 – Acción asociada escrita en SFC .....	205
Figura 11-22 – SFC inicial en CoDeSys .....	207
Figura 11-23 – Edición SFC. Nombre asociado a ramas en paralelo.....	210
Figura 11-24 – Símbolo de transición cuando tiene asociada una condición.....	211
Figura 11-25 – Añadir acción a un POU tipo SFC .....	213
Figura 11-26 – Acciones asociadas a un POU tipo SFC.....	213

Figura 11-27 – Visualización del ejercicio 6 .....	217
Figura 11-28 – Sistema de pesaje.....	218
Figura 11-29 – Visualización del ejercicio 9 .....	219
Figura 11-30 – Visualización del ejercicio 10 .....	220
Figura 11-31 – Visualización del ejercicio 12 .....	221
Figura 12-1 – Entrada analógica.....	227
Figura 12-2 – Escalado de una señal analógica.....	230
Figura 12-3 – Esquema de control de un proceso continuo desde un PLC.....	234
Figura 12-4 – Esquema de un controlador P .....	235
Figura 12-5 – Aproximación de la integral del error por el método de Euler .....	239
Figura 12-6 – Bloque funciona PID definido en CoDeSys.....	242
Figura 12-7 – Modulación PWM de una señal analógica.....	243
Figura 12-8 – Esquema de la válvula a controlar .....	247
Figura 12-9 – Esquema del proceso a controlar en Ejercicio 1.....	249
Figura 13-1 – Cruce con tranvía .....	254
Figura 13-2 – Cruce completo con tranvía .....	255
Figura 13-3 – Proceso a controlar en Automatización con Guía GEMMA .....	257
Figura 13-4 – Cuadro de control en Automatización con Guía GEMMA .....	258
Figura 13-5 – Modos de funcionamiento .....	260
Figura 13-6 – Ascensor .....	266
Figura 13-7 – Ascensor doble.....	268
Figura 13-8 – Ascensor doble completo .....	270
Figura 13-9 – Mezcladora con Guía GEMMA .....	272
Figura A-1 – Ventana para Gestión de Librerías .....	278
Figura A-2 – Ventanas de configuración de tareas .....	279

Figura B-1 – Ventana de edición de visualizaciones.....286

# ÍNDICE DE TABLAS

---

Tabla 4-1 Sintaxis de las variables de entrada/salida .....	81
Tabla 6-1. Tipos de contactos en CoDeSys.....	110
Tabla 6-2. Tipos de contactos para detección de flancos .....	111
Tabla 6-3. Tipos de bobinas .....	112
Tabla 6-4. Edición de contactos y bobinas .....	121
Tabla 6-5. Edición de bloques funcionales.....	123
Tabla 7-1. Operadores booleanos.....	134
Tabla 7-2. Operadores aritméticos y de comparación.....	136
Tabla 7-3. Operadores de saltos y llamadas a POUs .....	137
Tabla 8-1. Parámetros de entrada y salida en un temporizador .....	146
Tabla 8-2. Parámetros de entrada y salida en un contador de incremento .....	150
Tabla 8-3. Parámetros de entrada y salida en un contador de decremento .....	151
Tabla 8-4. Parámetros de entrada y salida en un contador de incremento-decremento .....	152
Tabla 10-1. Botones en la edición de graficos en FBD .....	178
Tabla 11-1. Edición de un grafo SFC .....	208
Tabla 11-2. Variables de entrada en el ejercicio 12 .....	222
Tabla 11-3. Variables de salida en el ejercicio 12 .....	223
Tabla 12-1. Variables de entrada asociadas a la visualización del ejercicio 1 .....	250
Tabla 13-1. Variables de entrada del cruce completo con tranvía .....	255
Tabla 13-2. Variables de salida del cruce completo con tranvía .....	256
Tabla 13-3. Variables de entrada de Automatización con guía GEMMA .....	264

Tabla 13-4. Variables de salida de Automatización con guía GEMMA .....	265
Tabla 13-5. Variables de entrada del ascensor.....	266
Tabla 13-6. Variables de salida del ascensor .....	267
Tabla 13-7. Variables de salida del ascensor .....	268
Tabla 13-8. Variables de entrada del ascensor doble .....	269
Tabla 13-9. Variables de salida del ascensor doble completo .....	270
Tabla 13-10. Variables de entrada del ascensor doble completo.....	271
Tabla 13-11. Variables de entrada del proceso de mezcla .....	274
Tabla 13-12. Variables de salida del proceso de mezcla.....	275
Tabla B-1. Elementos gráficos en las Visualizaciones.....	285

# 1. AUTÓMATAS PROGRAMABLES

---

**E**n este primer capítulo se presentan las nociones básicas relacionadas con la automatización necesarias para iniciarse en la programación de autómatas programables. El objetivo es recordar o en su caso, acercar al lector a los conceptos básicos que son necesarios para poder entender qué son los autómatas programables, cómo se usan y los procesos que se controlan con ellos. En cualquier caso, es altamente aconsejable que el lector tenga o vaya adquiriendo en paralelo al uso de este libro, un conocimiento de estos sistemas, que puede conseguir en obras que incluyen contenidos de carácter más teórico y tecnológico como [2], [3], [4].

## 1.1. Automatización y Control

La Automatización se puede definir como la utilización de técnicas y equipos para el control de un sistema, normalmente industrial, de tal forma que funcione de forma automática, al menos parcialmente.

Por tanto se habla de un concepto que implica *controlar un sistema o planta*, es decir, hacer que el comportamiento de dicho sistema cumpla unos determinados objetivos, unas especificaciones de funcionamiento, y que lo haga de forma autónoma, sin intervención humana o al menos con una intervención limitada. El elemento que sustituye al operador humano en la toma de decisiones es el *Sistema de Control*.

El comportamiento de un sistema va a estar caracterizado por magnitudes representativas de las que nos interesa su evolución en el tiempo, por ejemplo, temperaturas en un sistema térmico, presiones o caudales en un sistema hidráulico, tensiones o intensidades en un sistema eléctrico, o la posición de la cabina y el estado ON/OFF de los botones en un ascensor. En definitiva, estamos hablando de magnitudes o variables dinámicas, es decir, en las que su comportamiento cambia

con el tiempo.

Controlar un sistema implica actuar sobre él, es decir, determinar en cada momento los valores de un conjunto de variables sobre las que se puede actuar y que deben permiten modificar, y por tanto controlar, el comportamiento de la planta. Este conjunto de variables son las *actuaciones*, *señales de control* o *entradas* del sistema.

La Figura 1-1 muestra una estructura muy simple de control. El sistema de control, de acuerdo a las especificaciones, determina las acciones de control, pero sin conocer en ningún momento el estado de la planta. Este tipo de control se denomina *control en lazo abierto*.

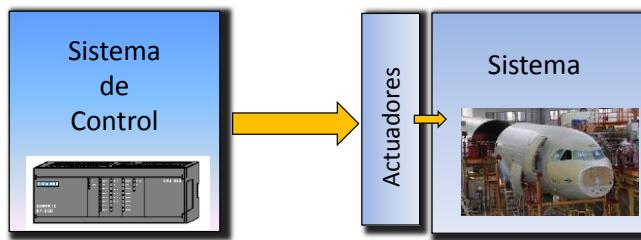


Figura 1-1– Control en lazo abierto.

Conocer cómo se está comportando en cada instante el sistema debe hacer mejorar la calidad del control, por ejemplo, permitiendo al sistema de control actuar frente a perturbaciones, a las que sería imposible que reaccionara con una estructura en lazo abierto. Esto se consigue dotando a la planta de un conjunto de sensores que midan las variables que se consideren de interés. Estas variables son las *salidas* del sistema. Si estas variables las utiliza el sistema de control para determinar las actuaciones se tiene el sistema de control en *lazo cerrado* (Figura 1-2).

En la figura se han incorporado en el sistema de control las interfaces (módulos de entrada y módulos de salida) que le permiten adquirir las señales provenientes o enviadas a la planta.

Nótese que los conceptos de entrada y salida dependen de si se consideran desde el punto de vista del sistema a controlar o del sistema de control. En este libro, cuando no se indique explícitamente lo contrario, se usará el punto de vista del sistema de control, es decir, *Entradas* son las señales que llegan al sistema de

control, y *Salidas* son las señales de control destinadas a los actuadores de la planta.

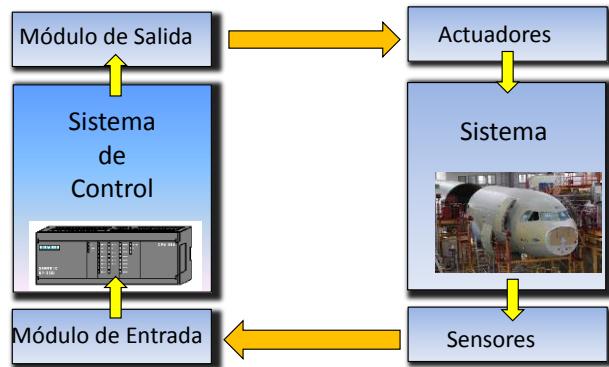


Figura 1-2– Control en lazo cerrado.

Las señales con las que se trabaja en estos sistemas de control, en relación a los valores que pueden adquirir dichas señales, son fundamentalmente de dos tipos:

- **Señales continuas o analógicas:** En un rango determinado de valores de la señal se tienen infinitos valores intermedios. Un ejemplo son las magnitudes físicas de un proceso, como presión, temperatura, caudal,...
- **Señales discretas:** Estas señales solo pueden tomar un número finito de valores. El caso más típico son las señales todo-nada, denominadas señales binarias o booleanas que solo pueden tener dos valores.

Las técnicas de diseño y la implementación de los sistemas de control van a depender en gran medida del tipo de señales con la que se trabaje. En este libro se va a tratar básicamente con señales binarias, que dan lugar a los sistemas de control denominados *Automatismos Lógicos*. Sin embargo, debido a la importancia que tiene actualmente el uso de señales continuas desde un autómata programable, se dedicará el Capítulo 12 al tratamiento de este tipo de señales y a la implementación en un autómata programable de algunas técnicas básicas de control de sistemas continuos.

## 1.2. Sensores y actuadores

Los sensores son dispositivos cuya función es transformar una determinada magnitud física de un proceso, tal como presión, temperatura, posición, etc., en otra señal normalmente eléctrica, que en el caso que nos trata, será enviada al sistema de control. Los actuadores realizan sobre el proceso una acción, normalmente mecánica, que implica un aporte de energía, en respuesta a una señal típicamente eléctrica proveniente del sistema de control.

Atendiendo al tipo de señal con la que operan, los sensores y actuadores pueden ser analógicos o discretos (todo-nada).

En esta sección se van describir brevemente algunos de estos dispositivos más frecuentes en el uso industrial y de los que se hace mención en los ejercicios y simulaciones que aparecen en este libro. Una descripción más completa de estos dispositivos se puede encontrar en [5] y [6].

### 1.2.1. Sensores y otras entradas al sistema de control

Existen multitud de sensores dependiendo de la magnitud física medida. Esta sección está centrada en los sensores todo-nada, que son los usados en automatismos lógicos. Alguno se los sensores de uso más frecuente son:

#### 1.2.1.1. Sensores de proximidad

Detectan la presencia de objetos en la cercanía del sensor. Habitualmente son del tipo todo-nada, aunque también se pueden incorporar en esta categoría sensores analógicos que proporcionan la distancia del objeto al sensor. Dependiendo del principio físico que utilicen para la detección, algunos necesitan contacto físico entre objeto y sensor y otros no. Algunos ejemplos son:

- **Finales de carrera o interruptores de posición:** Son interruptores que detectan la posición de un elemento móvil mediante accionamiento mecánico que cierra un contacto.
- **Detectores de proximidad inductivos:** Detectan objetos metálicos y no necesitan contacto. El principio de funcionamiento de un sensor inductivo de proximidad tiene que ver con la influencia de metales en un campo

electromagnético alterno (Ver Figura 1-3-a).

- **Detectores de proximidad capacitivos:** Detectan objetos metálicos y no metálicos. Su elemento sensor es un condensador y basan la detección en las variaciones de la capacidad cuando se aproxima un objeto.
- **Detectores fotoeléctricos.** Responden al cambio de intensidad en la luz. Requieren de un componente emisor que genera la luz, y un componente receptor. Según el aparato, actúan con la reflexión del haz luminoso o la interrupción del mismo (Ver Figura 1-3-b).

#### 1.2.1.2. Interruptores de nivel, presión, temperatura,..

El interruptor cambia de posición cuando se supera un determinado valor umbral en la magnitud que miden. En la Figura 1-3-c se muestra un interruptor de nivel basado en un flotador

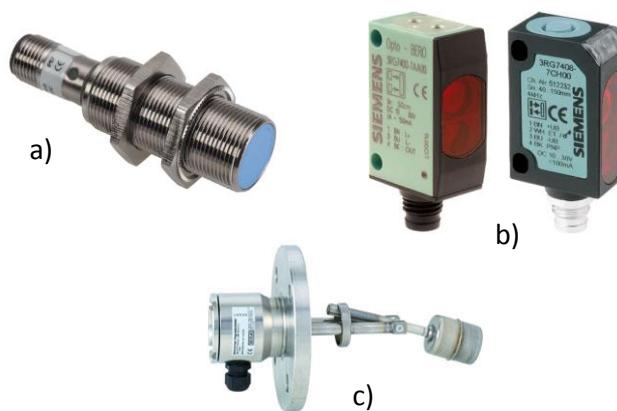


Figura 1-3- a) Sensor de proximidad inductivo AB Elektronik. b) Sensor fotoeléctrico de Siemens c) Interruptor de nivel de WIKA

### 1.2.1.3. Elementos de mando

Además de los sensores existen otros tipos de dispositivos que proporcionan entradas al sistema de control. Son los elementos que permiten al usuario dar órdenes al sistema, y entre los que se pueden encontrar pulsadores, interruptores, conmutadores, etc. Estos elementos pueden ser o bien dispositivos físicos, o bien programados en un sistema gráfico tipo SCADA. En la Figura 1-4 se muestran algunos de estos elementos del fabricante IDEC.



Figura 1-4– Elementos de mando de IDEC

### 1.2.2. Actuadores y otras salidas del sistema de control

Los actuadores son los dispositivos que se encargan de suministrar la energía a la planta para modificar los valores de las magnitudes que se están controlando a partir de una señal de mando que proviene del sistema de control. En muchos casos, esto se hace mediante una etapa intermedia que se denomina *preaccionador*. Se utiliza cuando la salida del sistema de control no tiene potencia o cuando hay que realizar una conversión de la señal. Éste sería el caso de un contactor para controlar la puesta en marcha de un motor.

Atendiendo a la tecnología usada, los actuadores son principalmente eléctricos, neumáticos o hidráulicos. De nuevo en esta sección se describirán algunos de los actuadores más utilizados del tipo todo-nada.

#### 1.2.2.1. Contactores y relés

Son dispositivos electromagnéticos que conectan o desconectan circuitos al excitar

una bobina de mando. La única diferencia entre ellos es que los contactores, al contrario de los relés, pueden accionar potencias elevadas. De esta manera, la salida de pequeña potencia del sistema de control puede alimentar la bobina, que abre o cierra un circuito de potencia elevada que permite activar por ejemplo un motor. En la Figura 1-5-b se muestra un contactor fabricado por Siemens.

### 1.2.2.2. Válvulas solenoides

Son válvulas electromecánicas diseñadas para controlar el caudal de un fluido a través de un conducto. Están controladas por una señal eléctrica que excita una bobina solenoide. La señal de control alimenta la bobina, que se encarga de abrir o cerrar la válvula (Ver Figura 1-5-a).

### 1.2.2.3. Actuadores neumáticos o hidráulicos

El actuador neumático (o hidráulico) más utilizado es el cilindro (Figura 1-5-c). Permite obtener un movimiento lineal de un émbolo en el interior del cilindro mediante la aplicación de presión a un extremo u otro de dicho émbolo.

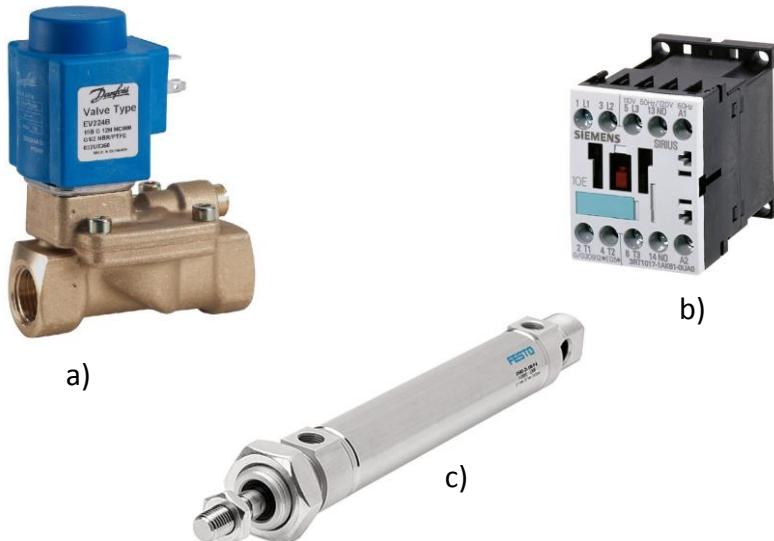


Figura 1-5– Actuadores. a) Válvula solenoide de Danfoss. b) Contactor Siemens.  
c) Cilindro neumático Festo.

Lo habitual es que la señal de control actúe sobre válvulas de distribución neumáticas intermedias que se encargan de establecer o cortar la conexión de aire comprimido entre sus terminales en función de la señal de control. Estás válvulas están conectadas a los terminales del cilindro por un lado y a la fuente del fluido por otro, de forma que de acuerdo a la señal de control se redirecciona el fluido que entra o sale del cilindro. En las secciones 3.8 y 3.9 se muestra un ejemplo de control de este tipo de sistemas.

#### 1.2.2.4. Señalizaciones

Un tipo de salida de los sistemas de control de suma importancia son las señalizaciones. Permiten notificar al operador cualquier condición de operación mediante avisos que fundamentalmente serán visuales (luces, leds, etc.) o sonoras (bocinas). Son de vital importancia en el caso de las alarmas.

### 1.3. Implementación de Automatismos Lógicos

Los automatismos lógicos son sistemas de control en los que las señales con las que se trabajan son de tipo todo-nada. Por tanto se trata de un sistema que a partir de los valores de entrada binarios debe proporcionar valores a las salidas también binarias. Se distinguen dos tipos de automatismos lógicos:

- **Combinacionales:** Los valores de la salida dependen exclusivamente de los valores de las entradas. Por tanto, independientemente de la técnica que se use, la lógica del automatismo consiste en implementar expresiones booleanas.
- **Secuenciales:** Los valores de salida dependen de las entradas pero también del estado interno en que en cada momento se encuentre el sistema. En su implementación se necesitan también elementos de memoria que permitan almacenar el estado en que se encuentre el automatismo.

Atendiendo a la forma de implementar la lógica de un automatismo se distingue:

- **Lógica cableada:** Está basado en uniones físicas entre los distintos componentes que realizan las operaciones lógicas o de memoria. Dentro de este tipo, lo más habitual es la utilización de contactos y relés con conexiones eléctricas. También existen sistemas de control basados en tecnología neumática, aunque son menos frecuentes.

- **Lógica programada:** El control se realiza mediante la ejecución de un programa en un sistema basado en un microprocesador. Lógicamente este sistema debe incluir los sistemas de adquisición de datos para conectar entradas y salidas a la planta.

Dentro de la lógica programada, los dispositivos más habituales son el PC industrial, los microcontroladores y los autómatas programables. Este libro está dedicado a la programación de estos últimos dispositivos. En las secciones siguientes se hace un resumen de las principales características y del modo de funcionamiento de estos dispositivos.

## 1.4. Definición de autómata programable

La norma IEC-1161, de la que se hará mención con profusión en este libro, define un Autómata Programable o PLC (Programmable Logic Controller) como “un sistema electrónico programable diseñado para ser utilizado en un entorno industrial, que utiliza una memoria programable para el almacenamiento interno de instrucciones orientadas al usuario, para implantar unas soluciones específicas tales como funciones lógicas, secuencia, temporización, recuento y funciones aritméticas con el fin de controlar mediante entradas y salidas, digitales y analógicas diversos tipos de máquinas o procesos”.

Por tanto estamos hablando básicamente de un computador para una aplicación muy concreta como es el control de procesos, lo que determina de una forma muy importante sus características tanto de hardware como de software:

- Debe estar diseñado para un ambiente industrial, y por tanto resistente a ambientes relativamente hostiles, con polvo, vibraciones, perturbaciones electromagnéticas, etc.
- Debe comunicarse con el proceso a controlar y con el usuario, por lo que debe disponer del hardware adecuado para la conexión de señales proveniente de sensores o dispositivos de mando del operario (pulsadores, selectores, etc.) y proporcionar señales a dispositivos de actuación sobre el proceso. En sus orígenes, los autómatas programables estaban orientados sobre todo al diseño de automatismos lógicos y por tanto gestionaban principalmente señales de tipo discreto. Sin embargo, con el paso de los años se han ido incorporando señales de tipo analógico, y cada vez es más habitual que el control de tipo

continuo de una planta también se haga desde el propio PLC. Por su creciente interés, en este libro se dedicará un capítulo al control de procesos utilizando señales analógicas.

- Los sistemas de adquisición de señales de entrada y salida también ha evolucionado en los últimos años. Cada vez es más frecuente que las señales, tanto de entrada como de salida esté conectadas a través de buses de campo. Por tanto los fabricantes se han visto en la necesidad de proporcionar módulos específicos capaces de gestionar señales provenientes de los distintos buses de campo que se encuentran en el mercado, tal como ModBus, ProfiBus o CANBus [7]. En una aplicación de cierta entidad lo habitual es que convivan señales cableadas y señales comunicadas por estos buses de campo.
- El sistema operativo debe ser capaz de gestionar los sistemas en tiempo real de acuerdo a la dinámica del proceso.
- Es habitual que la programación de estos dispositivos la realice un personal sin unos profundos conocimientos de programación informática convencional, pero con experiencia en diseño de automatismos con elementos tradicionales como los relés. Por este motivo, la totalidad de los autómatas programables disponen de un lenguaje de programación específico basado en lógica de contactos. Adicionalmente, disponen de lenguajes de programación similares a los de propósito general como puede ser el lenguaje C o lenguajes de tipo ensamblador.

## 1.5. Estructura externa de un autómata programable

Atendiendo al aspecto físico exterior del dispositivo, se pueden encontrar dos tipos: autómatas programables con estructura compacta y con estructura modular.

Los PLCs con estructura compacta se caracterizan porque todos sus elementos, incluidos las conexiones de entrada y salida se encuentran en un solo bloque. Son de utilidad cuando el número de señales cableadas es pequeño y sin previsiones de ampliación. En cualquier caso, la mayor parte de estos sistemas contemplan la posibilidad de conexión a bloques de ampliación de entrada o salida. Este tipo de PLC permite una carcasa más estanca y por tanto más adecuada a ambientes industriales especialmente hostiles. La Figura 1-6 presenta un PLC con estructura compacta, pudiéndose observar en los laterales las conexiones para las señales de

entrada y salida.



Figura 1-6– PLC Compacto Mitsubishi FX3G.

En los PLCs modulares, cada uno de los elementos (CPU, fuentes de alimentación, módulos de E/S, módulos de comunicaciones, etc.) es un dispositivo físico distinto que, de acuerdo a las necesidades del sistema de control, se conectan en rieles normalizados. Esta estructura permite de forma fácil adaptarse a las necesidades de diseño y a las posteriores ampliaciones, por ejemplo un aumento del número de señales de entrada o salida o la utilización de fuentes de alimentación redundantes. Otra ventaja de este tipo de configuración es que permite un funcionamiento parcial en el caso de avería en algún módulo no crítico, así como una rápida reparación mediante una simple sustitución de módulos. En la Figura 1-7 se muestra un autómata comercial de este tipo.

## 1.6. Estructura interna de un autómata programable

A nivel interno, un autómata programable no difiere mucho de cualquier otro computador: Una unidad central unida por buses internos a las interfaces de entrada/salida, periféricos y la memoria.



Figura 1-7 – PLC Modular Siemens Simatic S7-400.

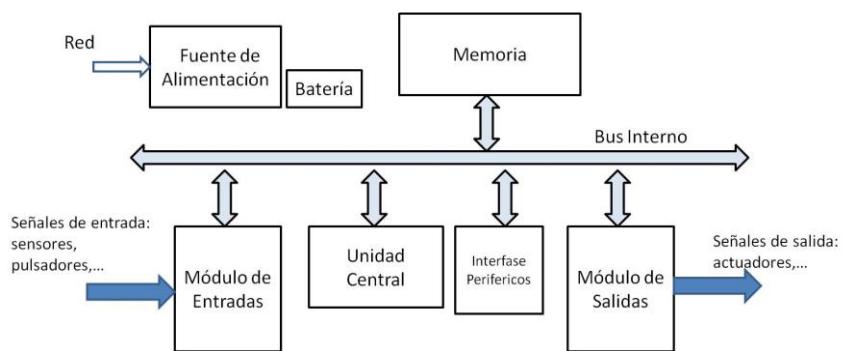


Figura 1-8 – Arquitectura de un PLC.

La Figura 1-8 muestra la arquitectura general de un autómata programable. La unidad central o CPU es un sistema basado en un micro-computador que se encarga de la ejecución de las instrucciones del programa del usuario, de la gestión de la memoria y de la transferencia de información con los módulos de entrada y salida. Debido a su importancia para comprender el funcionamiento de autómata, en las siguientes secciones se describen brevemente los módulos de memoria y los de entrada/salida.

### 1.6.1. Memoria

La estructura, utilización y tipo de la memoria del PLC depende del diseño del dispositivo, aunque en todos los casos se encuentran al menos las cinco zonas de memoria que se muestran en la Figura 1-9 y que se describen a continuación:

- **Sistema operativo o firmware.** Es un programa grabado por el fabricante y que se encarga del control a bajo nivel de los distintos módulos del PLC y de gestionar la ejecución del programa de usuario. Debe estar en una memoria no volátil, habitualmente de solo lectura tipo ROM. Actualmente, en los nuevos autómatas cada vez es más frecuente la utilización de tarjetas de memoria tipo SD o similar, que permiten una actualización del firmware cada vez que el fabricante desarrolla una nueva versión.
- **Memorias temporales del sistema.** Memoria tipo RAM donde se guardan variables temporales y registros del sistema. No es accesible por el usuario.
- **Memoria imagen de Entrada/Salida.** Cada una de las señales, tanto de entrada como de salida, conectada físicamente a los módulos de E/S tiene asociada una variable lógica en esta zona de memoria. El tipo de variable depende de la señal, siendo normalmente una variable booleana para las E/S discretas y de tipo entero para las señales analógicas. El programa de usuario accede a esta zona de memoria para leer o escribir las variables de entrada o salida respectivamente, realizándose el acceso a través de la dirección de la variable, que es fija para cada una de las entradas o salidas. El sistema operativo se encarga de actualizar los valores físicos y lógicos de cada una de las entradas o salidas. Para entender el funcionamiento del autómata programable es

importante conocer cómo y cuándo se realiza esta actualización. En las próximas secciones de este capítulo se detallará esta acción.

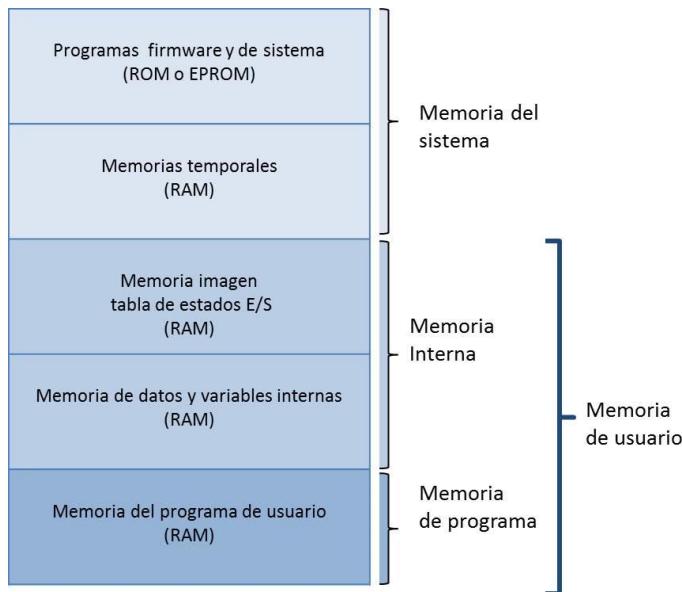


Figura 1-9 – Mapa de memoria de un PLC

- **Memoria de datos y variables de usuario.** En esta zona de memoria, también normalmente de tipo RAM, se almacenan las variables y los datos del programa de usuario.
- **Memoria de programa.** Es la zona donde se almacena el código escrito por el usuario. Este programa puede estar en memoria RAM aunque en este caso es imprescindible que tenga un respaldo de batería para evitar la pérdida de la información en caso de corte eléctrico. Cada vez es más habitual que el programa de usuario se encuentre en una tarjeta de memoria externa de tipo SD o similar, evitando de este modo la pérdida de información por corte de

suministro. Es importante hacer notar que la cantidad de memoria disponible en los autómatas programables es muy inferior a los valores actuales de los PCs de sobremesa o portátiles. A título de ejemplo, un PLC de gama alta como Modicom Quantum de Schneider tiene una capacidad de memoria de entre 400 y 800 Kb, ampliables a 7000 Kb. en caso de disponer de disponer de tarjeta de memoria adicional.

## 1.6.2. Módulos de Entrada/Salida

Son dispositivos que actúan de interface entre las señales del proceso, habitualmente en valores normalizados en tensión o intensidad, y las variables lógicas de la memoria imagen de E/S en el autómata programable.

La descripción detallada de estos circuitos queda fuera del alcance de este libro dedicado a la programación de los autómatas. En esa sección se proporciona una breve descripción de estos elementos, pero un análisis profundo de estos módulos se puede encontrar en obras como [3], [4] y en los manuales y fichas técnicas de los distintos fabricantes.

Las características de estos circuitos dependerán principalmente de si se trata de señales de entrada o salida y del tipo de señales que procesan, discretas o analógicas.

### 1.6.2.1. Entradas discretas

Son señales de tipo todo-nada que provienen principalmente de los dispositivos descritos en la sección 1.2.1. Existen módulos para señales tanto de corriente continua como alterna en tensiones normalizadas (12, 24, 48, 110 voltios en continua y 24, 48, 110, 220 en alterna). En la mayor parte de los casos cuentan con aislamiento galvánico para protección a sobretensiones o ruido eléctrico.

### 1.6.2.2. Salidas discretas

Proporcionan señal eléctrica a dispositivos del proceso como relés, contactores, electroválvulas., etc. (ver sección 1.2.2). Constructivamente las salidas lógicas pueden ser de distinto tipo, siendo las más habituales las salidas por relés, en las que cada una de éstas cuenta con un relé interno en el módulo que actúa sobre un

contacto interno que abre o cierra el circuito al que está conectado el dispositivo del proceso sobre el que actuar. Suelen tener también protección galvánica.

### 1.6.2.3. Entradas analógicas

Aunque en su origen los autómatas programables trataban exclusivamente con señales booleanas, ya es extraño el PLC que no incluye al menos la posibilidad de ser ampliado con módulos de entradas-salidas analógicas. Con estos módulos se podrán procesar en el PLC señales de tipo continuo, como temperaturas, presiones o posición de válvulas motorizadas, permitiendo realizar tareas como control continuo, por ejemplo con un controlador PID, o la supervisión de los valores máximos o mínimos de las variables del proceso. Actualmente en las aplicaciones de una cierta envergadura conviven señales de tipo lógico con señales de tipo analógico.

Los señales analógicas de los dispositivos del proceso suelen estar normalizadas, bien en tensión (por ejemplo, 0-10 V. ó 0-5 V.) o bien en intensidad, siendo lo más habitual 4-20 mA. Estas señales se transforman, en el caso de las entradas, a señales digitales, normalmente en código binario, cuya resolución dependerá del número de bits que se utilicen en la conversión. Para realizar esta conversión se utilizará un convertidor Analógico-Digital. En la Figura 1-10 se muestra el proceso de conversión realizado por un módulo de entrada analógica, en este caso una temperatura, utilizando un conversor de 8 bits. Para más detalles de este proceso, ver Capítulo 12.

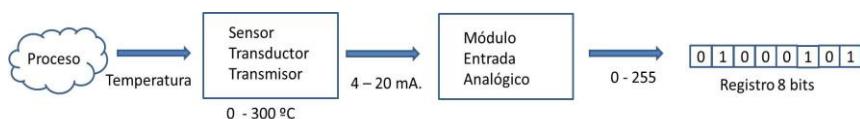


Figura 1-10 – Entrada analógica.

#### 1.6.2.4. Salidas analógicas

Se utilizan cuando el autómata programable debe proporcionar una señal analógica a dispositivos como variadores de velocidad, válvulas, etc. En este caso, los módulos de salida deben transformar una señal digital almacenada en una variable de tipo entero, en un valor eléctrico normalizado, para lo que se usa un convertidor Digital-Analógico. El proceso es el inverso al que se ha visto en el caso de las entradas, aunque la electrónica de estos dispositivos es muy distinta.

#### 1.6.3. Entornos de programación

La programación de un autómata programable se realiza desde dispositivos externos, que en la actualidad, en la práctica totalidad de los casos, se trata de computadores tipo PC con un software de programación específico (CoDeSys es un software de este tipo). La conexión física entre el autómata y el ordenador se realiza mediante algunos de los sistemas habituales de comunicación, típicamente USB. Desde el entorno de programación se realizan las siguientes tareas:

- Configuración del autómata
- Desarrollo de los programas usando editores de los distintos lenguajes.
- Transferencia del programa desde el PC al autómata o en sentido inverso
- Monitorización del programa. Con el programa ejecutándose en el autómata, se presentan en el PC los valores en tiempo real de las distintas variables.
- Simulación. Algunos entornos de programación tienen la posibilidad de ejecutar el programa del usuario en el propio PC, sin necesidad de transferirlo al autómata. Esta opción presente en CoDeSys se utiliza en los distintos ejemplos de este libro con objeto de que el lector pueda practicar la programación de estos dispositivos sin necesidad de conectarse a un autómata.

La utilización de consolas de programación específicas, muy habituales hace años, ha quedado en desuso.

### 1.7. Funcionamiento de un autómata programable

Es necesario tener siempre en mente que el objetivo de un autómata programable

es controlar un sistema en tiempo real, es decir, a grandes rasgos lo que debe realizar el PLC es actualizar “continuamente” el valor de las entradas que le llegan del proceso y de acuerdo al programa de control, actualizar el valor de las salidas también de una forma “continua”. Como se verá en esta sección, la actualización de entradas y salidas no se va a realizar de forma continua sino periódica debido a la forma en que trabaja un computador, pero sí debe hacerse tan frecuentemente como la dinámica del proceso exija.

El lector sin experiencia en la programación de estos dispositivos pero que haya utilizado lenguajes de programación convencionales debe ser especialmente cuidadoso en entender este funcionamiento, ya que las características de los programas va a ser muy distinta.

### 1.7.1. Ciclo del autómata programable

La ejecución del programa (o los programas) de usuario de un autómata es cíclico, es decir la secuencia de operaciones que realiza el programa del autómata se va repitiendo continuamente mientras el autómata continúe en funcionamiento. A grandes rasgos, en cada ciclo de funcionamiento el PLC realiza las siguientes operaciones (Ver Figura 1-11):

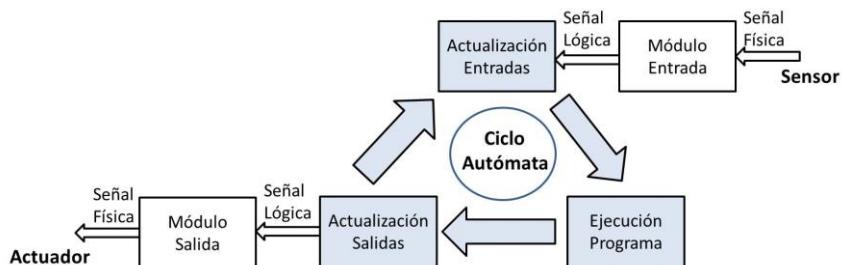


Figura 1-11 – Ciclo de funcionamiento de un autómata programable.

- Lectura de la señales de entrada:** se actualiza el valor de cada una de las variables de la tabla de memoria de entrada con el valor de la señal física

conectada en el módulo de entrada.

- **Ejecución del programa de control:** Se ejecuta el programa de usuario de forma secuencial, es decir desde la primera a la última línea de código.
- **Escritura de las señales de salida:** Se actualiza el valor de las señales eléctricas del módulo de salida con el valor lógico almacenado en la memoria de salida del PLC.

Teniendo en cuenta este modo de funcionamiento es necesario resaltar los siguientes aspectos:

- La actualización de salidas o entradas se realiza de forma periódica, una vez por ciclo. Por tanto el tiempo que transcurre entre dos actualizaciones depende de lo que dure el ciclo del autómata. Este intervalo de tiempo se denomina **Tiempo de Ciclo (Scan Time)**. Es obvio suponer que este tiempo debe ser lo suficientemente corto para que se detecten todos los cambios que se produzcan en las entradas. Típicamente, el tiempo de ciclo variará desde varios milisegundos a unos pocos cientos de milisegundos dependiendo de la dinámica más o menos rápida del proceso que se está controlando.
- El valor eléctrico de las entradas puede cambiar en cualquier momento, pero la actualización del valor lógico solo se hace una vez por ciclo, con lo que en determinados momentos este valor lógico puede estar desactualizado.
- Los programas de usuario trabajan exclusivamente con el valor lógico de la memoria de entrada, por tanto, en cada ejecución del programa se trabaja siempre con el mismo valor en cada entrada, independientemente de que en el transcurso de la ejecución del programa haya cambiado el valor físico del sensor.
- El valor físico de la salida se actualiza solamente al final del ciclo, independientemente de cuantas veces y donde se asigne valor a la variable lógica de salida en el programa.
- Entre otros factores, el tiempo de ciclo depende del tiempo que necesita la ejecución del programa de usuario. Por tanto, hay que tender a programas cortos, lo más secuenciales posible, y evitar dentro de lo posible bucles en los que el programa necesite mucho tiempo de ejecución o incluso pueda quedar atrapado. Es necesario tener siempre presente que el programa debe ejecutarse en un tiempo inferior al tiempo de ciclo máximo que se considere aceptable

para cada aplicación.

Las operaciones del autómata que se han presentado hasta el momento son las básicas. Sin embargo, en un autómata a estas actividades hay que añadirles las siguientes:

- En el arranque del autómata se realiza una verificación del hardware y se inicializan las variables del programa.
- Dentro de cada ciclo el sistema verifica que el programa se está ejecutando en tiempo inferior al establecido como tiempo de ciclo. En caso de que sea superior el sistema da un aviso o error. Esta tarea la realiza un temporizador interno que recibe el nombre de “watchdog”.
- También dentro de cada ciclo, el autómata realiza algunas verificaciones relacionadas con las conexiones de entrada-salida y de memoria.
- Finalmente, dentro de cada ciclo se necesita un tiempo para el intercambio de datos con los periféricos que tenga conectado el autómata.

En definitiva, el tiempo de ciclo de un autómata va a depender de:

- El código del programa de usuario
- Número de entradas y salidas
- Rutinas de chequeo y periféricos

### **1.7.2. Modos de funcionamiento del autómata programable**

Siempre teniendo en cuenta el funcionamiento general descrito en el apartado anterior, un autómata programable puede trabajar en varios modos. Cabe destacar, que habitualmente el código del programa de usuario puede descomponerse en distintos subprogramas y cada uno de ellos se puede ejecutar en un modo distinto. Los dos modos más importantes de funcionamiento son:

### 1.7.2.1. Modo normal o cíclico<sup>1</sup>

Se corresponde con el modo de funcionamiento básico, es decir, inmediatamente que se termina un ciclo comienza el siguiente sin solución de continuidad. En la Figura 1-12 se muestra este modo de funcionamiento.

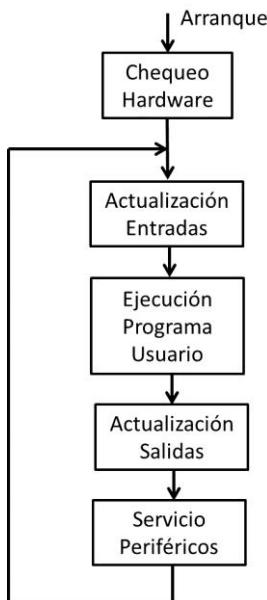


Figura 1-12 – Modo de funcionamiento normal o cíclico.

En CoDeSys, este modo de funcionamiento recibe el nombre de Freewheeling

### 1.7.3. Modo periódico

En este modo, la ejecución de los ciclos de ejecuta de forma periódica con tiempo fijo, de modo que dicho tiempo es fijado por el usuario. Lógicamente, el tiempo

<sup>1</sup> Conviene hacer notar que en la literatura y en la documentación de los distintos equipos existe cierta confusión en la manera de denominar a estos modos. En este caso hemos optado por utilizar la que consideramos más clara y se ha añadido la forma en que se denominan en el entorno Codesys.

periódico debe ser superior al tiempo necesario por al autómata para ejecutar un ciclo. Es una forma habitual de trabajo cuando en el programa se están controlando variables continuas. De esta manera se fija el tiempo de muestreo del controlador. En CoDeSys, este modo se denomina *cíclico*. La Figura 1-13 muestra el esquema de este tipo de funcionamiento.

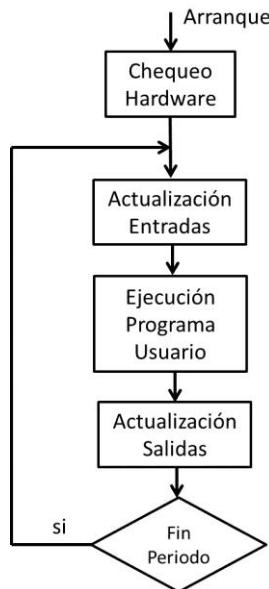


Figura 1-13 – Modo de funcionamiento periódico.

#### 1.7.4. Otros modos de funcionamiento

En función del autómata que se esté utilizando, es posible que presenten otros modos de funcionamiento complementarios, entre los que se pueden destacar:

##### 1.7.4.1. Tareas rápidas

En algunas ocasiones es necesario realizar algunas tareas críticas del control del sistema con una frecuencia mayor a la que marca el ciclo normal de funcionamiento del autómata.

Las tareas rápidas se ejecutan periódicamente por interrupciones con una prioridad mayor que el programa principal. El código debe estar diseñado para ejecutarse en un tiempo mucho más corto que el ciclo principal.

#### **1.7.4.2. Tareas lanzadas por eventos**

Son tareas que se ejecutan cuando se cumple una determinada condición, normalmente basadas en variables del programa o variables internas del PLC.



## 2. INTRODUCCIÓN A LA NORMA IEC-61131

---

**L**a norma IEC 61131 es un paso en la estandarización de los autómatas programables y sus periféricos, tanto en relación a hardware como a software, considerando aspectos en estos dispositivos como características funcionales, condiciones de servicio, seguridad, lenguajes de programación, comunicaciones, etc.

Los primeros trabajos en esta norma datan de 1979 y han continuado durante años hasta la actualidad en que todavía se sigue actualizando y apareciendo nuevos documentos. Este estándar surge como combinación y continuación de otros 10 estándares internacionales previos.

En la actualidad la norma está dividida en las siguientes 8 partes:

- **Parte 1: Información general de PLCs.** Establece definiciones básicas e identifica las principales características relevantes a la selección y utilización de autómatas programables y sus periféricos. Así se define la estructura funcional básica de un autómata programable, características de la CPU, modos de arranque, características de la memoria, características de los módulos de entrada y salida, de las comunicaciones, características generales de la interface hombre-máquina, de la programación, depuración y monitorización, etc.
- **Parte 2: Especificaciones de equipos y ensayos.** Esta segunda parte de la norma establece los requisitos constructivos eléctricos, mecánicos y ambientales de los distintos dispositivos. También estableces especificaciones funcionales de seguridad y protecciones. Finalmente define los tests para la certificación de los autómatas programables de acuerdo a esta norma.
- **Parte 3: Lenguajes de programación.** Define el modelo de software y los lenguajes de programación, incluyendo sintaxis, variables, etc. En concreto define cinco lenguajes distintos:

- Lenguaje de lógica de contactos (LD - Ladder Diagram).
  - Diagrama de bloque de funciones (FBD - Function Block Diagram).
  - Texto estructurado (ST - Structured Text),
  - Lista de instrucciones (IL - Instruction List),
  - Bloques de función secuenciales (SFC - Sequential Function Chart).
- **Parte 4: Guías para usuarios.** En esta parte se presentan las orientaciones necesarias para que los usuarios (industrias, empresas de ingeniería, programadores, etc.) puedan seleccionar, instalar y mantener los autómatas programables y sus periféricos.
  - **Parte 5: Comunicaciones.** En la quinta parte se define el modelo de comunicaciones, con los mecanismos que permiten las comunicaciones entre distintos autómatas programables y con los demás dispositivos de un sistema de automatización.
  - **Parte 6: Seguridad funcional.** En esta parte se especifican las características, tanto de hardware como de software, necesarias para que un autómata programable y sus periféricos puedan ser usados dentro un sistema seguro E/E/PE de acuerdo a la norma IEC-61508.
  - **Parte 7: Programación de control borroso.** Esta parte surge con el uso creciente de la lógica difusa en los sistemas de control. Define la estructura y el uso de un lenguaje para la programación de lógica difusa, el Fuzzy Control Language (FCL).
  - **Parte 8: Guía para la aplicación e implementación de lenguajes de programación.** Esta última parte proporciona orientaciones prácticas relacionadas con configuración y uso de lenguajes de programación definidos en la parte 3.

Más información y futuras actualizaciones sobre esta norma se pueden encontrar en [www.plcopen.org](http://www.plcopen.org). Este libro se centra en la tercera parte y principalmente en la sintaxis y utilización de los cinco lenguajes de programación.

## 2.1. Elementos básicos en la norma IEC-61131-3

Los sistemas de control ejecutados sobre autómatas programables pueden llegar a ser muy complejos, pudiendo consistir por ejemplo en varios PLCs que de forma coordinada controlan distintos subsistemas de la planta. El modelo de software de la norma IEC-61131-3 intenta contemplar todas estas posibles arquitecturas dando lugar a un modelo relativamente complejo.

El modelo de software que propone la norma está basado en una estructura jerárquica, en la que los elementos que la componen desde el nivel más bajo de la jerarquía hasta el más alto son los siguientes:

- **POU (*Program Organization Unit*):** son los elementos básicos de la programación, secciones de código que pueden ser escritas en cualquiera de los lenguajes que permite la norma.
- **Tarea (*Task*):** Las POUs estarán asociados a tareas, que son las que le asignaran las propiedades de ejecución en tiempo real. Por ejemplo, cada tarea especificará si es cíclica, periódica (y el tiempo del periodo) o lanzada por un evento. También se pueden asignar prioridades a las tareas. Todas las POUs que tenga asociada cada tarea se ejecutarán con las características de tiempo real especificadas.
- **Recurso (*Resource*):** Básicamente consiste en todos los elementos que se definen o ejecutan en una única CPU. Incluirá un conjunto de tareas con sus POUs asociadas, variables globales a todas las POUs, etc.
- **Configuración (*Configuration*):** Consiste en el conjunto de CPUs que constituyen el sistema de control, es decir, estará formado por un conjunto de recursos. La norma prevé posibilidad de comunicación entre los distintos recursos, definición de variables globales a todos los recursos de la configuración, etc.

Una completa descripción de este modelo de software se puede encontrar en [8].

Dado el carácter básico de este libro, en adelante solo se contemplarán arquitecturas simples con un solo recurso, y en general una única tarea cíclica, pero que puede constar de varias POUs que se ejecutan simultáneamente.

En CoDeSys la opción por defecto es una única tarea que se ejecuta de forma cíclica. En cualquier caso, para modificar esta configuración por defecto, consulte el Apéndice A, donde se muestra la forma de definir tareas, sus propiedades y

asociarles POU s en CoDeSys.

En el resto de la sección se introducen las POU s, variables de programación y sus lenguajes de programación.

## 2.2. Program Organization Units (POU)

Los bloques básicos de programación lo constituyen las POU s (Program Organization Unit). Una aplicación de usuario, normalmente denominada *Proyecto*, consta de una o varias POU s. Las POU s pueden estar escritos en cualquiera de los lenguajes de la norma y pueden llamar a otras POU s con o sin parámetros.

Las POU s pueden ser de tres tipos:

- Funciones (FUN)
- Bloques funcionales (FB)
- Programas (PROG)

Aunque posteriormente se verá con más detalle, la diferencia principal entre bloques funcionales y funciones radica en que estas últimas siempre devuelven el mismo valor si los parámetros de llamada son los mismos. Por el contrario, los bloques funcionales mantienen información interna a la POU que se utiliza para determinar el valor que devuelven. Un caso típico de estos últimos es un contador. Supóngase que se define con un único parámetro de entrada que da la orden de incrementar en una unidad el valor del contador y una salida que es el número de unidades contadas. Evidentemente, el valor de salida va a depender de la entrada pero también del valor interno almacenado dentro del bloque funcional (valor contado hasta ese instante de tiempo).

Los programas (PROG) constituyen los “programas principales”. En un Proyecto puede haber varios PROG. Son los únicos que tienen acceso a las señales físicas de Entrada/Salida y los únicos que tienen acceso a las variables globales.

## 2.3. Variables

Salvo algunas particularidades, el uso de las variables no difiere mucho del de

cualquier lenguaje de programación. En concreto, la norma permite el uso de distintos tipos de variables: booleanas, enteras, reales, etc. Las variables pueden ser definidas como globales a todas las POUs o locales a una POU. En cualquier caso, independientemente del o los lenguajes utilizados para programar, todas las variables usadas deben ser declaradas.

Mención especial merecen las variables de E/S. Como se describe en el Capítulo 1, el autómata dispone de una serie de señales de entrada y salida que están físicamente conectadas a los módulos de E/S. Asociada a cada una de estas señales físicas existe una variable a la que se accede desde los programas de usuario mediante su dirección y que es la que se utilizará para conocer el valor de una entrada o darle valor a una salida del autómata. Al igual que las variables globales, estas variables de E/S solo se pueden usar en los programas (PROG).

Cada una de las POUS tiene una zona de declaración de variables donde se escribirán las variables locales y los parámetros de entrada y salida de esa POU. Un ejemplo típico de la zona de declaración de una POU se muestra en el siguiente ejemplo:

---

```
VAR_INPUT
    Param1: BOOL;
    Param2 : INT;
END_VAR
VAR_OUTPUT
    Sal : REAL;
END_VAR
VAR
    Aux1 : STRING[10];
    Aux2 : BOOL;
END_VAR
```

---

Como se puede apreciar, están declaradas en distintos bloques correspondientes a las variables de entrada de la POU (VAR\_INPUT), de salida (VAR\_OUTPUT) y las variables locales a la POU (VAR).

## 2.4. Lenguajes de programación

En esta sección se hará una breve descripción de los cinco lenguajes que permite la norma. Cada uno de estos lenguajes se detallará en capítulos posteriores, donde se hará referencia tanto a la sintaxis definida por esta norma IEC como a la edición de programas utilizando el entorno de programación de CoDeSys.



Figura 2-1. Ejemplo del lenguaje LD

El **lenguaje de contactos (LD)**, conocido como *diagrama de escalera* o *ladder* es un lenguaje basado en los tradicionales diagramas de relés y contactos. Está orientado principalmente a aplicaciones con señales booleanas. Su origen está en Estados Unidos. La Figura 2-1 muestra un código simple que realiza la operación booleana  $Sal = (A \cdot B) + C$ . Nótese que una operación booleana AND se implementa en lógica de contactos con contactos en serie, y la operación OR con contactos en paralelo.

El **lenguaje de Diagramas de Bloques Funcionales (FBD)** es también un lenguaje gráfico basado en la conexión de bloques que representan los distintos operadores o POU's. Estas conexiones representan los parámetros de entrada y salida de los distintos programas o funciones, tal como se hace en los diagramas de circuitos electrónicos. Son muy utilizados en la industria de proceso. La Figura 2-2 muestra la misma expresión anterior en lenguaje FBD.

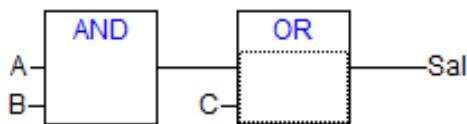


Figura 2-2. Ejemplo del lenguaje FBD

El **lenguaje de Lista de Instrucciones (IL)** es un lenguaje de tipo ensamblador simple apoyado en el uso de un acumulador. Tiene su origen en Alemania. El siguiente ejemplo muestra un trozo de código en lenguaje IL, en el que se implementa la misma operación lógica.

---

```
LD A
AND B
OR C
ST Sal
```

---

El **lenguaje de texto estructurado (ST)** es un lenguaje de más alto nivel con orígenes en los lenguajes de programación de propósito general Pascal y 'C'. Es el más adecuado para codificar expresiones complejas e instrucciones anidadas. Este lenguaje dispone de las estructuras típicas para bucles (FOR, REPEAT-UNTIL, WHILE-DO) y ejecución condicional (IF-THEN-ELSE; CASE). El siguiente ejemplo muestra la misma operación del ejemplo anterior escrita en lenguaje ST.

---

```
Sal:=A AND B OR C;
```

---

El último de los lenguajes es el **Diagrama Funcional Secuencial (SFC)**, que describe gráficamente el comportamiento de un programa de control mediante un

grafo de estados (etapas) y transiciones. Deriva de las Redes de Petri y Grafcet. Realmente no se trata de un lenguaje de programación, en el sentido que necesita hacer uso de rutinas escritas en los otros lenguajes para implementar acciones o condiciones lógicas.

# 3. PRIMEROS PASOS CON CoDeSys

---

**C**oDeSys es un entorno de programación de autómatas programables que permite la programación según la norma IEC-61131-3. Este entorno va a facilitar la edición de programas en los lenguajes que propone la citada norma, la depuración de los programas, la simulación, la comunicación con el PLC y la monitorización del programa que se ejecute en dicho dispositivo. En definitiva, permite realizar todos los pasos necesarios para el diseño de aplicaciones para autómatas programables.

CoDeSys es un software comercial desarrollado por 3S-Smart Software Solutions GmbH, empresa alemana independiente de fabricantes o plataformas hardware, aunque ya es posible desarrollar con él las aplicaciones para los productos de más de 250 fabricantes, siendo actualmente la plataforma más extendida en Europa para el desarrollo de aplicaciones de acuerdo a la norma IEC-61131-3.

En este capítulo se presenta una guía rápida en la que se muestra como ejecutar al software, se describe el entorno de programación de CoDeSys y se introduce el desarrollo de los primeros programas en uno de los lenguajes de la norma, el lenguaje de contactos. El lenguaje de contactos permite diseñar automatismos de forma similar a como tradicionalmente se ha realizado utilizando relés y contactos. Es un lenguaje con el que es fácil realizar diseños simples, está muy extendido y entendemos que cualquier persona interesada en la programación de autómatas programables debe estar familiarizada con él. En este capítulo se usarán algunos elementos básicos del citado lenguaje, aunque se estudiará con más profundidad en un capítulo posterior.

## 3.1. Algunos conceptos básicos en CoDeSys

CoDeSys organiza toda la información de cada aplicación que se va a ejecutar en el

autómata programable en *Proyectos*. Un proyecto incluye una o varias POU, toda la información necesaria para la configuración del PLC que se esté usando, las variables globales, interfaces gráficas para controlar y visualizar la ejecución del programa etc. Cada proyecto está guardado en un fichero con extensión *.pro*

En cuanto al código, un proyecto puede incluir una o varias POU. Las distintas POU de un proyecto pueden estar escritos en lenguajes diferentes y pueden ser de cualquiera de los tipos vistos anteriormente: Programas, Funciones o Bloques Funcionales.

Cuando se crea un nuevo proyecto, la primera POU que se define tiene la denominación PLC\_PRG. Esta POU que es de tipo PROGRAM, es el programa por donde empieza a ejecutarse el código (similar a la función *main* de C), y el resto de POU, si las hubiera, deberán ser llamadas, directa o indirectamente, desde ésta.

## 3.2. Descarga del software CoDeSys

CoDeSys es un programa comercial, pero desde la página web del desarrollador se puede descargar de forma gratuita el software con funcionalidad completa salvo la comunicación con los PLCs. Sin embargo, al disponer el software de la opción de ejecución en simulación, esta versión va a permitir trabajar en todos los lenguajes y con todas las funciones disponibles.

La web de la descarga de CoDeSys es <http://www.codesys.com/> accediendo a la sección *Download*.

**Nota Importante: La versión que se describe en este documento es la 2.3.**

Este libro se acompaña de una serie de ficheros de proyectos (con extensión *.pro* en CoDeSys) donde estarán ya implementados en simulación los sistemas a controlar con sus variables de entrada y salida del automatismo lógico (definidas como variables globales) y con los que el lector podrá analizar el funcionamiento o realizar los ejercicios que se muestran o proponen. Para abrir los ejercicios se hará doble-click sobre el nombre del fichero de extensión *.pro*, con lo que se abrirá CoDeSys con el ejercicio seleccionado. De esta manera el lector tendrá acceso a todas las funcionalidades y posibilidades de programación sin necesidad de disponer de un autómata programable. **Estos ejercicios se pueden descargar de ....**

### 3.3. Ejecución de CoDeSys

Cuando se ejecuta CoDeSys y se pretende crear un nuevo proyecto, en primer lugar se solicita información para configurar el PLC con el que se va a trabajar. La configuración consiste en caracterizar completamente el hardware disponible: CPU que se dispone, número y tipo de los módulos de entrada y salida, etc. Para los ejercicios de este libro se trabajará solamente en modo simulación, por lo que se utilizará una configuración genérica. En esta sección se muestra como escoger las opciones iniciales que permiten ejecutar los programas del autómata programable en modo simulación.

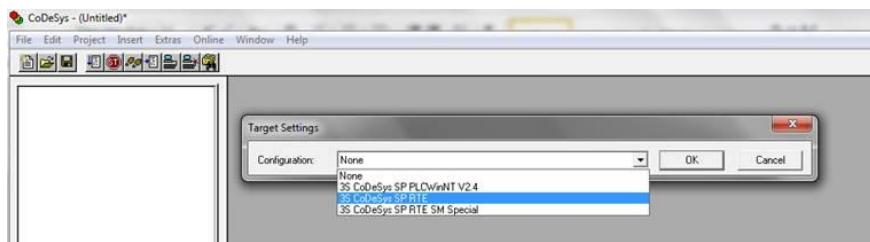


Figura 3-1 – Ventana de selección de la configuración

Una vez arrancada la aplicación, se abre una ventana en la que se solicitan los parámetros de configuración *Target Setting*. Desplegaremos las opciones y escogeremos la configuración 3S Codesys SP RTE (Figura 3-1)

En la siguiente pantalla se pulsa OK sin cambiar ningún parámetro de configuración (Figura 3-2).

Finalmente aparece la ventana de creación de POU, que incluye el nombre PLC\_PRG por defecto y en el que podemos escoger el lenguaje (en la Figura 3-3 se ha escogido lenguaje Ladder). Recuérdese que la primera POU que se cree de un proyecto debe ser PLC\_PRG, ya que es desde donde empieza a ejecutarse el código.

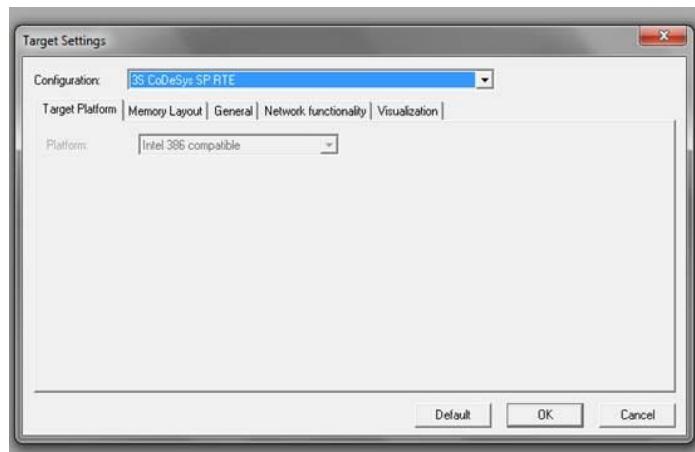


Figura 3-2. Ventana de selección de los parámetros de la configuración

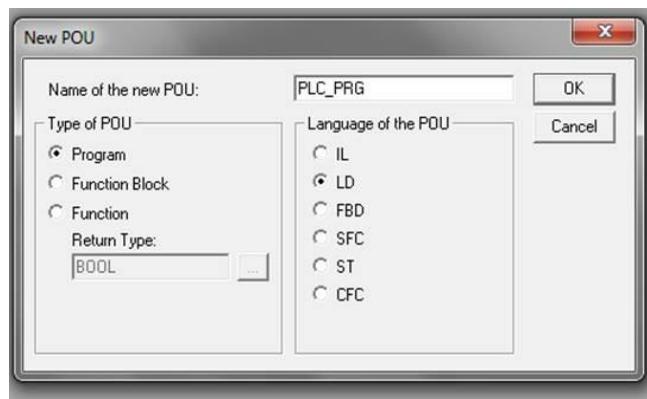


Figura 3-3 . Ventana de creación de una POU

### 3.4. Pantalla principal

En la Figura 3-4 se muestra la pantalla principal de CoDeSys que se abrirá, o bien al crear un proyecto nuevo, o al hacer doble-click sobre cualquier fichero de proyecto con extensión .pro.

La pantalla principal se encuentra dividida en dos zonas separadas por una barra vertical. La zona de la izquierda (Organizador de Objetos) permite acceder a los diferentes elementos que componen el Proyecto: POU's, variables globales, visualizaciones,... En la figura se aprecian los POU's que componen el Proyecto, pero adicionalmente, mediante las pestañas situadas en la parte inferior se puede acceder al resto de elementos.

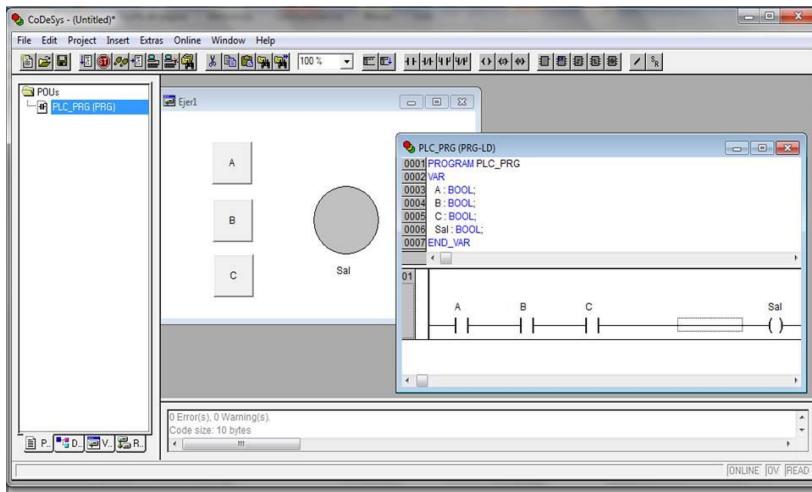


Figura 3-4 – Pantalla principal en CoDeSys

La zona de la derecha permite la edición de código, variable y visualizaciones. Una visualización es una pantalla gráfica cuyos elementos se pueden asociar a variables del autómata, de modo que pueden cambiar de color, moverse, realizar representaciones gráficas, etc., en función del valor que van tomando las variables del programa de usuario. Las visualizaciones se usarán para tener una

representación gráfica del proceso que se está controlando. En la figura se observa una visualización y una ventana de edición de código en lenguaje de contactos (LD). El editor variará dependiendo del lenguaje en el que se esté escribiendo el código. En la botonera de la parte superior se pueden observar botones específicos para el editor en lógica de contactos: contactos normalmente abiertos, normalmente cerrados, bobinas, etc. Esta botonera también cambiará dependiendo del lenguaje que se esté editando.

En la parte inferior se encuentra la *Ventana de Mensajes* donde aparecerán los mensajes de error, avisos, etc.

### 3.5. Primer programa en lógica de contactos

Esta sección describe cómo implementar un programa simple en lógica de contactos (LD). Para eso se va realizar paso a paso la escritura de un programa que realice la operación AND de tres entradas. Como bien se sabe, para ello es necesario conectar en serie las entradas y asignarlo al relé de salida.

Para realizar este programa vamos a usar el programa *Cap3\_1.pro* en el que las entradas se han asignado a interruptores denominados *A*, *B* y *C* y la salida a una señal luminosa denominada *Sal*.

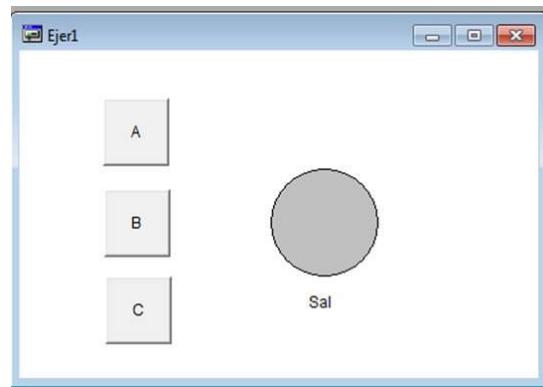


Figura 3-5 – Visualización ejercicio *Cap3\_1.pro*

Al abrir el fichero aparecen en la zona de edición dos ventanas. La primera es la de visualización en la que se representan los tres interruptores y la luz, tal como se aprecia en la Figura 3-5:

La segunda ventana (Figura 3-6) es la de edición en lógica de contactos. En la parte superior se encuentra la zona de declaración de variables. Cualquier variable que se utilice es necesario declararla.

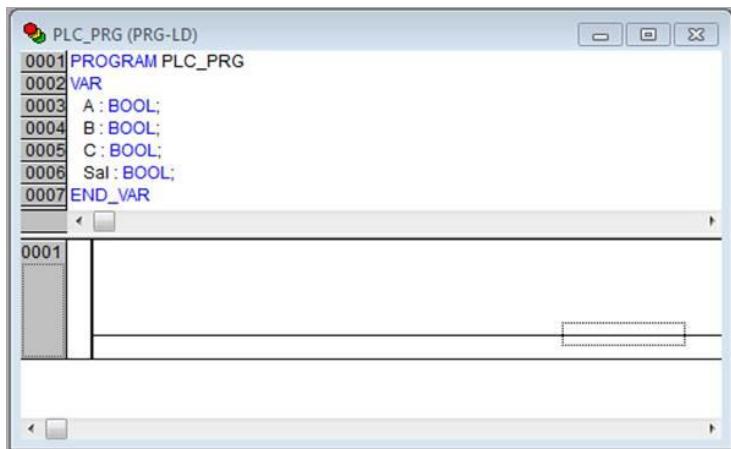


Figura 3-6 – Ventana de edición del ejercicio *Cap3\_1.pro*

Aunque posteriormente se volverá a los detalles sobre la declaración de variables, puede verse que están declaradas las tres variables de entrada y la variable de salida como booleanas de acuerdo al formato:

*Nombre\_variable : BOOL;*

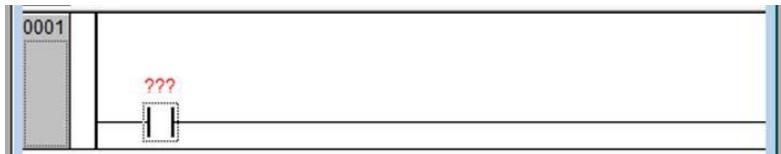
Debajo aparece una línea inicialmente vacía donde se irán colocando (usando el editor gráfico) los contactos en serie o paralelo en la parte izquierda de la línea y la bobina del relé en la parte derecha. En este ejercicio se usarán los contactos y relés básicos que aparecen en la barra de herramientas (si está seleccionada la ventana de edición), que se muestra en la Figura 3-7 y que representan de izquierda a derecha el contacto normalmente abierto, contacto normalmente cerrado, contactos en paralelo normalmente abierto y normalmente cerrado y la bobina.



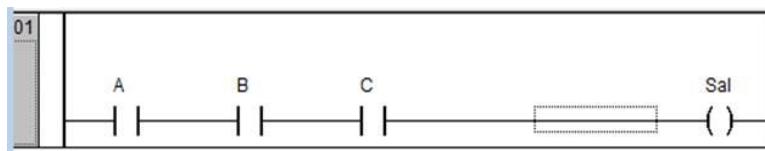
Figura 3-7 – Botones de edición en lógica de contactos

Para escribir el programa que realice la operación AND es necesario que esté activa la ventana de edición del programa en lógica de contactos y a continuación realizamos los siguientes pasos:

- Pulsamos el contacto normalmente abierto. Nos aparece un contacto en la línea y pulsando sobre las interrogaciones escribiremos el nombre de la variable de entrada, *A* en este caso.



- Pulsamos sobre la parte derecha de la línea para situar allí el cursor y volvemos a repetir la operación para los contactos *B* y *C*. Si no se coloca el cursor en dicha posición, los siguientes contactos se irán insertando a la izquierda del primero.
- Finalmente insertamos la bobina.



Una vez escrito el programa es necesario compilarlo. Para eso se selecciona la opción *Build* del menú *Project* (o F11). Si hubiera errores de compilación aparecerían en la ventana de mensajes.

Una vez compilado correctamente se procederá a la simulación del programa, para

lo que se realizan los siguientes pasos:

- Poner el programa en modo simulación: *Online > Simulation Mode*.
- Entrar en el modo ejecución: *Online > Login*
- Ejecutar el programa: *Online > Run*

Desde este momento el código se está ejecutando. Compruébese que la luz en la visualización se enciende cuando los tres pulsadores están activados y que no se enciende en cualquier otro caso.

Durante la ejecución se observa en la pantalla de edición del programa que aparecen en azul los contactos que en cada momento están cerrados, y lo mismo ocurre con las bobinas. Además, en la sección de declaración de variables se puede ver el valor (TRUE o FALSE) de cada una de las variables.

Para realizar cualquier modificación del programa, hay que salir previamente del modo ejecución, para lo que se deberá seleccionar:

*Online > Logout*

**Ejercicio 3.1** Escriba el código para implementar la operación OR de las tres variables de entrada

**Ejercicio 3.2** Escriba el código para implementar la operación  $Sal = (A \cdot B) + C$

Nota: Para hacer el paralelo de A y B hay que seleccionar los dos contactos (*Shift+Ratón Izq*).

### 3.6. Ejemplo 2. Puesta en marcha y parada de un motor

En este segundo ejercicio se pretende controlar el funcionamiento de un motor con dos pulsadores *M* y *P*. Al pulsar *M* el motor debe ponerse en marcha, y en ese estado debe continuar hasta que pulsemos *P*, momento en que el motor se detendrá. En caso de pulsar simultáneamente *M* y *P*, debe prevalecer la parada. En este ejemplo se verá como declarar variables nuevas y como realizar programas donde tenemos más de una línea de código.

La visualización de este ejercicio se encuentra en *Cap3\_2.pro*.

Al trabajar con pulsadores y no con interruptores, es necesario asegurarse que al

pulsar  $M$  el motor se ponga en marcha, pero al soltarlo, éste debe continuar en marcha mientras no pulsemos  $P$ . Esto se implementará utilizando un relé o variable auxiliar que denominaremos  $X$  y con un conexionado muy común denominado enclavamiento. En la Figura 3-8 se muestra el programa que será necesario escribir.

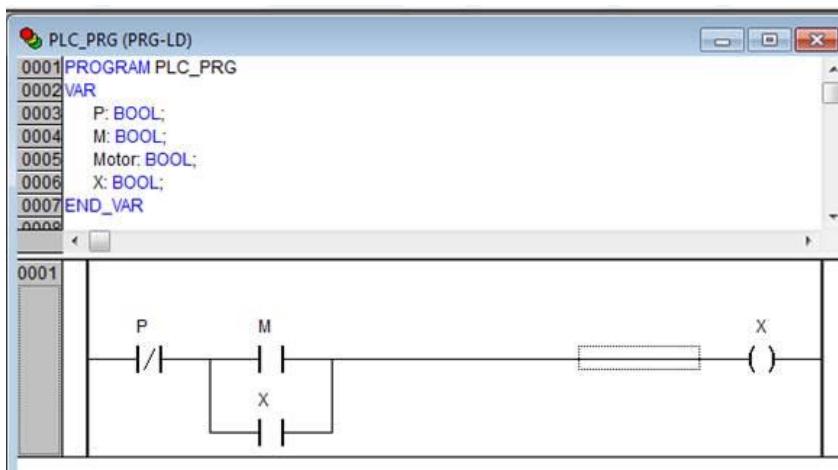


Figura 3-8 – Solución del Ejercicio 2 (I)

Como se observa, al conectar en paralelo  $M$  y un contacto asociado al relé  $X$ , si se pulsa  $M$  se cierra el circuito y activa el relé  $X$  con lo que se cierra el contacto asociado a dicho relé. Al soltar  $M$ , se mantiene cerrado el circuito a través del contacto de  $X$  con lo que el relé  $X$  sigue activado. Al pulsar el contacto normalmente cerrado  $P$ , se abre el circuito, se desactiva  $X$  y por tanto se vuelva a abrir su contacto asociado.

Para escribir el código es necesario declarar la nueva variable  $X$ , que se puede hacer de dos maneras:

- Escribiéndolo directamente en la zona de declaración de variables.
- Si se escribe  $X$  sobre un contacto y previamente no se ha declarado, aparecerá la ventana de declaración de variables (Figura 3-9), en la que podemos escribir el nombre y el tipo (BOOL). Al pulsar OK la variable se añadirá automáticamente a la zona de declaración.

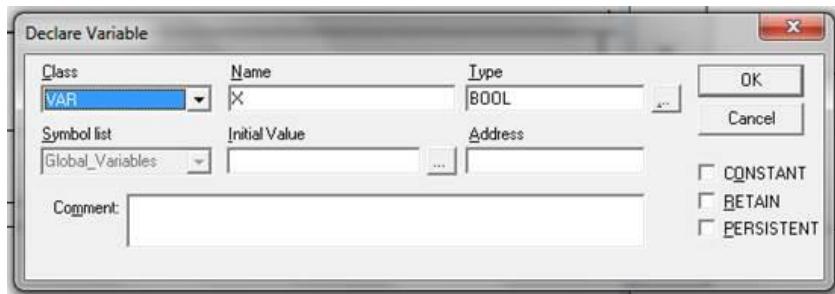


Figura 3-9 – Ventana de declaración de variables

A continuación es necesario añadir un segundo circuito que active la variable de salida *Motor*. En este caso *Motor* estará activo cuando lo esté X.

Cada uno de los circuitos se denomina en CoDeSys *Network*. Para añadir un circuito nuevo hay dos opciones:

- *Insert > Network (before) o Network (after)*
- <Ratón Derecho> + *Network (before) o Network (after)*

En cualquiera de los casos crea un nuevo circuito antes o después del circuito que se tenga seleccionado. El resultado final del programa se muestra en la Figura 3-10.

### 3.7. Ejemplo 3. Cambio del sentido de giro de un motor

En este ejemplo se dispone de un motor que puede girar en los dos sentidos. Se pretende controlar la puesta en marcha en los dos sentidos y la parada. Para ello se dispone de tres pulsadores, uno de parada P y dos de puesta en marcha, MI y MD que ponen en marcha el motor respectivamente en sentido anti horario y horario.

En la Figura 3-11 se muestra cómo se conectaría el motor para el cambio de sentido de giro. Como se puede observar se trata de un motor trifásico, de modo que para cambiar el sentido de giro es necesario intercambiar dos de las fases. Esto es lo que hacen los contactos asociados a los contactores KM1 y KM2. El modo de funcionamiento sería el siguiente:

- Si el contacto KM1 está cerrado (para lo que hay que activar la bobina del

contactor  $KM1$ ) gira en un sentido.

- Si el contacto  $KM2$  está cerrado (para lo que hay que activar el contactor  $KM2$ ) gira en sentido inverso.
- El motor está parado si los dos contactos están abiertos.
- Si están simultáneamente cerrados los dos contactos, se produciría un cortocircuito, por lo que el sistema de control deberá garantizar que esta situación nunca se produzca.

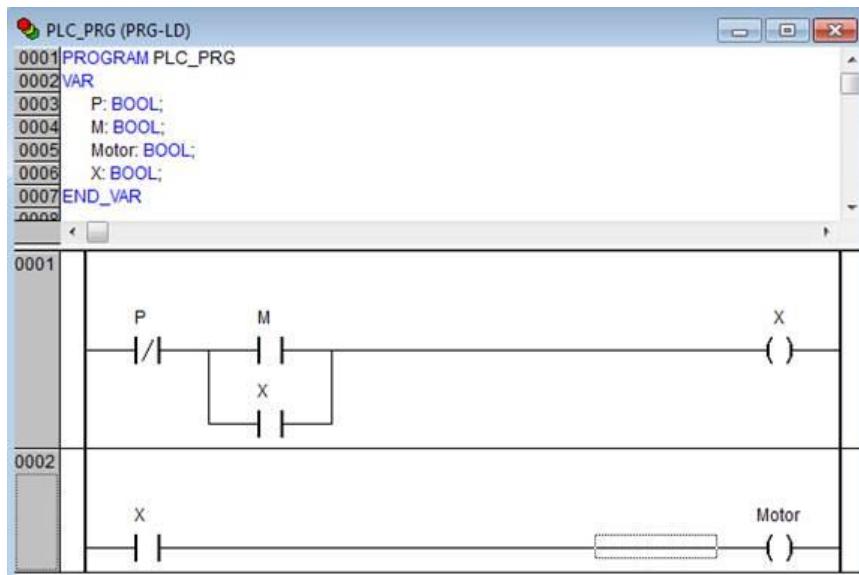


Figura 3-10 – Solución del Ejercicio 2 (II)

Por tanto, las señales de salida que ponen el motor en marcha en cada uno de los sentidos de giro son respectivamente las bobinas de los contactores<sup>2</sup>  $KM1$  y  $KM2$ . Estas serán las señales que deberá activar o desactivar el automatismo que se diseñará.

<sup>2</sup> Los contactores son relés que abren circuitos de potencia

Para realizar este ejercicio se utilizará el fichero *Cap3\_3.pro*. Al abrir el fichero se comprueba que hay algunas diferencias con los ejemplos anteriores. En primer lugar las variables de entrada y salida están declaradas como variables globales y por tanto son accesibles por todas las POU's que escribamos en este proyecto.

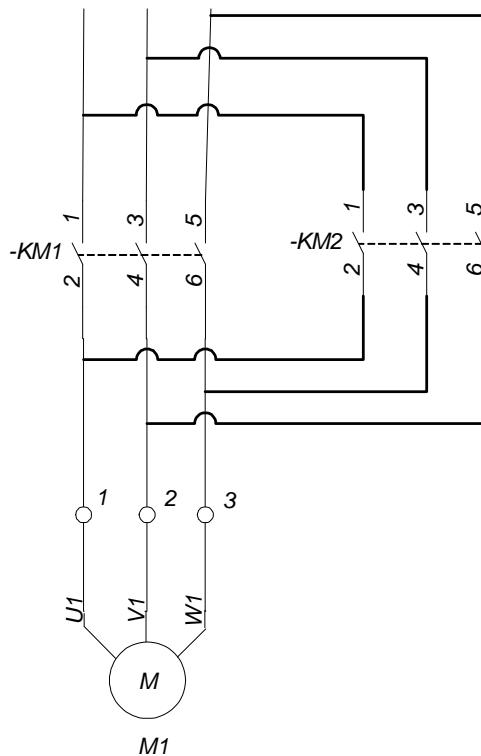


Figura 3-11 – Conexiónado para cambio de giro del motor

En este sentido conviene volver a recalcar que las tres señales de entrada y las dos señales de salida definidas anteriormente deberían estar conectadas a los módulos de E/S del autómata y acceder a ellas a través de sus correspondientes variables lógicas de E/S (tal como se definirá en el siguiente capítulo). A partir de este momento, y en el resto de ejemplos de este libro, al trabajar solo en simulación,

estarán definidas como variables globales, y estará indicando expresamente mediante comentarios cuándo se trata de variables de entrada o de salida. De este modo, si cualquiera de los ejercicios de este libro se quisiera implementar en un autómata conectado a señales físicas, las modificaciones que habría que hacer en el programa serían mínimas.

Para abrir la ventana de variables globales es necesario seleccionar la pestaña *Resources* del Organizador de Objetos y seleccionar *Global\_Variables* (Figura 3-12).

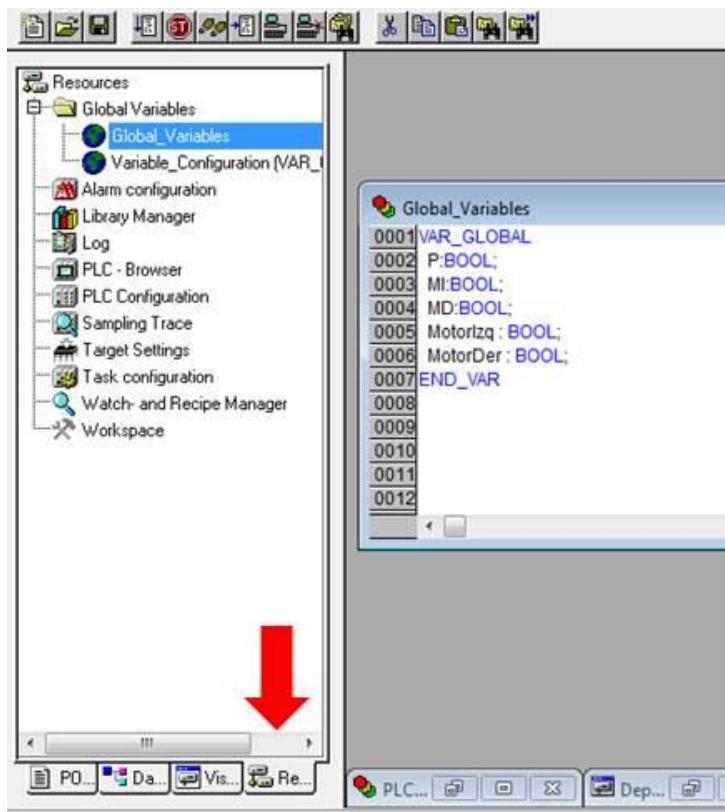


Figura 3-12 – Acceso a la ventana de variables globales

Además se puede observar en el proyecto abierto que hay dos POUs. La primera de ellas (PLC\_PRG) se encarga de realizar la animación en la ventana de visualización

y de llamar al programa en LD que se va a diseñar. Está escrito en texto estructurado (ST) y por ahora no nos ocuparemos de él. El segundo programa que se denomina *Control*, en LD, está vacío y diseñarlo es el objeto de este ejercicio.

Se propone la resolución de este ejemplo en dos casos distintos:

- **Caso 1:** Diseñe el sistema de control de modo que para cambiar de sentido de giro sea necesario parar previamente el motor con el pulsador *P*.
- **Caso 2:** Diseñe el sistema de modo que no sea necesario parar el motor para cambiar de sentido de giro.

Como se indicó anteriormente, hay que tener la precaución que en ningún momento se pueden tener simultáneamente activados las señales *KM1* y *KM2* ya que esto dañaría el motor.

En el fichero *Cap3\_3\_Res.pro* se encuentra resuelto el primer caso.

### 3.8. Ejemplo 4. Control de un cilindro neumático (I)

Se desea controlar un cilindro neumático de doble efecto, para lo que se dispone de una válvula 5/2, tal como se muestra en la Figura 3-13. El cilindro de doble efecto es un dispositivo neumático en el que el émbolo se mueve en un sentido u otro dependiendo de por cuál de las dos tomas que se muestran en la figura entre aire comprimido. La otra toma debe permitir la salida del aire.

La selección de la toma por la que entra el aire se realiza mediante una válvula neumática, representada en la parte inferior de la Figura 3-13. Hay muchos tipos de válvulas, pero la que se muestra en la figura es de 5 vías: las dos superiores están conectadas a cada una de las tomas del cilindro, la vía inferior central está conectada a la fuente de aire comprimido y las dos laterales abiertas a la atmósfera para permitir la salida de aire. Estas cinco vías se pueden conectar internamente de distintas formas que se denominan *posiciones*. La válvula de la figura tiene dos posiciones cada una representada en el esquema por un cuadrado en el que se dibuja la forma en que están dispuestas las conexiones. Esta válvula de 5 vías y 2 posiciones se denomina válvula 5/2. La válvula siempre estará en una de las dos posiciones.

Como se observa en la figura, la válvula se encuentra en la posición *p2* en la que el

aire comprimido entra por la toma derecha del cilindro, mientras que la toma izquierda está conectada a una de las vías abiertas a la atmósfera. En esta posición el cilindro se desplaza hacia la izquierda.

Si la válvula se cambia a la posición  $p_1$ , el aire entraría por la toma izquierda del cilindro mientras que la toma derecha quedaría conectada a la otra vía abierta permitiendo la salida de aire a la atmósfera. De esta manera el cilindro se desplazará a la derecha.

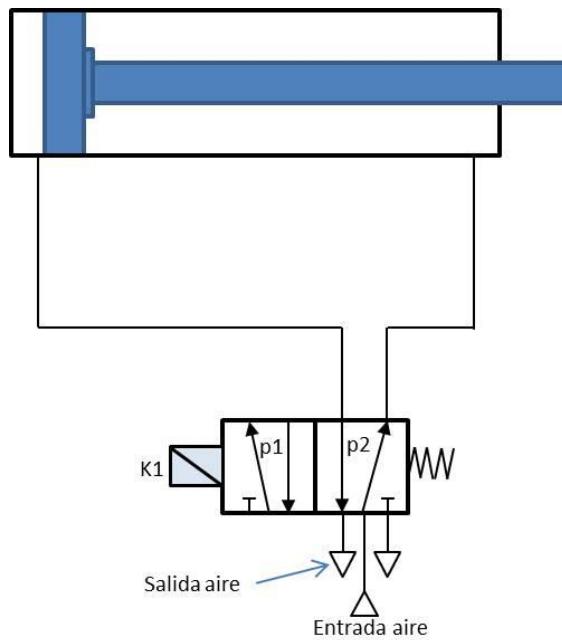


Figura 3-13 – Cilindro doble efecto y válvula 5/2

El cambio de posición se realiza de forma externa por distintos medios: manual, eléctrico, neumático, etc. En la figura, el cambio de la posición  $p_2$  a la  $p_1$  se realiza activando la bobina  $K1$ , mientras que el paso de la posición  $p_1$  a la  $p_2$  la realiza un muelle, que actúa en el momento que no está activada la bobina.

El funcionamiento del cilindro debe ser el siguiente (Ver Figura 3-14): Al activar el pulsador *Encendido*, el sistema de control deberá extender el cilindro hasta el final

del recorrido y volver a la posición de reposo (en la que se encuentra en la figura) donde se detendrá.

Para detectar los finales del recorrido del émbolo, el cilindro dispone de dos fines de carrera (*A* y *B*) en la figura. En la posición de reposo el cilindro tiene activado uno de los fines de carrera (*A*). Por tanto el sistema de control deberá actuar de forma adecuada sobre la bobina *K1*. Se trata pues de un automatismo con tres entradas (*Encendido*, *A* y *B*) y una salida *K1*.

El sistema se encuentra implementado en *Cap3\_4.pro* y se deja como ejercicio al lector.

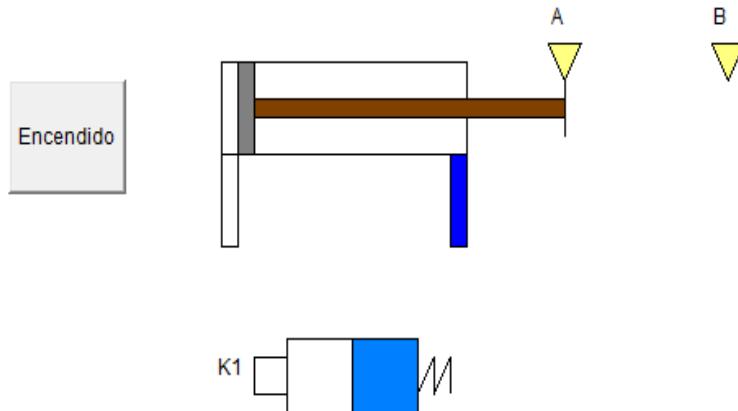


Figura 3-14 – Sistema a controlar en Ejemplo 4.

### 3.9. Ejemplo 5. Control de un cilindro neumático (II)

Este ejemplo consiste en repetir el mismo funcionamiento anterior pero considerando que la válvula no tiene retorno automático, sino que se mueve en los dos sentidos activando sendas bobinas. Para mover el cilindro a la derecha será necesario activar la bobina *K1* y para moverlo a la izquierda *K2*.

En la Figura 3-15 se representa el sistema, que se encuentra en el fichero *Cap3\_5.pro*. Este ejemplo se deja como ejercicio.

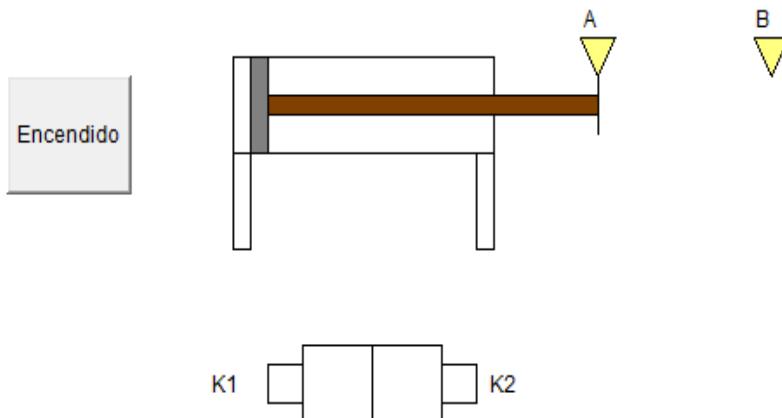


Figura 3-15 – Sistema a controlar en Ejemplo 5.

### 3.10. Ejemplo 6. Control del nivel de un depósito

Se desea controlar el nivel de un depósito. La entrada de agua se realiza mediante la activación de una bomba (señal de salida *Bomba*). El depósito tiene una toma de salida para satisfacer unos consumos, pero sobre este desagüe no se tiene control. La única acción de control que se realiza es encender o apagar la bomba. Así mismo se dispone de dos sensores de nivel *A* y *B*. El funcionamiento será el siguiente:

- Cuando el nivel alcance el sensor superior, la bomba se apagará.
- Cuando el nivel esté por debajo del sensor inferior, la bomba se encenderá.

El sistema automático se activará mediante el interruptor *Encendido*. La Figura 3-16 muestra el sistema.

Utilice el fichero *Cap3\_6.pro* para implementar su algoritmo de control. Para facilitar la comprensión del sistema, se encuentra implementado un control que activa la bomba con el botón de encendido.

¿Qué ocurriría si realizamos un sistema de control de nivel con un solo sensor de nivel? Realice el sistema de control y compruebe el funcionamiento.

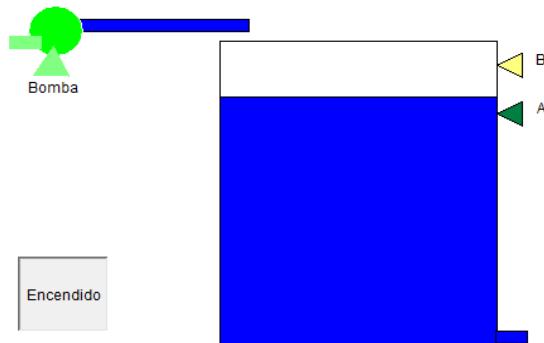


Figura 3-16 – Sistema a controlar en Ejemplo 6.



# 4. TIPOS DE DATOS Y VARIABLES

---

**E**n este capítulo se presentan los tipos de datos y el uso de variables que permite la norma IEC 61131-3. No pretende ser una “guía de referencia” completa, sino que se centrará en los elementos más comunes usados habitualmente y que se corresponden con un curso introductorio de programación de PLCs. Como se podrá comprobar, el tratamiento de los tipos de datos y declaración de variables se realizarán de una forma similar a la de cualquier otro lenguaje de programación convencional como el C. Se presupone al lector unos mínimos conocimientos en programación convencional, por lo que no se incidirá en aquellos aspectos que son comunes a dichos lenguajes y en muchos casos, en lugar de una descripción formal de la sintaxis, se recurrirá al uso de ejemplos.

## 4.1. Tipos de datos

En esta sección se van a definir los distintos tipos de datos previstos en la norma IEC-61131. Básicamente se analizarán los datos elementales, los datos derivados, con los que se podrán definir tipos de datos más complejos, y finalmente los datos de tipo genérico.

### 4.1.1. Tipos de datos elementales

Son los tipos de datos más usados en las aplicaciones de los autómatas programables. Los principales tipos de datos elementales que permite la norma, y que son muy similares a los que existen en cualquier lenguaje de programación de propósito general, son los que se indican a continuación:

- **Booleanos o cadenas de bits:**
  - BOOL, BYTE, WORD, DWORD, LWORD
- **Enteros**

- Con signo: INT, SINT, DINT, LINT
- Sin signo: UINT, USINT, UDINT, ULINT
- **Reales**
  - REAL, LREAL
- **Cadenas de caracteres**
  - STRING
- **Datos de tiempo**
  - TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME

En datos de tipo entero, real o cadena de bits, el primer carácter **D** indica doble, **L** long (largo) , **S** short (corto) y **U** unsigned (sin signo).

#### 4.1.2. Representación de datos

Con carácter general, los datos de tipo entero o real se expresan igual que en cualquier otro lenguaje de programación y por tanto no se comentará de forma expresa en este texto.

##### 4.1.2.1. Booleanos

Son variables de un solo bit, y por tanto solo pueden tomar dos valores. En general los valores que se admiten para las variables booleanas son TRUE y FALSE, aunque también se admite 1 y 0.

##### 4.1.2.2. Cadenas de bits

Los distintos tipos de variables de cadena de bits se diferencian en el número de bits. Por ejemplo, una variable tipo BYTE suele tener 8 bits, una tipo WORD 16 y una tipo DWORD tiene 32 bits. Se pueden expresar en distintas bases. Por ejemplo, 11, 16#0B y 2#0000\_1011 serían las codificaciones en base decimal, hexadecimal y notación binaria.

##### 4.1.2.3. Cadena de caracteres

El texto se expresará entre comillas simples

#### 4.1.2.4. Variables de tiempo

Son las variables que suponen una diferencia mayor con respecto a otros lenguajes de programación. La gestión de tiempo es de suma importancia en los autómatas programables, así por ejemplo, los temporizadores son uno de los elementos de más importancia en una gran parte de las aplicaciones que se realizan en estos dispositivos. Para facilitar esta gestión del tiempo, la norma IEC propone una serie de tipos de datos orientados específicamente a representar valores de tiempo.

En concreto, existen cuatro tipos de datos relacionados con la gestión del tiempo que son los siguientes:

- **TIME:** Son variables para indicar una duración o intervalo de tiempo. El formato se definirá con el siguiente ejemplo: T#2d3h34m19s34.5ms indica una duración de 2 días, 3 horas, 34 minutos, 19 segundos y 34.5 milisegundos. No es necesario incluir todas las unidades. Por ejemplo, T#10s representa una duración de 10 segundos.
- **DATE:** Fecha especificada en el formato D#año-mes-día. Por ejemplo, D#2014-06-10.
- **TIME\_OF\_DAY:** En formato TOD#horas:minutos:segundos:centésimas de segundo. Por ejemplo, TOD#13:23:45:44.
- **DATE\_AND\_TIME:** Se combinan DATE y TIME\_OF\_DAY. Por ejemplo, DT#2013-10-09-13:23:45:44.

#### 4.1.3. Tipos de datos derivados

Además de los datos elementales, el usuario puede definir sus propios tipos de datos. Estas definiciones se escriben entre TYPE y END\_TYPE. De esta manera se pueden definir nuevos tipos de datos enumerados, vectores, estructuras, variables con rango, etc., tal como se muestra en el Ejemplo 4-1.

En el ejemplo hay definido cinco tipos distintos. *Mireal* sería un tipo totalmente equivalente a LREAL. Las variables que se definan de tipo *Color* podrán tener únicamente tres valores (*rojo*, *amarillo* o *verde*). Las variables de tipo *Sensor* serán valores enteros, pero su valor deberá estar siempre entre -128 y 128. En caso contrario durante la ejecución daría un aviso o error. *Medida1D* es un tipo cuyas variables serán vectores de 45 elementos de tipo *Sensor*. *Medida2D* es una matriz de

variables de tipo *Sensor*. Finalmente el tipo *Miestructura* es un tipo de dato compuesto, y su definición y uso es muy similar al de las estructuras en lenguaje C.

#### *Ejemplo 4-1. Ejemplo de declaración de tipos de datos derivados*

---

```
TYPE
    MiReal : LREAL;
    Color : (rojo, amarillo, verde);      (* Tipo enumerado*)
    Sensor : INT (-128..128);           (*Entero con rango*)
    Medida1D : ARRAY [1..45] OF Sensor;   (*Vectores*)
    Medida2D : ARRAY [1..45,1..45] of Sensor; (*Matriz*)
    MiEstructura:
        STRUCT
            Valor: INT;
            Med1 : Medida1D;
            Luz : Color;
        END_STRUCT;
END_TYPE
```

---

Estos tipos de datos podrán ser utilizados posteriormente en las declaraciones de variables exactamente igual que los tipos elementales.

---

```
VAR
    Var1: Mireal;
    VarEst : MiEstructura;
END_VAR
```

---

#### **4.1.4. Tipos de datos genéricos**

La norma también incluye la posibilidad de utilizar tipos de datos genéricos. **Este tipo de datos NO está incluido en la presente versión de CoDeSys**, aunque se presenta aquí como parte que es de la norma.

Los tipos de datos genéricos permiten agrupar los tipos de datos elementales en grupos. Estos tipos de datos empiezan con el prefijo ANY. Por ejemplo ANY\_INT engloba a todos los tipos de enteros que se detallan en la lista anteriormente vista.

El uso fundamental de este tipo de datos es permitir definir funciones que permitan operar con más de un tipo de datos. Por ejemplo, si definimos la función SUMA, es muy deseable poder operar con cualquier dato de tipo entero o real. Para eso, definiremos sus parámetros de entradas del tipo ANY\_NUM. Algunos de los tipos enumerados son:

- ANY\_NUM : Todos los reales y enteros
- ANY\_INT: Todos los enteros, tanto con signo como sin signo
- ANY\_REAL: Todos los reales
- ANY\_BIT: Booleanos y cadenas de bits.

## 4.2. Variables

Todas las variables deben ser declaradas, o bien en la zona de declaración de variables de alguna de las POU's o bien como variables globales, tal como se explicó anteriormente. A continuación se muestra un ejemplo de la zona de declaración de variables de una POU.

---

```
VAR_INPUT
    Param1: BOOL;
    Param2 : INT;
END_VAR
```

```
VAR_OUTPUT
    Sal : REAL;
END_VAR
```

```
VAR
    Aux2 : BOOL;
END_VAR
```

---

Se puede comprobar que las variables están agrupadas en bloques de acuerdo a la clase de la variable. Tal como se ve en el ejemplo anterior, las variables pueden ser de varias clases, siendo las más importantes:

- **VAR:** Variables locales a la POU donde esté definida. Una variable declarada de esta forma solo podrá ser utilizada dentro de la POU donde está declarada.
- **VAR\_INPUT:** Parámetros de entrada a una POU.
- **VAR\_OUTPUT:** Parámetro de salida de una POU. Se utiliza para devolver valores desde una POU.
- **VAR\_IN\_OUT:** Simultáneamente de entrada y salida. Se pueden utilizar tanto para pasarle valores a la entrada como para devolver valores.
- **VAR\_GLOBAL:** Son variables globales a todas las POUs y por tanto se puede usar desde cualquiera de ellas. En CoDeSys se declaran en la lista de variables globales. Otros entornos de programación permite declarar esta clase de variables dentro de una POU. En ese caso, en el resto de los POUs donde se vaya a utilizar, hay que declararla como VAR\_EXTERNAL.

#### 4.2.1. Atributos

Además de las clases de variables definidas en la sección anterior, a éstas se les pueden añadir unos atributos definidos en la norma. Algunos de estos atributos son:

- **CONSTANT:** La variable no puede ser modificada durante la ejecución del programa.
- **RETAIN:** La variable se almacenan en un back-up mantenido por baterías, de manera que mantienen su valor incluso en el caso que el PLC pierda la alimentación
- **PERSISTENT:** Mantienen el valor de la variable solo después de la recarga del código (download) pero no en una pérdida de alimentación.

Un ejemplo de declaración de una variable con el atributo CONSTANT sería:

---

```
VAR CONSTANT
    Variable : BOOL;
END_VAR
```

---

#### 4.2.2. Variables de Entrada/Salida

Como se ha indicado anteriormente, cada una de las conexiones físicas de E/S que llegan al autómata tiene una variable lógica asociada. Esta variable lógica es la que se usará en el código y servirá para leer los valores de las entradas físicas y asignarle valor a las salidas. Es por tanto necesario establecer la correspondencia entre cada señal física de E/S y su correspondiente variable lógica.

Tabla 4-1 Sintaxis de las variables de entrada/salida

Dirección			Explicación
%			Primer carácter obligatorio
	I Q M		Entrada Salida Memoria auxiliar
	X B W D L		Si no aparece el tercer carácter, es tipo bit bit byte word Doble word Long word
		v.w.x.y.z	Dirección que depende del fabricante: número de señal, número de tarjeta, número de bastidor,..

Para ello existen dos alternativas:

- Usar el nombre de la dirección física según las especificaciones de la norma.
- Usar un nombre de variable definido por el usuario, que será necesario asociar a la correspondiente dirección física. Esta alternativa permite facilitar la legibilidad del código, asignando las variables de E/S con un nombre significativo, por ejemplo *SensorTemperatura* o *LuzAlarma*.

Las direcciones físicas tienen el formato que se indica en la Tabla 4-1. Empiezan por el carácter "%" seguido de I, Q o M dependiendo de si la variable es de entrada, salida o interna; otra letra indicando el tipo de dato y finalmente la dirección física en la que se encuentra cableada la señal.

Por ejemplo, %I10 hace referencia a la entrada booleana número 10, mientras que %QW3.1 es la salida de tipo WORD número 1 correspondiente al módulo 3. En cualquier caso, para conocer las direcciones específicas de un autómata concreto hay que recurrir a las especificaciones del fabricante.

Si en lugar de utilizar la dirección física, se quiere asignarle un nombre distinto a la variable, en la declaración de estas variables se utilizará la palabra AT de acuerdo al formato indicado en el ejemplo siguiente. Una vez usado este tipo de declaración, para acceder a dicha variable se pueda utilizar de modo indistinto la dirección física o el nombre de variable.

---

```
VAR
    MiEntrada AT %I2 : BOOL;
    MiSalida AT %Q7 : BOOL;
END_VAR
```

---

En los ejercicios propuestos en este libro, al trabajar siempre en simulación, las variables de E/S estarán siempre definidas como variables globales, y se usará y declarará exclusivamente el nombre de variable. Nótese que si se quisiera ejecutar cualquiera de estos programas en un autómata programable, simplemente será necesario añadir de la manera arriba indicada (con AT), una dirección física a cada una de los nombre de variables que se correspondan con entradas o salidas. El código no será necesario modificarlo, ya que el acceso a las variables de E/S se puede seguir realizando con el nombre de la variable. Por ejemplo, en el ejemplo 2 del capítulo anterior, será necesario hacer la siguiente modificación:

---

```
VAR_GLOBAL
    M AT %I1: BOOL;          (* Pulsador de marcha *)
    P AT %I7: BOOL;          (* Pulsador de parada *)
    Motor AT %Q1: BOOL;      (* Activación del motor *)
END_VAR
```

---

#### 4.2.3. Valores iniciales

En la declaración de variables se pueden especificar valores iniciales a dichas variables. El formato se muestra en el ejemplo siguiente:

---

```
VAR
    Var1 : BOOL := FALSE;
    Var2 : INT := 12;
END_VAR
```

---

#### 4.2.4. Matrices y estructuras

Tal como se detalló anteriormente, es posible declarar variables de tipo vector o matriz. El formato se muestra a continuación con un ejemplo.

---

```
VAR
    Vector : ARRAY [0..10] OF BOOL;
    Matriz : ARRAY[1..20,2..100] OF INT;
END_VAR
```

---

Para acceder a los elementos de un vector o matriz se realiza de la siguiente forma:

---

```
Vector[5]
Matriz[2,65]
```

---

Las estructuras son tipos de datos compuestos que están formados por otros tipos de datos. El uso de las estructura en la norma IEC-1161-3 es muy similar al del lenguaje C. Como se indicó en la sección anterior, las estructuras se definen con TYPE, tal como se muestra en el ejemplo.

---

```
TYPE
  MiEstructura:
  STRUCT
    Var1 : BOOL;
    Vector1 : ARRAY [0..10] OF INT;
  END_STRUCT
END_TYPE
```

---

Una vez definidas las estructuras, se pueden declarar variables de dichos tipos, tal como se indica a continuación.

---

```
VAR
  VarEstruc : MiEstructura;
  VectorEstruc : ARRAY[1..100] OF MiEstructura
END_VAR
```

---

En el ejemplo anterior, se ha definido un tipo de datos denominado *Miestructura* que consta de un dato booleano y de un vector de 11 enteros. Posteriormente se han declarado dos variables utilizando el tipo *Miestructura*, una variable y un vector de 100 elementos.

Para acceder a las variables internas de una estructura se utiliza el operador “.”, la forma de hacerlo es exactamente igual que en el lenguaje C. En el siguiente ejemplo

se muestran algunos casos.

---

```
VarEstruc.Var1
VarEstruct.Vector[5]
VectorEstruc[23].Var1
VectorEstruc[45].Vector[2]
```

---

El lector deberá conocer a qué dato individual accede cada una de las variables anteriores.

### 4.3. Definición de tipos de datos derivados en CoDeSys

La definición de tipos de datos derivados en CoDeSys se realiza desde la pestaña *Data Types* del Organizador de Objetos (Ver Figura 4-1). En CoDeSys cada tipo de datos derivado se define de forma individual con un objeto nuevo desde esta pestaña. Para añadir un tipo nuevo, el cursor debe estar en el Organizador de Objetos y con el botón derecho del ratón abrir el menú contextual. Se selecciona la opción *Add Object*. A continuación se le asigna un nombre al nuevo tipo de datos y se edita el dato desde la ventana de edición.

En la Figura 4-2 se muestra la ventana de edición de un tipo de datos enumerado y otro de tipo estructura.

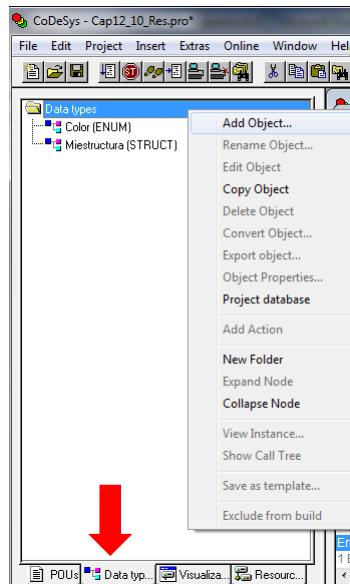


Figura 4-1 – Nuevos tipos de datos en CoDeSys.

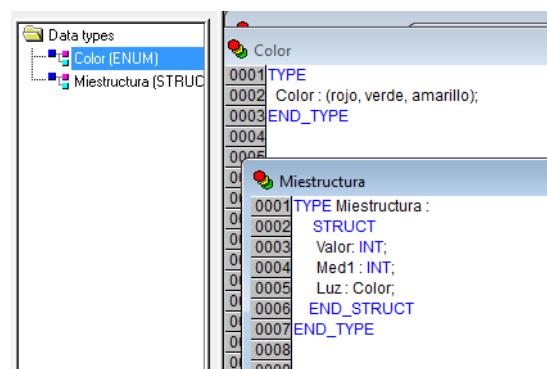


Figura 4-2 – Ventana de edición de nuevos tipos de datos en CoDeSys.

# 5. UNIDADES DE ORGANIZACIÓN DE PROGRAMAS (POU)

---

**L**a norma IEC 61131-3 denomina a los bloques de código que definen el Proyecto como *Program Organization Units* (POU). Un Proyecto estará compuesto por una o varias POUs, que pueden estar escritos cada una de ellas en alguno de los distintos lenguajes de programación de entre los 5 que fija la norma. En este capítulo se analizarán las características de los tres tipos de POUs que define la norma: funciones, bloques funcionales y programas. La mayor parte de los ejemplos explicativos están escritos en lenguaje de texto estructurado ST. Son lo suficientemente simples para que puedan ser entendidos con un mínimo conocimiento en programación.

## 5.1. Tipos de POUs

El equivalente a una POU en la programación convencional serían las funciones, subrutinas y programas. La norma define tres tipos de POUs que difieren entre ellos en algunas características que se describirán en este capítulo. Los tres tipos de POUs son:

- **Funciones (FUN):** Pueden tener varios parámetros de entrada, pero devuelven un único dato que puede ser de cualquiera de los tipos elementales o derivados que se vieron con anterioridad, incluyendo lógicamente un dato de tipo estructura. No guarda ninguna información interna, por lo que si se llama dos veces con los mismos valores en los parámetros de entrada, devuelve siempre el mismo valor de salida. En su código pueden hacer llamadas a otras funciones, pero no a bloques funcionales o programas.
- **Bloques Funcionales (FB):** También pueden tener parámetros de entrada, pero pueden devolver más de un valor de salida, que se definen todos ellos en su cabecera. La diferencia fundamental con relación a las funciones es que pueden

guardar información interna, de modo que los valores de salida van a depender tanto de los valores de los parámetros de entrada como de las variables que definen el estado interno del FB. En su código pueden tener llamadas a funciones u otros bloques funcionales, pero no a programas. Como veremos, la forma de llamarlas es radicalmente distinta a las FUN, ya que están basadas en la creación de lo que se denominan *Instancias*.

- **Programas (PRG):** Los programas también pueden tener parámetros de entrada y de salida. Los programas pueden llamar a funciones, bloques funcionales y otros programas, pero como se dijo anteriormente, no pueden ser llamados desde funciones o bloques funcionales. Un ejemplo significativo en CoDeSys es el programa PLC\_PRG, programa que debe estar presente en cualquier proyecto y es por el que empieza la ejecución del proyecto, de forma equivalente a como lo hace *main* en lenguaje C.

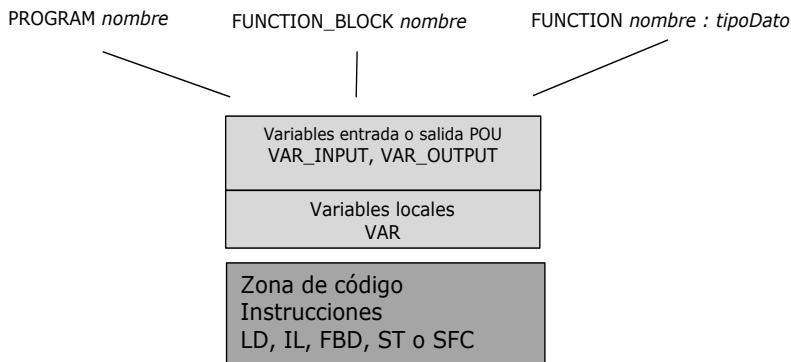


Figura 5-1 – Estructura de un Programa Organization Unit (POU).

La estructura general de un POU está representada en la Figura 5-1. Fundamentalmente se distinguen dos zonas, la zona de declaración y la zona de código. La zona de declaración siempre empieza con la línea de declaración del nombre del POU y su tipo (PROGRAM, FUNCTION\_BLOCK o FUNCTION). En el caso de la función se especifica también el tipo de datos que devuelve. A continuación se declaran las variables de entrada y salida (cuando proceda) y finalmente las variables locales. Dependiendo del lenguaje que se escoja, la zona de edición de código será textual o gráfica.

## 5.2. Declaración de variables en CoDeSys

Para declarar las variables de una POU, CoDeSys tiene dos posibilidades que ya fueron usadas en el Capítulo 3:

- Escribir texto en la zona de declaración de variables en el formato que se ha indicado con anterioridad.
- Usar el asistente de declaración de variables.

En este último caso, al introducir en la zona de código alguna variable que no ha sido previamente declarada, aparecerá una ventana donde es posible definir todo lo necesario para la declaración de la variable. Esta ventana también aparece pulsando <SHIFT-F2>. Si al pulsarlo, el cursor se encuentra encima de una variable, aparecerá la ventana con los datos de dicha variable. Al pulsar OK automáticamente se actualizará la zona de declaración de variables del POU.

La Figura 5-2 muestra la ventana del asistente de declaración de variables.

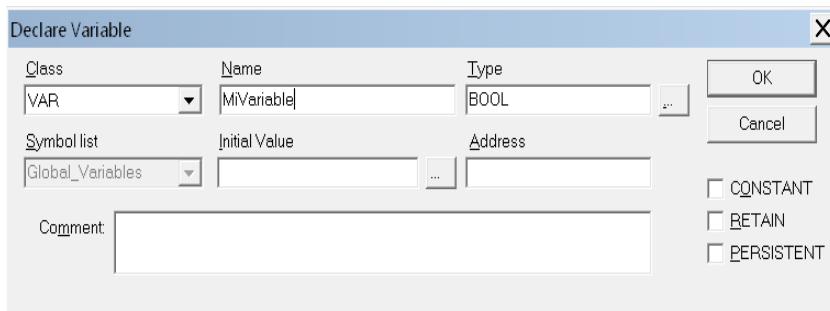
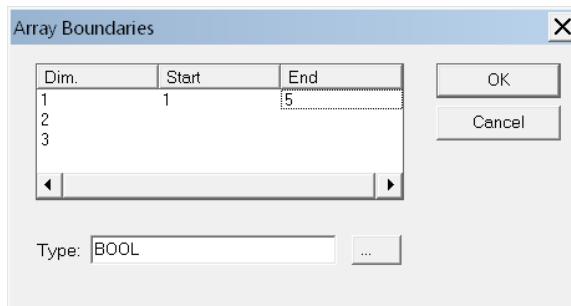


Figura 5-2 – Ventana de asistente de declaración de variables.

Los datos básicos que se introducen para la declaración de cada variable son:

- **Class:** Se indica si la variable es VAR, VAR\_INPUT, VAR\_GLOBAL...
- **Name:** Nombre de la variable
- **Type:** El tipo de dato de la variable. Se puede escribir directamente o seleccionar pulsando el botón de la derecha. Se puede acceder tanto a los tipos elementales como a los derivados. Si se escoge el tipo ARRAY

aparecerá una nueva ventana que permite introducir las dimensiones y el tipo de elemento



- **Initial Value:** Se escribirá un valor inicial en función del tipo de dato declarado. Si el dato es de tipo ARRAY el valor inicial se puede dar de forma cómoda utilizando el botón que aparece a la derecha de este campo
- **Address:** Se utiliza cuando se quiere asociar la variable a una determinada dirección física, normalmente de E/S. Se corresponde con la instrucción AT vista anteriormente.
- Atributo CONSTANT, RETAIN, PERSISTENT

### 5.3. Bloques funcionales

Los bloques funcionales (FB) son muy utilizados en la práctica. Pueden ser llamados desde programas u otros bloques funcionales y por otro lado, desde dentro de un bloque funcional se pueden llamar a otros bloques funcionales o a funciones.

El concepto de *Instancia* es el elemento clave en los FB y es lo que los distingue fundamentalmente de otros POU's. La idea de instancia es la misma que se utiliza en la declaración de variables. Si tenemos un tipo de variables, por ejemplo INT, podemos declarar variables de este tipo, tantas como queramos, cada una identificada con un nombre de variable y cada una, en cada momento de la ejecución “guarda” un valor distinto.

```
Mientero : INT;
```

Si se declara una variable de tipo INT, por ejemplo *MiEntero*, se puede decir que

*MiEntero* es una *instancia* de un dato entero tipo INT.

Esta idea es la que se aplica a los bloques funcionales. Si se define un tipo de bloque funcional, por ejemplo *Contador*, con una sola entrada booleana y cuyo objetivo es contar el número de flancos de subida en la señal de entrada, se pueden definir varias *instancias* de tipo *Contador*, y se puede usar cada una de ellas para contar el número de flancos de subida de señales distintas. En la ejecución del programa, cada uno de ellos almacenará el número de flancos de subida de la señal conectada a su correspondiente entrada, que evidentemente serán en general distintos unos de otros.

En definitiva, igual que en programa se pueden tener muchas variables INT cada una con una información distinta (aunque el mismo tipo de información en todas), también se pueden tener muchas *instancias* del bloque funcional de tipo *Contador*, cada una con un valor interno distinto, pero todas estructuralmente iguales.

De hecho, la declaración de las instancias de bloques funcionales se realiza exactamente igual que la declaración de variables.

---

```
VAR
    ContA : Contador;
    ContB: Contador;
END_VAR
```

---

Una vez declarada, la instancia puede ser usada dentro de la POU donde ha sido declarada. También es posible declararlas como VAR\_GLOBAL, en cuyo caso puede ser utilizada desde todos los POUs.

En la literatura, es habitual que con el término *Bloque Funcional* se haga referencia tanto a la instancia como al tipo (y por tanto al código escrito que la define). Es importante resaltar que se trata de dos conceptos distintos.

En el Ejemplo 5-1 se define un bloque funcional que implementa un biestable S-R con prioridad al Set. Tendrá dos variables de entrada (definidas en el bloque VAR\_INPUT) y una de salida (en el bloque VAR\_OUTPUT), que se corresponde con el estado del biestable. Si hubiera variables locales al bloque funcional se declararían en un bloque VAR.

**Ejemplo 5-1. Código del bloque funcional Biestable**


---

```

FUNCTION_BLOCK Biestable
VAR_INPUT
    EntS : BOOL;
    Entr: BOOL;
END_VAR
VAR_OUTPUT
    Estado : BOOL :=TRUE;
END_VAR
-----
IF (EntS=TRUE) THEN
    Estado:=TRUE;
ELSE
    IF (Entr=TRUE) THEN
        Estado:=FALSE;
    END_IF
END_IF

```

---

En el fichero *Cap5\_1.pro* está implementado este bloque funcional y se puede comprobar su funcionamiento con la ayuda de la visualización asociada, en la que hay dos botones (*BotonR* y *BotonS*) mientras que el estado del biestable se muestra con el círculo rojo o verde (*Salida*). Estas tres variables son las de E/S y están definidas como variables globales.

Aunque después se volverá sobre el tema, en el ejemplo, el bloque funcional (es decir, la instancia) está llamada desde el programa PLC\_PRG (escrito en FBD). Puede comprobar que el programa únicamente consiste en la definición de una instancia de tipo *Biestable* que hemos denominado *MiBiestable*, y que el código solo incluye una llamada con el nombre de la instancia (*MiBiestable*) y con los correspondientes parámetros de entrada y salida.

Téngase en cuenta que durante la ejecución, esta instancia de bloque funcional está siendo llamada en cada ciclo del autómata, es decir una vez cada pocos milisegundos. En cada ejecución, se comprueba si está a TRUE *EntR* o *EntS*, y si no lo está ninguna (cosa que ocurre en la mayor parte de los ciclos) el biestable no cambia de estado. Compruébese que el valor inicial que tiene asignado la variable *Estado* solamente se utiliza en el primer ciclo, es decir, la primera vez que se ejecuta la instancia.

### 5.3.1. Estructura de datos asociada a un bloque funcional

Cuando se crea un bloque funcional, automáticamente es creada una estructura de datos con el mismo nombre que el bloque funcional y que incluye los datos de entrada y salida de dicho bloque. En el ejemplo anterior se crearía la siguiente estructura:

---

```
TYPE Biestable
STRUCT
(* Entradas *)
    EntS : BOOL;
    EntR : BOOL;
(* Salidas *)
    Estado :BOOL;
END_STRUCT
END_TYPE
```

---

Por tanto, cuando se hace la declaración de una instancia

MiBiestable: Biestable;

automáticamente se crea una variable de dicho tipo de estructura con el mismo nombre que la instancia, con la que se puede acceder a los parámetros de entrada y salida de la instancia. El acceso se hace del mismo modo que se haría con cualquier otra estructura, es decir, *MiBiestable.EntS* o *MiBiestable.Estado*.

Adicionalmente, a lo largo de la ejecución el sistema almacena los valores de todas las variables locales de cada instancia, además de los de entrada y salida. Es necesario tener esto en mente cuando en un Proyecto de utilizan bloques funcionales con gran cantidad de variables locales, por ejemplo matrices grandes, y además, se crean muchas instancias de ese FB, ya que se podría llegar a tener problemas de falta de memoria.

### 5.3.2. Llamada al bloque funcional

Para llamar a un bloque funcional desde otro programa o bloque funcional, es

necesario en primer lugar declarar una instancia en la zona de declaraciones de la POU (o como variable global). La llamada se hace con el **nombre de la instancia (no del bloque funcional)** con los correspondientes parámetros de entrada y salida. La llamada va a depender del lenguaje utilizado y se verá más en detalle posteriormente, aunque a título de ejemplo se muestra una llamada en lenguaje estructurado ST desde el programa PLC\_PRG en el Ejemplo 5-2.

En dicho ejemplo se supone que existen dos pulsadores (*BotonS* y *BotonR*) que se quiere que realicen respectivamente el SET y el RESET y que cuando el biestable está activado (*Estado*=TRUE) se encienda la salida *LuzSalida*. Nótese que a cada uno de los parámetros de entrada se le asocia su correspondiente valor utilizando el operador “:=” y a los de salida mediante el operador “=>”

#### *Ejemplo 5-2. Llamada al bloque funcional con la instancia MiBiestable*

---

```
PROGRAM PLC_PRG
VAR
    BotonS : BOOL;
    BotonR : BOOL;
    LuzSalida : BOOL;
    MiBiestable : Biestable;
END_VAR
-----
MiBiestable(EntS:=BotonS, EntR:=BotonR, Estado=>LuzSalida);
```

---

Otra alternativa sería utilizar la estructura definida anteriormente indicada. En el Ejemplo 5-3 se muestra una forma alternativa a la utilizada en el ejemplo anterior. Sobre este tema se incidirá más tarde cuando se analicen las llamadas a bloques funcionales desde los distintos lenguajes de la norma.

La misma llamada utilizando el lenguaje de bloque FBD se muestra en la Figura 5-3.

*Ejemplo 5-3. Llamada al bloque funcional con la instancia MiBiestable (II)*

```
PROGRAM PLC_PRG
VAR
    BotonS : BOOL;
    BotonR : BOOL;
    Salida : BOOL;
    MiBiestable : Biestable;
END_VAR
-----
MiBiestable.EntS:=BotonS;
MiBiestable.EntR:=BotonR;
MiBiestable();
Salida:=MiBiestable.Estado
```

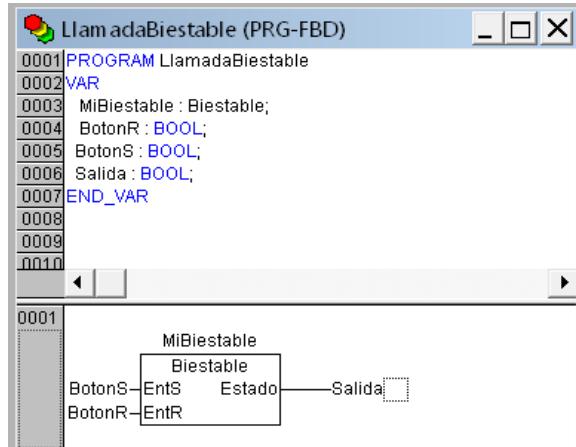


Figura 5-3 – Llamada a un bloque funcional en lenguaje FBD.

### 5.3.3. Restricciones en el uso de bloques funcionales

- Un bloque funcional puede llamar a funciones u otros bloques funcionales pero no a programas.
- No se pueden hacer asignaciones a direcciones físicas del PLC (%Q, %I). Con esto se consigue que los bloques funcionales sean independientes del hardware.

## 5.4. Funciones

Las funciones en la norma IEC son similares a las de cualquier otro lenguaje de programación como el C. Pueden tener varios parámetros de entrada pero devuelven un único dato que puede ser de cualquier tipo. Una función a la que se le suministra los mismos parámetros de entrada devuelve siempre el mismo valor, al contrario que ocurría en los bloques funcionales que también dependían del estado interno. Tampoco se crean instancias, ni estructura de datos asociada, ni almacena los valores de las variables internas.

A título de ejemplo se muestra una función que suma dos valores. El valor lo devuelve en una variable que se llama igual que la función:

---

```
FUNCTION  suma:INT
VAR_INPUT
    sumando1: INT;
    sumando2: INT;
END_VAR
-----
suma:=sumando1+sumando2; (* en lenguaje ST *)
```

---

Como se puede observar, en la primera línea de la declaración de la función se especifica el tipo de dato que devuelve dicha función.

La llamada a la función desde otra POU se realizaría en lenguaje ST de la siguiente manera:

```
Valor:=SUMA(Sumando1:=1, Sumando2:=3);
```

La norma también permite hacerlo de una forma más simplificada del siguiente modo:

```
Valor:=SUMA(1, 3);
```

La llamada a la función en lenguaje FBD sería de la forma que se indica en la Figura 5-4.

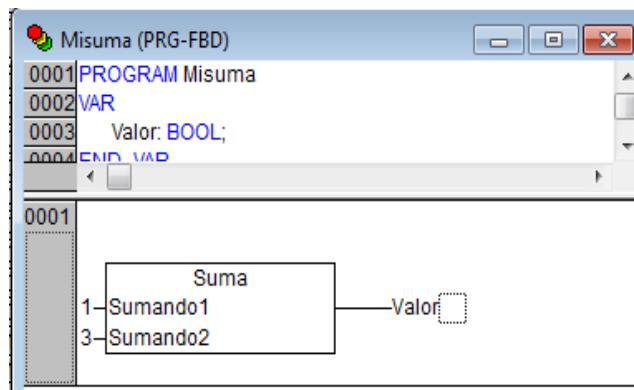


Figura 5-4 – Llamada a una función en lenguaje FBD.

#### 5.4.1. Diferencia en el funcionamiento entre funciones y bloques funcionales

Para aclarar la diferencia entre una función y un bloque funcional, en el fichero *Cap5\_ej2.pro* está escrita, tanto como función como en forma de bloque funcional, una rutina que a una variable interna (*contador*) le suma una unidad. El objetivo de este ejemplo es didáctico, ya que la función declarada no tendría ninguna utilidad práctica. Nótese las diferencias que existen tanto en la declaración como en la llamada. En ambos casos la variable está inicializada a 0. Compruebe que el funcionamiento de ambas es totalmente distinto debido a la “memoria” del bloque funcional.

Es necesario tener en cuenta que el programa estará ejecutándose de forma cíclica y que en cada ciclo se ejecuta tanto la función como el bloque funcional. Nótese que

en el caso del bloque funcional se guarda el valor interno de la variable local *contador*, de forma que en cada ciclo del autómata se ejecuta el bloque funcional y por tanto incrementa en una unidad *contador*. Por tanto, este bloque funcional está contando el número de ciclos que ejecuta el autómata. En el caso de la función, el valor de la variable interna se inicializa en cada ejecución a cero y cuando se ejecuta la función la variable contador pasará a valer 1.

#### 5.4.2. Restricciones en el uso de funciones

- Una función puede llamar a otras funciones, pero no a bloques funcionales ni programas
- Al igual que en los bloques funcionales, no se pueden hacer asignaciones a direcciones físicas del PLC (%Q, %I).

### 5.5. Programas

Las POU's de tipo PROGRAM constituyen los “programas principales”. En un Proyecto puede haber uno o varios programas ejecutándose. Los programas pueden tener tantos parámetros de entrada y de salida como se quiera. Las variables locales se guardan de una ejecución a otra (como en los bloques funcionales) independientemente de que se llamen desde POU's distintas, pero a diferencia de estos no se definen instancias (o también se puede ver como si se creara una única instancia global).

En CoDeSys la ejecución de un Proyecto empieza por el programa denominado PLC\_PRG. Desde este programa se pueden lanzar tantos otros programas, bloques funcionales o funciones como se quiera.

Los parámetros de entrada y salida se definen de una forma similar a la de los bloques funcionales.

### 5.6. Creación de POU's en CoDeSys

Como se ha ido indicando con anterioridad, un proyecto puede tener tantos POU's como se necesite. Cuando se crea un proyecto nuevo, por defecto se tiene una única

POU de tipo PROGRAM, que puede escribirse en cualquier lenguaje, denominado PLC\_PRG.

Para crear nuevos POUs seleccionaremos la pestaña POU del organizador de objetos. Con el botón derecho del ratón se obtiene el menú de POUs. (Ver Figura 5-5). Al seleccionar “Add Object...” aparece la ventana de creación de POUs ya vista anteriormente con lo que podemos elegir el tipo, el lenguaje y el nombre del nuevo POU.

Con este mismo menú, es posible realizar acciones con los POUs como renombrarlos, borrarlos, copiarlos, organizarlos, etc.

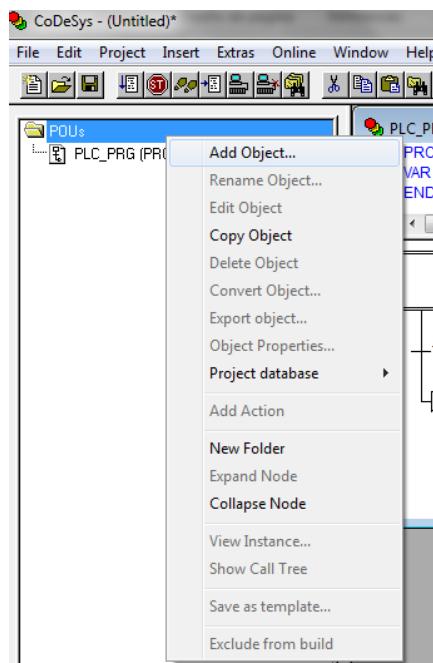


Figura 5-5 – Añadir una nueva POU.

## 5.7. Funciones y bloques funcionales estándar

CoDeSys (y la norma IEC 61131-3) proporciona un gran número de funciones y

bloques funcionales predefinidos que puede ser usados en cualquier proyecto. En esta sección se nombrarán algunos de los más usado en la práctica. Una lista completa se puede obtener en la ayuda de CoDeSys.

### 5.7.1. Algunas funciones estándar

Algunas de las funciones predefinidas en la norma de uso más habitual son:

#### 5.7.1.1. Funciones de conversión de tipo

Nombre	Descripción
*_TO_**	* y ** son dos tipos de variable. Convierte una variable de tipo * a otra de tipo **
TRUNC	Trunca un valor real convirtiéndolo a entero

Por ejemplo, en el primer caso, existen funciones como INT\_TO\_REAL, WORD\_TO\_INT, etc.

#### 5.7.1.2. Funciones numéricas

En general, las funciones matemáticas definidas en CoDeSys admiten distintos tipos de variable en los parámetros de entrada y en el de salida. Para conocer los tipos admitidos en cada función, consulte la ayuda de CoDeSys.

Nombre	Descripción
ABS	Devuelve el valor absoluto de un número
SQRT	Devuelve la raíz cuadrada de un número
LN	Devuelve el logaritmo neperiano de un número
LOG	Devuelve el logaritmo en base 10 de un número
EXP	Devuelve una exponencial en base $e$
SIN	Devuelve el seno de un número
COS	Devuelve el coseno de un número
TAN	Devuelve la tangente de un número
ASIN	Devuelve el arco seno de un número
ACOS	Devuelve el arco coseno de un número
ATAN	Devuelve el arco tangente de un número

### 5.7.1.3. Funciones aritméticas

Las funciones aritméticas también admiten en general distintos tipos de variables en los parámetros de entrada y en el de salida. Dependiendo de la función puede haber uno, dos o más parámetros de entrada, e incluso alguna de las funciones como ADD o MUL pueden tener un número variable de parámetros de entrada. Esto se muestra en la descripción de la función, donde los parámetros de entrada se indican con  $IN1, IN2, \dots, INn$  y el de salida como  $F$ . Para llamar a estas funciones también se admite el uso de operadores (cuando se programa en lenguaje ST). El símbolo se muestra en la columna *operador*.

Nombre	Operador	Descripción
ADD	+	$F=IN1+IN2+\dots+INn$ (número de entradas variable)
MUL	*	$F=IN1*IN2*\dots*INn$ (número de entradas variable)
SUB	-	$F=IN1 - IN2$
DIV	/	$F=IN1 / IN2$
MOD		Resto de una división; $F=IN1-(IN1/IN2)*IN2$
EXPT	**	$F = IN1^{IN2}$
MOVE		Asignación $F=IN$

El operador MOVE es de uso frecuente en los lenguajes gráficos LD y FBD para asignación de variables no booleanas, por ejemplo enteras o reales.

#### 5.7.1.4. Máximos y Mínimos

Estas funciones también admiten varios tipos de datos y tienen un número variable de parámetros de entrada.

Nombre	Descripción
MIN	Calcula el mínimo entre los parámetros de entrada
MAX	Calcula el máximo entre los parámetros de entrada

#### 5.7.1.5. Operaciones booleanas

Realizan operaciones booleanas bit a bit, es decir, las operaciones booleanas se realizan entre los bits que ocupan la misma posición en los parámetros de entrada y el resultado se almacena en la misma posición del parámetro de salida. Los operadores pueden ser de tipo BOOL, BYTE, WORD o DWORD. En los operadores AND, OR y XOR se admiten un número variable de parámetros de entrada.

Nombre	Descripción
AND	Operación AND bit a bit de las entradas
OR	Operación OR bit a bit
XOR	Operación OR-exclusiva.
NOT	Negación del parámetro de entrada

### 5.7.1.6. Selección binaria y multiplexores

Nombre	Descripción
SEL	Tiene tres parámetros, el primero ( $G$ ) es booleano y los otros dos ( $IN1, IN2$ ) de cualquier tipo: $F=IN1$ si $G=0$ , $IN2$ si $G=1$
MUX	La primera entrada ( $K$ ) es entera, el resto ( $IN1, \dots, INn$ ) de cualquier tipo: Si $K=i$ , devuelve $F=INi$

### 5.7.1.7. Funciones de comparación

En CoDeSys, estas funciones tienen dos parámetros de entrada, que puede ser de cualquier tipo y devuelven un valor booleano.

Nombre	Descripción
GT	$F=TRUE$ si $IN1 > IN2$ , $F=FALSE$ en caso contrario
GE	$F=TRUE$ si $IN1 \geq IN2$ , $F=FALSE$ en caso contrario
LT	$F=TRUE$ si $IN1 < IN2$ , $F=FALSE$ en caso contrario
LE	$F=TRUE$ si $IN1 \leq IN2$ , $F=FALSE$ en caso contrario
EQ	$F=TRUE$ si $IN1 = IN2$ , $F=FALSE$ en caso contrario
NE	$F=TRUE$ si $IN1 \neq IN2$ , $F=FALSE$ en caso contrario

En la norma IEC-61131 se contempla que el número de parámetros sea mayor que

dos. Poniendo como ejemplo la función GT, la salida será:

$$F = \text{TRUE} \text{ si } \forall i, IN_i > IN(i+1)$$

Es decir, se tiene que cumplir que para cualquier entrada, su valor sea mayor que el de la siguiente entrada.

### 5.7.2. Algunos bloques funcionales estándar

También la norma establece una serie de bloques funcionales estándar y que por tanto, se encuentran ya implementados en CoDeSys. En este apartado no se va a detallar su funcionamiento, ya que dada la importancia que tienen, especialmente los contadores y temporizadores, se dedicará un capítulo para detallar su funcionamiento. La lista de los bloques funcionales estándar más importantes es:

---

Nombre	Descripción
SR	Biestable con SET dominante
RS	Biestable con RESET dominante
R_TRIG	Detección de flancos de subida
F_TRIG	Detección de flancos de bajada
CTU	Contador de incremento
CTD	Contador de decremento
CTUD	Contador de incremento y decremento
TON	Temporizador de conexión
TOF	Temporizador de desconexión
TP	Temporizador de tiempo fijo

---

En la sección de ejercicios de este capítulo, se propone la implementación de algunos bloques funcionales similares a los bloques estándar.

## 5.8. Ejercicios

### 5.8.1. Bloque funcional para Marcha-Parada de un motor

Se trata de realizar básicamente el mismo problema de marcha-parada de un motor de la sección 3.6. Sin embargo, en este caso se supone que es necesario controlar tres motores, y por tanto para ahorrar recursos de programación se va a realizar un bloque funcional *ControlMotor* con dos entradas (*Marcha* y *Paro*) y una salida (*Motor*). El programa se realizará en lenguaje de contactos LD tomando como base el ejercicio anteriormente citado.

El programa principal deberá controlar los tres motores usando el bloque funcional. El programa principal PLC\_PRG se escribirá en ST, de acuerdo a como se ha explicado en este capítulo.

Una vez creado el bloque funcional se deberán crear tres instancias, que habrá que llamar desde el programa principal, y cuyos parámetros de entrada y salida se asociarán a los pulsadores y salidas de cada uno de los motores.

Utilizar el fichero *Cap5\_3.pro* donde ya están definidas las variables de E/S (los dos botones de control de cada motor y la orden de puesta en marcha o parada de cada motor).

### 5.8.2. Bloque funcional Contador

El objetivo es diseñar un Bloque Funcional denominado *Contador* que se escribirá en ST y que deberá contar los flancos de subida que se den en una señal. El bloque funcional tendrá dos entradas:

- *Entrada* (BOOL): Se conecta a la señal en la que se quiere contar los flancos de subida.
- *Reset* (BOOL): Cuando la señal se ponga a TRUE, el contador se reseteará, es decir el valor del contador se pondrá a cero.

Y de una salida:

- Salida (INT): Almacenará el número de flancos de subida registrados en *Entrada* desde el último *Reset*.

Para implementar el bloque funcional se hará uso del bloque R\_TRIG, que es un tipo de bloque funcional estándar según la norma IEC. Este bloque funcional detecta flancos de subida en su señal de entrada. Cuando registra un flanco de subida, la salida del bloque funcional se pone a TRUE. La salida a TRUE sólo se mantiene durante un ciclo del autómata. El resto del tiempo la salida está a FALSE. Por tanto, el bloque funcional R\_TRIG tendrá un parámetro de entrada y uno de salida:

- CLK (BOOL): Entrada del bloque funcional, señal sobre la que se detectan flancos de subida.
- Q (BOOL): Salida del bloque funcional. Está a TRUE un ciclo después de detectar un flanco de subida en CLK y a FALSE todo el resto del tiempo.

Para probar el contador a realizar se usará el fichero *Cap5\_4.pro*. En él se representa un sistema con dos pulsadores (*Boton* y *BotonReset*) y un visualizador en el que se representa la variable entera *Valor*. Todas estas variables están definidas como variables globales.

Se pretende que el contador cuente los flancos de subida en la señal *Boton*, es decir las veces que pulsamos el botón (realmente las veces que pasa de no-pulsado a pulsado). Cuando se pulse *BotonReset*, el valor del contador pasará a cero. En la variable visualizada *Valor* deberá aparecer el valor del contador.

En el fichero *Cap5\_4\_Res.pro* el lector puede encontrar el ejercicio resuelto.

**NOTA:** Aunque se verá posteriormente con más detalle, en ST una sentencia del tipo IF THEN ELSE tiene la siguiente sintaxis:

IF var=TRUE THEN

...

ELSE

...

END\_IF

Y la sentencia de asignación es:

Variable:=Valor;

### 5.8.3. Bloque funcional de detección de flancos

Este ejercicio consiste en reemplazar el bloque funcional R\_TRIG por uno que será necesario diseñar denominado *DeteccionFlanco*, con la misma entrada y salida y el mismo comportamiento que el bloque funcional R\_TRIG.

El programa se hará en lenguaje LD. Nótese que en el bloque funcional será necesario utilizar una variable interna (por ejemplo, *CLK\_OLD*) en la que se almacene el último valor de la entrada *CLK*. El programa deberá:

- Poner la salida *Q* a TRUE solo si *CLK* es TRUE y *CLK\_OLD* es FALSE
- Hacer el valor de *CLK\_OLD* igual al de *CLK* para el siguiente ciclo.

Utilice el mismo fichero que en el ejercicio anterior para probarlo (*Cap5\_4.pro*)

El ejercicio está resuelto en *Cap5\_5\_Res.pro*.

Diseñe y pruebe bloques funcionales para detectar flancos de bajada y para detectar ambos tipos de flancos.



# 6. PROGRAMACIÓN EN LD

El lenguaje de lógica de contactos, de escalera o *ladder* representa los circuitos eléctricos de contactos y relés que tradicionalmente se han utilizado en el diseño de automatismos. Está especialmente orientado a aplicaciones con señales booleanas.

Un diagrama LD está acotado por dos barras verticales. En el símil eléctrico, la barra de la izquierda tiene tensión, y el estado abierto o cerrado de los contactos abre o cierra el circuito activando o no la bobina, que se encuentra en la parte derecha.

Un programa en LD está estructurado en redes (*networks*) que se ejecutan, en general, de arriba abajo (Ver Figura 6-1. Nótese que cada red no es más que una instrucción booleana en el que se le asigna valor a una variable (la bobina).

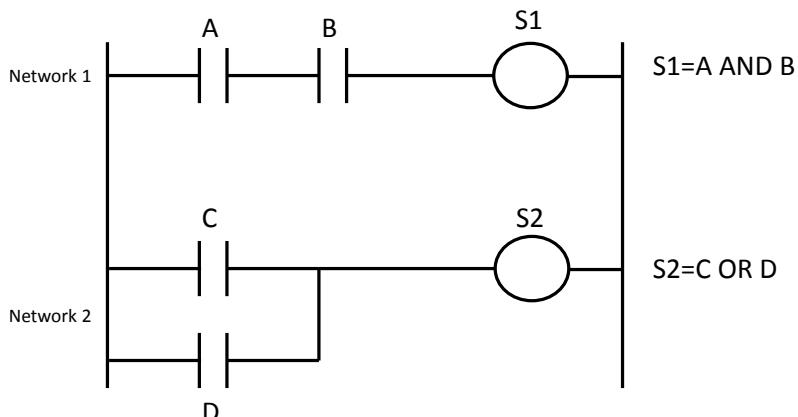


Figura 6-1 – Lógica de contactos. Diagrama LD.

Los elementos que pueden aparecer en una red son:

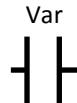
- Bobinas y contactos
- Conexiones (en serie, paralelo)
- Saltos
- Elementos gráficos de llamadas a funciones o bloques funcionales

## 6.1. Contactos y bobinas

Son los elementos más habituales en este tipo de diagramas. En cuanto a los contactos, fundamentalmente existen dos tipos, normalmente abiertos y normalmente cerrados (Tabla 6-1):

- **Normalmente abierto:** El contacto está cerrado cuando la variable (*Var* en la figura) tiene valor TRUE. Es decir, cuando la variable está a TRUE deja pasar corriente.
- **Normalmente cerrado:** El contacto está cerrado cuando la variable *Var* tiene valor FALSE.

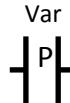
Tabla 6-1. Tipos de contactos en CoDeSys

Designación	Representación
Normalmente Abierto	Var 
Normalmente Cerrado	Var 

La norma permite otros dos tipos de contactos que sirven para detectar flancos de subida o flancos de bajada. Estos dos tipos de contactos no están implementados en

la actual versión de CoDeSys, aunque como veremos posteriormente su funcionalidad puede ser implementada con la ayuda de los bloques funcionales R\_TRIG y F\_TRIG. Son los que se representan en la Tabla 6-2.

Tabla 6-2. Tipos de contactos para detección de flancos

Designación	Representación
Contacto tipo P. Detección flanco de subida	Var 
Contacto tipo N. Detección flanco de bajada	Var 

- **Contacto tipo P:** El contacto está cerrado sólo si la variable *Var* tiene valor TRUE y tenía el valor FALSE en el ciclo anterior (el contacto está cerrado un solo ciclo en cada flanco de subida de la señal *Var*)
- **Contacto tipo N:** El contacto está cerrado sólo si la variable *Var* tiene valor FALSE y tenía el valor TRUE en el ciclo anterior (está cerrado un solo ciclo en cada flanco de bajada de la señal *Var*)

En cuanto a las bobinas, la norma contempla cuatro tipos, que están representadas en la Tabla 6-3 y que son los siguientes:

- **Bobina:** El valor de la conexión izquierda de la bobina se transfiere a la variable *Var*.
- **Bobina negada:** El valor invertido de la conexión izquierda se transfiere a la variable *Var*.
- **Bobina de ajuste o de Set:** La variable *Var* está a TRUE si la conexión de la izquierda está a TRUE. Si la variable *Var* está a FALSE el valor de la bobina no cambia.
- **Bobina de restablecimiento o de Reset:** La variable *Var* está a FALSE si la

conexión de la izquierda está a TRUE. Si la variable *Var* está a FALSE el valor de la bobina no cambia.

Nótese que las bobinas de Set y Reset funcionan como un biestable RS: Una variable se activa con una bobina tipo Set y se desactiva con una bobina tipo Reset.

Tabla 6-3. Tipos de bobinas

Designación	Representación
Bobina	Var 
Bobina negada	Var 
Bobina de ajuste o de set	Var 
Bobina de restablecimiento o Reset	Var 

### 6.1.1. Ejemplos

En el fichero *Cap6\_1.pro* se puede comprobar el funcionamiento de los distintos tipos de bobinas y contactos. El programa escrito en el ejemplo se muestra en la Figura 6-2.

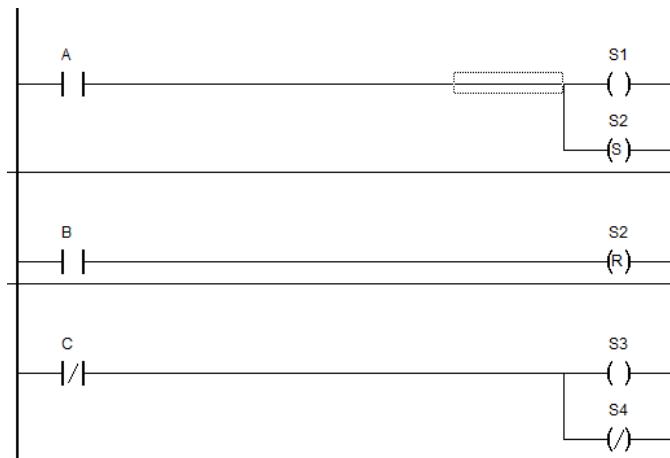


Figura 6-2 – Ejemplo de contactos y bobinas

El funcionamiento es el siguiente:

- La bobina  $S_1$  es una bobina normal. Está encendida si  $A$  está a TRUE y apagada si  $A$  está a FALSE.
- Sin embargo  $S_2$  es una bobina de Set-Reset. Nótese que asociada a la variable  $S_2$  hay dos bobinas, una de Set y otra de Reset. Cuando  $A$  se pone a TRUE  $S_2$  también, sin embargo cuando  $A$  se pone a FALSE la bobina sigue a TRUE. Para apagar la bobina hay que poner a TRUE la correspondiente bobina de Reset. En este caso se consigue con la señal  $B$ .
- Por otro lado, la señal  $C$  activa una bobina normal y otra negada. Compruebe su funcionamiento

## 6.2. Bloques funcionales

En un diagrama LD también se pueden incluir bloques funcionales. Normalmente, la conexión a la red de estos bloques se hace a través de sus señales de entrada o salida booleana. Los bloques más típicos usados en este tipo de diagrama son los

contadores, temporizadores y bloques de detección de flancos de subida o bajada.

### 6.2.1. Ejemplos

En la Figura 6-3 se muestra un ejemplo utilizando el bloque funcional visto en el capítulo anterior para detección de flancos de subida R\_TRIG.

El bloque funcional F\_TRIG tiene como entrada una señal booleana. La salida solamente se pone a TRUE cuando el valor de la entrada es TRUE y en el ciclo anterior era FALSE (es decir un flanco de subida). En el resto de los casos la salida está a FALSE. El ejemplo de la figura está implementado en *Cap6\_2.pro*. La salida del bloque alimenta una bobina normal y una de Set. En el momento que aparece un flanco de subida en la entrada A la salida se pone a TRUE pero solo durante un ciclo. Esto hace que al probar el programa la bobina no se ve nunca o casi nunca encendida (es tan corto el tiempo que no da tiempo a refrescar la visualización). Sin embargo se comprueba que ha estado a TRUE ya que la bobina tipo Set sí aparece activada.

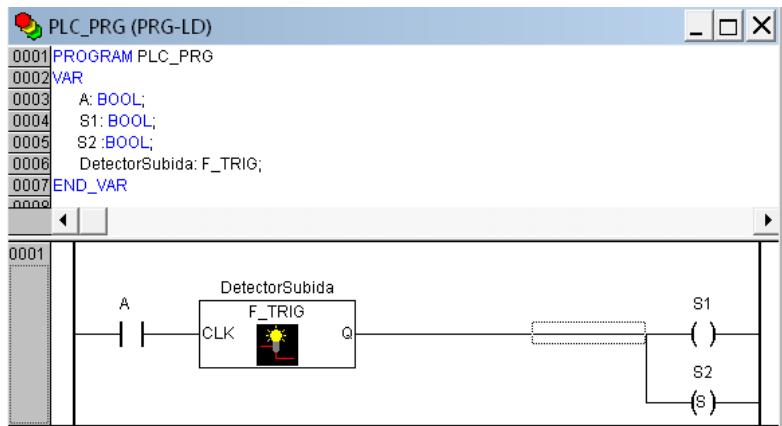


Figura 6-3 – Ejemplo de bloque funcional en LD

En el ejemplo de la Figura 6-4 se ha utilizado un bloque temporizador. En este ejemplo la salida S1 se activa 2 segundos después de que lo haga la entrada (Ver *Cap6\_3.pro*). Para apagar la bobina será necesario poner la entrada A a FALSE. Los

temporizadores se verán con detalle en el Capítulo 8.

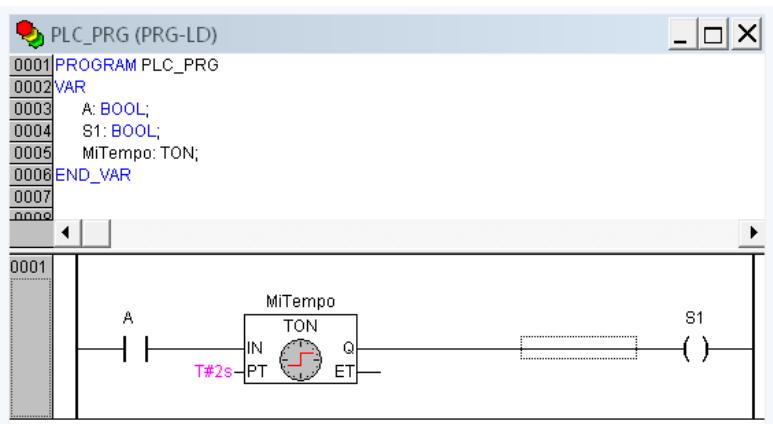


Figura 6-4 – Ejemplo de bloque funcional temporizador en LD

### 6.3. Entrada EN en funciones y bloques funcionales

En el lenguaje LD es posible utilizar bloques que dispongan de la entrada EN. En la mayor parte de las funciones y bloques funcionales se puede optar por incluir un bloque con la entrada EN.

La entrada EN es una señal booleana que se utiliza para la habilitación o deshabilitación de bloques funcionales o funciones durante la ejecución del programa. Para que el bloque se ejecute normalmente, es necesario que la entrada EN esté a TRUE. Si la entrada EN está a FALSE el bloque no se ejecuta.

Cuando se selecciona esta opción, es la entrada EN la que siempre se conecta a la red. Al resto hay que asignarle los valores directamente mediante el nombre de una variable o una constante.

### 6.3.1. Ejemplos

En el programa de la Figura 6-5 , que se encuentra implementado en *Cap5\_4.pro*, el temporizador solamente estará funcionando si la entrada A está activa.

Nótese que la conexión a la red se realiza con la entrada EN y que en las otras dos entradas se le ha asignado una variable y una constante. La salida tampoco está conectada la red y también se le ha asignado una variable.

Compruébese que cuando el bloque funcional no esté activo, las variables internas mantienen su valor. Para comprobarlo, active A, a continuación active B y manténgalo hasta que se active S2 y finalmente desactive A. Véase que la salida S2 sigue activada aunque el temporizador no está funcionando (compruébelo desactivando B).

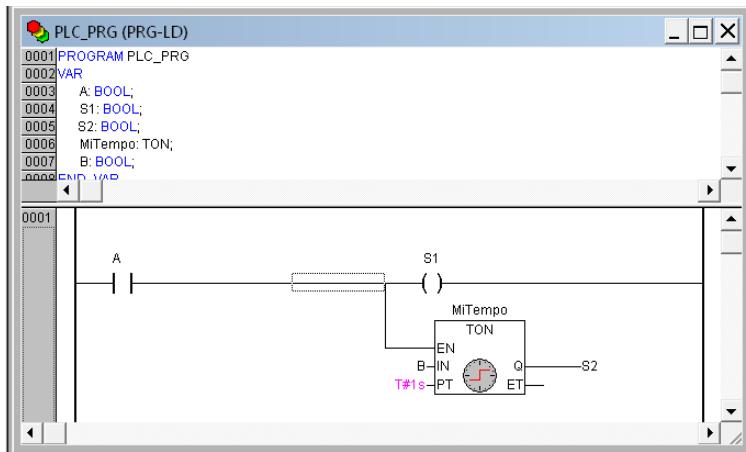


Figura 6-5 – Ejemplo de uso de la entrada EN con un temporizador

Aunque el lenguaje LD está diseñado para trabajar principalmente con señales booleanas, la entrada EN va a permitir realizar en este lenguaje operaciones utilizando otro tipo de variables.

En el ejemplo *Cap6\_5.pro* se muestra un ejemplo de este uso, cuyo programa aparece en la Figura 6-6. La instrucción MOVE se utiliza para realizar una asignación de valor a una variable, es decir, el valor de la variable entrada se le asigna a la variable de salida. Por ejemplo, la primera red sería equivalente a la

## asignación:

Tiempo:=T#5s

que se realizaría solamente cuando la variable A fuera cierta. En definitiva, las tres primeras instrucciones de la figura permiten ejecutar un temporizador de tiempo variable en función del valor TRUE/FALSE de A.

La última línea implementa un contador simple que cuenta el número de flancos de subida en la señal C. Nótese que el operador ADD se utiliza en este caso para sumar dos valores enteros.

En cualquier caso, cuando las operaciones con señales no booleanas son numerosas, suele ser preferible utilizar otro lenguaje más adaptado a variables no booleanas como el FBD o el ST.

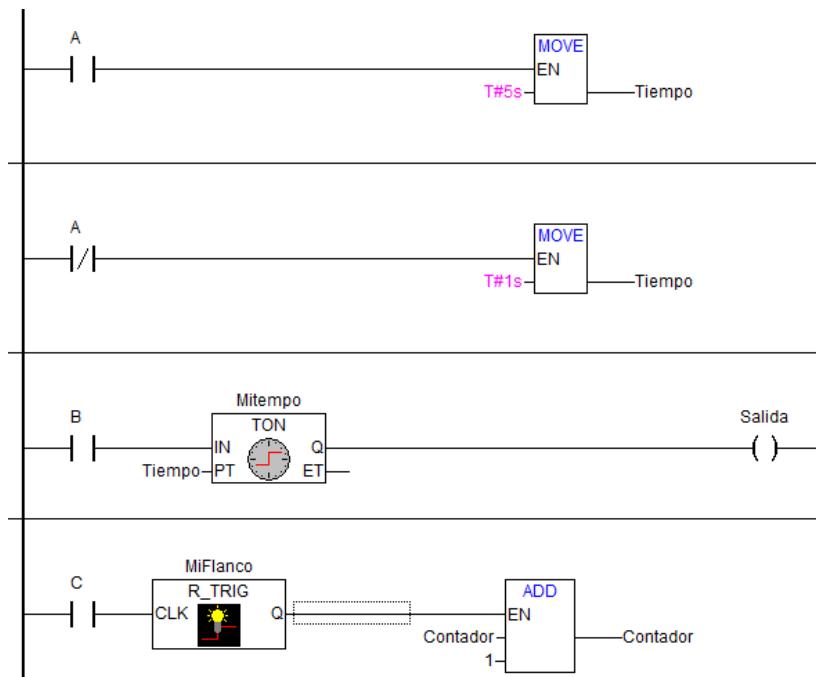


Figura 6-6 – Ejemplo de uso de la entrada EN con funciones

## 6.4. Saltos

Las opciones que hay en LD de saltar el orden secuencial de las instrucciones (de arriba a abajo) son muy limitadas. Una posibilidad es la realización de saltos condicionales a otra red en función del valor de una señal. Para especificar los puntos a las que se llega mediante saltos (que se van a corresponder con las redes) es necesario especificar etiquetas.

La segunda posibilidad es la salida del POU en cualquier punto del programa con *Return*. Funciona de la misma manera que habitualmente en los lenguajes de programación.

### 6.4.1. Ejemplo

El programa *Cap6\_6.pro*, que aparece en la Figura 6-7, realiza la operación OR o la operación AND de las señales *A* y *B* en función del valor de la señal *Selec*. Si *Selec* es TRUE hace la operación OR y AND si es FALSE.

En la primera red se observa el salto. Si la expresión lógica a la izquierda del salto (el valor de *Select* en este caso) es cierta, realiza el salto hasta la red en la que se encuentra la etiqueta (la red 0004). En la tercera red hay un salto incondicional, es decir, la condición lógica siempre es cierta. Esto evita que después de realizar la operación AND realice la operación OR y por tanto el resultado sea siempre la operación OR.

**NOTA IMPORTANTE:** Hay que tener mucho cuidado con asignar en un mismo programa dos veces valor a una misma variable de salida (en la misma o distintas POU), ya que la primera asignación se sobrescribe con la segunda. Por ejemplo, si se quiere hacer un programa que la salida *Luz* se encienda si está activa la señal *A* o la señal *B*, el programa de la Figura 6-8 sería incorrecto.

Efectivamente, como la ejecución es de arriba abajo, el valor de la salida *Luz* siempre sería el de la entrada *B*, es decir, el valor de *A* no afectaría para nada si *Luz* se activa o no.

En su lugar, es necesario establecer una única asignación que incluya la condición lógica completa. En este caso, el programa correcto se muestra en Figura 6-9.

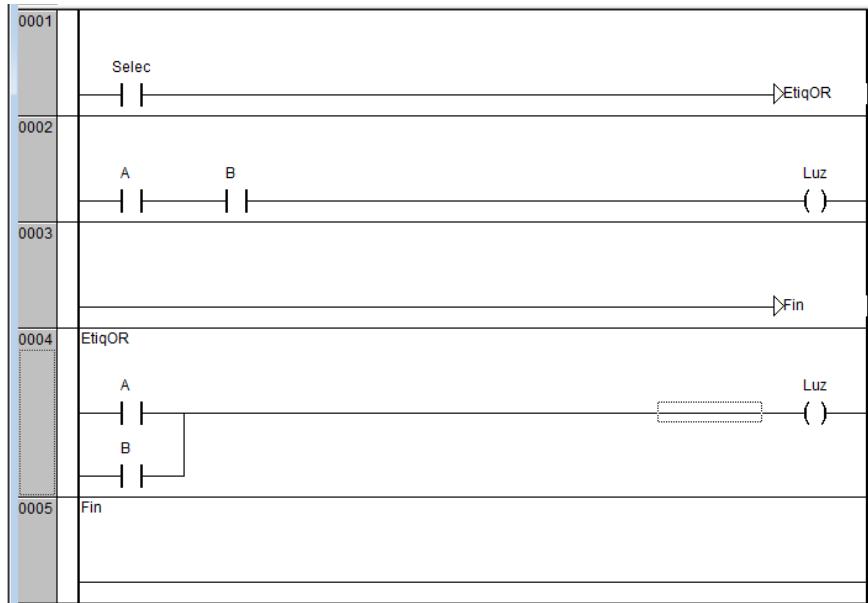


Figura 6-7 – Ejemplo del uso de saltos en LD

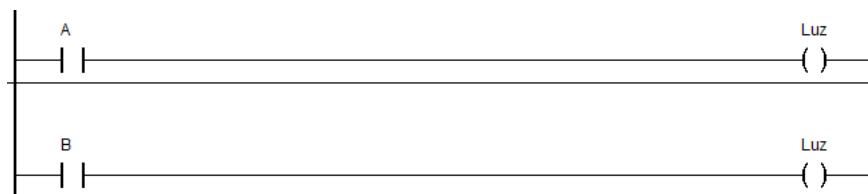
Figura 6-8 – Asignación incorrecta para activar *Luz* con *A* o con *B*



Figura 6-9 – Asignación correcta para activar *Luz* con *A* o con *B*

## 6.5. Editor de LD en CoDeSys

En esta sección se describen algunos de los comandos más importantes para editar un programa en lenguaje de contactos usando el entorno CoDeSys.

Los comandos de edición se encuentran en el menú de contexto (botón derecho del ratón). A algunos de estos comandos más usados se puede acceder desde botones del Menú de Herramientas.

Como se ha indicado anteriormente, la ventana de edición gráfica está dividida en redes (network). Para la edición, se dispone de un cursor gráfico (recuadro de línea punteada). Dentro de una red, el cursor gráfico puede estar sobre un elemento (contactos, bobinas o bloques) o sobre la línea de conexión entre los contactos y las bobinas.

### 6.5.1. Añadir nuevas redes

*Botón Ratón Derecho -> Network (before)* o *Botón Ratón Derecho -> Network (after)* dependiendo de dónde se quiera añadir la nueva red, si antes o después de la red seleccionada.

### 6.5.2. Edición de contactos y bobinas

La Tabla 6-4 muestra las principales acciones para la inserción de contactos o bobinas.

Tabla 6-4. Edición de contactos y bobinas

Acción	Menú contextual	Botón
Insertar contacto normalmente abierto	<i>Contact</i>	
Insertar contacto normalmente cerrado	<i>Contact (negated)</i>	
Insertar contacto normalmente abierto en paralelo	<i>Parallel Contact</i>	
Insertar contacto normalmente cerrado en paralelo	<i>Parallel Contact (negated)</i>	
Insertar bobina	<i>Coil</i>	
Insertar bobina tipo set	<i>'Set' Coil</i>	
Insertar bobina tipo reset	<i>'Reset' Coil</i>	
Negar contacto o bobina (seleccionarlo previamente)	<i>Negate</i>	
Cambiar tipo bobina Normal/Set/Reset	<i>Set/Reset</i>	

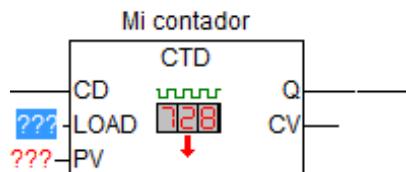
La posición del cursor de edición es importante a la hora de insertar elementos.

- Si está seleccionado un contacto, el nuevo contacto se insertará a su izquierda
- Para colocarlo el último a la derecha hay que seleccionar el final de la línea.
- Para colocar un contacto paralelo se selecciona el/los contactos con los que se quiere que esté en paralelo.
- Si se selecciona el final de la línea, lo inserta en paralelo con todo lo que haya en la línea
- Las bobinas solo pueden ir a la derecha de todos los contactos.
- Se pueden insertar varias bobinas, pero siempre irán todas en paralelo a la derecha de la línea.

### 6.5.3. Edición de bloques funcionales

Exactamente igual que en el caso de los contactos y bobinas, los bloques funcionales se insertan en el entorno gráfico de CoDeSys con el menú contextual o los botones. Las distintas acciones relacionadas con bloques funcionales figuran en la Tabla.

Nótese que en todos los casos, la entrada que se conecta a la red es la primera entrada de tipo booleano. Al resto de entradas hay que asignarle valores, escribiendo directamente el nombre de una variable o constante (Ver Figura 6-4). Para esto, seleccione con el ratón la etiqueta de la señal de entrada o salida y escriba el nombre de una variable o constante. Lo mismo ocurre cuando hay más de una salida.



De nuevo, la posición en que se insertará el bloque funcional dependerá de la posición en que se encuentre el cursor gráfico.

Tabla 6-5. Edición de bloques funcionales

Acción	Menú contextual	Botón
Insertar bloque funcional	<i>Function Block...</i>	
Insertar bloque funcional o función con entrada EN	<i>Box with EN</i>	
Insertar bloque funcional R_TRIG	<i>Rising edge detection</i>	
Insertar bloque funcional F_TRIG	<i>Falling edge detection</i>	
Insertar bloque funcional temporizador TON	<i>Timer (TON)</i>	

#### 6.5.4. Edición de saltos en CoDeSys

La creación de un salto se realiza con la opción *Jump* del menú contextual (Botón derecho del ratón). Es necesario escribir el nombre de una etiqueta junto al símbolo del salto. El lugar de destino se marca en la esquina superior izquierda de una red, donde se escribirá el mismo nombre de etiqueta que en el salto.

La opción *Return* también está disponible en el menú contextual. En este caso, no se usan las etiquetas.

## 6.6. Ejemplo

En este ejemplo se muestra cómo implementar un automatismo lógico en LD a partir de su grafo de estados. El objetivo es diseñar un sistema de control para el sistema de la Figura 6-10. Al pulsar el botón de inicio (*START*), el carrito debe ir

hasta *B* y luego volver a *A*. En *A* y *B* existen dos sensores de fin de carrera. Para mover el carrito se dispone de dos señales de salida, *MOTOR\_DE* y *MOTOR\_IZ*, que activan respectivamente el movimiento a la derecha y a la izquierda. Se trata por tanto de un sistema con tres entradas (*START*, *A*, *B*) y dos salidas (*MOTOR\_DE*, *MOTOR\_IZ*).

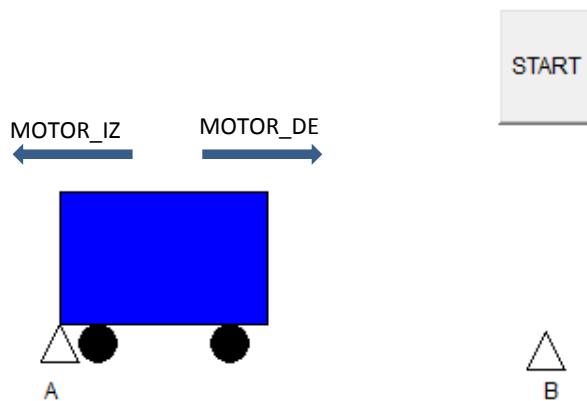


Figura 6-10 – Sistema a controlar

El ejercicio está resuelto en *Cap6\_7.pro*. La solución está basada en la implementación de un grafo de estados en LD con tres estados *Reposo* (*Re*), Moviendo a la derecha (*Der*) y moviendo a la izquierda (*Izq*). El grafo de estados se muestra en la Figura 6-11. Como se observa, las transiciones entre estados están condicionadas a que se activen las señales de entrada. Las salidas están asociadas a los estados, así por ejemplo, si el sistema está en estado *Der* se deberá activar la salida *MOTOR\_DE*, mientras que si está en estado *Res* no se activa ninguna salida.

El programa que se diseñe deberá memorizar en cuál de los tres estados se encuentra el sistema, para lo que se usarán las variables locales booleanas *Re*, *Iz* y *De*, de forma que sólo estará a TRUE la variable asociada al estado en que se encuentre el sistema. Por otro lado, el estado (y por tanto su variable asociada) se deberá actualizar en función de en qué estado se encuentre en cada momento y los valores de las señales de entrada de acuerdo al grafo de la Figura 6-11. Finalmente

deberá activar las salidas en función del estado en que se encuentre en cada instante.

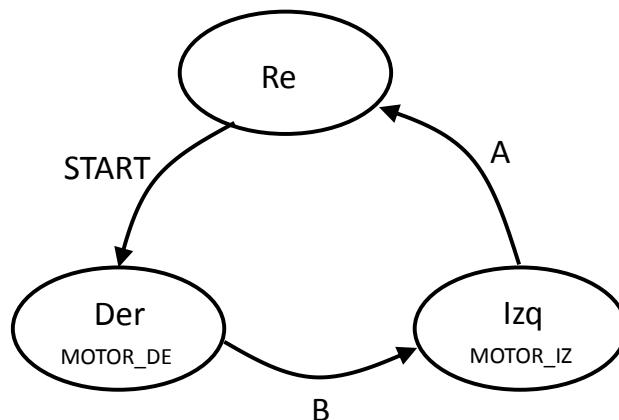


Figura 6-11 – Grafo de estados

El programa se muestra en la Figura 6-12 (también en *Cap6\_7.pro*). El programa está dividido en tres secciones, diferenciadas con colores distintos en la figura.

La primera sección que consta de la primera línea sirve para activar el estado inicial. Si todos los estados están inactivos (solo ocurre al iniciar el programa ya que por defecto las variables de inicializan a FALSE) se activa el estado de reposo.

La segunda sección implementa cada una de las tres transiciones del grafo. Se coloca una red por cada transición. Si se está en un estado (su variable está a TRUE) y se cumple la condición de transición, se desactiva el estado de salida de la transición y se activa el estado de llegada.

Nótese que la activación de estados se realiza con bobinas de Set y Reset. De esta manera, cuando se dan las condiciones de activación de un estado, se entra en dicho estado (activando un Set), y el sistema se mantiene en ese estado hasta que se cumplan una transición de salida (en cuyo caso se aplicará una bobina Reset de ese estado). Observe que el sistema siempre está en un único estado (hay una y solo

una de las variables de estado a TRUE).

La tercera sección activa las condiciones de salida. Se coloca una asignación por cada una de las salidas. Conviene volver a recordar que utilizar varias asignaciones a una misma variable de salida en general dará lugar a un comportamiento no deseado.

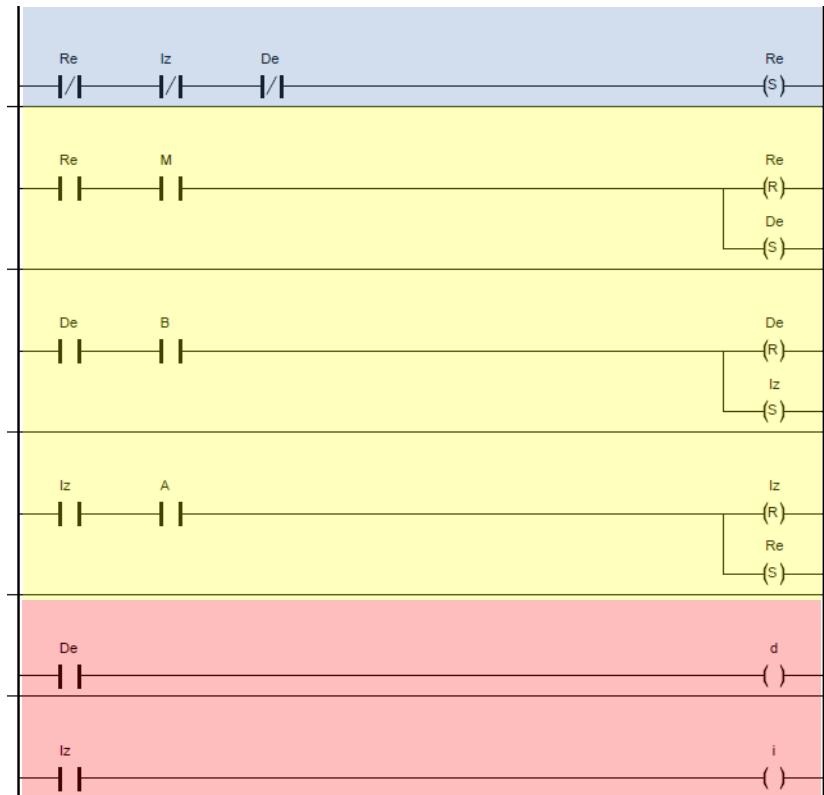


Figura 6-12 – Programa en LD

Nótese que esta misma estructura de programa en tres secciones serviría para implementar cualquier automatismo del que se disponga su grafo de estado.

## 6.7. Ejercicios

### 6.7.1. Ejercicio 1

Escriba un programa que encienda o apague una luz con tres interruptores. Al pulsar cualquiera de ellos deberá cambiar el estado de la luz. Utilice el fichero *Cap6\_8.pro*.

### 6.7.2. Ejercicio 2

Se desea diseñar un sistema de control para controlar el semáforo de los vehículos en un paso a nivel. Para detectar la proximidad del tren se dispone de dos sensores (*A* y *B*) situados a 1 Km. a cada lado del paso a nivel. Los trenes pueden llegar por los dos lados, pero siempre se cumplen las siguientes condiciones:

- La longitud de los trenes es menor de 2 Km.
- La separación entre dos trenes en la misma dirección es superior a 2 Km.
- No pueden llegar trenes en sentido contrario.

En definitiva nunca hay más de un tren en el paso a nivel.

El sistema de control debe activar la luz verde (*salverde*) si no hay ningún tren en el sistema, y la luz roja (*salrojo*) desde el momento en que un sensor detecta un tren hasta que sale completamente del paso a nivel. Se pide:

- Diseñar un grafo de estados que controle el sistema
- Codificarlo con lógica de contactos

El programa se escribirá en la sección “Automatismo” del fichero *Cap6\_9.pro* que se proporciona.

### 6.7.3. Ejercicio 3

Se desea controlar el nivel del depósito de la Figura 6-13 de forma que el nivel se mantenga entre los sensores de nivel *A* y *B*, de tal manera que se minimice el

número de conexiones y desconexiones de las bombas, por lo que en su funcionamiento normal, el nivel estará oscilando entre los sensores *A* y *B*. Para ello se dispone de dos bombas (*Bomba1* y *Bomba2*). El funcionamiento debe ser el siguiente:

- El sistema de control empezará a funcionar cuando se active el botón *Encendido*
- Si el nivel está por encima de *B* no funcionará ninguna de las bombas.
- Si el nivel está por debajo de *A* funcionan las dos bombas
- Si el nivel está entre *A* y *B* funcionará una única bomba, pero la elección de la bomba que funciona sola se realizará de forma alternativa. Es decir, si la última bomba en funcionar sola fue *Bomba1* la siguiente en funcionar sola será la *Bomba2*, y a la inversa.

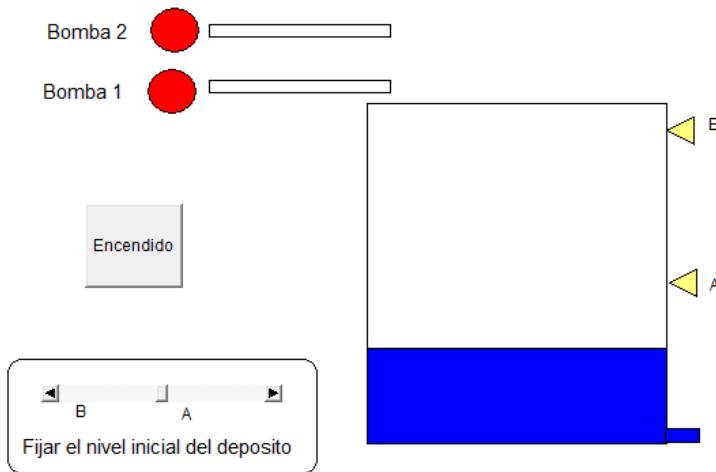


Figura 6-13 – Visualización del ejercicio 3

El sistema de control debe funcionar independientemente de cual sea el nivel inicial del depósito. Para cambiar el nivel inicial se dispone de una barra de deslizamiento donde están marcados los valores de nivel correspondiente a los niveles de *A* y *B*.

Téngase en cuenta que la barra hay que ajustarla al valor deseado antes de ejecutar el programa.

El programa se realizará en lenguaje LD en el fichero que se proporciona *Cap6\_10.pro*. El programa se escribirá en la POU denominada *Control*.



# 7. PROGRAMACIÓN EN IL

---

**E**l segundo de los lenguajes de la norma IEC 61131-3 que se va a describir es el de Lista de Instrucciones (IL). Se trata de un lenguaje de tipo ensamblador con un juego de instrucciones relativamente corto. Al contrario de LD, que está más orientado a variables booleanas, con el lenguaje IL se puede trabajar de forma cómoda con variables de cualquier tipo.

Se trata de un lenguaje textual en el que cada comando se escribe en una línea. También se admiten líneas en blanco. Cada línea de comandos tiene el siguiente formato:

Etiqueta:              Operador/Funcióñ              Operandos

La etiqueta con su delimitador ":" es opcional y se usa en las instrucciones de salto. La lista de operandos consiste en ninguna, una o varias constantes o variables (depende del operador) separadas por comas. También, especialmente en el caso de llamada a funciones o bloques funcionales, los operandos pueden ir entre paréntesis. Se pueden añadir comentarios encerrándolo entre los delimitadores ("\* y \*").

En el lenguaje IL, al igual que ocurre con todos los ensambladores, las operaciones están apoyadas en el uso de un acumulador, en este caso denominado CR (Current Result). La característica principal es que no es de ningún tipo de variable determinado, sino que se puede usar con variables de cualquier tipo. Evidentemente, el programador debe asegurarse de que las operaciones sean compatibles con el dato guardado en el CR.

Las instrucciones básicas para el uso del acumulador son:

- **LD:** Tiene un único operador que es el nombre de una variable del programa, que puede ser de cualquier tipo. La instrucción carga en el acumulador el contenido de la variable.
- **ST:** También tiene como único operador el nombre de una variable, también de cualquier tipo. En este caso, el contenido del acumulador se almacena en la

variable.

En realidad el acumulador se gestiona como una pila, es decir, con la operación LD se coloca el contenido de una variable encima de la pila de CR y la operación ST almacena en una variable el dato que esté encima de la pila y lo elimina de la pila CR. Como se verá posteriormente, la gestión como pila permitirá realizar operaciones con paréntesis. Un ejemplo del uso de estos operadores sería:

---

```
LD %I7
AND %I8
ST %Q1
```

---

La primera instrucción carga en el acumulador el valor que tiene la entrada %I7 (crea el dato en CR), la segunda hace la operación AND entre el contenido del acumulador y la variable de entrada %I8 y almacena el resultado en el acumulador (modifica el valor del dato del acumulador creado anteriormente), mientras que la última almacena el contenido del acumulador en la variable de salida %Q1 (elimina el dato del acumulador, por tanto si el acumulador estaba inicialmente vacío, terminará la última operación de nuevo vacío). En este caso el acumulador está trabajando con datos de tipo booleano.

Este otro ejemplo suma dos números enteros, de forma que el acumulador está trabajando con enteros.

---

```
VAR
  Sum1:INT;
  Sum2:INT;
  Result:INT;
END_VAR
-----
LD Sum1
ADD Sum2
ST Result
```

---

En general en el CR se pueden almacenar:

- Datos de tipo elemental
- Datos de tipo derivado
- Instancias de bloques funcionales

## 7.1. Operadores básicos y llamadas a funciones

Además de los operadores vistos en la sección anterior y algún otro que se verá a continuación, en general los operadores en IL coinciden con las funciones que se presentaron en el Capítulo 5. Por tanto, teniendo en cuenta que cualquier operación en IL se trata realmente de una llamada a función, es importante entender cómo se realiza dicha llamada.

Para llamar a una función se utiliza simplemente el nombre de la función. Para suministrarle los parámetros se procede de la siguiente manera:

- El primer parámetro debe estar en el acumulador antes de la llamada.
- A partir del segundo se ponen en la llamada, tras el nombre de la función y separados por comas.
- El valor que devuelve, la función lo coloca en el acumulador.

Lógicamente, esto es aplicable tanto a funciones estándar predefinidas como a funciones de usuario.

En el siguiente ejemplo se muestran dos formas equivalentes de realizar la operación AND de tres variables booleanas y almacenarlas en la variable *Sal*.

---

LD A	LD A
AND B	AND B,C
AND C	ST Sal
ST Sal	

---

Aunque la mayor parte de los operadores se enumeraron en el Capítulo 5, en esta

sección se volverán a indicar los más usados en los programas indicando en este caso la forma en que llamarlos con los operadores y acumulador. Los operadores booleanos más usados son los mostrados en Tabla 7-2.

Tabla 7-1. Operadores booleanos

Nombre	Operador	Descripción
AND	Uno o más	Operación AND entre CR y los operadores.
OR	Uno o más	Operación OR entre CR y los operadores.
XOR	Uno o más	Operación OR exclusivo entre CR y los operadores.
S	Uno	Pone el operador a TRUE si el acumulador está a TRUE
R	Uno	Pone el operador a FALSE si el acumulador está a TRUE
NOT	Ninguno	Negación del acumulador

Además alguno de estos operadores puede ser completado con un carácter adicional llamado modificador que cambia el significado de la operación. Estos modificadores son:

- **N** : Negación del operando
- **C** : Ejecución condicional del operador. Solo se ejecuta si el acumulador está a TRUE
- **(** : Para comenzar niveles de paréntesis

El siguiente ejemplo realiza la operación  $Res = \overline{\overline{A}} + \overline{\overline{B}}$

---

```
LDN A (* Carga en el acumulador el valor negado de A*)
ORN B (*OR del el acumulador y el contenido negado de B *)
STN Res (* Guarda en Res el valor del acumulador negado *)
```

---

En cuanto al uso de paréntesis, cuando se detecta el modificador '(' se crea un nuevo valor que pone encima de la pila CR (el valor anterior del acumulador queda en el segundo nivel de la pila), realiza las nuevas operaciones y cuando encuentra el cierre de paréntesis se realiza la correspondiente operación entre los dos primeros datos sobre la pila. Se puede realizar más de un nivel de paréntesis. El siguiente ejemplo realiza la operación  $Res = Var1 \text{ AND } (\overline{Var2} \text{ OR } Var3)$ .

---

```
LD Var1
AND ( Var2
NOT
OR Var3
)
ST Result
```

---

Cuando se llama al operador ')', encima de la pila está el resultado de  $Var2 \text{ OR } Var3$  y el segundo dato en la pila es  $Var1$ , realizándose entre ellos la operación AND, y dejando como único dato en la pila dicho resultado. Finalmente la operación ST elimina el dato de la pila CR, quedando ésta vacía, tal como estaba inicialmente.

Dicho programa también se puede escribir de la siguiente manera:

---

```
LD Var1
AND(
LD Var2
NOT
OR Var3
)
ST Result
```

---

En la Tabla 7-2 se muestran los operadores aritméticos y de comparación más usados.

Estos operadores hacen la correspondiente operación o comparación entre el

contenido del acumulador y el operador (variable o constante). En el caso de las comparaciones en el acumulador deja el valor booleano TRUE o FALSE, mientras que en las operaciones, el resultado de la operación es del mismo tipo que los operadores.

Tabla 7-2. Operadores aritméticos y de comparación

Nombre	Operador	Descripción
ADD	Uno o más	Suma CR y los operadores.
SUB	Uno	Resta CR y el operador
MUL	Uno o más	Multiplica CR y los operadores.
DIV	Uno	Divide el acumulador entre el operador
GT	Uno	Escribe TRUE en CR si CR es mayor que el operador, si no, escribe FALSE
GE	Uno	Escribe TRUE en CR si CR es mayor o igual que el operador, si no, escribe FALSE
EQ	Uno	Escribe TRUE en CR si CR es igual que el operador, si no, escribe FALSE
NE	Uno	Escribe TRUE en CR si CR es distinto al operador, si no, escribe FALSE
LT	Uno	Escribe TRUE en CR si CR es menor que el operador, si no, escribe FALSE
LE	Uno	Escribe TRUE en CR si CR es menor o igual que el operador, si no, escribe FALSE

Finalmente, en la Tabla 7-3 aparecen los operadores de saltos a etiquetas y llamadas a programas y funciones. En la sección 7.3 se muestra un ejemplo con el uso de alguna de estas instrucciones.

Tabla 7-3. Operadores de saltos y llamadas a POU

Nombre	Operador	Descripción
JMP	Uno	Salto a la etiqueta (operador).
JMPC, JMPN	Uno	Salto condicional a la etiqueta si el acumulador es TRUE (JMPC) o FALSE (JMPN)
CAL	Uno o más	Llamada a programa o bloque funcional
CALC, CALN	Uno o más	Llamada condicional si el acumulador es TRUE (CALC) o FALSE (CALN)
RET	Ninguno	Return de una función o bloque funcional
RETC, RETN	Ninguno	Return condicional si el acumulador es TRUE (RETC) o FALSE (RETN)

## 7.2. Uso de funciones y bloques funcionales

### 7.2.1. Llamada a funciones

Las llamadas a funciones definidas por el usuario se realizan de la misma manera que se ha visto en el apartado anterior. En el Ejemplo 7-1 se muestra la definición de una función que calcula la media de tres números enteros.

#### Ejemplo 7-1. Función que calcula la media de tres enteros

---

```

FUNCTION Media3 : INT
VAR_INPUT
    Valor1: INT;
    Valor2 : INT;
    Valor3 : INT;
END_VAR
-----
LD          Valor1
ADD        Valor2, Valor3
DIV        3
ST         Media3

```

---

Para llamar a esta función para calcular la media de 10, 20 y 31 desde un programa IL, se hace de la siguiente manera:

---

LD	10
Media3	20,31
ST	Result

---

La función deja el resultado en el acumulador y con el operador ST se ha almacenado en la variable *Result*.

### 7.2.2. Llamada a bloques funcionales

La llamada a un bloque funcional se realiza mediante el operador CAL (llamada incondicional), CALC (sólo se hace la llamada si el acumulador es TRUE) o CALN (sólo se realiza si el acumulador es FALSE). Recuérdese que la llamada se hace con el **nombre de la instancia**. Para indicar los parámetros de entrada y salida hay dos métodos:

- **Primer método:** Indicando la lista de parámetros de entrada y salida entre paréntesis en la llamada
- **Segundo método:** Asignando valores a los parámetros de entrada anteriormente a la llamada a la función y a los parámetros de salida posteriormente. Esto se realiza utilizando la estructura de datos asociada a cada instancia que se vio anteriormente en la sección 5.3.2

Veamos estos dos procedimientos con un ejemplo. Recordemos el bloque funcional *Biestable* definido anteriormente y cuyos parámetros de entrada y salida son los siguientes:

---

```
FUNCTION_BLOCK Biestable
VAR_INPUT
    EntS : BOOL;
    Entr: BOOL;
END_VAR
VAR_OUTPUT
    Estado : BOOL :=TRUE;
END_VAR
```

---

La llamada a una instancia que denominaremos *MiBiest* desde otro programa o bloque funcional se puede realizar de las siguientes maneras:

### 7.2.2.1. Primer método

Para cada uno de los parámetros de entrada se indica el nombre que tiene dicho parámetro en la definición del bloque funcional, “:=” y el valor que se le asigna (variable o constante). Para los parámetros de salida es igual pero utilizando “=>”. El orden en que se pongan los parámetros es indiferente.

---

```
VAR
    MiBiest : Biestable;
    Senals:BOOL;
    SenalR:BOOL;
    MiEstado:BOOL;
END_VAR
-----
CAL MiBiest(
    EntS:=Senals,
    Entr:=Senals,
    Estado=>MiEstado
)
```

---

### 7.2.2.2. Segundo método

En este segundo procedimiento se hace uso de la estructura que se crea de forma

automática al crear una instancia de bloque funcional.

---

```

VAR
    MiBiest : Biestable;
    SenalsS:BOOL;
    SenalR:BOOL;
    MiEstado:BOOL;
END_VAR
-----
LD SenalsS
ST MiBiest.EntS
LD SenalR
ST MiBiest.Entr
CAL MiBiest      (* Llamada al bloque funcional *)
LD MiBiest.Estado
ST Miestado

```

---

La llamada al bloque funcional se realiza sin parámetros, pero previamente se le han dado valor a los parámetros de entrada mediante la estructura asociada a la instancia. Después de la llamada al bloque funcional se asigna valor a las variables asociadas a las salidas del bloque funcional.

### 7.2.2.3. Métodos mixtos

La norma también admite soluciones intermedias, es decir llamar algunos de los parámetros con un procedimiento y otros con otro.

---

```

LD SenalR
ST MiBiest.EntR
CAL MiBiest(
    EntS:=SenalsS
)
LD MiBiest.Estado
ST MiEstado

```

---

## 7.3. Saltos

Los saltos en IL proporcionan la única manera de cambiar el orden secuencial de las instrucciones. El siguiente código resuelve en IL el ejemplo de *Cap6\_6.pro*, que realiza la operación OR o la operación AND de las señales *A* y *B* en función del valor de la señal *Selec*. Si *Selec* es TRUE hace la operación OR y AND si es FALSE.

---

```
PROGRAM PLC_PRG
VAR
    Selec: BOOL;
    A: BOOL;
    B: BOOL;
    Luz: BOOL;
END_VAR
-----
LD Select (* Almacena Select en el acumulador *)
JMPc OperOR (* Salta si Select era TRUE *)
LD A
AND B
ST Luz
RET      (* Return, sale del programa *)
OperOr: LD A
OR B
ST Luz
```

---

## 7.4. Ejercicios

### 7.4.1. Ejercicio 1

Realizar el ejercicio *Cap6\_8.pro* de encendido de una luz desde tres puntos en lenguaje IL. Para ello deberá borrar el programa PRG\_PLC que se proporciona en LD y crear uno nuevo en IL. Estas operaciones se realizan desde el Organizador de Objetos.

### 7.4.2. Ejercicio 2

En el fichero *Cap7\_1.pro* se muestra un sistema con un depósito que se llena con dos pozos con bomba y cuya visualización se muestra en la Figura 7-1. El depósito tiene dos sensores de nivel *DI* y *DS*. Del mismo modo, cada uno de los pozos tiene un sensor para saber si el pozo tiene suficiente agua para bombeo. Estos sensores son *N1* el del pozo 1 y *N2* el del segundo pozo. Finalmente cada una de las bombas se activa respectivamente con las señales de salida *B1* y *B2*.

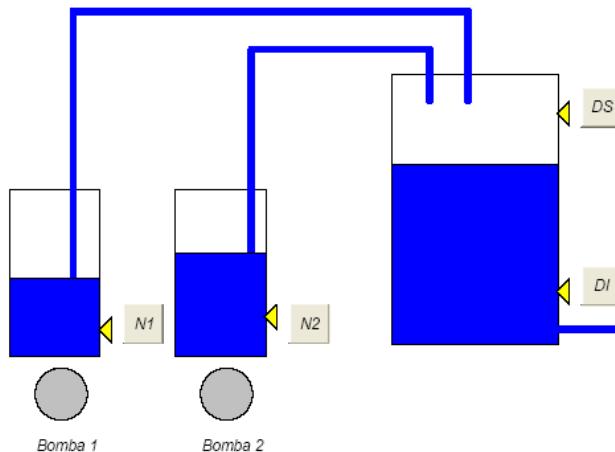


Figura 7-1 – Visualización del ejercicio 2

Se deberá escribir un programa en IL para que el funcionamiento del sistema sea el siguiente:

- Si el nivel del depósito está por encima de *DS*, las bombas deberán estar paradas
- Si el nivel del depósito está por debajo de *DS* y el nivel del pozo 1 por encima de *N1*, funciona la bomba 1.
- Si no hay agua en pozo 1 pero el nivel de pozo 2 está por encima de *N2*,

funciona la bomba 2

- Si el nivel del depósito por debajo de DI, funcionan las dos (si tienen agua)

En la animación no se simula dinámicamente, sino que el estado de los sensores se activa a mano por medio de los correspondientes botones, tal como se ve en la figura.

Nota: Compruébese que se trata de un automatismo combinacional, es decir los valores de las salidas  $B1$  y  $B2$  se pueden obtener como expresiones booleanas de las cuatro entradas.

### 7.4.3. Ejercicio 3

Disponemos de un depósito en el que hay colocados tres sensores, un interruptor de presión, otro de nivel y otro de temperatura (*Señales Nivel, Pres y Temp*). Estos tres sensores se activan cuando su correspondiente magnitud alcance un valor límite prefijado.

Se pretende diseñar un sistema de gestión de alarmas en IL, en el que se encienda un aviso cuando alguno de estos sensores esté activo (*Señales LuzNiv, LuzPres y LuzTemp*)

Además debe encender una luz de alarma cuando se dé una de estas tres condiciones:

- Estén activados simultáneamente las señales de nivel y temperatura alta.
- Estén activados simultáneamente las señales de nivel y presión alta.
- Estén activados simultáneamente las señales de temperatura y presión alta.

Esta luz de alarma (*Alarma*) no se debe apagar cuando dejen de cumplirse las condiciones que la activaron. La señal de alarma se mantendrá hasta que el operario pulse el botón de reconocimiento de alarma (*ACK*).

En el simulador que se proporciona en *Cap7\_2.pro*, el valor de los sensores se simulará mediante los botones P, N y T. En la Figura 7-2 se muestra la visualización del ejercicio.

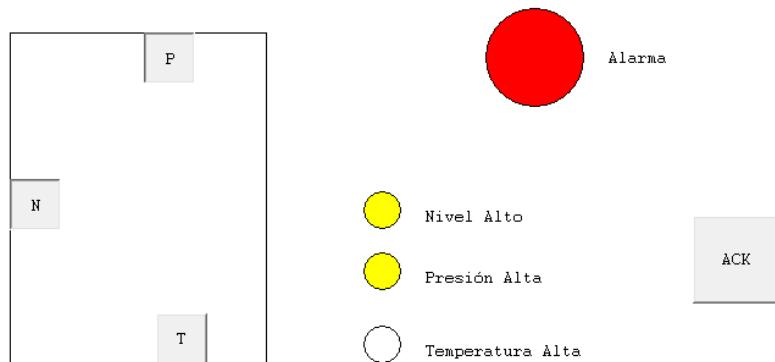


Figura 7-2 – Visualización del ejercicio 3

# 8. TEMPORIZADORES Y CONTADORES

---

**A**ntes de describir los tres lenguajes restantes de la norma IEC 61131-3, se va a dedicar un capítulo a los dos tipos de bloques funcionales definidos en la norma que más habitualmente se usan en la práctica: los temporizadores y los contadores. La norma contempla varios tipos tanto de temporizadores como de contadores. En este capítulo se describirá su uso y se propondrán varios ejercicios en los que se podrá comprobar su uso.

Es importante hacer notar que se tratan de bloques funcionales, por lo tanto su uso ha de ceñirse a lo visto sobre los bloques funcionales en los capítulos anteriores. Los ejemplos se verán indistintamente en lenguaje LD o IL.

## 8.1. Temporizadores

La temporización es una tarea primordial en la mayor parte de los sistemas de automatización, por ejemplo, cuando se quiere mantener una señal activada durante un determinado tiempo y que éste se puede prefijar. Existen tres tipos de temporizadores en la norma implementados como bloques funcionales denominados TON, TOF y TP. Todos ellos tienen dos entradas (IN y PT) y dos salidas (Q y ET), diferenciándose en su funcionamiento. La entrada IN se utiliza para el control del temporizador, fundamentalmente para determinar cuándo hay que empezar a temporizar. La entrada PT se utiliza para indicar el tiempo que se ha de temporizar. La salida Q cambia de valor cuando ha transcurrido el tiempo prefijado en función del tipo de temporizador que se use y la salida ET es un reloj en la que se indica el tiempo que ha transcurrido en la temporización. En la Tabla 8-1 se muestran los tipos de datos de cada una de las entradas y salidas del temporizador.

Tabla 8-1. Parámetros de entrada y salida en un temporizador

Parámetro	Tipo	
IN	BOOL	Entrada
PT	TIME	Entrada
Q	BOOL	Salida
ET	TIME	Salida

En los siguientes apartados se describe el funcionamiento de los tres tipos de temporizadores.

### 8.1.1. Temporizador de conexión (TON)

El funcionamiento de este temporizador consiste en que la salida Q se pone a TRUE cuando ha transcurrido un tiempo prefijado dado, que se proporciona en la entrada PT, desde que se da la orden de comienzo de temporización.

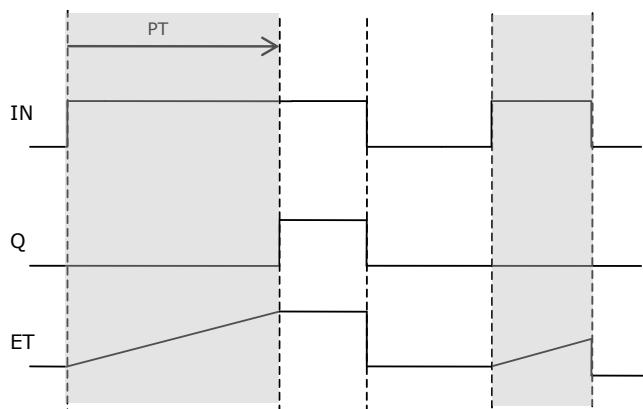


Figura 8-1 – Cronograma del temporizador TON

Para empezar a contar el tiempo debe ponerse la entrada IN a TRUE y mantenerse en ese valor durante toda la temporización. En ese caso, cuando transcurre el tiempo PT, la salida Q se pone a TRUE.

Para volver a temporizar es necesario previamente reiniciar el temporizador (es decir, poner Q a FALSE y EV a 0). Esto se consigue poniendo la entrada IN a FALSE. Una vez hecho esto, si se vuelve a poner IN a TRUE volverá a comenzar la temporización. La Figura 8-1 muestra un cronograma de las distintas variables. Nótese que si durante la temporización IN se pone a FALSE, la temporización se cancela.

### 8.1.2. Temporizador de Tiempo Fijo (TP)

El temporizador TP funciona como un generador de un pulso de duración PT en Q cuando se detecta un flanko de subida en IN. En este caso, no es necesario mantener IN a TRUE durante la temporización. La Figura 8-2 muestra el cronograma de un temporizador TP.

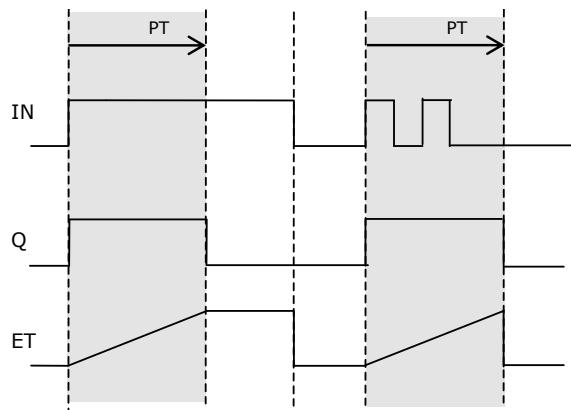


Figura 8-2 – Cronograma del temporizador TP

Cuando se produce un flanko de subida en IN, el temporizador empieza a contar en milisegundos, la salida Q se pone a TRUE hasta que el valor de EV valga PT.

Cuando transcurre el tiempo PT, Q se pone a FALSE y EV se mantiene con el valor TP hasta que la entrada IN se ponga a FALSE. En ese momento el contador vuelve a estar en el estado inicial a la espera de un nuevo flanco de subida en IN para volver a temporizar.

Nótese que en este temporizador, una vez que ha comenzado a temporizar, la salida IN no lo resetea hasta que haya transcurrido el tiempo PT.

### 8.1.3. Temporizador de desconexión (TOF)

En el temporizador de desconexión, la señal de salida Q está a TRUE desde que la señal IN cambia al valor TRUE hasta que haya transcurrido un tiempo PT desde que la señal IN vuelve a cambiar a FALSE. Por tanto este elemento comienza a temporizar cuando detecta un flanco de bajada en IN. Si durante el periodo de temporización hay un flanco de subida en IN, la temporización se pone a 0. En la Figura 8-3 se muestra un cronograma del funcionamiento del temporizador.

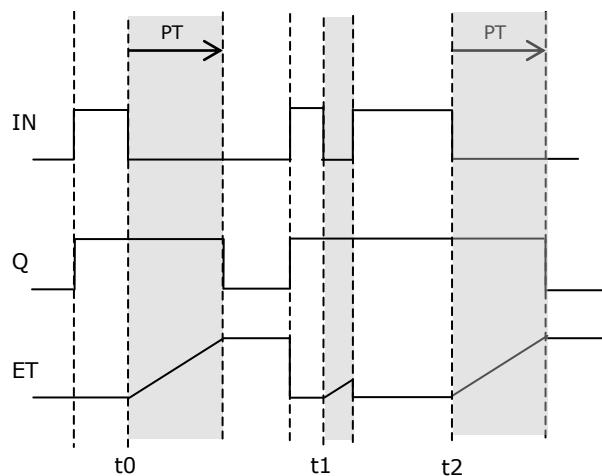


Figura 8-3 – Cronograma del temporizador TOF

El temporizador empieza a contar cuando se produce un flanco de bajada en IN (instantes  $t_0$ ,  $t_1$  y  $t_2$ ). Sin embargo, el temporizador que comienza en  $t_1$  se resetea

antes de llegar a PT debido a que se produce un flanco de subida en IN.

#### 8.1.4. Uso de los temporizadores

El uso de los temporizadores es igual que el de cualquier otro bloque funcional, por lo que es necesario definir instancias. Por ejemplo, un temporizador TON en IL se usará de la siguiente forma:

---

```
VAR
    Nom_tempo: TON;
    VarBOOL1:BOOL;
    VarBOOL2:BOOL;
    varTIME:BOOL;
END_VAR
-----
CAL    Nom_tempo(IN :=VarBOOL1,
                  PT:= T#5s,
                  Q=> VarBOOL2,
                  EV=>varTIME)
```

---

En este ejemplo, el tiempo se ha definido como una constante (5 segundos), aunque también se puede usar una variable de tipo TIME.

Por otro lado, en lenguaje LD, al insertar un temporizador se conecta a la línea por las señales IN y Q. El mismo código en Lenguaje de Contactos LD se muestra en la Figura 8-4.

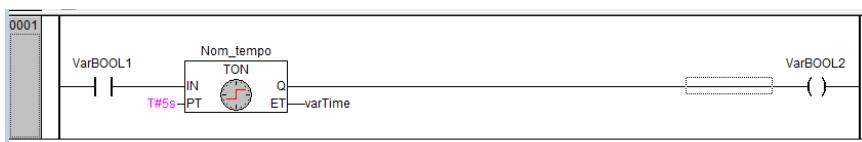


Figura 8-4 – Uso de un temporizador TON en lenguaje LD

## 8.2. Contadores

Al igual que los temporizadores, los contadores también son bloques funcionales predefinidos. Estos bloques funcionales cuentan el número de flancos de subida en una señal booleana (una de las entradas). Hay tres tipos definidos en la norma IEC: contadores de incremento (CTU), de decremento (CTD) y de incremento-decremento (CTUD).

### 8.2.1. Contador de incremento (CTU)

Este bloque funcional tiene tres entradas y dos salidas, de los tipos que se muestran en la Tabla 8-2 . El significado de cada una de las entradas es el siguiente:

Tabla 8-2. Parámetros de entrada y salida en un contador de incremento

Parámetro	Tipo	
CU	BOOL	Entrada
RESET	TIME	Entrada
PV	INT	Entrada
Q	BOOL	Salida
CV	INT	Salida

- **CU:** El contador se incrementa en una unidad cuando se produce un flanco de subida en CU, salvo si RESET es TRUE.
- **RESET:** Inicializa el contador. El valor del contador CV se pone a 0 y Q a FALSE cuando RESET se pone a TRUE.
- **PV:** Valor a contar. Cuando el contador alcanza este valor la salida Q toma el valor TRUE.
- **Q:** Valor a TRUE si el valor actual CV es mayor o igual que PV.
- **CV:** Valor actual del contador

Si una vez que el contador alcance el valor CV siguen llegando flancos de subida en

CU, el valor del contador sigue incrementándose y el valor de Q continuará a TRUE.

### 8.2.2. Contador de decremento (CTD)

Es muy similar al anterior salvo que cada vez que llega un flanco de subida el contador se decremente en una unidad, de forma que la salida Q se pone a TRUE cuando el valor interno del contador llega a 0. En la Tabla 8-3 se muestran los tipos de los parámetros de entrada y salida del bloque funcional.

Tabla 8-3. Parámetros de entrada y salida en un contador de decremento

Parámetro	Tipo	
CD	BOOL	Entrada
LOAD	TIME	Entrada
PV	INT	Entrada
Q	BOOL	Salida
CV	INT	Salida

El significado de cada parámetro y su funcionamiento es el siguiente:

- **CD:** El contador se decremente en una unidad cuando se detecta un flanco de subida en CD, salvo si LOAD es TRUE o el valor actual CV está a 0
- **LOAD:** Inicializa el contador. El valor del contador CV se pone a PV y Q a FALSE cuando LOAD se pone a TRUE.
- **PV:** Valor a contar.
- **Q:** Toma el valor TRUE cuando el valor actual del contador CV alcanza el valor 0
- **CV:** Valor actual del contador

### 8.2.3. Contador de incremento-decremento

Este contador puede incrementarse y decrementarse dependiendo de en qué entrada se detecte un flanco de subida. Es una combinación de los dos anteriores. Los parámetros de este bloque funcional se muestran en la Tabla 8-4.

Tabla 8-4. Parámetros de entrada y salida en un contador de incremento-decremento

Parámetro	Tipo	
CU	BOOL	Entrada
CD	BOOL	Entrada
RESET	BOOL	Entrada
LOAD	BOOL	Entrada
PV	INT	Entrada
QU	BOOL	Salida
QD	BOOL	Salida
CV	INT	Salida

Como se puede comprobar, los parámetros de entrada y salida se corresponden con la unión de las señales de los otros dos contadores:

- **CU:** El contador se incrementa en una unidad con un flanco de subida en CU, salvo si RESET y/o LOAD es/son TRUE.
- **CD:** El contador se decrementa en una unidad con un flanco de subida en CD, salvo si RESET y/o LOAD es/son TRUE. En caso de conflicto prevalece CU.
- **RESET:** Inicializa a 0 el contador. Si está a TRUE, CV se pone a 0, QU a FALSE y QD a TRUE
- **LOAD:** Inicializa el contador al valor PV. El valor del contador CV se pone a PV, QU a TRUE y QD a FALSE cuando LOAD se pone a TRUE. En caso de que tanto RESET como LOAD estén a TRUE prevalece RESET

- PV: Valor a contar.
- QU: A TRUE si el valor actual CV es mayor o igual que PV
- QD: A TRUE si el valor actual CV es 0
- CV: Valor actual del contador.

#### 8.2.4. Uso de los contadores

Con respecto al uso de contadores, de nuevo indicar que se realiza igual que en cualquier otro bloque funcional. Lo único que hay que reseñar en el editor de CoDeSys es que cuando se utiliza en LD, al crear una instancia de bloque funcional se conectará automáticamente a la línea de conexión la primera entrada y la primera salida. Al resto hay que asignarle la señal con el nombre de la variable, tal como se indica en la Figura 8-5.

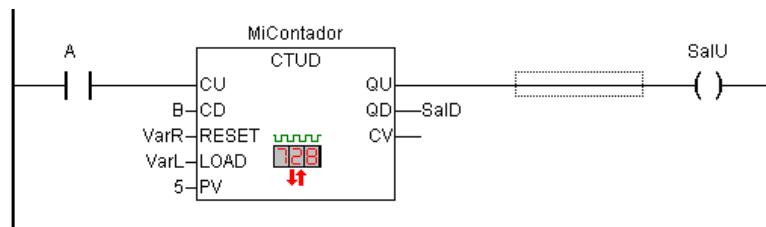


Figura 8-5 – Uso de un contador CTUD en lenguaje LD

### 8.3. Ejercicios

#### 8.3.1. Ejercicio 1

Diseñe un bloque funcional en lenguaje LD que permita disponer de un intermitente (Bloque funcional *Intermitente*) que se mantenga en valor alto y bajo el mismo intervalo de tiempo. Tendrá dos entradas y una salida:

- Entrada *Encendido* (BOOL): El intermitente funcionará cuando esta entrada

esté a TRUE. Si esta señal está a FALSE la salida del intermitente estará a FALSE.

- Entrada *Tiempo* (TIME): Tiempo que se mantiene la señal de salida tanto en valor alto (TRUE) como bajo (FALSE).
- Salida *InterOut* (BOOL): Salida del intermitente

Diseñe el bloque funcional en el fichero *Cap\_1.pro* y pruébelo con el interruptor (señal *Interruptor*) como señal de activación y *Luz* como señal de salida. Será pues necesario definir el bloque funcional, y a continuación definir y usar una instancia en el programa PLC\_PRG, también en LD.

Después de implementar su propia solución, en el fichero *Cap8\_1\_Res.pro* hay implementada otra solución. Explique el funcionamiento de esta solución.

### **8.3.2. Ejercicio 2**

Diseñe un bloque funcional denominado *IntermDos* de manera que los tiempos que está en valores altos y bajos sean en general diferentes. Tendrá 3 entradas y una salida:

- Entrada *Encendido* (BOOL): El intermitente funcionará cuando esté a TRUE. Si está a FALSE la salida estará a FALSE
- Entrada *TArriba* (TIME): Duración de la salida del intermitente en valor TRUE
- Entrada *TAbajo* (TIME): Duración de la salida del intermitente en valor FALSE
- Salida *InterOut* (BOOL): Salida del intermitente

Utilice para la implementación el fichero *Cap8\_2.pro*. Deberá definir el bloque funcional y definir y usar una instancia en el programa PLC\_PRG, también en LD para probarlo con las señales de Entrada/Salida *Interruptor* y *Luz*.

### **8.3.3. Ejercicio 3**

Implementar los bloques funcionales de los ejercicios 1 y 2 en lenguaje de lista de

instrucciones IL.

### 8.3.4. Ejercicio 4

Diseñar un sistema de control en LD para un semáforo, con un interruptor que lo pone en marcha (*Habilita*) y tres luces (*Rojo*, *Ambar* y *Verde*). Las luces deberán estar encendida respectivamente (4, 1 y 3 segundos). Todas las luces deberán estar apagadas cuando el semáforo no esté en marcha. Utilice el fichero *Cap8\_3.pro*.

Después de implementar su diseño, analice y explique el funcionamiento de la solución dada en *Cap8\_3.Res.pro*

### 8.3.5. Ejercicio 5

El objetivo es diseñar el sistema de control de un semáforo para vehículos y peatones del sistema del fichero *Cap8\_4.pro*. El sistema contará con dos interruptores de entrada:

- Interruptor General (*Interruptor*): Enciende o apaga el funcionamiento del semáforo
- Interruptor de Tipo de funcionamiento (*FuncNormal*)
  - Si está a 1: Funcionamiento normal del semáforo
  - Si está a 0: Sólo funciona la luz ámbar de forma intermitente.

Las salidas del sistema son las tres luces del semáforo de vehículos (*RojoCoche*, *AmbarCoche*, *VerdeCoche*) y las dos del de peatones (*RojoPeaton*, *VerdePeaton*).

El programador podrá cambiar (cambiando un programa del autómata) el tiempo en que el semáforo de coches está en rojo y el tiempo en que está en verde. La luz ámbar siempre estará 5 segundos. En intermitencia, estarán las luces 0.5 segundos encendido y 0.5 segundos apagado.

En funcionamiento normal la temporización será la mostrada en la Figura 8-6.

Para la realización de la aplicación, se realizarán las siguientes fases:

- 1.- Escribir en lenguaje IL un programa (de tipo PROGRAM) denominado

*CalculoTiempos* que permita al usuario definir los tiempos en que el semáforo de coches estará en verde, ámbar y rojo y calcule los tiempos del semáforo de peatones. No es necesario escribir una interface de usuario para escribir los valores, sino que éstos se cambiarán escribiendo directamente en el código de este programa.

2.- Escribir un bloque funcional en LD (*SemaforoCocheNormal*) que realice la secuenciación del semáforo de coches de acuerdo al cronograma anterior. Tendrá una entrada:

- **Habilita:** Tipo BOOL. Para que se ponga en marcha esta señal debe estar a TRUE.

Y tres salidas correspondientes a las tres luces (*Rojo*, *Ámbar* y *Verde*). Para este diseño el lector se basará en el programa realizado en el ejercicio anterior, aunque deberá transformarlo para usarlo como bloque funcional.

3.- Escribir un bloque funcional en LD (*SemaforoPeatonNormal*) que realice la secuenciación del semáforo de peatones de acuerdo al cronograma anterior. Tendrá una entrada:

- **Habilita:** Tipo BOOL. Para que se ponga en marcha esta señal debe estar a TRUE.

Y dos salidas correspondientes a las dos luces (*Rojo* y *Verde*). Para la intermitencia, usará el bloque funcional realizado en los ejemplos anteriores.

4.- Escribir un bloque funcional en LD (*SemaforoCochelIntermitente*) que realice la secuenciación del semáforo de coches cuando está la luz ámbar intermitente. Tendrá una entrada:

- **Habilita:** Tipo BOOL. Para que se ponga en marcha esta señal debe estar a TRUE.

Y tres salidas correspondientes a las tres luces (*Rojo*, *Ámbar* y *Verde*). Para la intermitencia, usará el realizado en los ejemplos anteriores.

5.- Escribir el programa principal (PLC\_PRG) en IL o LD que utilizando los bloques funcionales y programas anteriores realice el control completo del semáforo.

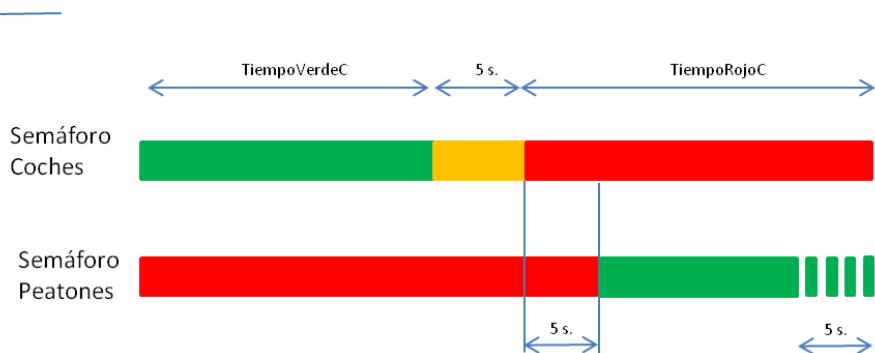


Figura 8-6 – Temporización de los semáforos

### 8.3.6. Ejercicio 6

Diseñar en IL el sistema de acceso a un teatro que se encuentra en el fichero *Cap8\_5.pro*. El teatro dispone de dos puertas, una para entrada y otra para salida. En la entrada se encuentra el sensor *A* y en la salida el *B*.

El sistema deberá tener un recuento de las personas que hay dentro. Dependiendo de ese número se activará el semáforo en la entrada del teatro de la siguiente manera:

- Menos de 9: Luz verde (*LuzVerde*)
- Entre 9 y 11: Luz amarilla (*LuzAmarilla*)
- Más de 12: Luz roja (*LuzRoja*)

Con objeto de que en la visualización aparezca el número de personas (ver Figura 8-7), se usará el contador de incremento-decremento ya definido en las variables globales denominado *Contador*. Para simular la entrada o salida de personas, pulsar los botones *A* o *B*.

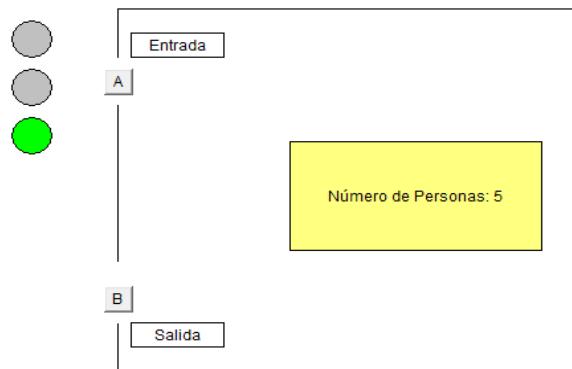


Figura 8-7 – Visualización del ejercicio 6.

### 8.3.7. Ejercicio 7

El objetivo es diseñar un sistema de control para el acceso a un museo con aforo limitado (*Cap8\_6.pro*) con una única puerta que se utilizará tanto para entrada como para salida. El sistema de acceso está compuesto por dos células fotoeléctricas próximas (*A* y *B*) que se ponen a TRUE cuando una persona la cruza (Ver la animación a velocidad baja para analizar el comportamiento de las células cuando entran o salen personas). Para ver correctamente la animación, asegúrese que en el mismo directorio del fichero de CoDeSys se encuentra el directorio *DibujosEj6* que incluye las imágenes de las personas.

La capacidad máxima se almacena en la variable global *Aforo*, que está declarada en el archivo que se proporciona pero a la que hay que darle valor. En la entrada del museo hay un semáforo, de modo que si el número de personas dentro es inferior a *Aforo* menos dos, está encendida la salida *LuzVerde*, si está entre *Aforo* menos dos y *Aforo*, se enciende la salida *LuzAmarilla* y si es mayor o igual a *Aforo* se enciende la salida *LuzRoja* que impide el paso de más visitantes.

Para contar las personas en el interior se utiliza un contador de incremento-decremento denominado *Contador*, ya declarado como variable global en el fichero proporcionado. Además se utiliza otro contador de incremento *Visitas*, también

declarado como variable global, para el recuento del número total de visitas. Utilice estos contadores para que el valor se refleje en las animaciones.

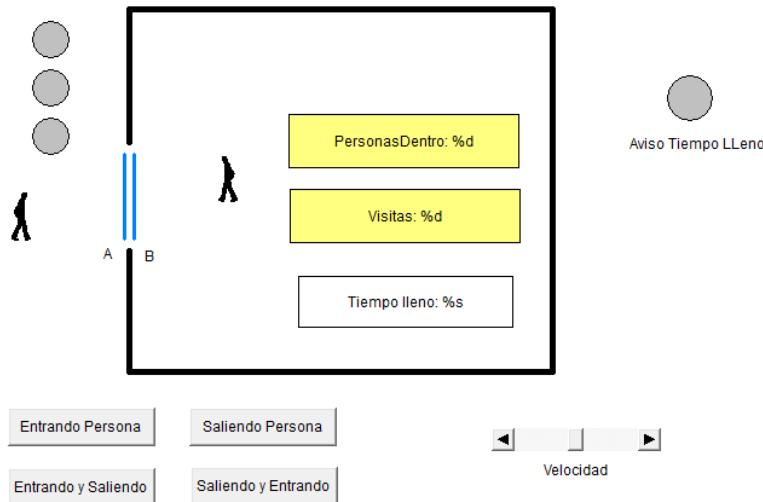


Figura 8-8 – Visualización del ejercicio 7.

La realización del programa de control se realizará en el siguiente orden:

1.- En primer lugar se considerará que no pueden entrar y salir dos personas simultáneamente, es decir sólo se considerarán entrada de una persona o salida de una persona (los dos botones superiores en la Figura 8-8). Para ello se diseñará:

- Un programa principal en IL ya creado (incluye una única instrucción que permite realizar la animación y que no se debe borrar). El programa a diseñar debe gestionar los contadores y activar las luces de salida de acuerdo al estado de *Contador*.
- Un programa en LD (hay que crearlo) que gestione las entradas y salidas. Es decir, a partir de los valores que se reciban en las entradas A y B, debe proporcionar las señales de incremento y decremento de los contadores.

2.- Incluir en el programa la gestión de un temporizador denominado *Temporizador* ya definido. El temporizador se activa cuando el museo está lleno. Cuando el

museo esté lleno durante más de un cierto valor de tiempo preestablecido (elegido por el programador) se debe encender el piloto *AvisoLleno*, que verá el vigilante, con objeto de que “anime” a los visitantes a acelerar su visita.

3.- Completar el programa con la posibilidad de que dos personas puedan salir y entrar simultáneamente (dos botones inferiores de la animación de la Figura 8-8).

### 8.3.8. Ejercicio 8

Este ejercicio es similar al anterior, salvo que en lugar de una única puerta hay dos puertas, de manera que las dos sirven tanto de entrada como de salida, tal como se muestra en la Figura 8-9. El sistema se encuentra modelado en *Cap8\_7.pro*. Para resolverlo se va a diseñar un bloque funcional que detecte personas en puertas del tipo del ejercicio y se definirán dos instancias para cada una de las puertas.

Para la realización del sistema de control se diseñará un bloque funcional en lenguaje LD (se puede adaptar el programa que se hizo en el ejercicio anterior) para la gestión de las entradas y salidas del tipo de puertas del ejercicio. El bloque funcional será del tipo *SistemaES* y tendrá:

- Dos entradas denominadas *EntA* y *EntB* que se corresponden con cada una de las dos células fotoeléctricas.
- Dos salidas denominadas *EntraPersona* y *SalePersona* deben dar un flanco de subida (pasar de FALSE a TRUE) cuando entre o salga respectivamente una persona. Estas señales se usarán para los contadores.

El programa principal deberá gestionar las dos instancias de tipo *SistemaES* correspondientes a cada una de las puertas.

Para una correcta visualización de la animación asegúrese de que la carpeta *DibujosE6* se encuentra en el mismo directorio que el fichero *.pro*.

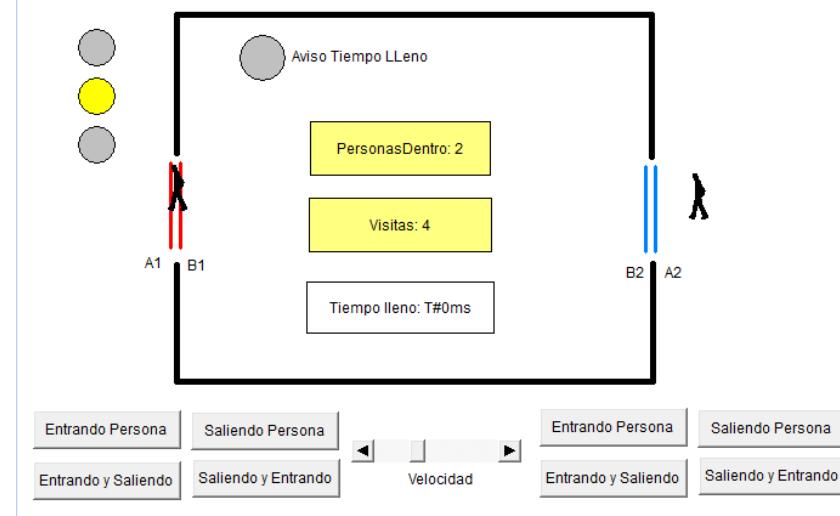


Figura 8-9 – Visualización del ejercicio 8.



# 9. PROGRAMACIÓN EN ST

---

Este capítulo está dedicado a la programación en Texto Estructurado (ST), un lenguaje de más alto nivel que los anteriores, y que tiene una sintaxis similar al Pascal o al C, aunque con un juego de instrucciones mucho más simple. Conviene siempre tener presente que a pesar de las mayores posibilidades de programación que dispone ST, son programas que funcionan en un PLC y que por tanto se ejecutan de forma cíclica en un tiempo reducido (como es sabido, con un tiempo de ejecución de algunos milisegundos cada ciclo en aplicaciones normales). Al disponer el lenguaje ST de estructuras de control más complejas como bucles FOR o WHILE, si no se programa con cuidado, es posible que el programa quede atrapado (o al menos se quede mucho tiempo) en uno de estos bucles. Por tanto, el programador en este lenguaje debe tener un especial cuidado en que el programa se ejecute en el tiempo máximo establecido.

Es definitiva, se trata de un lenguaje adecuado para algoritmos de control complejos y para la utilización de distintos tipos de variables.

## 9.1. Instrucciones en ST

Un programa en ST consiste en una serie de instrucciones separadas por punto y coma (;). Por tanto, es posible escribir varias instrucciones en una única línea o una instrucción que ocupe varias líneas.

Los comentarios van encerrados por los delimitadores (\* y \*), pudiendo aparecer en cualquier parte del texto.

Teniendo en cuenta que las instrucciones del lenguaje ST son similares a cualquier lenguaje de alto nivel y que el lector debe tener unos mínimos conocimientos en programación, el juego de instrucciones se va a describir muy brevemente, acudiendo en cada caso básicamente a un ejemplo. Una descripción con un mayor nivel de detalle la puede encontrar en [8] y [9]

### 9.1.1. Asignación

Para la asignación se utiliza el operador “:=”, por ejemplo

```
A:=B+10.7;
```

### 9.1.2. Sentencia IF

Se utiliza para la selección de alternativas por medio de condiciones booleanas.  
Ejemplo:

---

```
IF A>10 AND B<>10 THEN
    RES:=17;
ELSEIF C<=4 THEN
    RES:=100;
ELSE
    RES:=200;
END_IF;
```

---

### 9.1.3. Sentencia CASE

La sentencia CASE permite la elección entre distintos bloques de sentencias dependiendo del valor de una expresión entera. Ejemplo:

---

```
CASE Selec OF
    1 : RES:=11;
    2 : RES:=12;
    4 : RES:=14;
    ELSE RES:=20;
END_CASE;
```

---

Si la variable entera *Selec* tiene los valores 1, 2 ó 4 ejecuta las sentencias

correspondientes, en caso contrario ejecuta las sentencias del bloque ELSE.

#### 9.1.4. Sentencia FOR

Ejecuta las sentencias del bloque dentro del bucle FOR de acuerdo a las condiciones de inicio, fin e incremento del bucle. Ejemplo:

---

```
FOR I:=1 TO 100 BY 2 DO
  Vec[I]:=I+1;
  G := Mifuncion();
END_FOR;
```

---

#### 9.1.5. Sentencia WHILE

Ejecuta un bucle, de forma que la condición de salida está situada al comienzo de dicho bucle. En el ejemplo, se estarán ejecutando las dos instrucciones del interior del bucle mientras el valor de la variable *m* sea mayor que cero.

---

```
m:=10;
WHILE m>0 DO
  Vec[m]:=m+1;
  m:=m-1;
END WHILE;
```

---

#### 9.1.6. Sentencia REPEAT

Es otra sentencia de bucle, pero con la condición de salida situada al final de éste. Ejemplo:

```
REPEAT
    m:=i*j;
    i:=i+1;
UNTIL i>1000
END_REPEAT;
```

---

### 9.1.7. Sentencia EXIT

Permite salir de cualquier bucle sin necesidad de que se haya cumplido la condición de salida.

## 9.2. Uso de funciones y bloques funcionales

### 9.2.1. Llamadas a funciones

Para llamar a una función desde otro POU se utiliza simplemente el nombre de la función. Para suministrarle los parámetros se procede de forma similar a cualquier lenguaje de programación convencional. A continuación se ven los mismos ejemplos que anteriormente se vieron en el lenguaje IL.

La función que calcula la media de tres números escrita en ST sería:

---

```
FUNCTION Media3 : INT
VAR_INPUT
    Valor1: INT;
    Valor2 : INT;
    Valor3 : INT;
END_VAR
-----
Media3:=(Valor1+Valor2+Valor3) 3;
```

---

Si se quiere llamar a la función para calcular la media de 10, 20 y 31 desde un programa ST, se hace de la siguiente manera:

```
Result:=Media3(10,20,31);
```

### 9.2.2. Llamada a bloques funcionales

Como ya se ha explicado, la llamada a un bloque funcional se realiza con el **nombre de la instancia**. Para indicar los parámetros de entrada y salida se introducen entre paréntesis con el nombre formal del parámetro seguido de '':=' si es de entrada o '=>' si es de salida seguido de una expresión con el valor que se quiera dar a dicho parámetro. Los parámetros se pueden introducir en cualquier orden.

En lenguaje ST, al igual que se vio en el lenguaje IL, también se puede hacer la llamada mediante la utilización de la estructura de datos asociada a cada instancia de bloque funcional.

Recordemos el bloque funcional *Biestable* definido anteriormente y cuyos parámetros de entrada y salida son los siguientes:

```
FUNCTION_BLOCK Biestable
VAR_INPUT
    EntS : BOOL;
    Entr: BOOL;
END_VAR
VAR_OUTPUT
    Estado : BOOL :=TRUE;
END_VAR
```

En el programa desde el que se quiera llamar a una instancia de dicho bloque funcional se define la instancia y se realiza la llamada por los mismos métodos que se vieron en el caso del lenguaje IL. La llamada utilizando el primer método, es decir asignando los parámetros de entrada y salida desde la propia línea de llamada a la instancia sería:

---

```

VAR
    MiBiest : Biestable;
    SenalS:BOOL;
    SenalR:BOOL;
    MiEstado:BOOL;
END_VAR
-----
MiBiest(EntS:=SenalS,EntR:=SenalR,Estado=>MiEstado);

```

---

Con el segundo método, es decir, usando la estructura de datos asociada a la instancia:

---

```

VAR
    MiBiest : Biestable;
    SenalS:BOOL;
    SenalR:BOOL;
    MiEstado:BOOL;
END_VAR
-----
MiBiest.EntS:=SenalS;
MiBiest.EntR:=SenalR;
MiBiest();
MiEstado:=MiBiest.Estado;

```

---

Tal como se vio con el lenguaje IL, también se permiten soluciones mixtas en la que algunos parámetros se pasan en la llamada a la función y otros a través de la estructura de datos.

### 9.2.3. Sentencia RETURN

Se utiliza en funciones o bloque funcionales para salir de ella desde cualquier punto. En cualquier caso hay que tener en cuenta que en el caso de las funciones es

necesario asignarle valor antes de ejecutar RETURN, ya que en caso contrario no devolvería ningún valor.

## 9.3. Ejercicios

### 9.3.1. Ejercicio 1

Diseñar un bloque funcional escrito en ST para activar un display de 7 segmentos como el de la Figura 9-1. El bloque funcional tendrá una entrada:

- *Dígito*: Valor en decimal que se quiere enviar al display. Será de tipo entero (INT) y si no estuviera entre los valores 0 y 9 el bloque funcional dará una señal de error.

Las salidas serán las siguientes:

- *a, b, c, d, e, f, g*: Señales booleanas correspondientes a cada uno de los siete segmentos del display, tal como aparecen representados en la figura.
- *Error*: Señal booleana que se activará si la señal de entrada no es un valor entre 0 y 9. En ese caso no se encenderá ninguno de los segmentos.

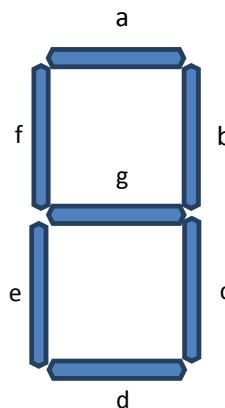


Figura 9-1 – Display de siete segmentos

El bloque funcional se probará en *Cap9\_1.pro*. En el fichero se dispone de un display de siete segmentos cuyos segmentos se corresponden con las variables de salida *a0*, *b0*, *c0*, *d0*, *e0*, *f0* y *g0* (definidas como variables globales) y una luz que se activará con la señal de error del bloque funcional (*No valido*). El dígito en decimal se introducirá en el recuadro indicado en la animación (Variable global de entrada *Numero*). La llamada al bloque funcional se realizará desde el programa principal, también en lenguaje ST.

El ejercicio también se encuentra resuelto en *Cap9\_1\_Res.pro*.

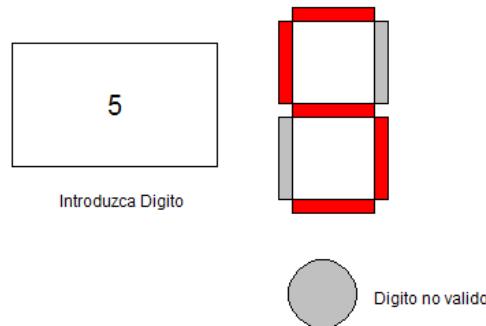


Figura 9-2 – Visualización del ejercicio 1

### 9.3.2. Ejercicio 2

Utilizando el bloque funcional del ejercicio anterior, realizar un programa en ST que represente número de tres cifras. Para ello se dispone de tres displays, el de las centenas cuyos segmentos se activan con las señales *a0* a *g0*, el de las decenas que se activan con las señales de *a1* a *g1* y el display de las unidades con señales *a2* a *g2*. Si el número no está comprendido entre 0 y 999 se encenderá la luz de error. Implementar el diseño en el fichero *Cap9\_2.pro*.

### 9.3.3. Ejercicio 3

Es habitual que al adquirir un display de siete segmentos, éste venga provisto de un circuito conversor de código BCD a siete segmentos. Es decir, en estos dispositivos, las entradas de cada display serán los cuatro dígitos del código BCD del dígito que se quiere representar. En este ejercicio se realizarán las siguientes actividades:

- a) Implementar en ST un bloque funcional que se comporte como el circuito conversor, es decir, deberá tener cuatro entradas booleanas (el código BCD del número a representar) y siete salidas (los 7 segmentos)
- b) Repetir el ejercicio 2 anterior pero realizando el paso intermedio por el código BCD, para lo que se utilizará el bloque funcional realizado en a).

Para implementar el programa se usará el mismo fichero *Cap9\_2.pro*.



# 10. PROGRAMACIÓN EN FBD

El lenguaje de Diagramas de Bloques Funcionales (FBD) tuvo su origen en el campo del procesamiento de señales. Se trata de un lenguaje gráfico en el que las distintas funciones, programas, bloques funcionales u operaciones aritméticas o lógicas se representan mediante bloques con entradas y salidas. Las conexiones entre bloques representan las señales.

La estructura de un programa es similar a la del lenguaje LD. Está estructurado en redes (networks) de forma que en cada una de ellas realiza operaciones mediante conexiones de bloques. A estas redes también se le pueden asignar etiquetas que se pueden usar para realizar saltos en el código.

En la Figura 10-1 se muestra un programa escrito en FBD en CoDeSys con dos redes.

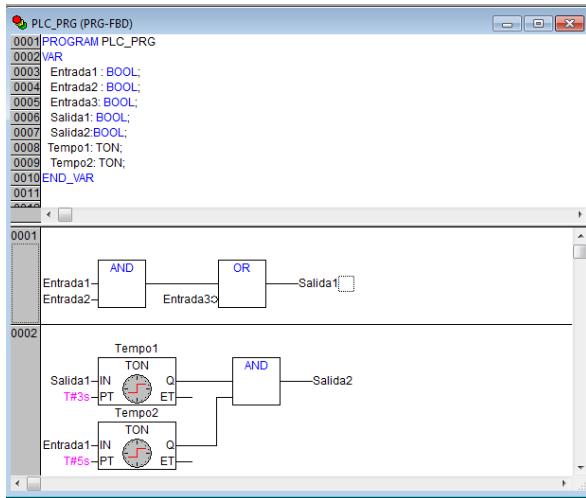


Figura 10-1 – Ejemplo de programa en FBD en CoDeSys

Al igual que en el caso del editor gráfico de LD, la edición del gráfico se realiza mediante comandos y no mediante la libre colocación de bloques (como por ejemplo en Simulink). Esto le resta flexibilidad a la edición pero permite realizar siempre redes correctas desde el punto de vista sintáctico.

## 10.1. Cómo trabajar con bloques en CoDeSys

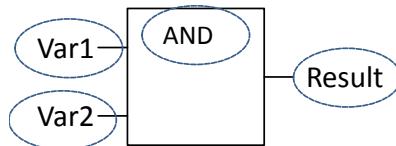
En esta sección se describen las operaciones básicas que permite CoDeSys para la realización de un programa en lenguaje FBD. Antes de analizar los comandos, es importante conocer las distintas posiciones en que puede estar el cursor gráfico en el editor, ya que el efecto que tenga cada comando va a depender de la posición en que se encuentre el cursor.

### 10.1.1. Posición del cursor gráfico

El cursor en el editor gráfico de FBD, que se representa mediante un recuadro con línea discontinua, se puede colocar en las posiciones que se describen en esta sección. Lógicamente, la selección de la posición se realiza con el ratón.

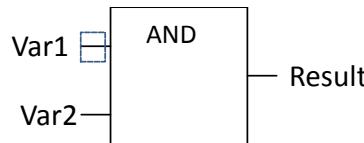
#### 10.1.1.1. Posición 1

El cursor puede estar en cada campo de texto, incluyendo nombre de variables o de operadores, funciones o bloques funcionales.



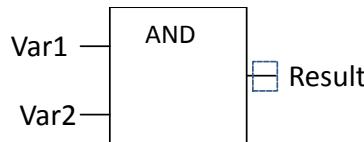
#### 10.1.1.2. Posición 2

El cursor se encuentra en la entrada de un bloque



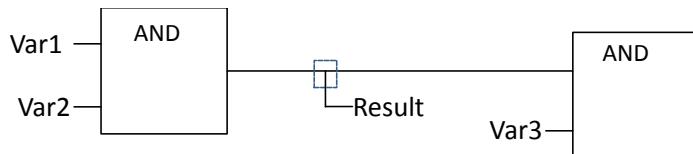
#### 10.1.1.3. Posición 3

En esta posición, el cursor se encuentra a la salida de un bloque



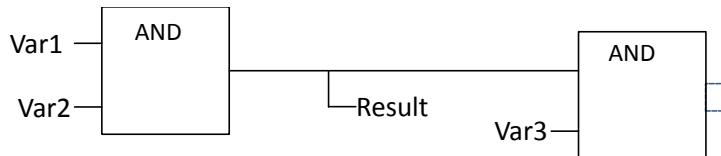
#### 10.1.1.4. Posición 4

El cursor se coloca en una bifurcación para una asignación o salto.



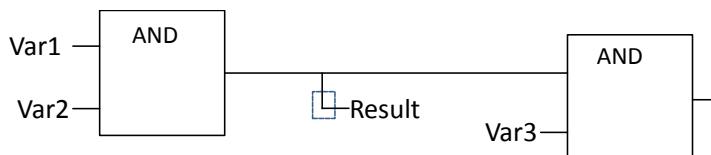
#### 10.1.1.5. Posición 5

En la posición más a la derecha de la red



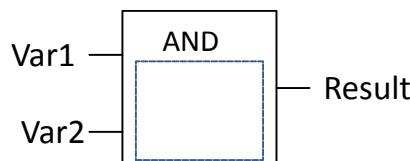
#### 10.1.1.6. Posición 6

Delante de una asignación



### 10.1.1.7. Posición 7

En esta última posición, el cursor está en el interior de un bloque.



## 10.1.2. Comandos en FBD

Al igual que en otros lenguajes en CoDeSys, a los distintos comandos se puede acceder o bien mediante los botones en el menú de herramientas, mediante el menú contextual (botón derecho del ratón) o en los menús *Insert* o *Extras*. En la Tabla 10-1 se muestran los botones correspondientes a los distintos comandos.

### 10.1.2.1. Comando Assign

Permite asignar a una determinada señal (un punto de las líneas de conexión) una determinada variable. Se puede realizar con el cursor en las posiciones 2, 3 y 5 descritas en el apartado anterior. A continuación se sustituirán las “???” por el nombre de la variable que se desee.

### 10.1.2.2. Comando Jump

Realiza un salto a una red (network). Esta red a la que se desea ir debe tener una etiqueta, que al igual que en lenguaje LD se escribe en la esquina superior izquierda de la red. Esta acción se realiza con el cursor en las posiciones 2, 3 y 5. En lugar de las interrogaciones habrá que introducir el nombre de la etiqueta.

### 10.1.2.3. Comando Return

Realiza un RETURN de la función o bloque funcional en que se está. En las mismas posiciones que en los dos casos anteriores.

### 10.1.2.4. Comando Box

Inserta en la posición del cursor un operador, función o bloque funcional. Por defecto se inserta el bloque AND pero esto se puede cambiar sin más que poner el

cursor en el nombre del bloque y escribir un nuevo tipo de bloque o bien utilizar el asistente (F2). En el caso de funciones y bloques funcionales aparecerán los nombres de las entradas y salidas. En el caso de bloques funcionales también habrá que dar nombre a la instancia (nombre que aparece sobre el bloque).

El bloque se insertará en una posición que depende de la posición del cursor:

**Posición 2 o posición 3:** El bloque se intercalará en esa posición conectándose automáticamente la primera entrada y la primera salida del bloque.

**Posición 5:** El nuevo bloque se coloca a continuación del último bloque.

#### **10.1.2.5. Comando *Input***

Permite añadir nuevas entradas a los bloques que lo permitan, por ejemplo en los bloques que permiten trabajar con un número variable de operandos como AND, OR, ADD,..

El cursor deberá estar en las posiciones 2 o 7.

#### **10.1.2.6. Comando *Output***

Sirve para hacer una nueva asignación cuando el cursor está en una posición de asignación, como por ejemplo, la posición 4. Por tanto sirve para asignarle el valor de una señal a distintas variables.

#### **10.1.2.7. Comando *Negate***

Se utiliza para negar entradas, salidas, saltos o RETURN. Cuando se seleccione en el gráfico aparecerá un círculo asociado al elemento que se ha negado.

#### **10.1.2.8. Comando *Set/Reset***

Con esta instrucción se pueden definir salidas de tipo SET, RESET o normales. La idea es similar a las bobinas SET y RESET que se vieron anteriormente en LD. En las salidas de tipo SET aparece un recuadro con una S, en las de tipo RESET un recuadro con una R y en las normales no aparece nada.

Tabla 10-1. Botones en la edición de gráficos en FBD

Menú contextual	Botón
Comando <i>Input</i>	
Comando <i>Output</i>	
Comando <i>Box</i>	
Comando <i>Assign</i>	
Comando <i>Jump</i>	
Comando <i>Return</i>	
Comando <i>Negate</i>	
Comando <i>Set/Reset</i>	

## 10.2. Lenguaje CFC

CoDeSys dispone de un segundo lenguaje de tipo Diagrama de Bloques denominado CFC (Continuous Function Chart). El editor de este lenguaje es más flexible que el FBD, permitiendo colocar los bloques en cualquier posición de la red y conectarlos manualmente. Al no ser un lenguaje contemplado en la norma IEC no se va a describir en este libro.

## 10.3. Ejercicios

### 10.3.1. Ejercicio 1

Se desea diseñar un sistema de seguridad para una máquina, de modo que mientras esté en funcionamiento, un operario debe confirmar que la está atendiendo cada cierto tiempo mediante la pulsación de un botón. Si el operario no pulsa el botón cada cierto tiempo (10 segundos) se encenderá una luz de alarma, y si continua durante otros 5 segundos sin pulsarla la máquina se pondrá en modo pausa, estado en el que continuará hasta que pulse el botón. El ejercicio se puede realizar en *Cap10\_1.pro*.

La máquina tiene dos entradas booleanas:

- *Start*, pulsador que pone en marcha la máquina. El pulsador estará en color verde si la máquina está encendida.
- *ControlOK* interruptor de dos posiciones que en la animación aparecen en dos colores, azul y celeste.

También tiene dos salidas booleanas

- *Warning*. Señal luminosa que se enciende cuando se produce una alarma
- *Stop*: Señal luminosa que se activa cuando la máquina se detiene.

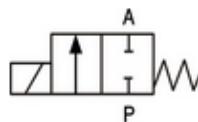
#### Solución

Una solución al problema se encuentra en *Cap10\_1\_Res.pro*. La primera de las redes gestiona la salida *Warning*. Se ha utilizado un temporizador de desconexión TOF. La idea es reiniciar el temporizador (IN a FALSE) cuando se detecte un flanco de subida o un flanco de bajada de *ControlOK* (al ser un interruptor puede estar en cualquier posición). Por otro lado, la entrada del temporizador siempre estará a TRUE cuando *Start* está a FALSE (bloque OR más a la derecha con la entrada *Start* negada). De este modo se garantiza que el temporizador no funcione con la máquina apagada.

La segunda de las redes gestiona la señal *Stop* con un contador TON activado por *Warning*.

### 10.3.2. Ejercicio 2

Se desea controlar una válvula todo-nada neumática. La válvula dispone de dos sensores de fin de carrera para detectar cuándo está completamente abierta o cerrada. Estos sensores son *FCA* y *FCC* para la apertura y cierre respectivamente. El control de la válvula se hace mediante una válvula de distribución neumática 2/2 normalmente cerrada con retorno automático, como la representada en la figura.



Por tanto su comportamiento es el siguiente. Si la bobina (señal *Bobina*) está activa, deja pasar el aire y la válvula principal se abre (posición de la izquierda en el dibujo). Si por el contrario la bobina está desactivada, el muelle la lleva a la posición de la derecha en el dibujo, cortando el aire, y la válvula principal se cierra. Es decir, para abrir la válvula y mantenerla abierta es necesario tener la bobina activada y para cerrarla y mantenerla cerrada se deberá mantener la bobina desactivada.

Se desea también detectar los fallos en la válvula todo-nada. Se consideran los siguientes:

- Si manteniendo la señal *Bobina* durante 6 segundos, la válvula no se abre completamente.
- Si con la bobina desactivada, la válvula no se cierra completamente en 10 segundos.

Para el control de la válvula se va a diseñar un bloque funcional en lenguaje FBD con los siguientes parámetros de entrada:

- *OrdenApertura*: La válvula debe abrirse con un flanco de subida en este señal (por ejemplo pulsar un botón) y se mantendrá abierta hasta que se dé la orden de cierre.
- *OrdenCierre*: La válvula se cerrará con un flanco de subida en esta señal, manteniéndose cerrada mientras no se dé una orden de apertura.
- *FCAper*: Fin de carrera de válvula abierta
- *FCCier*: Fin de carrera de válvula cerrada

Los siguientes parámetros de salida del bloque funcional serán:

- *ActBobina*: Actuación sobre la bobina de la válvula 2/2
- *Error*: Se pondrá a TRUE cuando se detecten los errores arriba descritos.

El sistema se probará con el fichero *Cap10\_2.pro*. La Figura 10-2 muestra el sistema. Se creará el bloque funcional (para lo que habrá que crear una nueva POU) y una instancia de dicho bloque funcional también en FBD para controlar la válvula de la figura, en el programa *ControlValvula* ya creado pero vacío.

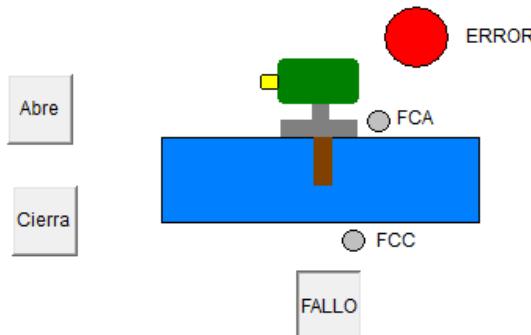


Figura 10-2 – Visualización de la válvula todo-nada

Las variables de E/S del sistema a controlar y que se muestran en la figura son:

- Entradas
  - *Abre*: Señal del pulsador que provoca la apertura de la válvula.
  - *Cierra*: Señal del pulsador que provoca el cierre de la válvula.
  - *FCA*: Fin de carrera de apertura de la válvula de la figura.
  - *FCC*: Fin de carrera de cierre de la válvula de la figura.
- Salidas
  - *Bobina*: Señal de activación de la bobina de la válvula de la figura.

- *ErrorValvula*: Señal luminosa de la figura para aviso de fallo.

Para simular los fallos se usará el interruptor *FALLO*. Cuando se pulse, la válvula se bloqueará permitiendo comprobar si funciona la gestión de fallos. Nótese que no es una señal de E/S del automatismo lógico, sino que forma parte de la simulación del proceso.

### 10.3.3. Ejercicio 3

Este ejercicio es similar al anterior, salvo que en este caso se pretende controlar una electroválvula motorizada. Por tanto, en este caso para abrir la válvula habrá que activar la señal *MotorAper* hasta que la válvula llegue a su fin de carrera, momento en el que habrá que parar el motor. Para cerrar, se realiza igual pero con la señal *MotorCier*.

Se volverá a diseñar un bloque funcional en FBD con los mismos parámetros de entrada, pero los parámetros de salida se corresponderán con las señales de apertura y cierre del motor. Por tanto los parámetros de entrada serán:

- *OrdenApertura*: La válvula debe abrirse con un flanco de subida en este señal (por ejemplo pulsar un botón) y se mantendrá abierta hasta que se dé la orden de cierre.
- *OrdenCierre*: La válvula se cerrará con un flanco de subida en esta señal, manteniéndose cerrada mientras no se de una orden de apertura.
- *FCAper*: Fin de carrera de válvula abierta
- *FCCier*: Fin de carrera de válvula cerrada

Y los parámetros de salida:

- *ActMotorAper*: Actuación sobre el motor para apertura.
- *ActMotorCier*: Actuación sobre el motor para cierre.
- *Error*: Se pondrá a TRUE cuando se detecten los errores descritos en el ejercicio anterior.

El sistema se probará con el fichero *Cap10\_3.pro*. Se creará el nuevo bloque funcional y una instancia también en FBD en el programa *ControlValvula* ya creado pero vacío. El sistema a controlar se muestra en la Figura 10-3.

Las variables de E/S del sistema a controlar y que se muestran en la figura son:

- Entradas

- *Abre*: Señal del pulsador que provoca la apertura de la válvula.
- *Cierra*: Señal del pulsador que provoca el cierre de la válvula.
- *FCA*: Fin de carrera de apertura de la válvula de la figura.
- *FCC*: Fin de carrera de cierre de la válvula de la figura.

- Salidas

- *MotorAper*: Señal de activación del motor en el sentido que permite abrir la válvula.
- *MotorCier*: Señal de activación del motor en el sentido inverso, y por tanto permitirá cerrar la válvula.
- *ErrorValvula*: Señal luminosa de la figura para aviso de fallo.

Para simular los fallos se usará el interruptor *FALLO*. Cuando se pulse, la válvula se bloqueará.

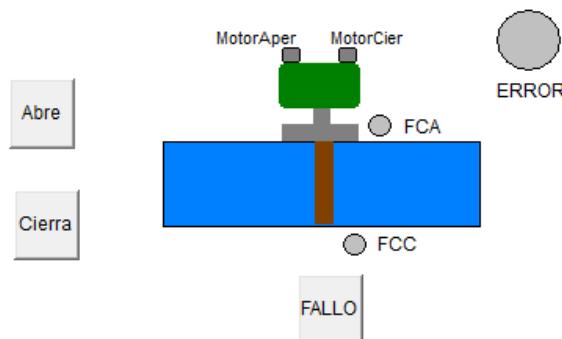


Figura 10-3 – Visualización de la válvula motorizada



# 11. PROGRAMACIÓN EN SFC

El último de los lenguajes que contempla la norma IEC-61131-3 es SFC (Sequential Function Chart). Este lenguaje básicamente es muy similar al muy extendido GRAFCET [10], basado en las Redes de Petri [11]. Se trata de una metodología de diseño de automatismos lógicos secuenciales basado en un grafo de estados, compuesto por etapas a las que se le asocia las acciones de control a realizar en cada momento, y transiciones que permiten la evolución dinámica del sistema en función de los valores que van tomando las entradas del sistema. En la Figura 11-1 se muestra un ejemplo de un proceso simple secuencial.

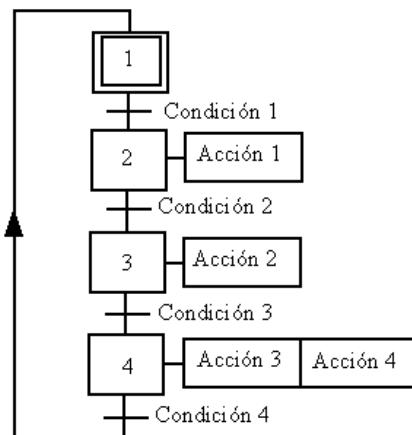


Figura 11-1 – Ejemplo en GRAFCET

Aunque un curso en profundidad de GRAFCET queda fuera del alcance de este libro, este capítulo empezará con una breve revisión de los elementos del GRAFCET y su modo de funcionamiento.

## 11.1. Introducción a GRAFCET

Como se indicó anteriormente, GRAFCET es un método gráfico de representación de automatismos. La representación gráfica está compuesta de tres elementos: etapas, transiciones y arcos orientados.

### 11.1.1. Etapas

Una etapa corresponde a un estado de funcionamiento del automatismo en el que se determinan las condiciones de operación del sistema. Se representa mediante un cuadrado (Ver Figura 11-2)

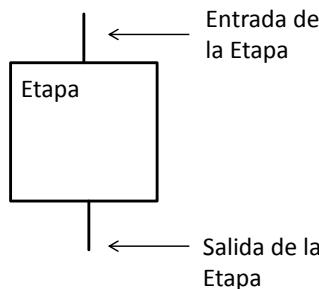


Figura 11-2 – Representación de una etapa

En cada instante la etapa puede estar:

- Activa o Marcada: Se suele representar con un punto en el interior del cuadrado o cambio de color del cuadrado
- Inactiva o Desmarcada

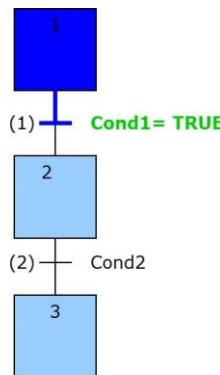
El estado del Automatismo Lógico, y por tanto su comportamiento, queda definido por las etapas que están activas en cada instante, que tiene que ser como mínimo una, pero puede haber varias etapas activas simultáneamente. Cuando se diseña un GRAFCET es necesario realizar el marcado inicial de al menos una etapa.

### 11.1.2. Transiciones

Las transiciones van a permitir el cambio del estado de activación de una o varias etapas, es decir, permitirán la evolución dinámica del grafo de acuerdo a unas determinadas reglas de evolución que se verán posteriormente. Se representan mediante una barra horizontal. Las transiciones pueden tener asociada una condición lógica (receptividad), que normalmente estará relacionada con las entradas del sistema de control, y que se colocan junto a la transición.

Una transición puede estar:

- **Validada:** si todas las etapas inmediatamente precedentes y conectadas directamente a la transición (etapas de entrada a la transición) están activas.
- **No validada:** Cuando hay al menos una etapa de entrada a la transición que no está validada
- **Franqueable:** Cuando la transición está validada y se verifica la condición lógica asociada.



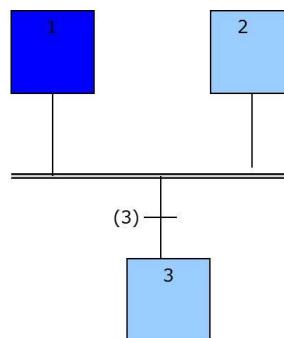
Transición 1: Franqueable

Transición 2: No validada

Figura 11-3 – Transiciones (I)

La transición se franquea obligatoriamente cuando está validada y se verifica la receptividad asociada. En la Figura 11-3 se representa un grafo en el que las etapas activas están representadas por un color oscuro. Se puede observar una transición validada y franqueable (1) y otra no validada (2).

En la Figura 11-4 la transición no se encuentra validada, ya que una de las etapas de entrada no está activa.



Transición 3: No Validada

Figura 11-4 – Transiciones (II)

### 11.1.3. Arcos orientados

Los arcos orientados unen las etapas con las transiciones o las transiciones con las etapas pero NUNCA unirán dos transiciones o dos etapas entre sí. Marcan las vías que van a seguir los marcados de etapas y por tanto la evolución dinámica del sistema de control.

Se representan con líneas verticales y horizontales, desaconsejándose dentro de lo posible las líneas oblicuas. Por convención el sentido es de arriba a abajo, y por tanto, aunque los grafos son orientados no se usarán flechas cuando se representen arcos en este sentido. Cuando se quieran representar arcos en los sentidos contrarios se usarán flechas, o siempre que convenga para una mayor claridad del

grafo.

Se utilizará el doble trazado horizontal si hay que reagrupar varios arcos ligados a la misma transición, tal como se aprecia en la Figura 11-4.

#### **11.1.4. Reglas de evolución**

GRAFCET permite modelar el comportamiento dinámico de un sistema de control mediante la evolución de las etapas activas de acuerdo a una serie de reglas simples. Como se indicó anteriormente, el conjunto de etapas activas en cada instante de tiempo determinan el estado en que se encuentra el sistema. Esta evolución viene determinada por las siguientes cinco reglas:

##### **Regla 1. Situación inicial**

La situación inicial del GRAFCET caracteriza el comportamiento inicial de la red y se corresponde con el conjunto de las etapas activas inicialmente y que permitirá el comienzo del funcionamiento. Es imprescindible que haya una o varias etapas activas inicialmente ya que de otra manera el sistema no podrá evolucionar.

##### **Regla 2. Condición de franqueo de una transición**

La evolución del estado en cada momento del GRAFCET está asociada al franqueo de una transición, que no se puede producir a menos que se den estas dos condiciones:

- la transición esté validada,
- la receptividad asociada sea cierta.

En estas condiciones la transición es franqueable y es obligatoriamente franqueada.

##### **Regla 3. Evolución de las etapas activas**

Esta regla es la que cambia el estado de activación de las etapas. El franqueo de una transición implica simultáneamente la activación de todas las etapas de salida de dicha transición y la desactivación de todas sus etapas de entrada. Como se indicó anteriormente, las etapas de entrada de una transición son aquellas para las que existe un arco que va de la etapa a la transición, mientras que las etapas de salida de una transición son aquellas para las que existe un arco de la transición a la etapa.

#### Regla 4. Evolución simultánea

Si en un instante determinado hay varias transiciones simultáneamente franqueables, éstas se franquean se forma simultánea de acuerdo a la tercera regla.

#### Regla 5. Activación prioritaria

Si en el curso de una evolución, una misma etapa debe ser al mismo tiempo activada y desactivada, ésta finalmente permanecerá activada.

En las siguientes figuras se muestran algunos ejemplos de evolución de acuerdo a la regla número 3. La Figura 11-5 es un proceso secuencial en el que la primera etapa está activa y la receptividad es cierta, por lo que la transición se franquea tal como muestra la figura. En la situación final el sistema no evolucionará hasta que la receptividad correspondiente a la transición 2 sea cierta.

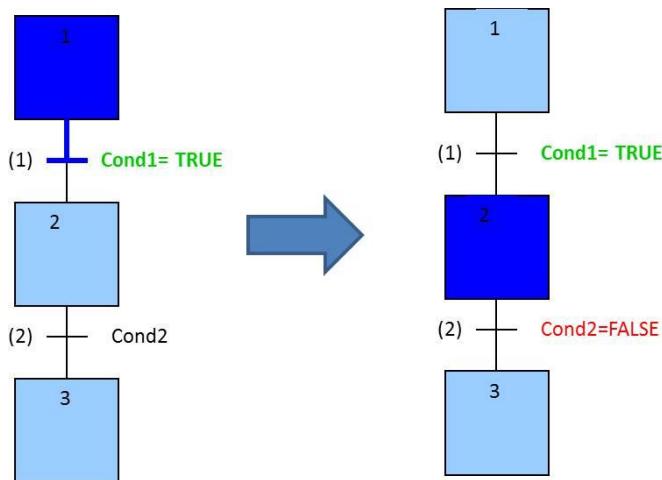


Figura 11-5 – Reglas de evolución (I)

El ejemplo de la Figura 11-6 muestra que el número de etapas activas no se conserva necesariamente en el franqueo de una transición, sino que depende del número de etapas de entrada y salida de dicha transición. En la figura, la transición,

que no tiene receptividad y por tanto de franqueará en el momento que esté validada, tiene dos etapas de entrada y una de salida.

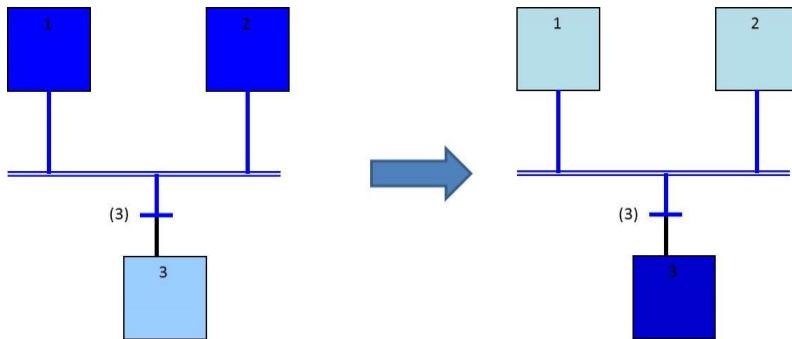


Figura 11-6 – Reglas de evolución (II)

La Figura 11-7 muestra el caso de la regla número 5. En la situación inicial tanto la transición 1 como la 2 son franqueables. El franqueo de la transición 1 implica la activación de la etapa 2, pero el franqueo de la etapa 2 implica la desactivación de la misma etapa 2, que por tanto simultáneamente se le va a dar una orden de activación y otra contradictoria de desactivación. Por la regla número 5, la activación es prioritaria por lo que, tal como se muestra en la figura, la etapa queda activada.

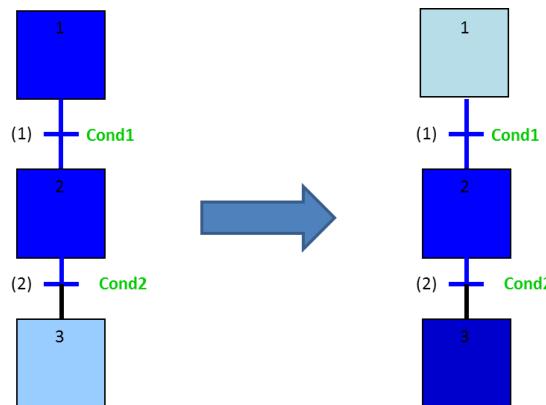


Figura 11-7 – Reglas de evolución 5

## 11.2. Estructuras básicas en GRAFCET

El modelado de un proceso concreto se realizará mediante el conexionado de las etapas y transiciones. En general, cada etapa se corresponderá con una parte del proceso en el que se realizarán una serie de actuaciones determinadas. El modo de conectar estas etapas servirá para determinar si estas partes del proceso se ejecutan secuencialmente, en paralelo, de forma sincronizada, etc.

Hay una serie de estructuras básicas que se utilizan de forma habitual y que se pasarán a describir en esta sección.

### 11.2.1. Secuencia única

Permite modelar procesos secuenciales, en que los distintos pasos se realizan uno detrás de otro de manera que se pasa de uno al siguiente cuando se cumple una determinada condición. La estructura se muestra en Figura 11-8

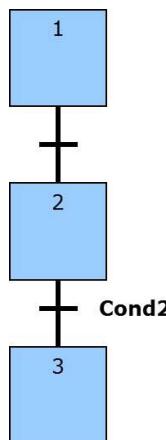


Figura 11-8 – Estructura secuencial

### 11.2.2. Secuencia simultanea

Se utiliza para modelar procesos que se ejecutan en paralelo. En la Figura 11-9, si la etapa 1 está activa y se cumple *Cond1* se activarán las etapas 2 y 3, con lo que empezarán a ejecutarse en paralelo los procesos asociados a cada una de las ramas. También se le conoce como estructura AND.

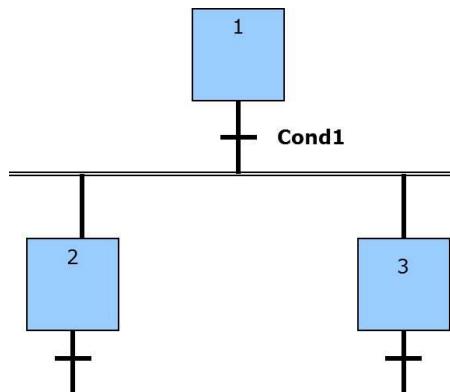


Figura 11-9 – Estructura de secuencia simultanea

### 11.2.3. Selección de secuencia

Con la estructura de selección de secuencia se podrá elegir entre la realización de un proceso u otro de acuerdo a que se verifiquen determinadas condiciones. En el caso de la Figura 11-10, si la etapa 1 está activa, se activarán las etapas 2 y/o la 3 dependiendo de las condiciones *Cond1* y *Cond2*.

Si sólo una de las condiciones es cierta, solo se ejecuta una de las secuencias. Si son ciertas las dos condiciones, se ejecutan las dos secuencias. Esta última situación se denomina *parallelismo interpretado* y no es muy recomendable, ya que es habitual que den lugar a comportamientos no deseados al final de cada una de las ramas.

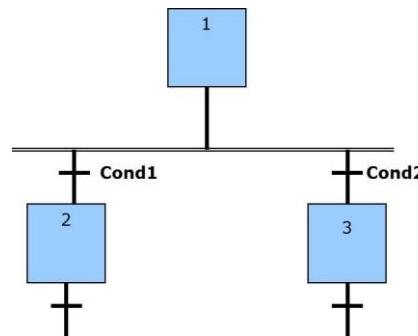


Figura 11-10 – Estructura de selección de secuencia

Por tanto es muy recomendable obligar a que sólo una de las condiciones sea cierta simultáneamente. Es lo que se denomina *selección exclusiva*, que se puede realizar de varias maneras:

- **Por orden físico:** El propio proceso lo impide por ser físicamente imposible que se verifiquen simultáneamente las dos condiciones. Este sería el caso en el que una de las ramas estuviera asociada a la activación de un determinado sensor mientras que la otra estuviera asociada a que dicho sensor no estuviera activado.
- **Por orden lógico:** Se consigue la exclusividad mediante la negación de la condición de la otra rama. Por ejemplo, tal como se ve en la Figura 11-11 , si se pretende que vaya por una rama cuando se produzca la condición  $A$  y por la otra cuando se dé la condición  $B$ , se utilizarán las siguientes condiciones lógicas

$$(T1): A \cdot \bar{B}$$

$$(T2): B \cdot \bar{A}$$

De esta manera sólo se puede dar una de ellas. Nótese que si simultáneamente  $A$  y  $B$  fueran ciertas, no se ejecutaría ninguna de las dos ramas.

- **Por prioridad:** Consiste en darle prioridad a alguna de las dos condiciones:

$$(T1): A$$

$$(T2): B \cdot \bar{A}$$

De esta manera no se pueden dar las dos condiciones simultáneamente, pero en el caso que  $A$  y  $B$  fueron ciertas a la vez, se ejecutaría la rama con prioridad, en este caso la  $A$ .

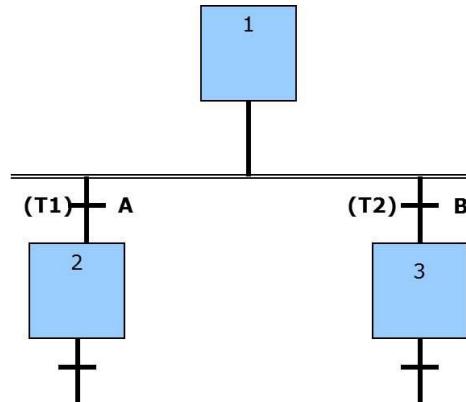


Figura 11-11 – Selección exclusiva

#### 11.2.4. Salto de etapa

La estructura de salto de etapa se muestra en la Figura 11-12. Dependiendo del valor de la condición  $A$ , si la etapa 1 está activa, se activaría o bien la etapa 2 ( $A$  TRUE) o bien la etapa 3 ( $A$  FALSE). Es decir, dependiendo del valor de la condición, se saltaría una parte del proceso

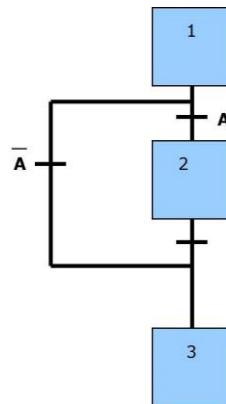


Figura 11-12 – Estructura de salto de etapa

### 11.2.5. Repetición de secuencia

Esta estructura se muestra en la Figura 11-13. Si la etapa 2 está activa, dependiendo del valor de la condición  $A$ , o bien se continuará con la etapa 3 ( $A$  TRUE) o bien, se repetirán las etapas 1 y 2 ( $A$  FALSE).

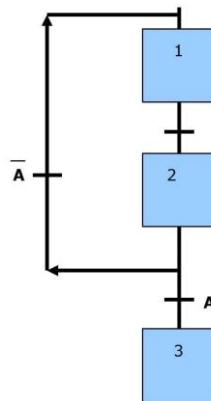


Figura 11-13 – Repetición de secuencia

### 11.3. Acciones en GRAFCET

Cada etapa puede tener asociada una serie de acciones que se ejecutan cuando dicha etapa está activa. Estas acciones normalmente pueden ser de dos tipos:

- Activación de variables, habitualmente señales de salida del automatismo lógico.
- Ejecución de un código, que puede estar escrito en cualquiera de los lenguajes de programación que admite la norma, incluido el propio SFC.

La forma en que se ejecuten las acciones va a depender del tipo de acción, tipos que se describen más adelante en esta sección.

En GRAFCET, las acciones asociadas a una etapa se representan por un recuadro colocado a la derecha de la etapa (Ver Figura 11-14).

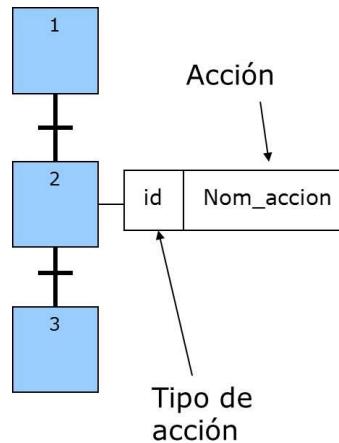


Figura 11-14 – Representación de acciones en GRAFCET

Los tipos de acciones más habituales y su identificador según la norma IEC-61131 son:

- **N:** Acción continua. La acción está activa mientras lo está la etapa. Es decir,

si la acción es una variable booleana toma el valor TRUE mientras la etapa está activa, y si se trata de código, se ejecutará mientras la etapa esté activa. Es el funcionamiento normal.

- **S:** Set. La acción se activa y permanece ejecutándose hasta un Reset. Es decir, la acción continua activa independientemente de que la etapa que la habilitó deje de estar activa. Sólo se detendrá cuando en otra etapa se ejecute un Reset de esa misma acción.
- **R:** Reset. La acción es desactivada. Como se indicó anteriormente funciona en coordinación con activación de acciones de tipo Set.
- **L:** Duración limitada. La acción se activa durante un tiempo especificado por el usuario, y como máximo el tiempo de activación de la etapa. Es decir, empieza a ejecutarse en el momento de la activación de la etapa y termina con lo que ocurra antes entre la desactivación de la etapa o el fin del tiempo especificado.
- **D:** Retardada. También es necesario suministrar un parámetro de tiempo. La acción se activa un tiempo después de la activación de la etapa y a partir de ese momento permanece activa mientras lo está la etapa.
- **P Pulso.** La acción se ejecuta una sola vez cuando se activa la etapa.

## 11.4. GRAFCET en la norma IEC-1161-3

GRAFCET es un lenguaje básicamente gráfico, aunque la norma IEC permite tanto su representación gráfica como textual. En cualquier caso, es preferible la representación gráfica porque da una idea más intuitiva de la parte del proceso que se está ejecutando. En este libro se hará referencia exclusivamente a la representación gráfica.

A continuación se detallará la forma en que la norma permite implementar los elementos fundamentales de un SFC: las etapas, transiciones y las acciones asociadas a etapas.

### 11.4.1. Etapas

Las etapas se representan de acuerdo a la norma mediante un rectángulo. Si tiene asociada acciones, a la derecha se representa otro rectángulo unido a la etapa donde se indica la acción. Este segundo rectángulo se omite si no hay acciones asociadas. En la Figura 11-15 se muestra una etapa con acción.

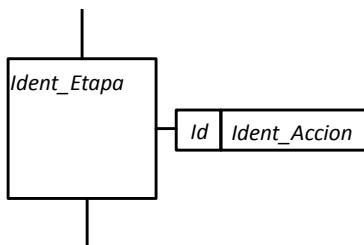


Figura 11-15 – Representación de una etapa con acción

Cada etapa se identifica obligatoriamente con un identificador, que debe cumplir las mismas normas que un nombre de variable. Este nombre debe ser único, es decir, no debe haber otras etapas o variables con el mismo nombre. La identificación de las acciones se analizará en una sección posterior.

Cada vez que se crea una etapa con su correspondiente identificador, automáticamente se crean dos variables asociadas a ella cuyos valores se van actualizando automáticamente en función de la evolución del GRAFCET. Son variables que no es necesario declararlas, que se pueden leer y utilizar sus valores, pero a las que no se les deben asignar valores por el usuario. Suponiendo que se ha definido una etapa con identificación *Ident\_Etapa*, se crean las siguientes variables:

- *Ident\_Etapa.X*: Variable booleana que indica el estado de activación de la etapa. Si la etapa está activa, la variable está a TRUE y si no está activa la variable está a FALSE.
- *Ident\_Etapa.T*: Variable de tipo TIME. En cada momento muestra el tiempo que ha transcurrido desde la activación de la etapa. Si nunca ha estado activa, el tiempo que muestra es cero. En caso contrario, cuando no está activa, muestra

el tiempo en que estuvo activa durante su última activación.

Tanto el identificador de la etapa como sus correspondientes variables *.X* y *.T* son variables locales al POU donde se utilizan.

### 11.4.2. Transiciones

Las condiciones asociadas a las transiciones deben ser expresiones booleanas. Las transiciones se pueden especificar de varias maneras, aunque no todas están permitidas en todos los entornos de desarrollo:

- La condición lógica se escribe directamente junto a la transición, escrita en lenguaje ST, LD o FBD. CoDeSys sólo permite escribirla en ST, mientras que por ejemplo, el entorno UnityPro de Schneider permite también las otras posibilidades. En la Figura 11-16 se muestra un ejemplo.

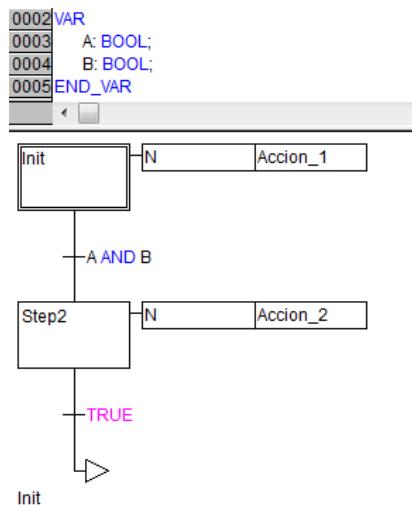


Figura 11-16 – Condición lógica escrita en ST en una transición

- Se utiliza un identificador de transición que hace referencia a un código escrito en LD, FBD, ST o IL. En la figura se muestra un identificador *Trans0* y el código

escrito en lenguaje LD. Nótese que en CoDeSys, cuando se utiliza esta forma de indicar condiciones lógicas, aparece un pequeño triángulo negro en la transición. Del mismo modo, cuando se escribe el programa no se realiza la asignación de la condición lógica a ninguna variable (o bobina en LD).

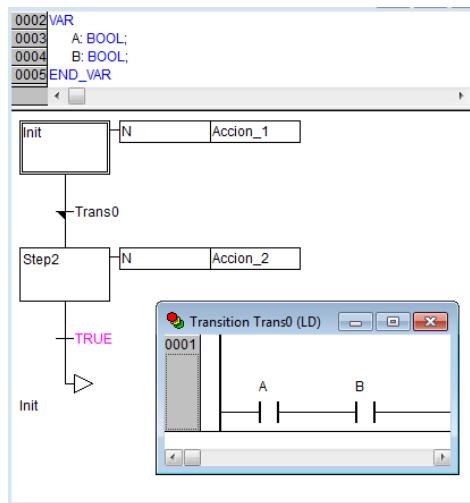


Figura 11-17 – Condición lógica asociada al identificador *Trans0* escrita en LD en una transición

### 11.4.3. Acciones

Cada etapa puede tener o no tener acciones, y si las tiene, puede tener una o varias. Cada una de estas acciones puede consistir en:

- Una asignación a una variable booleana
- Una secuencia de instrucciones programadas en IL, ST, LD, FBD o SFC.

En el primer caso, en una acción tipo N, la acción es simplemente el nombre de la variable booleana, de modo que ésta estará a TRUE si la etapa está activa y a FALSE si la etapa está inactiva. Véase el funcionamiento en *Cap11\_1.pro*.

Si la acción no es de tipo N el funcionamiento es distinto. Por ejemplo, en *Cap11\_2.pro* (Figura 11-18) se muestra el uso de las acciones S y R para dar valor a una variable booleana. La variable *Salida* estará a TRUE mientras las etapas *Step2* y *Step3* están activas.

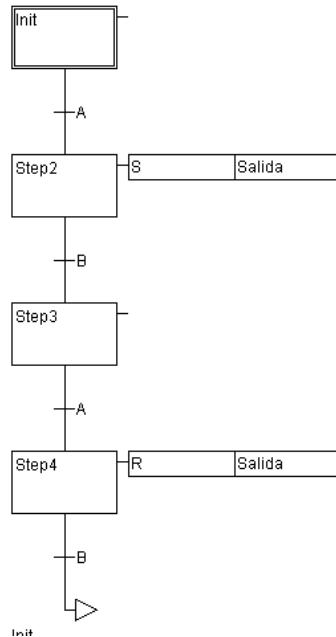


Figura 11-18 – Acciones de tipo S y R

En el caso de que la acción sea un código, cuando una etapa está activa, las instrucciones de la acción de dicha etapa se ejecutan. Recuérdese que el modo de funcionamiento de un programa escrito en SFC es igual que en cualquier otro lenguaje, es decir, el código se ejecuta cíclicamente. En el caso de SFC en cada ciclo del autómata se realiza lo siguiente:

- Si fuera necesario, se realiza la actualización del marcado del GRAFCET de acuerdo al estado actual del marcado, las condiciones lógicas y las reglas de evolución del GRAFCET. Nótese que habrá ciclos (muchos) donde no se cumplan condiciones para el cambio del marcado y el marcado de GRAFCET permanece igual.

- Ejecutar las instrucciones de las acciones de las etapas activas en cada ciclo.

Por tanto, téngase en cuenta que normalmente las etapas van a estar activas durante un número de ciclos considerables y por tanto cada acción se va a ejecutar mucha veces de forma cíclica.

Si en una acción hay bloques funcionales (por ejemplo, contadores y temporizadores), cuando la acción no se ejecuta, sus valores no se van actualizando, pero el bloque funcional no desaparece y los valores internos o salidas no cambian por el hecho de que no se está ejecutando.

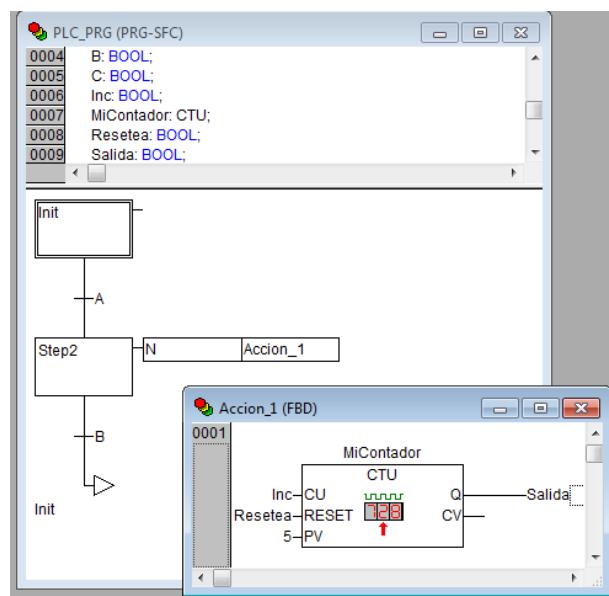


Figura 11-19 – La acción asociada es código en FBD que consiste en llamada a un contador

En el archivo *Cap11\_3.pro* se encuentra el programa simple de la Figura 11-19. Se trata de un GRAFCET con dos etapas y en la segunda se ha definido una acción. La evolución del GRAFCET se hace mediante las entradas *A* y *B* mientras que el control del contador se hace con las entradas *Inc* y *Resetea*. El valor de salida del contador se almacena en la variable *Salida*.

Compruebe que mientras está activa la etapa *Init* el contador no responde a sus señales de entrada. Si pulsamos *A* y activamos la etapa *Step2* el contador sí responde a las entradas y va actualizando el valor interno de contador y la variable *Salida*. Si mediante *B* desactivamos de nuevo *Step2*, el contador de nuevo deja de responder, pero el valor interno y los de salida no cambian.

El fichero *Cap11\_4.pro* es el mismo programa pero la acción incluye un temporizador en vez de un contador (Figura 11-20). Compruébese el comportamiento y verá que cuando se desactiva la etapa *Step2* mientras se está temporizando, las variables internas del temporizador se detienen, pero cuando lo volvamos a activar comprobamos que las variables internas se actualizan ¡sumándole el tiempo en que la etapa no ha estado activa!

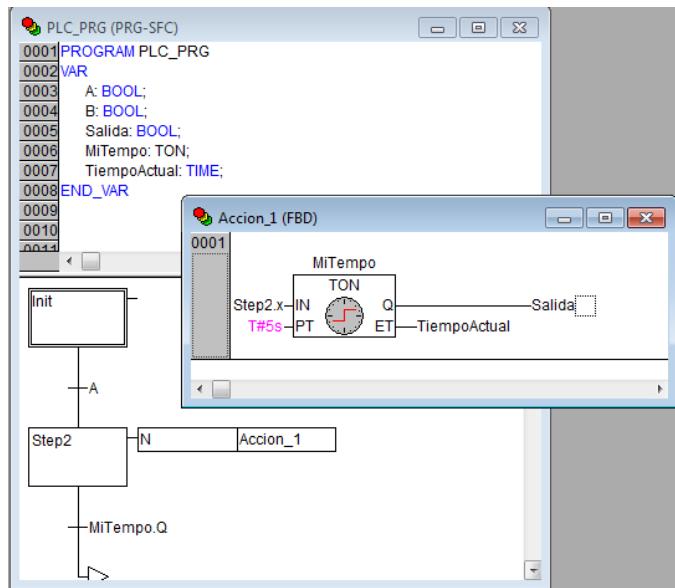


Figura 11-20 – La acción asociada es código en FBD que consiste en llamada a un temporizador

En *Cap11\_5.pro* se cambia el programa para que la etapa *Step2* se mantenga activa durante 5 segundos. Nótese que la IN del temporizador está asociada a la variable

*Step2.x* y que la condición de la transición de salida es la propia salida del temporizador. ¿Funcionaría si la entrada IN del temporizador se pone a TRUE, con lo que siempre estará a este valor? Pruébelo experimentalmente y explique el funcionamiento observado.

En relación a cómo se ejecutan las acciones, como regla general:

- Las acciones asociadas a la etapa se ejecutan al menos una vez desde el momento en que la etapa se ha activado.
- Después de la desactivación de una etapa, las acciones de dicha etapa se ejecutan una vez más para asegurar que los temporizadores terminen, se resetean variables, etc. En el ejemplo anterior, se ha ejecutado una vez el bloque funcional TON una vez desactivada *Step2*, haciendo que IN (es decir *Step2.x*) fuera FALSE y por tanto se resetee el temporizador.

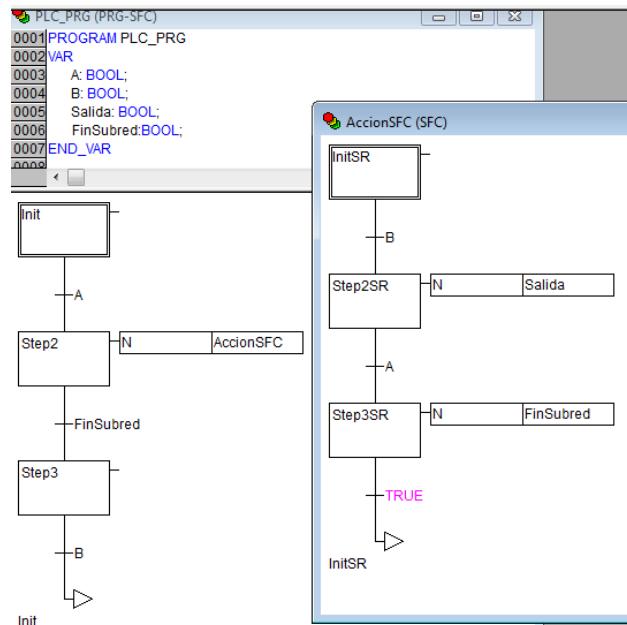


Figura 11-21 – Acción asociada escrita en SFC

Como se indicó anteriormente, una acción también puede ser un programa en SFC,

dando lugar a una estructura jerárquica de los GRAFCET. En *Cap11\_6.pro* (ver Figura 11-21) se muestra un ejemplo simple. Normalmente, es necesario sincronizar la terminación de la subred con la desactivación de la etapa del GRAFCET principal que lo activó. En el ejemplo se hace mediante la variable *FinSubRed*. Cuando en la última etapa de la subred la variable *FinSubRed* se pone a TRUE, activa la condición de la transición de salida de *Step2* y por tanto dicha etapa se desactiva.

## 11.5. GRAFCET en CoDeSys

En esta sección se describe cómo se crea y edita un GRAFCET utilizando el entorno de programación de CoDeSys.

En primer lugar indicar que CoDeSys permite la edición de GRAFCET de acuerdo a la norma IEC, pero también permite usar lo que se denominan “etapas simplificadas” cuya sintaxis queda fuera de la citada norma. En este libro, nos centraremos en la norma IEC. Por tanto, antes de comenzar la edición es necesario definir qué tipo de GRAFCET vamos a utilizar. Para utilizar la norma IEC, es necesario que la librería *Iecsfclib* esté incluida en el proyecto (Ver Apéndice A para incluirla si no lo está). Seguidamente será necesario activar el flag “*Use IEC-Steps*”. Desde el menú *Extras*:

*Extras -> Use IEC Steps*

### 11.5.1. Edición de un GRAFCET

CoDeSys dispone de un editor gráfico para la implementación de programas en SFC. Se trata de un editor un tanto “rígido” que no está basado en la libre colocación de los distintos elementos (etapas, transiciones y arcos) sino en una serie de comandos que van añadiendo elementos a la red, con lo que se consigue que la red siempre tenga una sintaxis válida. Como en los lenguajes anteriores, los comandos están en la línea de botones del menú de herramientas o en el menú contextual (botón derecho del ratón) al que se accede cuando está abierta la ventana de edición.

Al crear un nuevo SFC, automáticamente se crea una red simple compuesta por una etapa inicial (*Init*) una transición y un salto al estado inicial. A partir de esta

red básica se añadirán elementos (Ver Figura 11-22).

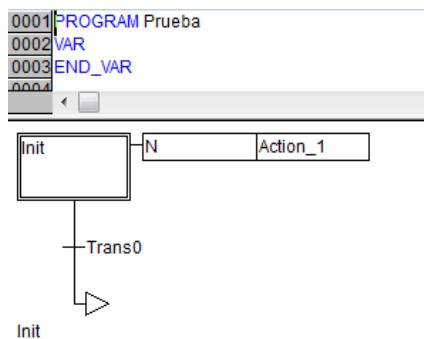


Figura 11-22 – SFC inicial en CoDeSys

Un GRAFCET en CoDeSys sólo podrá tener una etapa inicial (la que crea automáticamente) y que está marcada con un doble recuadro. A dicha etapa se le puede cambiar el nombre.

Para añadir nuevos elementos es necesario marcar con el cursor la posición donde se incluirán, para lo cual es necesario seleccionar previamente uno o varios elementos. Los elementos se seleccionan pulsándolos con el ratón, apareciendo un recuadro con línea discontinua alrededor del elemento o los elementos marcados. Para seleccionar más de uno, añadir nuevos elementos con *Cntr-Ratón Izq.*

Hay que tener en cuenta que no se va a permitir realizar acciones que lleven a un GRAFCET no válido. Por ejemplo, si seleccionamos una etapa y se intenta eliminar, no se podrá realizar la operación (daría lugar a dos transiciones conectadas). Tendríamos que eliminar, como mínimo, una etapa y una transición. Tampoco se podrá añadir una sola etapa o transición, siempre será necesario añadirlas de forma conjunta. En general, si se intenta realizar una operación que dé lugar a un GRAFCET no válido, el sistema no lo realizará y emitirá un aviso sonoro.

Cada vez que se crea una etapa o transición se le asigna automáticamente un nombre. Dichos nombres se pueden cambiar sin más que pulsar con el ratón sobre el nombre y cambiarlo. Recuérdese que en cada POU el nombre de cada etapa debe ser único y no coincidir con otros nombres de variables o POUs,

### 11.5.2. Comandos básicos para edición del GRAFCET

En este apartado se describen los principales comandos. El siguiente conjunto de comandos se podrá ejecutar pulsando el botón de comando, desde el menú contextual o desde el menú principal *Insert*. La Tabla 11-1 indica los botones del menú de herramientas para cada acción.

Tabla 11-1. Edición de un grafo SFC

Acción	Menú contextual	Botón
Inserta etapa y transición antes del cursor	<i>Step-Transition (before)</i>	
Inserta etapa y transición después del cursor	<i>Step-Transition (after)</i>	
Insertar rama para selección de secuencias a la derecha	<i>Alternative branch (right)</i>	
Insertar rama para selección de secuencias a la izquierda	<i>Alternative branch (left)</i>	
Insertar rama para secuencias simultaneas a la derecha	<i>Parallel branch (right)</i>	
Insertar rama para secuencias simultaneas a la izquierda	<i>Parallel branch (left)</i>	
Insertar salto	<i>Jump</i>	
Insertar Transición y salto	<i>Transition-Jump</i>	

#### 11.5.2.1. Step-Transition (before)

Inserta una etapa y una transición antes del bloque seleccionado.

### **11.5.2.2. Step-Transition (after)**

Inserta una etapa y una transición detrás del bloque seleccionado

### **11.5.2.3. Alternative Branch (right)**

Inserta una rama paralela a la derecha de los elementos seleccionados que permite implementar una selección de secuencias (ver sección 11.2.3). La rama insertada va a contener exclusivamente una transición. Esto determina qué elementos hay que tener seleccionados para realizar la acción, por ejemplo, una única transición o una transición-etapa-transición. Así, si seleccionamos una única etapa no se podrá realizar esta operación ya que daría lugar a un GRAFCET no válido.

### **11.5.2.4. Alternative Branch (left)**

Igual que el comando anterior pero la rama se inserta a la izquierda de la selección.

### **11.5.2.5. Parallel Branch (right)**

Inserta una rama paralela a la izquierda de los elementos seleccionados que permite implementar una secuencia simultánea (ver sección 11.2.2). En este caso incluirá una rama con una sola etapa, por lo que la selección debe ser o una sola etapa, o etapa-transición-etapa, etc.

### **11.5.2.6. Parallel Branch (left)**

Lo mismo que la anterior pero a la izquierda de la selección.

### **11.5.2.7. Jump**

Se utiliza para implementar saltos de un lugar del GRAFCET hacia una etapa. El salto debe ir etiquetado con el nombre de la etapa a la que se desea saltar. De nuevo la red resultante debe ser válida, por lo que sólo se podrá realizar si el bloque seleccionado termina en una transición. Tampoco se podrá realizar desde una transición situada en medio de una secuencia (dejaría desconectada la parte inferior), por lo que sólo se puede hacer con la última transición de una rama.

### **11.5.2.8. Transition-Jump**

Similar a la anterior, pero inserta una transición seguida del salto. Por tanto, solo se podrá realizar si se selecciona una etapa al final de una rama.

### 11.5.2.9. Paste Parallel Branch

En primer lugar se selecciona el trozo de red que se desea copiar (Ctrl-C), a continuación se selecciona el trozo de red junto al que se va a crear una nueva rama paralela con el contenido copiado y finalmente se selecciona esta opción.

### 11.5.2.10. Add name to Parallel Branch

Permite asignar un nombre a una rama, con lo que es posible hacer un salto que dé comienzo a varios procesos en paralelo, tal como se muestra en la Figura 11-23, de forma que cuando se dispara *Trans0* comenzarán en paralelo los procesos de *Step2* y *Step3*. El nombre de la rama se le puede asignar en el momento de crearla. Para hacerlo posteriormente es necesario marcar todas las ramas en paralelo.

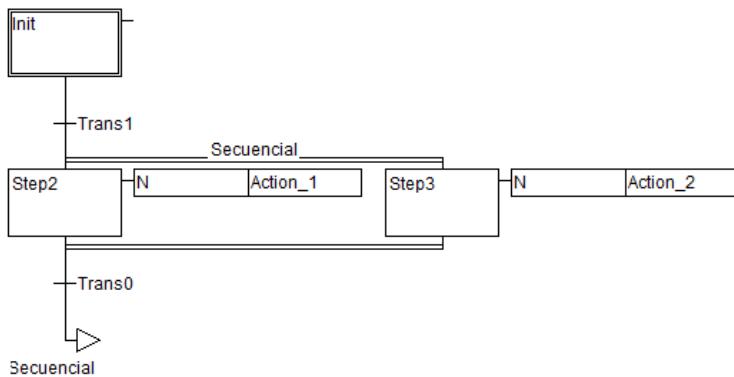


Figura 11-23 – Edición SFC. Nombre asociado a ramas en paralelo

### 11.5.3. Edición de transiciones

En CoDeSys todas las transiciones deben tener una condición lógica asociada, de forma que incluso cuando se desee que la condición lógica siempre se verifique, será necesario poner expresamente la constante TRUE.

Como se indicó anteriormente, existen dos formas de asociar condiciones lógicas a una transición:

- Escribir directamente en ST la condición junto a la transición, por ejemplo  
 $(A > 20) \text{ AND } (B < 10)$   
 Donde A y B pueden ser variables de tipo entero.
- Escribir la condición lógica en lenguaje LD, IL, ST o FBD. Para ello se asignará un identificador a la transición y se escribirá el código en el lenguaje elegido.

Para esta segunda opción, se escribirá junto a la transición el identificador que se le deseé asignar a la transición. Nótese que cuando se crea una transición automáticamente se le asigna un nombre, aunque este nombre se puede cambiar sin más que editar el nombre. Al escribir un nombre, y dependiendo de las opciones establecidas, es posible que aparezca la ayuda a la declaración de variable. Si es el caso, el identificador de transición **no** se declara como variable, y por tanto no se usará la citada ayuda.

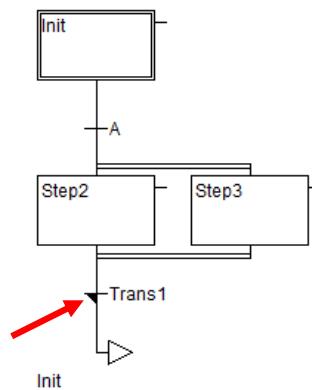


Figura 11-24 – Símbolo de transición cuando tiene asociada una condición

Una vez seleccionada la transición, se escoge la opción *Zoom Action/Transition* (en el menú contextual del botón derecho o en el menú *Extras*). Se selecciona el lenguaje requerido y se edita la condición en la correspondiente ventana.

Una vez editada y cerrada la ventana de edición se comprueba que el ícono de la transición ha cambiado (triángulo negro en Figura 11-24).

Sea cual sea el lenguaje elegido, nunca se deberá realizar una asignación en la

declaración de la transición (bobina en LD, instrucción ST en lenguaje IL, o el operador ‘:=’ en lenguaje ST). En el caso de LD o FBD sólo se puede utilizar una única red (Network).

En cualquier momento se puede volver a editar la condición lógica sin más que volver a seleccionar la opción *Zoom Action/Transition*.

Para eliminar la condición lógica se seleccionará *Clear Action/Transition*.

#### 11.5.4. Edición de acciones

Como se ha indicado anteriormente, las acciones pueden ser o una simple variable booleana o una lista de instrucciones que se ejecuta cuando la etapa esté activa.

En este último caso, antes de asociar una acción a una etapa es necesario asignar a un identificador de acción un código escrito en cualquiera de los lenguajes. Al contrario que las condiciones lógicas de las transiciones, las acciones de este tipo no están asociadas a una etapa concreta, sino al POU, de modo que la misma acción se puede utilizar en varias etapas distintas.

Para crear una acción, se selecciona en el árbol de POUs (ventana izquierda) el POU tipo SFC al que se quiere asignar la acción, y con el botón derecho del ratón se accede al menú contextual, seleccionando la opción *Add Action* (Figura 11-25). Seguidamente se selecciona el lenguaje en el que se escribirá y el nombre de la acción apareciendo una ventana de edición desde la que se escribirá la acción. En la ventana de POUs aparecerán asociadas a cada POU de tipo SFC todas sus acciones (Figura 11-26).

Una vez que las acciones están creadas se pueden asociar a la etapa o etapas que se requiera. Para esto, después de seleccionar la etapa correspondiente, se usa la opción *Associate Action* (menú contextual con el botón ratón derecho o Menú Extras). Se deberá escoger el tipo de acción (N,S,R,...) y la variable booleana o el identificador de la acción que se le quiera asignar. A una misma etapa se pueden asignar hasta 9 acciones distintas.

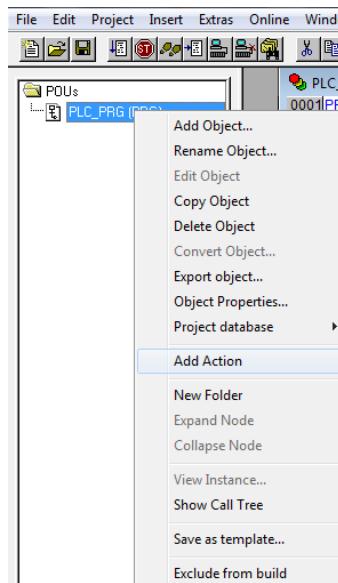


Figura 11-25 – Añadir acción a un POU tipo SFC

Para eliminar la asociación de una etapa con una acción se utiliza la opción *Clear Action/Transition*. Si hay varias, aparecerá una ventana que permite escoger cuál es la que se desea eliminar. Téngase en cuenta que por defecto, al crear una etapa, se hace con una acción asociada con un identificador no definido. Si no se va a utilizar acciones en dicha etapa, será necesario eliminarla.

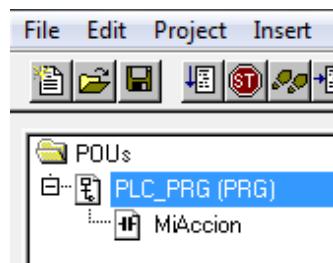


Figura 11-26 – Acciones asociadas a un POU tipo SFC

## 11.6. Ejercicios

### 11.6.1. Ejercicio 1

Diseñar un sistema de control en GRAFCET para un semáforo. El semáforo se activa con un pulsador *M* y se debe desactivar después de que se hayan ejecutado un determinado número de ciclos verde-amarillo-rojo. Los tiempos que están encendidas cada una de las luces son en general distintos. El sistema a controlar está implementado en *Cap11\_7.pro*. Para contabilizar el número de ciclos del semáforo se usará el contador CTU ya definido CONTA, con objeto de que su valor aparezca en la visualización.

#### Solución

En el fichero *Cap11\_7.Res.pro* se encuentra implementada una solución al problema.

En el programa hay implementadas dos acciones (*contadores* y *temporizadores*) que gestionan respectivamente el contador y los temporizadores (tres). Las dos acciones son de tipo S, están implementadas en IL y están asociadas a la etapa inicial. Esto significa que estas acciones estarán siempre ejecutándose durante toda la ejecución del programa.

En el caso del contador (ver acción *contadores*), su señal de RESET está controlada por *Init.x* de modo que cada vez que se active el estado Inicial se reiniciará. Debe contar una vez por ciclo, por lo que la señal CU se ha asociado a *SepVerde.x*, es decir contará cuando *StepVerde* pase de desactivado a estado activado. También se podría haber asignado a las etapas asociadas al amarillo o al rojo.

La acción *temporizadores* gestiona los tres temporizadores. Se usan dos posibles formas de pasar parámetros, o bien en la misma línea de la llamada (rojo y amarillo) o bien asignándole los valores antes o después de la llamada y llamar al bloque funcional TON sin argumentos. Las dos formas son equivalentes.

El control de cada temporizador (entrada IN) se realiza con la variable asociada a su etapa correspondiente (*StepVerde.x* para el temporizador de luz verde,...).

### 11.6.2. Ejercicio 2

Modifique el ejemplo anterior para que:

- En lugar de una única acción para los temporizadores asociada a la etapa inicial, haya tres acciones para cada una de las luces y que cada una de estas acciones esté asociada a su correspondiente etapa.
- El semáforo se podrá desactivar de dos maneras, la primera como en el ejercicio anterior, es decir, cuando ocurran un determinado número de ciclos, y en segundo lugar mediante el pulsador *P*. Cuando se desactive mediante *P* el contador debe reiniciarse, y por tanto la señal de RESET del contador es necesario cambiarla. El pulsador *P* se podrá utilizar en cualquier momento.

Utilice el fichero *Cap11\_8.pro* para desarrollar el ejercicio.

### 11.6.3. Ejercicio 3

Mismo enunciado que en el ejercicio 2, pero en esta caso, al pulsar *P* el sistema no se desconecta inmediatamente sino al final del ciclo verde-amarillo-rojo que se esté ejecutando. Es decir, si se pulsa *P* mientras está encendida la luz amarilla, el sistema no se desactiva hasta que termine el tiempo de encendido de la luz roja.

### 11.6.4. Ejercicio 4

Se desea realizar el movimiento de un carrito (*Cap11\_9.pro*) desde la posición marcada por el sensor de presencia *A* hasta el sensor *B* y vuelta a *A*. El movimiento comenzará cuando el interruptor START esté a TRUE, de modo que si cuando el carrito vuelve a *A* el interruptor sigue a TRUE, continuará con el movimiento de vaivén. El carrito se mueve con dos señales de salida que permiten mover el motor en las dos direcciones (*MOTOR\_IZ* y *MOTOR\_DE*). Implementar el programa en GRAFCET en el POU *Automatismo*.

### 11.6.5. Ejercicio 5

Diseñar un sistema de control para dos carritos que operan de forma sincronizada (*Cap11\_10.pro*). Cada uno de los carritos se deberá mover entre los sensores de presencia *A1* y *B1* (el primer carrito) y entre *A2* y *B2* (el segundo). El movimiento sincronizado comienza con el interruptor general *START*. Cada uno de los dos carritos tiene dos señales de salida que permiten moverse en las dos direcciones (*MOTOR\_I Zi* y *MOTOR\_DEi*, con  $i=1,2$ ).

El funcionamiento es el siguiente. Cuando *START* se pone a TRUE comienza el movimiento de los dos carritos hacia la izquierda (sus velocidades en general podrán ser distintas). Una vez que llegan a *B*, no comenzarán el movimiento de retorno a *A* hasta que los dos carritos hayan llegado, es decir, el carrito más rápido espera en *B* hasta que llegue el otro antes de reanudar su vuelta a *A*. En *A* de nuevo se volverán a sincronizar, es decir, hasta que no están los dos en *A*, no podrán a moverse hacia *B*. El programa en GRAFCET se implementará en el POU *Automatismo*. Comprobar que el programa funciona adecuadamente, independientemente de cuál de los carritos sea más rápido.

### 11.6.6. Ejercicio 6

Un móvil (recuadro verde) se puede mover en un recinto ABCD (Ver Figura 11-27). El móvil tiene 4 salidas que permiten realizar el movimiento en las cuatro direcciones (*Arriba*, *Abajo*, *Izquierda* y *Derecha*) y sobre el móvil hay colocada una luz (*Luz*). Para detectar que el móvil ha llegado a cualquiera de los cuatro lados del rectángulo ABCD, se dispone de sensores (*X1*, *X2*, *Y1* e *Y2*) que se activan cuando el móvil toca en cualquier punto del lado correspondiente. Además se dispone de un pulsador *M* para lanzar el proceso. El sistema debe operar de la siguiente manera:

Al pulsar *M*, el móvil debe hacer el recorrido A-C-D-A y durante el trayecto C-D-A la luz del móvil debe estar encendida. En general, las velocidades de movimiento vertical y horizontal son distintas (se pueden ajustar manualmente con los selectores situados a la izquierda de la figura), por lo que el movimiento en diagonal A-C normalmente llegará primero a una de las paredes *X2* o *Y2*. En ese caso, se continuará el movimiento junto a la pared hasta llegar a C.

El programa se implementará en el POU en GRAFCET que ya está creado en el fichero *Cap11\_11.pro*.

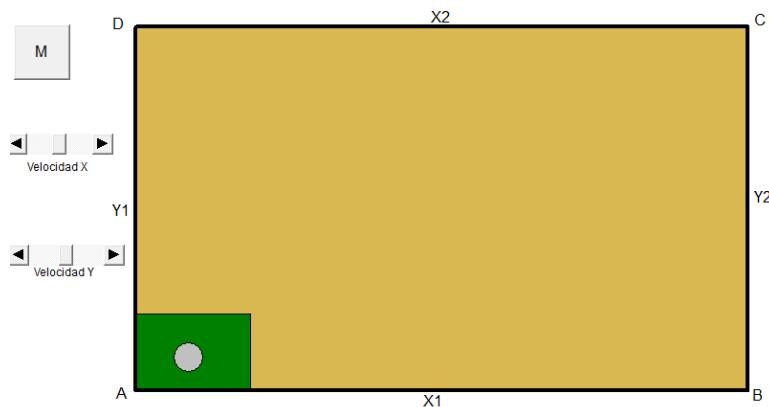


Figura 11-27 – Visualización del ejercicio 6

### 11.6.7. Ejercicio 7

El enunciado es el mismo que en el apartado anterior. La diferencia está en que debido a desajuste en los motores, el móvil tiene tendencia a separarse de los bordes en el recorrido C-D y en el D-A. Las perturbaciones las simularemos pulsando los botones *Pert* que harán que el móvil se separe de los laterales correspondientes. El objetivo es el mismo que antes, pero cuando aparece una perturbación el controlador debe volver a pegar el móvil a la pared. Se trata de que a pesar de las perturbaciones, el móvil esté el mayor tiempo posible pegado a las paredes X2 y Y1 en los movimientos C-D y D-A. Para la realización del ejercicio utilice *Cap11\_12.pro*.

### 11.6.8. Ejercicio 8

Se desea controlar el sistema de pesaje de la Figura 11-28 [1]. El sistema a controlar está implementado en *Cap11\_13.pro*. El sistema consta de un depósito superior con dos compuertas que se abren activando las señales S1 y S2 respectivamente.

Por otro lado el producto que cae del depósito, lo hace sobre una báscula, que proporciona dos señales discretas  $L1$  y  $L2$  cuando se alcanzan dos pesos determinados ( $L2$  mayor que  $L1$ ). Finalmente hay un pistón con retorno automático para vaciar el producto de la báscula. Este se acciona con la orden  $S3$  y tiene un fin de carrera  $FC$  cuando el cilindro está extendido

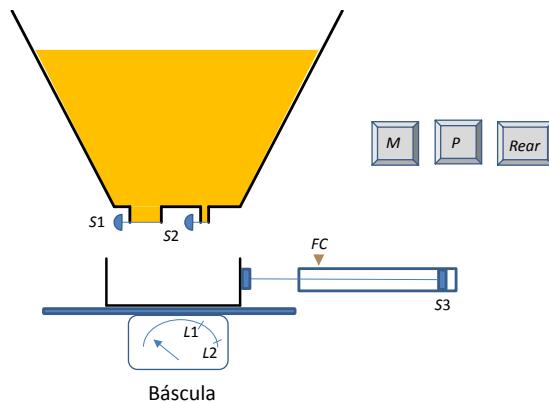


Figura 11-28 – Sistema de pesaje

Una pulsación en  $M$  deberá provocar la apertura de las dos compuertas. Cuando la aguja del peso llegue a  $L1$ , debe cerrarse la compuerta  $S1$ . Cuando la aguja llegue a  $L2$ , se cerrará la compuerta de afinado  $S2$ . En ese momento se deberá proceder al vaciado utilizando el pistón.

Al accionar el pulsador de emergencia  $P$ , se cerrarán las dos compuertas en cualquier momento del ciclo y se parará éste. Para reanudar el proceso se pulsará el botón de rearne ( $Rear$ ), continuando el proceso en la fase donde se suspendió.

El programa se escribirá en el POU *bascula*.

### 11.6.9. Ejercicio 9

Se pide diseñar un sistema de control en GRAFCET para el sistema de la Figura

11-29 [1]. El móvil deberá realizar un movimiento de vaivén continuado desde el momento que se pulsa la orden de puesta en marcha (*M*). Los límites del movimiento están marcados por dos fines de carrera *A* y *B*. El motor se mueve mediante las órdenes *MOTOR\_IZ* y *MOTOR\_DE*.

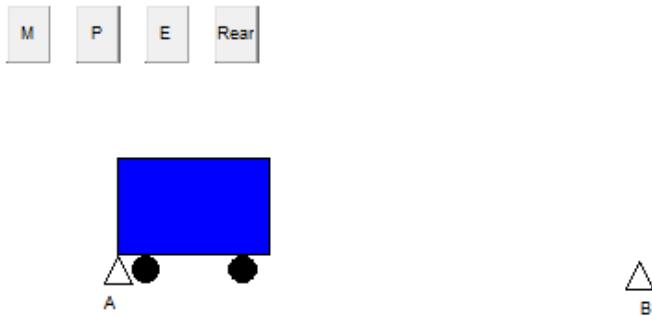


Figura 11-29 – Visualización del ejercicio 9

Al pulsar el botón *P* se debe detener el motor, pero no en el acto, sino al final del movimiento de vaivén ya iniciado. Al pulsar el mando de emergencia *E* se producirá el retorno inmediato a la posición de origen y no podrá ser puesto en marcha al menos que previamente se pulse el botón de rearne (*Rear*)

Se utilizará para desarrollar el ejercicio el fichero *Cap\_11\_14.pro*. El controlador se implementará en el POU *Automatismo* ya existente.

### 11.6.10. Ejercicio 10

Se desea diseñar un sistema de control para el ascensor simple de tres plantas de la Figura 11-30. El sistema de control se implementará en *Cap11\_15.pro*. El ascensor tiene un sensor en cada planta (*S<sub>i</sub>*) que detecta cuando el ascensor se encuentra en dicho lugar. La llamada al ascensor se hace solo desde el exterior mediante los cuatro pulsadores (*P<sub>i</sub>*), una correspondiente a cada planta. El motor se puede mover en las dos direcciones utilizando las variables *Motor\_arriba* y *Motor\_abajo*. El funcionamiento es muy simple. El ascensor debe acudir a la planta del correspondiente pulsador que le llama, donde permanecerá hasta que se le solicita

que acuda a otra planta. El automatismo se implementará en un POU denominado *Ascensor*.

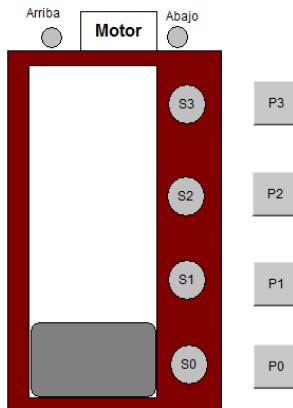


Figura 11-30 – Visualización del ejercicio 10

### 11.6.11. Ejercicio 11

Se desea diseñar en GRAFCET el mecanismo de apertura y cierre de una puerta de garaje. Para ello se dispone de un motor de apertura y cierre (*Señales Abrir* y *Cerrar*) y de un pulsador *P* para controlar el sistema. También dispone de dos fines de carrera, *FCC* (puerta cerrada) y *FCA* (puerta abierta). El sistema de control se realizará en *Cap11\_16.pro*. El modo de funcionamiento es el siguiente:

- Cuando se pulsa *P* y la puerta está cerrada, se inicia la maniobra de apertura. Si está abierta se inicia la maniobra de cierre.
- Si durante la apertura se pulsa *P*, la puerta se detiene. Estará detenida hasta que o bien se pulse nuevamente *P* o bien pasen 10 segundos. En cualquiera de los casos la puerta comenzaría la maniobra de cierre.
- Del mismo modo, si durante el cierre se pulse *P* la puerta se detendría, hasta que o bien se pulse *P* o bien transcurran 10 segundos. En cualquiera de los casos la puerta comenzaría la apertura.

- d) La puerta estará abierta un máximo de 30 segundos.
- e) El sistema deberá contar el número de veces que la puerta ha estado completamente abierta.

El programa en GRAFCET se escribirá en el POU ya creado denominado *Grafset*.

Lógicamente muchas de las transiciones del GRAFCET estarán asociadas a la señal *P*. Compruebe que para un correcto funcionamiento del programa es necesario que dichas transiciones estén asociadas a la detección de un flanco de subida en *P* en lugar de únicamente a la señal *P*.

### 11.6.12. Ejercicio 12

El proceso de la Figura 11-31 consiste en mezclar cantidades exactas de dos productos con agua y mezclarlas. El sistema consta de dos tanques superiores, respectivamente con los productos A y B. Un tanque intermedio en el que se mezclarán los dos productos anteriores y un tanque inferior de agua que dispone de un agitador.

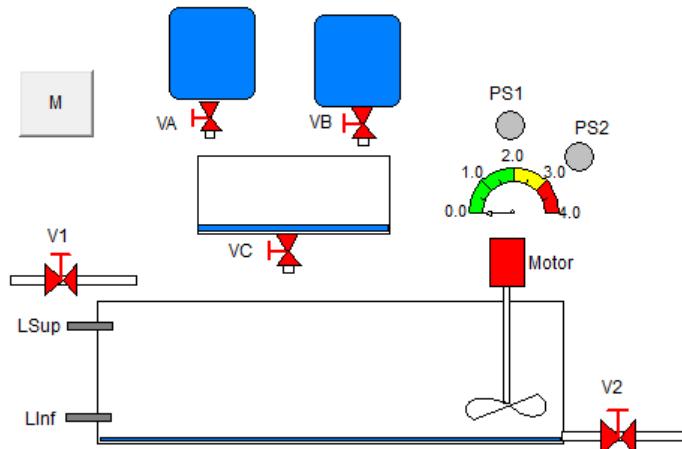


Figura 11-31 – Visualización del ejercicio 12

El proceso deberá realizar las siguientes operaciones:

- Llenar el tanque de agua hasta el nivel *LSup*

- En paralelo con lo anterior se llenará el depósito intermedio de A y B de la siguiente manera: En primer lugar se llenará de producto A abriendo la válvula  $VA$ , hasta que el peso llegue a 2 (se activará la señal  $PS1$ ). En ese momento se cierra la válvula  $VA$  y se abre la  $VB$  para continuar con el llenado hasta llegar a un peso de 3 (Se activará la señal  $PS2$ ), momento en el que se cerrará la válvula  $VB$ .
- En el momento en que las dos operaciones anteriores estén realizadas, se abrirá la válvula  $VC$  para verter la mezcla en el depósito inferior. Esta válvula estará abierta un tiempo suficiente para vaciar el depósito. Dicho tiempo, que será una constante, lo fijará el programador.
- Activar el agitador mediante la señal *Motor* durante 10 segundos.
- Vaciar el contenido del depósito inferior hasta llegar al nivel  $Linf$ .

Esta serie de operaciones se tendrá que repetir 3 veces (que se controlarán mediante un contador). La puesta en marcha se realizará mediante el pulsador  $M$ . Es decir, al pulsar  $M$  se realizará tres veces el proceso arriba indicado y el sistema se detendrá.

El programa de control se implementará en el POU *Automatismo*.

NOTA: La visualización dispone de una serie de interruptores para la activación manual de las válvulas. Estos botones tienen el objetivo de permitir al programador entender mejor el proceso, pero NO se utilizarán en el programa a realizar.

En las siguientes tablas se resume el conjunto de entradas y salidas del proceso.

Tabla 11-2. Variables de entrada en el ejercicio 12

Nombre	Tipo	Descripción
$M$	BOOL	Interruptor de puesta en marcha
$LInf$	BOOL	Sensor nivel bajo en depósito inferior
$LSup$	BOOL	Sensor nivel alto en depósito inferior
$PS1$	BOOL	Señal de peso igual a 2 alcanzado en depósito intermedio
$PS2$	BOOL	Señal de peso igual a 3 alcanzado en depósito intermedio

Tabla 11-3. Variables de salida en el ejercicio 12

Nombre	Tipo	Descripción
<i>V1</i>	BOOL	Válvula de entrada a depósito inferior
<i>V2</i>	BOOL	Válvula de salida del depósito inferior
<i>VA</i>	BOOL	Válvula de salida depósito de producto A
<i>VB</i>	BOOL	Válvula de salida depósito de producto B
<i>VC</i>	BOOL	Válvula de salida depósito intermedio
<i>Motor</i>	BOOL	Motor del agitador



# 12. USANDO SEÑALES ANALÓGICAS

---

Los tipos más habituales de señales con las que trabaja un autómata programable son las señales booleanas y las señales analógicas. En los capítulos anteriores, la práctica totalidad de las señales de entrada y salida que se han utilizado han sido de tipo booleano, ya que se ha estado tratando el diseño de automatismos lógicos. Sin embargo con la evolución de los Autómatas Programables, éstos han ido adquiriendo funcionalidades adicionales que les permiten realizar tareas que van mucho más allá del diseño e implementación de los catados automatismos lógicos. Entre ellas cabe destacar la utilización de sistemas de adquisición de datos, tanto de entrada como salida, para señales analógicas.

En la mayor parte de los procesos industriales medianamente complejos conviven las señales digitales con señales analógicas provenientes de sensores que miden magnitudes de tipo continuo, como presión, temperatura, velocidad, grado de apertura de una válvula, etc., por lo que es fundamental saber gestionar y procesar este tipo de variables desde un autómata programable.

El objetivo de este capítulo es hacer una introducción a las posibilidades que proporciona la norma IEC-1161-3 y en particular CoDeSys, para el diseño de sistemas de control basados en señales analógicas.

## 12.1. Señales analógicas

Los autómatas programables están conectados con el proceso mediante los módulos de entrada-salida. Como se ha visto en el Capítulo 1, estos módulos se encargan de traducir entre las señales, normalmente eléctricas, que se leen de los sensores o se envían a los actuadores del proceso y las variables de entrada/salida en el autómata programable.

Estos módulos de entrada-salida son dispositivos electrónicos cuyo diseño es distinto dependiendo de la naturaleza de la señal que procesen, fundamentalmente discreta o analógica y de entrada o de salida. En el caso de las señales analógicas, estos módulos realizan la conversión analógica a digital (conversión A/D) en las entradas o digital a analógica (D/A) en las salidas.

Las señales analógicas con las que trabajan los dispositivos de campo están normalizadas, siendo lo más habitual una señal de 4 a 20 mA. cuando la señal es de intensidad, y de 0 a 10 v. cuando se trata de una señal de tensión (también se trabaja habitualmente con otros rangos de tensión, como por ejemplo de -5 a 5 voltios).

Es importante recordar que en los módulos de E/S los valores de las señales analógicas se convierten a cadenas de bits. El número de bits que se utilice para almacenar la variable analógica determinará la *Resolución*, que es una característica de los módulos de E/S que es necesario tener muy en cuenta en el procesado de la señal.

Considérese un módulo de E/S de 8 bits. Es inmediato comprobar que los valores que se pueden almacenar en esos 8 bits van desde 0 hasta 255. Si la señal analógica que se está almacenando es de 0 a 10 voltios, esta diferencia de 10 voltios se puede dividir en 255 intervalos:

$$\frac{10 - 0}{255} = 0.0392 \text{ v.}$$

Por tanto, al almacenar una variable analógica en 8 bits se producirá una pérdida de información, En concreto, en el PLC no se reconocerán variaciones en la señal analógica menores de estos 0.0392 v.

Al adquirir un módulo de E/S es muy importante verificar que la resolución es igual o superior a la que se necesita en la aplicación. La gama de resoluciones que se encuentran en los productos comerciales es muy amplia, pudiendo ir de 4 a 32 bits, siendo valores típicos 12 o 16 bits.

### **12.1.1. Entradas analógicas: del proceso al autómata**

En la Figura 12-1 se muestra el proceso de conversión de una determinada señal, en este caso, una temperatura, desde el proceso hasta al módulo de E/S.

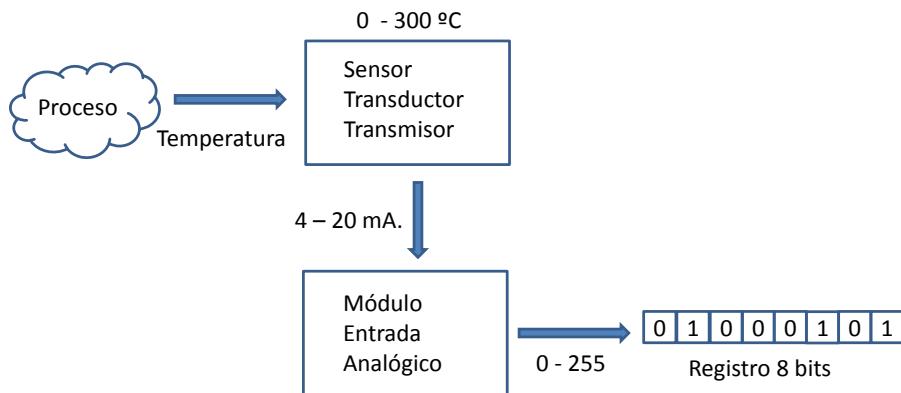


Figura 12-1 – Entrada analógica

En este proceso se pretende medir una determinada temperatura, para lo que se dispone de un sensor-transductor capaz de medir el rango de temperaturas que se desea, en este caso desde 0 hasta 300 °C. Este dispositivo convierte una señal de temperatura dentro de dicho rango en una señal eléctrica en intensidad de 4 a 20 mA. Finalmente, es conectada a un módulo de E/S de un PLC con un registro de 8 bits.

Por ejemplo, si en un determinado momento se mide una temperatura de 170 °C, la señal eléctrica normalizada será:

$$170 \cdot \frac{20-4}{300-0} + 4 = 13.066 \text{ mA.}$$

Esta señal eléctrica en el módulo de entrada se convertirá al siguiente valor digital:

$$(13.066 - 4) \cdot \frac{255}{20-4} = 144 = 2\#10010000$$

Este será el valor almacenado en el registro de 8 bits del módulo de entrada.

Como último paso, el valor digital es almacenado en la correspondiente posición de la zona de memoria de E/S y se podrá acceder a ella utilizando la sintaxis que define la norma IEC-1131-3 para las variables de entrada y salida.

Lo habitual es que las señales analógicas de E/S se almacenen en variables de tipo entero, principalmente WORD o DOUBLE WORD, por lo que las variables de

entrada se identificarán en el caso más simple con %IW<sub>i</sub>, donde i es un número entero que identifica la posición de la entrada. Si se almacenaran en DOUBLE WORD, la identificación sería de la forma %ID<sub>i</sub>.

Nótese que aunque una variable de tipo WORD tiene normalmente 16 bits permitiendo valores de 0 a 65535, se almacenarán en variables de este tipo cualquier dato proveniente de registros de menos de 16 bits. Los bits restantes (los más significativos) desde el número de bits del registro hasta 16 no deberán usarse. Por ejemplo, si se dispone de un módulo de entrada de 12 bits, al almacenarlo en una variable de tipo WORD, solo se usarán los bits de 0 al 11, mientras que del 12 al 15 no se utilizarán.

### **12.1.2. Salidas analógicas: del autómata al proceso**

En el caso de las salidas analógicas, el módulo de salida realiza la conversión digital – analógica, resultando, desde el punto de vista funcional, un proceso inverso al que se analizó en el caso de las entradas digitales.

El programa de control será el encargado de actualizar, en cada ciclo de ejecución del autómata, las variables de la zona de memoria del PLC correspondiente a las salidas. En este caso, la norma define una sintaxis de la forma %QW<sub>i</sub> o %QD<sub>i</sub> si se están usando respectivamente variables de tipo WORD o DOUBLE WORD. De nuevo, es importante asegurarse de utilizar solamente el número de bits que se corresponden con el registro del módulo de salida, dejando los demás a cero.

Por ejemplo, supóngase que se quiere controlar una válvula con un módulo de salida que utiliza registros de 12 bits. Esto permite codificar un total de  $2^{12} = 4096$  posiciones distintas. Si suponemos que el valor 10 se corresponde con la válvula completamente cerrada (0%) y el valor 4095 se corresponde con la válvula completamente abierta (100%) y que la variable de salida asociada es %QW1 ¿Qué valor será necesario poner en dicha variable para que la válvula se abra al 25%?

$$25 \cdot \frac{4095 - 10}{100 - 0} + 10 = 1031 = 2\#0000010000000111$$

Al final de cada ciclo del autómata, el valor de dicha variable se pasará al registro del módulo de salida analógica, que lo convertirá en un valor eléctrico de corriente o tensión de acuerdo a las especificaciones de la tarjeta. Este valor llegará al

actuador que abrirá la válvula en el porcentaje requerido.

## 12.2. Tratamiento de señales analógicas en el PLC

En la sección anterior se han analizado las variables de entrada o salida con las que trabajará el programa de control implementado en el PLC en función de la variable a medir (o sobre la que actuar), de su rango de valores y de las características de los módulos de entrada y salida.

Sin embargo, es habitual que en el programa de control se prefiera trabajar con los valores de ingeniería de la señal analógica en lugar de con los valores de las variables de entrada o salida. Por ejemplo, es necesario realizar esta conversión para la presentación de valores al usuario o para el diseño de controladores que usan dichas variables.

En esta sección se van a diseñar algunos de los bloques funcionales básicos que permiten realizar esta conversión.

### 12.2.1. Cambio de tipo de variable

En el caso de las entradas digitales, es necesario convertir las variables de entrada de tipo WORD (o DWORD) a variables de tipo REAL. Para las salidas analógicas, si éstas se han procesado como reales, será necesario pasar de REAL a WORD.

En este caso, la norma IEC-1161, y por tanto Codesys ya dispone de una familia de funciones estándar que permiten el cambio entre los distintos tipos de variables. Los nombres de las funciones son de la forma *tipo1\_TO\_tipo2*.

Por tanto, las funciones a utilizar en este caso sería WORD\_TO\_REAL para las entradas y REAL\_TO\_WORD para las salidas.

### 12.2.2. Escalado de señales analógicas

Para convertir las señales analógicas desde el valor almacenado en las variables de entrada a valores de ingeniería, es necesario realizar un escalado de dichas señales. Lo mismo ocurre para transformar las variables de salida desde valores de

ingeniería al valor guardado en las variables de salidas. En ambos casos, para realizar la conversión es necesario conocer:

- Valores límites en las variables de entrada/salida o número de bits utilizados para almacenar el valor.
- Valores límites de los valores de ingeniería que se corresponden con los valores del punto anterior.

Tal como se muestra en la Figura 12-2, el escalado consiste en una transformación lineal.

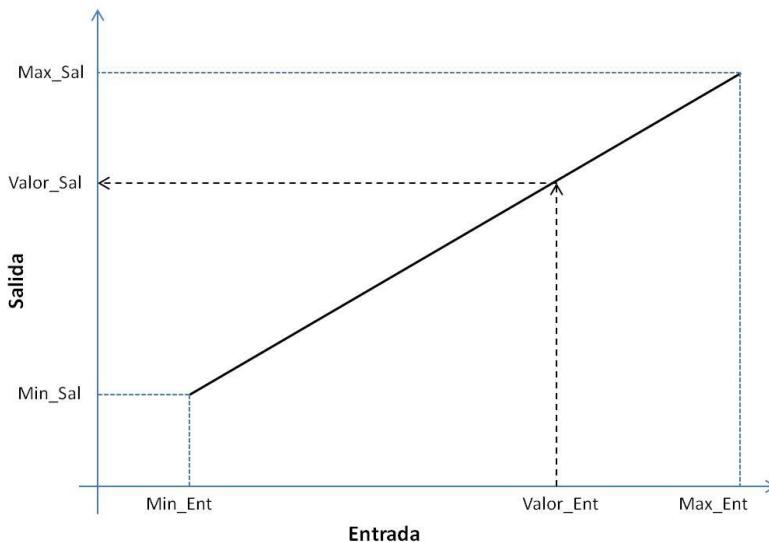


Figura 12-2 – Escalado de una señal analógica

Por tanto, el valor de salida puede obtenerse fácilmente de la siguiente expresión:

$$Valor\_Sal = (Valor\_Ent - Min\_Ent) \cdot \frac{Max\_Sal - Min\_Sal}{Max\_Ent - Min\_Ent} + Min\_Sal \quad (12.1)$$

El siguiente paso será construir un bloque funcional denominado *Escalado*, escrito en lenguaje ST y que se podrá utilizar en todas los programas donde se necesite.

El bloque funcional tendrá los siguientes parámetros de entrada y salida:

- **Parámetros de Entrada**

- *Valor\_Ent* : REAL. Valor a convertir.
- *Min\_Ent* : REAL. Valor mínimo de la entrada.
- *Max\_Ent* : REAL. Valor máximo de la entrada.
- *Min\_Sal* : REAL. Valor mínimo de la salida.
- *Max\_Sal* : REAL. Valor máximo de la salida.

- **Parámetros de Salida**

- *Valor\_Sal* : REAL. Valor convertido.
- *Error* : BOOL Se activa cuando el valor de entrada queda fuera de límites.

El código del bloque funcional se muestra en el Ejemplo 12-1.

---

**Ejemplo 12-1. Código del bloque funcional Escalado**

---

```
FUNCTION_BLOCK Escalado
VAR_INPUT
    Valor_Ent : REAL;      (* Valor a convertir *)
    Min_Ent : REAL;        (* Valor mínimo de la entrada *)
    Max_Ent : REAL;        (* Valor máximo de la entrada *)
    Min_Sal : REAL;        (* Valor mínimo de la salida *)
    Max_Sal : REAL;        (* Valor máximo de la salida. *)
    ValError : REAL:=-1;   (*Valor al que poner la salida en
                           caso de error. Por defecto -1*)
END_VAR
VAR_OUTPUT
    Valor_Sal : REAL;     (*. Valor convertido *)
    Error : BOOL;         (* Se activa cuando el valor de entrada
                           Queda fuera de límites *)
END_VAR
VAR
    pendiente: REAL;
END_VAR
```

---

## Continuación

---

```
(* Determinación del estado de error *)
Error:= (Valor_Ent>Max_Ent) OR (Valor_Ent<Min_Ent);

IF NOT Error THEN
    (* Si no hay error se calcula la salida *)
    pendiente:=(Max_Sal-Min_Sal)/(Max_Ent-Min_Ent);
    Valor_Sal:=(Valor_Ent-Min_Ent)*pendiente+Min_Sal;
ELSE
    (* Si hay error a la salida se le da el valor -1 *)
    Valor_Sal:=-1;
END_IF
```

---

Nótese que la entrada *ValError* sirve para determinar el valor de la salida del bloque funcional cuando se detecta un error. Por defecto, el valor es  $-1$ . En caso de utilizar este bloque funcional para dar valor a una variable de salida del PLC habría que proporcionarle el valor correspondiente a la posición segura del dispositivo de salida.

Este bloque funcional se encuentra implementado en el fichero *Cap12\_1.pro*, donde también se incluye en el programa principal la llamada a una instancia de dicho bloque y una visualización desde la que se le pueden ir dando valores a la entrada y viendo las salidas obtenidas.

### 12.2.3. Ejemplo

Se desea controlar en modo manual la apertura de una válvula desde un SCADA. El valor que introduce el operario en el SCADA llega al autómata por la entrada *IW1* de 8 bits, mientras que la señal de salida a la válvula se proporcionará por la variable de salida *QW1* de 12 bits. La apertura de la válvula en valores de ingeniería está medida en tanto por ciento, de modo que la válvula totalmente cerrada (0%) se corresponde con el valor 0 tanto en *IW1* como en *QW1*, y la válvula completamente abierta se corresponderá con los valores máximos de cada uno de los registros. La variable en valores de ingeniería se almacenará en la variable *REAL AperturaValvula*.

El problema se encuentra resuelto en *Cap12\_2.pro*. Hace uso del bloque funcional de

escalado definido anteriormente y de la conversión de valores entre WORD y REAL. Para el caso de la instancia del bloque funcional que da valor a QW1 el valor que se ha tomado para el caso de error en 0, es decir, válvula cerrada. El programa está realizado en FBD.

## 12.3. Control de sistemas continuos en un PLC

En esta sección se presentan algunos bloques funcionales y ejemplos básicos que pueden ser de interés en el control de sistemas continuos utilizando PLCs. Se va a presentar la implementación de controladores básicos, como por ejemplo el PID, así como su utilización en el control de un proceso. En los ejemplos que se proponen, los procesos a controlar estarán simulados en un POU que se ejecutará junto al programa de control.

Como es bien conocido, dado un proceso en el que existe una variable  $Y$  que se desea controlar, y una actuación  $U$  sobre el proceso que tiene efecto con una cierta dinámica sobre la variable  $Y$ , el objetivo de un controlador es determinar la señal de actuación  $U$  sobre el proceso para que la variable de salida a controlar  $Y$  siga una determinada referencia  $SP$  con unas determinadas especificaciones.

En la Figura 12-3 se muestra la estructura general de un sistema de control implementado en un PLC. Como se puede observar, al PLC le llegará una señal analógica de entrada (la variable  $Y$  a controlar) y proporcionará una salida analógica (la actuación  $U$ ). La referencia  $SP$  habitualmente la proporcionará un operario desde un sistema tipo SCADA o bien puede ser un valor dado por el propio sistema de control que se ejecuta en el PLC.

Un tema importante a la hora de implementar un controlador en un PLC es el *Tiempo de Muestreo*. Cuando el controlador se ejecuta en un computador, que es el caso de un PLC, la lectura de la señal  $Y$  y la obtención de un nuevo valor para la actuación  $U$  se realiza cada cierto intervalo fijo de tiempo que se denomina *tiempo de muestreo*. La determinación por el usuario de este valor va a depender de la dinámica del sistema a controlar.

Si el controlador está implementado en una POU, por defecto, ésta se ejecuta cada ciclo del autómata, cuyo tiempo de ejecución a priori es desconocido y depende entre otros factores, de la complejidad del programa ejecutándose en el PLC. Por tanto, en el caso de los controladores, es aconsejable asociar la POU en la que está

escrito a una tarea en la que el modo de ejecución no sea cíclico sino periódico. El valor del tiempo periódico sería el tiempo de muestreo. Para una adecuada configuración de las tareas en CoDeSys véase el Apéndice A.

A continuación se verán algunos ejemplos y se propondrán ejercicios de los controladores básicos P, PI y PID.

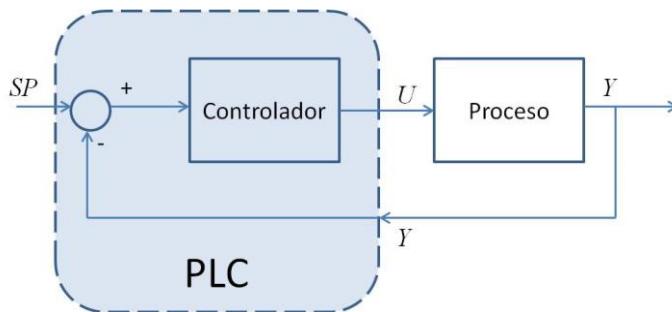


Figura 12-3 – Esquema de control de un proceso continuo desde un PLC

### 12.3.1. Controlador P

En primer lugar de va a realizar un bloque funcional en el que se implemente un controlador P simple.

En el controlador proporcional, la expresión matemática que permite obtener el valor de la actuación  $U$  es la siguiente:

$$U = K \cdot (SP - Y) + Offset \quad (12.2)$$

donde  $K$  es un parámetro a ajustar.

La señal de actuación deberá estar limitada entre un valor máximo y otro mínimo. Si el valor obtenido en la actuación es menor que el valor mínimo o mayor que el máximo, se le dará a la salida el correspondiente valor límite.

Además, el controlador puede estar en dos modos de funcionamiento

- Automático: La salida se calcula mediante la ecuación (12.2). En este modo, el controlador estaría funcionando en lazo cerrado de acuerdo al esquema de la Figura 12-3.
- Manual: El control opera en lazo abierto. El valor de la salida se obtiene directamente de la entrada del bloque funcional *Manual\_Ent* mediante la expresión:

$$U = Man\_Ent + Offset \quad (12.3)$$

La elección entre un modo y otro se realiza con la entrada booleana del bloque funcional *Manual*. Si esta señal está a TRUE funcionará en modo manual y si está a FALSE, en modo automático.

La Figura 12-4 muestra el esquema que se va a realizar en el bloque funcional del controlador P:

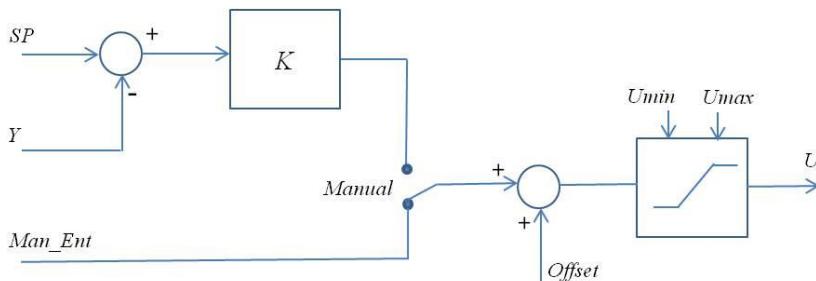


Figura 12-4 – Esquema de un controlador P

Por tanto, el bloque funcional tendrá las siguientes entradas y salidas:

- **Entradas:**
  - *SP* : REAL. Set Point, valor deseado de la variable a controlar.

- $Y$  :REAL. Variable a controlar. El valor lo proporciona un sensor en el proceso.
- $K$  : REAL. Parámetro del controlador proporcional. Por defecto tendrá el valor 1.0.
- $Umin$  : REAL. Valor mínimo de la salida. Por defecto se le da el valor 0.
- $UMax$  : REAL. Valor máximo de la salida. Por defecto, tendrá un valor 100.0.
- $Offset$  :REAL. Cantidad a sumar a la señal de salida proporcionada por el controlador según ecuaciones (12.2) y (12.3). Por defecto tendrá un valor 0.
- $Manual$  : BOOL. TRUE modo manual, FALSE modo automático.
- $Manual\_Ent$  : REAL. Valor para la salida según ecuación (12.3) en modo manual

- **Salidas**

- $U$  : REAL. Salida del controlador.
- $Lim$  : BOOL. Estará a TRUE si se superan los límites  $Umin$  o  $Umax$  en la salida

El código de bloque funcional de un controlador P se muestra en el Ejemplo 12-2.

#### **12.3.1.1. Ejemplo de un controlador P**

En el fichero *Cap11\_3.pro* hay implementado un sistema de control completo con un controlador P para controlar un proceso. El proceso se corresponde con un sistema de primer orden que se simula en el POU *Proceso*. El proceso actualiza las variable *IW1* (salida del proceso y entrada del PLC) y *QW1* (entrada al proceso y salida del PLC). Las dos variables están definidas como variables globales y son de tipo WORD y se supone que se han utilizado registros de 12 bits (valores de 0 a 4095). En valores de ingeniería, tanto la entrada como la salida están limitadas entre -10 y 10.

**Ejemplo 12-2. Código del bloque funcional Controlador P**


---

```

FUNCTION_BLOCK ControladorP
VAR_INPUT
    SP : REAL;          (* Set Point *)
    Y :REAL;            (* Variable a controlar. *)
    K : REAL := 1;      (* Parámetro del controlador *)
    Umin : REAL :=0;   (* Valor mínimo de la salida *)
    UMax : REAL:=100;  (* Valor máximo de la salida *)
    Offset :REAL:=0;   (* A sumar a la salida del controlador *)
    Manual : BOOL;     (* TRUE modo manual, FALSE modo automático*)
    Manual_Ent : REAL; (*Valor para la salida en modo manual *)
END_VAR
VAR_OUTPUT
    U : REAL;          (*Salida del controlador. *)
    Lim : BOOL;        (*A TRUE si se superan los límites Umin o Umax en
                        la salida *)
END_VAR
-----
IF Manual THEN
    U:=Manual_Ent+Offset;  (* Salida en modo manual *)
ELSE
    U:=K* (SP-Y)+Offset;  (* Salida en modo automático *)
ENDIF
(* Saturación de la salida *)
IF U>UMax THEN
    U:=UMax;
ELSE
    IF U<Umin THEN
        U:=Umin;
    END_IF
ENDIF
LIM:=(U>UMax) OR (U<Umin);

```

---

En el fichero indicado, el sistema de control está implementado en el POU *Control*, que lee como variable de entrada *IW1* y proporciona valor mediante el controlador P a *QW1*. Los pasos que se realizan en este POU son:

- Transformación de *IW1* a valor real
- Escalado de la señal de entrada a valores de ingeniería, en este caso, de -10 a 10.
- Calculo de la actuación en valores de ingeniería con el controlador P usando el

bloque funcional. La salida también está limitada entre -10 y 10.

- Escalado de la actuación al rango 0-4095
- Transformación de la variable real a WORD y almacenamiento en QW1.

En este caso no se ha fijado un tiempo de muestreo determinado sino que se ha considerado el de la duración del propio ciclo del autómata. Es decir, el autómata está funcionando en modo cíclico y no periódico.

La visualización permite el cambio del parámetro  $K$ , del set-point  $SP$ , poner el sistema en modo manual o automático y fijar el valor de la entrada en modo manual *Manual\_Ent*. Todas estas variables se corresponden con variables de entrada. Modifique estos valores y observe el comportamiento del proceso. Compruebe que el controlador P siempre tiene error en régimen permanente, y que disminuye al aumentar  $K$ .

### 12.3.2. Controlador PI

En un controlador PI la salida  $U$  se obtiene como se muestra en la ecuación (12.4):

$$\begin{aligned} \text{error} &= SP - Y \\ U &= K_p \cdot (\text{error} + \frac{1}{T_N} \int \text{error}) + \text{Offset} \end{aligned} \quad (12.4)$$

Como se observa, con respecto al controlador P se añade un sumando proporcional a la integral del error, siendo  $K_p$  y  $T_N$  los dos parámetros a ajustar del controlador.

Con respecto a la implementación en un bloque funcional, la dificultad más importante que aparece es el cálculo del término integral del error. En este bloque funcional se va a utilizar una aproximación basada en el método de Euler, que como es conocido, usa la aproximación de la integral que aparece en la Figura 12-5.

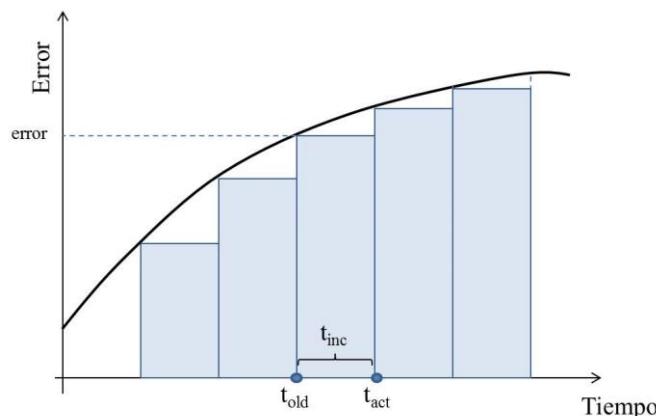


Figura 12-5 – Aproximación de la integral del error por el método de Euler

De nuevo se va a considerar el autómata funcionando en modo cíclico. Para calcular el error se necesita el intervalo de tiempo  $t_{inc}$  que es el tiempo de ciclo del autómata. Por tanto se necesita un procedimiento para calcular dicho tiempo, para lo que se va a hacer uso de la función predefinida TIME(), que devuelve el tiempo transcurrido en milisegundos desde el último arranque del sistema. El valor que devuelve es del tipo TIME. Nótese que este procedimiento será válido tanto para el autómata funcionando en modo cíclico como periódico, aunque en este último caso se podría simplificar el código.

Para poder operar con el tiempo, lo más simple es transformarlo en una variable de tipo WORD. Con esta puntuación, el código de un PI simple quedaría tal como se indica en el Ejemplo 12-3.

### 12.3.2.1. Ejemplo de un controlador PI

En el fichero *Cap12\_4.pro* hay implementado un sistemas de control completo con un controlador PI para controlar el mismo proceso que se vio en el caso del controlador P. La visualización permita modificar los parámetros del controlador.

**Ejemplo 12-3. Código del bloque funcional Controlador PI**


---

```

FUNCTION_BLOCK ControladorPI
VAR_INPUT
    Y :REAL;          (* Variable a controlar. *)
    SP : REAL;        (* Set Point*)
    Kp : REAL := 1;   (* Parámetro del controlador*)
    TN : REAL:=1;    (* Parámetro del controlador. Tiempo integral*)
    Umin : REAL :=0; (* Valor mínimo de la salida *)
    UMax : REAL:=100; (* Valor máximo de la salida. *)
    Offset :REAL:=0;  (* Cantidad a sumar a la salida*)
    Manual : BOOL;   (* TRUE modo manual, FALSE modo automático*)
    Manual_Ent : REAL;(*Valor para la salida modo manual *)
END_VAR
VAR_OUTPUT
    U : REAL;         (*Salida del controlador. *)
    Lim : BOOL;       (*TRUE si se superan los límites Umin o Umax U *)
END_VAR
VAR
    tact: TIME;      (* Instante de tiempo actual en formato TIME *)
    tactw: DWORD;    (* Instante de tiempo actual en formato DWORD *)
    tinc: REAL;      (* Tiempo de ciclo del autómata *)
    toldw: DWORD;    (* Instante de tiempo del ciclo anterior *)
    error: REAL;     (* SP - Y *)
    ierror: REAL;    (* Integral del error *)
    Ciclo1:BOOL :=TRUE; (* TRUE si es el primer ciclo *)
END_VAR
-----
(* Calcula el instante de tiempo actual (milisegundos) *)
tact:=TIME();
tactw:=TIME_TO_DWORD(tact);
(* Calcula el intervalo de tiempo desde el último ciclo *)
tinc:=DWORD_TO_REAL(tactw-toldw);
(* Almacena el valor actual de tiempo para al próximo ciclo *)
toldw:=tactw;

(* Salida en modo manual *)
IF Manual THEN
    U:=Manual_Ent+Offset;
    (* En Manual no se calcula el error *)
    ierror:=0;
ELSE
    (* Calculo error **)
    error:=SP-Y;
    (* Calculo de la integral del error usando el método de Euler*)

```

---

---

```

(* En el primer ciclo no se calcula.told no tiene valor valido *)
IF NOT Ciclol THEN
    ierror:=ierror+tinc*error;
END_IF
(* Calculo de la salida del controlador *)
U:=Kp*(error+(1/TN)*ierror)+Offset;
END_IF

(* Solo se ejecuta en el primer ciclo *)
IF Ciclol THEN
    U:=Offset;
    Ciclol:=FALSE;
END_IF

(* Calculo de las saturaciones de la salida *)
IF U>Umax THEN
    U:=Umax;
ELSE
    IF U<Umin THEN
        U:=Umin;
    END_IF
END_IF

(* Calcula el valor LIM *)
LIM:=(U>Umax) OR (U<Umin);

```

---

### 12.3.3. Controlador PID

El controlador PID calcula la salida del controlador con la siguiente expresión:

$$\begin{aligned}
 error &= SP - Y \\
 U &= K_p \cdot (error + \frac{1}{T_N} \int error) + T_D \frac{d(error)}{dt} + \text{Offset}
 \end{aligned} \tag{12.5}$$

en el que en relación al controlador PI, se le ha añadido un término proporcional a la derivada del error.

#### 12.3.3.1. Ejemplo de un controlador PID

Utilizando el código del controlador PI, implemente un controlador PID. Implémente en el fichero *Cap12\_5.pro* y utilícelo para controlar el proceso que se

estudió en los apartados anteriores. En dicho fichero se ha mantenido el controlador PI para que sirva de referencia. También es necesario diseñar el programa *Control* de forma similar a los ejemplos anteriores del P y PI. Para dar valores a las entradas del controlador PID utilice las variables ya declaradas en el programa *Control* ya que son las que están asociadas a la visualización que se proporciona.

### 12.3.3.2. Ejercicio con el controlador PID de CoDeSys

En el entorno Codesys (y en la práctica totalidad de los entornos) se proporciona un bloque funcional predefinido para un PID muy similar al que se ha diseñado. El nombre del bloque funcional es PID y se encuentra en la librería Util.lib. En caso de que la librería no esté accesible, se deberá incorporar al Proyecto tal como se indica en el Apéndice A,

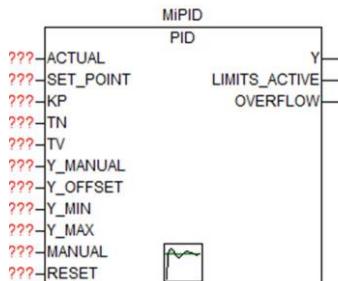


Figura 12-6 – Bloque funcional PID definido en CoDeSys

Estudie el significado de las entradas y salidas del bloque funcional que se muestra en Figura 12-6 utilizando la ayuda de CoDeSys y utilícelo en el fichero *Cap11\_5.pro* para controlar el proceso.

## 12.4. Bloques funcionales para el control de procesos

En este apartado se presentan algunos ejemplos y ejercicios para el desarrollo de bloques funcionales de interés en el caso de control de proceso de sistemas

continuos. Aunque algunos de ellos se encuentran resueltos, se recomienda al lector que haga su propio desarrollo, lo que le permitirá practicar en el manejo de bloques funcionales con señales analógicas.

### 12.4.1. Generador de salida PWM

El objetivo de este bloque funcional será la generación de una señal modulada por ancho de pulsos (PWM) a partir de una señal analógica. Las señales PWM son de uso habitual para actuar sobre ciertos dispositivos, como pueden ser los motores.

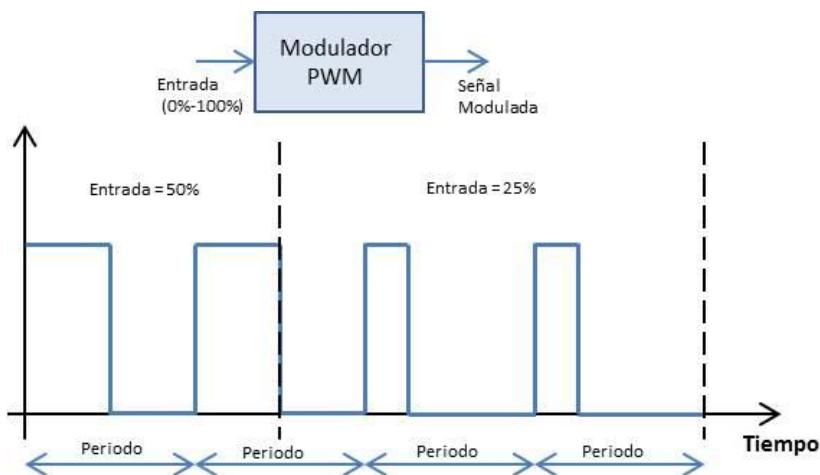


Figura 12-7 – Modulación PWM de una señal analógica

La Figura 12-7 muestra la modulación PWM de una señal analógica cuyo valor está comprendido entre 0% y 100%. La señal modulada es una señal binaria de periodo fijo de forma que el tiempo que durante un período la señal está a valor alto es proporcional a la señal de entrada.

En el fichero *Cap12\_6.pro* se encuentra implementado un bloque funcional que realiza esta modulación. En este bloque funcional, el valor mínimo del período de la señal viene dado por el tiempo de ciclo del autómata. Los parámetros de entrada de dicho bloque son:

- *Entrada:* REAL. Señal a modular, entre 0 y 1.
- *Periodo:* REAL. Valor del periodo de la señal modulada en milisegundos.

Por otro lado, la salida del bloque funcional es la señal modulada que se proporciona en la variable booleana *Q*.

El código del bloque funcional hace uso de otro bloque funcional denominado *Pulso*. Este bloque funcional genera un pulso de un ciclo del autómata de anchura y con un periodo dado por su parámetro de entrada. El código de ambas funciones se encuentra en el fichero anteriormente indicado.

### 12.4.2. Control de una válvula motorizada

En esta sección se va a desarrollar un bloque funcional simple cuyo objetivo es el control de una válvula motorizada. Las funciones que realiza el bloque funcional serán principalmente:

- Seleccionar el modo manual o automático. En modo manual el operario fijará la posición de la válvula desde la consola, o en este caso desde la visualización de CoDeSys. En modo automático la consigna de posición de la válvula vendrá del programa de control, por ejemplo de un PID.
- Realimentación y tratamiento de las señales que llegan de la válvula motorizada, por ejemplo, el valor real de la posición de la válvula o una señal de error.
- Fijación de la señal de consigna a la válvula en caso de aviso de error en la válvula.
- Fijación de la señal de consigna a la válvula en caso de enclavamiento. Un ejemplo de enclavamiento se daría si aguas arriba de la válvula existe otra válvula de tipo todo-nada. La válvula motorizada solo debe funcionar en caso de que la válvula todo-nada esté abierta. En caso de que la válvula todo-nada estuviera cerrada, la válvula de regulación deberá estar enclavada y en una posición de cerrada.

Para el caso de esta válvula simple, los parámetros de entrada que se van a considerar para el bloque funcional son:

- *Manual* (BOOL): Si está a TRUE el modo es manual, en caso contrario

Automático.

- *SPManual* (REAL). Consigna que se manda a la válvula en modo manual. El valor estará entre 0 y 100.
- *SPAAuto* (REAL). Consigna que se manda a la válvula en modo automático. El valor estará comprendido entre 0 y 100.
- *PV* (REAL): Señal que llega de la válvula de la posición real que tiene ésta. Se utiliza para comprobar la diferencia entre el valor de consigna que se envía y el valor que realmente tiene la válvula. Puede servir para detectar un mal funcionamiento de la válvula.
- *ErrorValvula* (BOOL): Señal que llega de la válvula indicando un funcionamiento incorrecto de ésta.
- *SPError* (REAL): Valor de posición a enviar a la válvula en caso de error. Debe ser la posición de seguridad.
- *Encl* (BOOL): Señal que viene de otro dispositivo para enclavar la posición de la válvula.
- *SPEnclav* (REAL): Valor de posición a enviar a la válvula en caso de enclavamiento.

Los parámetros de salida del bloque funcional serían:

- *SP* (REAL) Consigna que se envía a la válvula. El valor estará entre 0 y 100.
- *DIF* (REAL). La diferencia entre SP y PV
- *Alarma* (BOOL): Situación de alarma.

En este control simple la situación de mal funcionamiento de la válvula se dará o bien cuando llegue una señal de error de la válvula o bien cuando durante 1 minuto se mantenga  $SP-PV>0.3$ . Esta condición sirve a título de ejemplo, ya que normalmente las condiciones de error de una válvula suelen ser más complejas.

Nótese que las entradas y salidas del bloque son variables reales. Lógicamente cuando haya que conectarlas a salidas y entradas del PLC será necesaria escalarlas y convertirlas a WORD tal como se ha realizado en secciones anteriores.

El código del bloque funcional se muestra en el Ejemplo 12-4.

*Ejemplo 12-4. Código del bloque funcional para el control de una válvula motorizada*

---

```

FUNCTION_BLOCK ControlValvula
VAR_INPUT
    Manual : BOOL;
    SPMManual : REAL;
    SPAuto : REAL;
    PV : REAL;
    ErrorValvula : BOOL;
    SPError : REAL;
    Encl : BOOL;
    SPEncl : REAL;
END_VAR
VAR_OUTPUT
    SP:REAL;
    DIF : REAL;
    Alarma : BOOL;
END_VAR
VAR
    TempoError: TON; (* Temporizador para detectar error *)
    INTtempo: BOOL; (* Entrada al temporizador *)
END_VAR
-----
IF MANUAL THEN
    SP:=SPManual;
ELSE
    SP:= SPAuto;
ENDIF
(* Se le da prioridad al enclavamiento *)
IF Encl THEN SP:=SPEncl; END_IF
DIF:=SP-PV;
INTtempo:= ABS(DIF)>0.3;
TempoError(IN:=INTtempo, PT:=T#1m);

Alarma:=ErrorValvula OR TempoError.Q;
(* Prioridad máxima a la salida de error *)
IF (Alarma) THEN SP:=SPError; END_IF

```

---

#### 12.4.2.1. Ejemplo

Para probar el funcionamiento del bloque funcional anterior, en el fichero

*Cap12\_7.pro* se ha realizado una simulación y una visualización para el control de la válvula motorizada de la Figura 12-8 (Válvula 1).

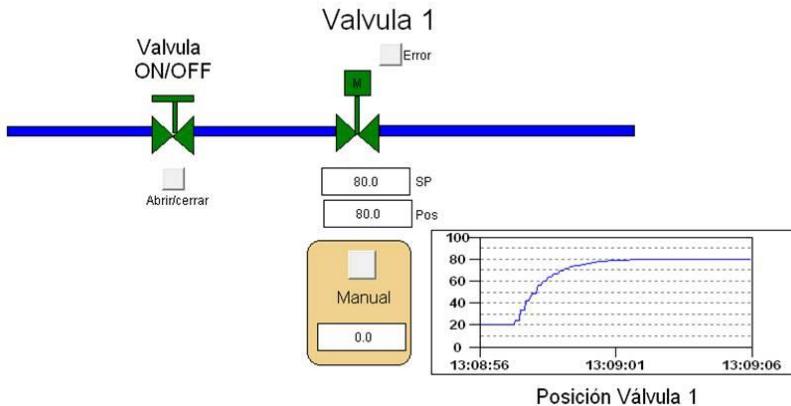


Figura 12-8 – Esquema de la válvula a controlar

Todas las señales analógicas de entrada/salida del autómata están codificadas con 12 bits (0 – 4095) y son las siguientes (Ver figura):

- **Entradas**

- *IW1\_Posicion* : WORD; Posición de la válvula 1
- *IX2\_Error* : BOOL; Error en la Válvula 1
- *Man* : BOOL; Variable de entrada asociada al botón del panel de control de la válvula. La válvula estará en modo manual si esta variable tiene valor TRUE.
- *RefManual* : REAL; Valor de referencia de la válvula cuando está en modo manual. Este valor se proporciona en la correspondiente entrada numérica del panel de control de la válvula.

- **Salidas**

- *QW1\_SPPosicion* : WORD; Consigna de posición para la válvula.

La válvula ON/OFF se puede abrir y cerrar con el botón de la imagen aunque queda fuera del sistema de control a diseñar.

Para la realización de este ejercicio se deberá:

- Implementar el bloque funcional *ControlValvula* definido en esta sección.
- Crear una instancia del bloque funcional *ControlValvula* para la Válvula 1 de la visualización.
- El programa de control se escribirá en la POU denominada *ProgramaControl* en el lenguaje que el lector estime oportuno. (no está creado en el fichero).
- El modo manual se controla desde el recuadro de la visualización, donde se le pueden dar valor a las variables de entrada *Man* y *RefManual*. En modo automático la posición de la válvula debe seguir una referencia que cambia en escalón entre 20% y 80% cada 10 segundos. Esta función está implementada en el bloque funcional *SenalAuto* ya creado.
- Desde la visualización se puede simular el error de la válvula pulsando el botón *Error*. Ante un error, la válvula deberá abrirse completamente.
- En caso de que la válvula ON/OFF de la figura esté cerrada deberá enclavarse la Válvula 1 cerrando completamente la válvula.

El sistema de control está resuelto en *Cap12\_7\_Res.pro*, aunque se insta al lector a resolver el problema por sus propios medios.

## 12.5. Ejercicios

### 12.5.1. Ejercicio 1

Se desea diseñar un sistema de control de caudal mediante un controlador PI. Se dispone de una válvula motorizada y de un caudalímetro. El esquema del proceso, similar al del ejercicio anterior, se muestra en la Figura 12-9. En el fichero *Cap12\_8.pro* se encuentra implementado el simulador, variables globales,

visualización y definición de bloques funcionales básicos. Al igual que en el ejercicio anterior las entradas analógicas estarán comprendidas entre 0 y 4095. Las entradas al autómata programable provenientes del proceso serán:

- *IW1\_Posicion* : WORD; Posición de la válvula 1
- *IX2\_Error* : BOOL; Error en la Válvula 1
- *IW2\_Caudal* : WORD; Lectura del caudalímetro. Mide entre 0 y 100 l/minuto.

A estas entradas habrá que añadir las entradas asociadas a los cuadros de control de la visualización, tanto de la válvula 1 como del controlador PI. Estas variables se muestran en la Tabla 12-1.

Y las salidas serán:

- *QW1\_SPPosicion* : WORD; Consigna de posición para la válvula 1

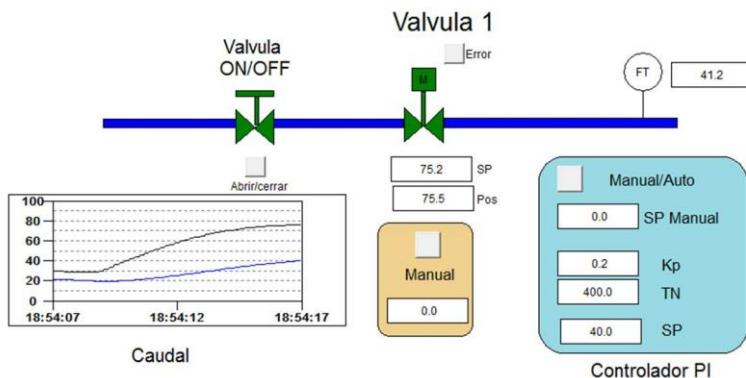


Figura 12-9 – Esquema del proceso a controlar en Ejercicio 1.

Se hará uso de los bloques funcionales *ControladorPI* y *ControlValvula*, ya que la válvula 1 se deberá comportar de la misma manera que en el ejercicio anterior, es

decir, considerar estados de error, enclavamientos, modo manual o automático, etc. Del mismo modo el PI se podrá poner en modo Manual y Automático. Todas las acciones de control del operario se realizarán desde la visualización.

Tabla 12-1. Variables de entrada asociadas a la visualización del ejercicio 1

Nombre	Tipo	Descripción
<i>Panel de Control de la Válvula 1</i>		
<i>Man</i>	BOOL	Modo Manual/Automático de la Válvula 1
<i>RefManual</i>	REAL	Valor de consigna de posición en Válvula 1 en Manual
<i>Panel de Control del Controlador PI</i>		
<i>ManPI</i>	BOOL	Modo Manual/Automático del PI
<i>ManPI_Ent</i>	REAL	Valor de entrada del PI Manual (lazo abierto)
<i>KpPI</i>	REAL	Parámetro Kp del PI
<i>TNPI</i>	REAL	Tiempo Integral del PI
<i>SPPI</i>	REAL	Set Point del PI en modo lazo cerrado

### 12.5.2. Ejercicio 2

En este ejercicio, el mismo proceso del apartado anterior se va a controlar desde un proceso secuencial implementado en GRAFCET. El sistema deberá realizar lo siguiente:

- En el estado inicial el controlador PI estará en modo manual con una consigna de posición de 0% (válvula cerrada)
- Cuando se pulse el botón de entrada *IX3\_Boton*, el controlador PI deberá funcionar en automático con un SP=40 l/min. durante 30 segundos.
- Seguidamente se deberá poner en modo manual enviando una consigna

de posición a la válvula del 30%. En este estado estará durante 40 segundos.

- Finalmente, se volverá a poner en automático con un SP=10 l/min. durante 30 segundos.

El ejercicio sin resolver está en *Cap12\_9.pro* y resuelto en *Cap12\_9\_Res.pro*.

### 12.5.3. Ejercicio 3

Como se puede observar, el ejercicio anterior plantea un nuevo problema: desde el momento en que los ajustes del PI (modo manual y set-points) los fija el programa de control secuencial en GRAFCET, deja de estar operativo el cuadro de control en la visualización.

En este ejercicio se pretende que se pueda dar el control del PI al programa secuencial o al operador mediante el panel de control. Esta es una situación habitual cuando se dispone de un sistema de supervisión SCADA (realmente la visualización se puede considerar una versión muy simplificada de un SCADA).

Para conseguir esto se va a añadir en el panel de control un interruptor adicional que define el *Propietario* del PI. El propietario puede ser o bien el operador (el PI se gestiona desde el cuadro) o bien el proceso, de forma que el PI se gestionaría desde un programa de control externo (en este caso escrito en GRAFCET).

En este ejercicio se pretende que se puedan gestionar desde ambos sitios el modo manual/automático y los respectivos set-points en ambos modos. Esto va a implicar algunos cambios en el bloque funcional PI

- Añadir una nueva entrada *Propietario* que se le dará valor desde el panel de control y que determina quien gestiona el PI, TRUE el operador, FALSE el programa de control.
- Duplicar las señales que se quieren controlar en las dos situaciones.
  - La señal *Manual* se sustituye por *ManualOperador* y *ManualProceso*.
  - SP se sustituye por *SPOperador* y *SPProceso*.
  - *Manual\_Ent* por *Manual\_EntOperador* y *Manual\_EntProceso*.

En este ejercicio, los parámetros de ajuste *Kp* y *TN* del PI solo se pueden cambiar

desde el panel de control.

### Solución

En el fichero *Cap12\_10\_Res.pro* se encuentra una solución a este ejercicio. En el panel de control se ha añadido un nuevo botón para determinar el propietario. El nuevo bloque funcional para implementar el PI denominado *PIControlDoble* se ha realizado mediante una llamada al bloque *ControladorPI*. Este nuevo bloque funcional selecciona en función del *Propietario*, las señales a utilizar en el *ControladorPI*.

Estudie la solución y verifique que el comportamiento es el especificado.

# 13. PROBLEMAS

---

**E**n este capítulo se proponen ejercicios para poder practicar, con un nivel de dificultad algo mayor que los ejercicios vistos en los capítulos anteriores. En todos ellos se indica el fichero *.pro* donde se encuentra implementada el proceso y están definidas como variables globales las entradas y salidas. El programa principal PLC\_PRG está escrito en ST e incluye una llamada al programa que ejecuta la animación. Se deberá hacer en ese mismo fichero la llamada al programa que se diseñe para realizar el control.

## 13.1. Cruce con tranvía

Se desea diseñar el sistema de control para los semáforos de la rotonda de la figura (rotonda de la Avenida del Cid de Sevilla), donde se combina el paso de los coches con el tranvía. Es necesario controlar dos semáforos, uno para el tráfico de vehículos (SC) y otro para los peatones (SP). Los tranvías pueden circular en los dos sentidos. La entrada y salida de tranvías en el cruce se detectan por los sensores S1 y S2. La animación incluye dos botones que simula la llegada de tranvía por cada uno de los lados. El fichero *.pro* y gráficos necesarios para la animación se encuentran en la carpeta *Tranvia*.

El modo de funcionamiento deseado es el siguiente:

- Si no hay tranvía en el cruce, los semáforos funcionarán de forma normal: El semáforo SC estará 45 segundos en rojo, 3 segundos en amarillo y 60 segundos en verde. El de peatones estará sincronizado con el anterior, se modo que estará en rojo mientras SC esté en verde o amarillo y en verde cuando esté en rojo SC. Los últimos 10 segundos de luz verde del semáforo de peatones será intermitente.
- Cuando llegue un tranvía, el semáforo de vehículos SC se pondrá en rojo y el de peatones en verde. Esta situación se mantendrá hasta 10 segundos después de que el tranvía salga del cruce, aunque durante estos 10 segundos el de

peatones estará intermitente. Pasado este tiempo, el semáforo de vehículos se pondrá en verde y comenzará el funcionamiento normal descrito en a).

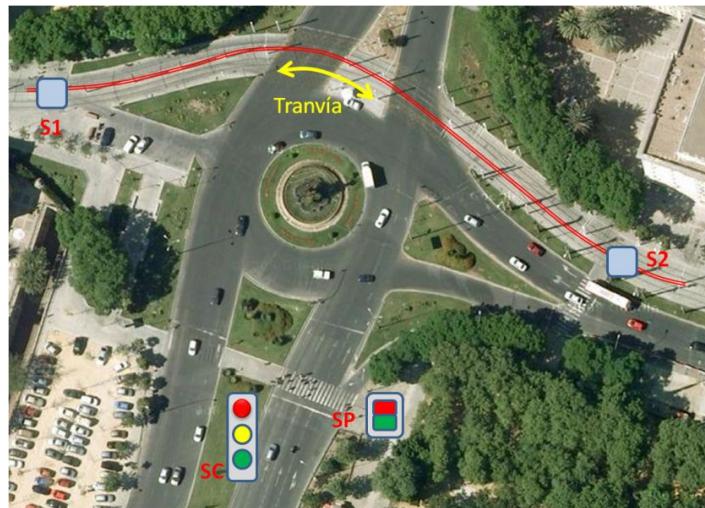


Figura 13-1 – Cruce con tranvía

## 13.2. Cruce completo con tranvía

Para el mismo cruce del ejercicio anterior, se va a realizar el control completo del cruce, para lo que será necesario actuar sobre 4 semáforos, cada uno de ellos con las tres luces para coches y las dos para peatones, tal como se muestra en la Tabla 13-2. Los semáforos 1, 3 y 4 controlan la entrada en la rotonda, mientras que el 2 es de salida de la rotonda. Para las otras dos salidas de la rotonda no se utilizarán semáforos. Los ficheros para realizar el ejercicio están en la carpeta *Tranvía2*.

El programador tiene libertad absoluta para diseñar este sistema de control tanto en el modo de funcionamiento como en los lenguajes de programación escogidos, de forma que se comporte como es habitual en un cruce de estas características.

Las entradas y salidas del sistema están listadas en Tabla 13-1 y Tabla 13-2 respectivamente.

Tabla 13-1. Variables de entrada del cruce completo con tranvía

Nombre	Tipo	Descripción
Arranque	BOOL	Señal de activación del sistema
S1	BOOL	Sensor de presencia del tranvía 1
S2	BOOL	Sensor de presencia del tranvía 2

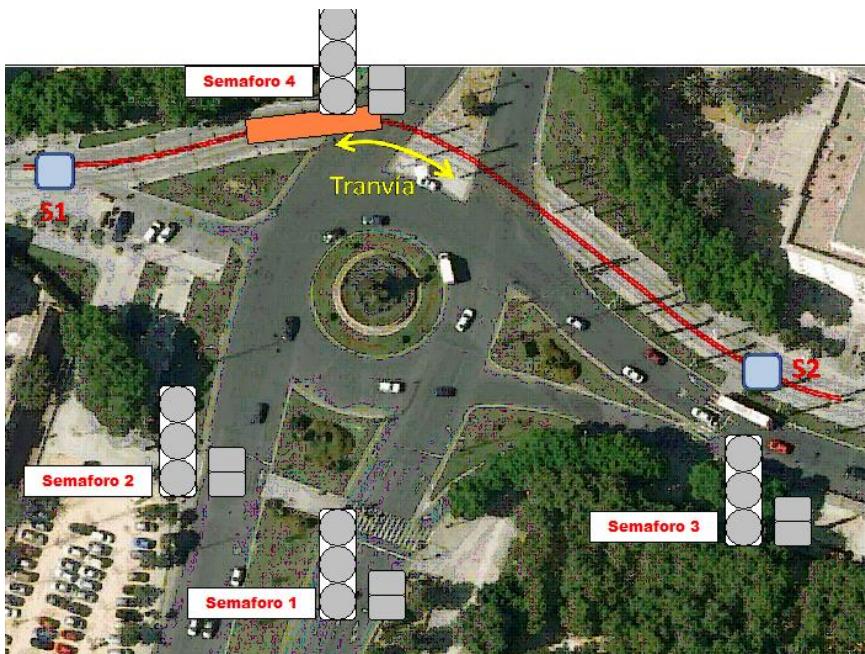


Figura 13-2 – Cruce completo con tranvía

Tabla 13-2. Variables de salida del cruce completo con tranvía

Nombre	Tipo	Descripción
<i>Sem1CRojo</i>	BOOL	Luz Roja Semáforo Vehículos 1
<i>Sem1CAmbar</i>	BOOL	Luz Ámbar Semáforo Vehículos 1
<i>Sem1CVerde</i>	BOOL	Luz Verde Semáforo Vehículos 1
<i>Sem1PRojo</i>	BOOL	Luz Roja Semáforo Peatones 1
<i>Sem1PVerde</i>	BOOL	Luz Verde Semáforo Peatones 1
<i>Sem2CRojo</i>	BOOL	Luz Roja Semáforo Vehículos 2
<i>Sem2CAmbar</i>	BOOL	Luz Ámbar Semáforo Vehículos 2
<i>Sem2CVerde</i>	BOOL	Luz Verde Semáforo Vehículos 2
<i>Sem2PRojo</i>	BOOL	Luz Roja Semáforo Peatones 2
<i>Sem2PVerde</i>	BOOL	Luz Verde Semáforo Peatones 2
<i>Sem3CRojo</i>	BOOL	Luz Roja Semáforo Vehículos 3
<i>Sem3CAmbar</i>	BOOL	Luz Ámbar Semáforo Vehículos 3
<i>Sem3CVerde</i>	BOOL	Luz Verde Semáforo Vehículos 3
<i>Sem3PRojo</i>	BOOL	Luz Roja Semáforo Peatones 3
<i>Sem3PVerde</i>	BOOL	Luz Verde Semáforo Peatones 3
<i>Sem4CRojo</i>	BOOL	Luz Roja Semáforo Vehículos 3
<i>Sem4CAmbar</i>	BOOL	Luz Ámbar Semáforo Vehículos 3
<i>Sem4CVerde</i>	BOOL	Luz Verde Semáforo Vehículos 3
<i>Sem4PRojo</i>	BOOL	Luz Roja Semáforo Peatones 3
<i>Sem4PVerde</i>	BOOL	Luz Verde Semáforo Peatones 3

### 13.3. Automatización con Guía GEMMA

El objetivo de este ejercicio es diseñar los mecanismos de puesta en marcha-parada, modo manual-automático y paradas de emergencia de un proceso. Para ello se va a utilizar la metodología de la guía GEMMA [12].

El trabajo se va a realizar fundamentalmente utilizando GRAFCET, de modo que se diseñará un GRAFCET general en el que aparezcan los estados necesarios de la guía GEMMA y que permitirá acceder a los distintos modos: modo automático, modo manual, modo de emergencia, etc., que normalmente también estarán escritos en GRAFCET.

En el directorio GEMMA que se suministra está el fichero *.pro* y algunos gráficos que deben estar en el mismo directorio que el fichero *.pro* para una correcta visualización del ejercicio.

### 13.3.1. Proceso a controlar

El proceso a controlar es muy simple, de modo que en este ejercicio el trabajo se pueda centrar en los modos de funcionamiento.

Se trata de un mecanismo simple de movimiento de piezas mediante dos cilindros, de modo que cada uno de ellos tiene dos fines de carrera y dos acciones posibles: extender o retroceder el émbolo. El proceso se muestra en la Figura 13-3.

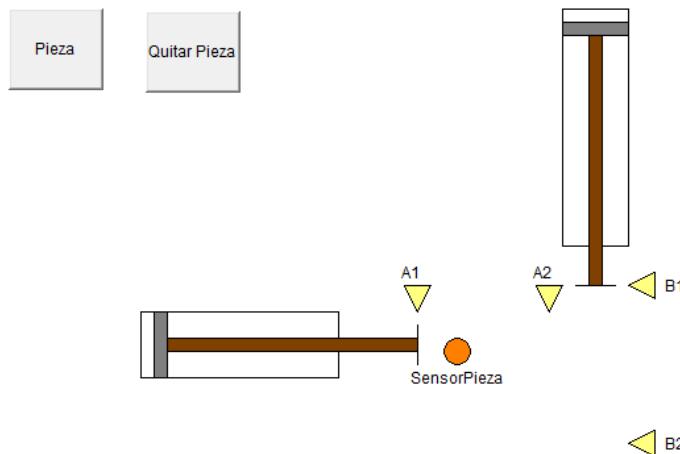


Figura 13-3 – Proceso a controlar en Automatización con Guía GEMMA

La alimentación de las piezas es manual por parte de un operario, para lo que la animación denominada *proceso* tiene un botón *Pieza* que colocará una pieza en la posición inicial. Si a lo largo del proceso fuera necesario retirar manualmente una pieza, se utilizará el botón *Quitar Pieza*. Recuérdese que estos dos botones no son entradas del automatismo sino que se usarán para simular acciones directas del operario.

El funcionamiento normal en modo automático será el siguiente: Cuando el operario ponga una pieza sobre *SensorPieza* el cilindro 1 se extenderá desde el fin de carrera *A1* hasta *A2*. En ese momento y simultáneamente se extenderá el cilindro 2 desde *B1* hasta *B2* y retrocederá el cilindro 1. Finalmente retrocederá el cilindro 2, volviendo a la posición inicial.

Además dispondrá de un modo manual, en el que se podrán mover los dos cilindros hacia delante y atrás con 4 botones.

El sistema incluye los dos sensores siguientes:

- *SensorPieza*: TRUE si hay una pieza en la posición inicial
- *PiezaEnSistema*: TRUE si hay alguna pieza dentro del proceso.

Para el control del sistema se ha diseñado un cuadro de control, representado en la Figura 13-4.

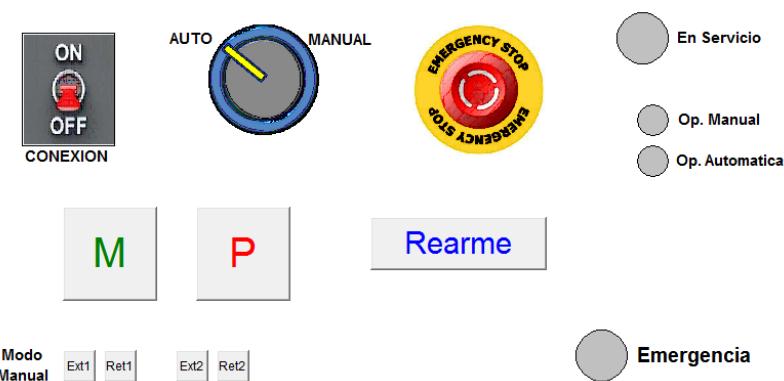


Figura 13-4 – Cuadro de control en Automatización con Guía GEMMA

El citado cuadro de control consta de los siguientes elementos:

- Un interruptor de conexión general (*CONEXION*)
- Un selector de modo automático o manual (*MANUAL*). TRUE modo manual, FALSE modo automático.
- Pulsadores de puesta en marcha o parada (*M* y *P*). Se utilizan tanto en modo manual como automático.
- Pulsador de Emergencia (*EMERGENCIA*).
- Pulsador de rearme (*REARME*). Se utiliza para volver al funcionamiento normal después de una emergencia.
- Botones de modo manual (*BotonExt1*, *BotonExt2*, *BotonRet1*, *BotonRet2*). Mueven los dos cilindros en los dos sentidos.
- Señales luminosas de conexión, en operación manual, en operación automática y estado de emergencia (*EnServicio*, *EnOperacionManual*, *EnOperacionAuto*, *LuzEmergencia*)

### 13.3.2. Modos de funcionamiento

El sistema de control deberá contemplar los modos de funcionamiento que se indican en la Figura 13-5.

#### 13.3.2.1. Arranque

El sistema estará inicialmente en “Estado de Desconexión”, del que se saldrá mediante el interruptor general (*CONEXION*). En primer lugar realizará un proceso de arranque (A6 en la guía GEMMA, tal como se muestra en la Figura 13-5) que consistirá en llevar los dos cilindros a su posición inicial. Además no deberá salir de este modo mientras haya alguna pieza en el sistema que no esté en la situación inicial (*PiezaEnSistema*=TRUE y *SensorPieza*=FALSE).

Una vez cumplidas estas condiciones, el sistema pasará al estado de “Listo” (A1 de la guía)

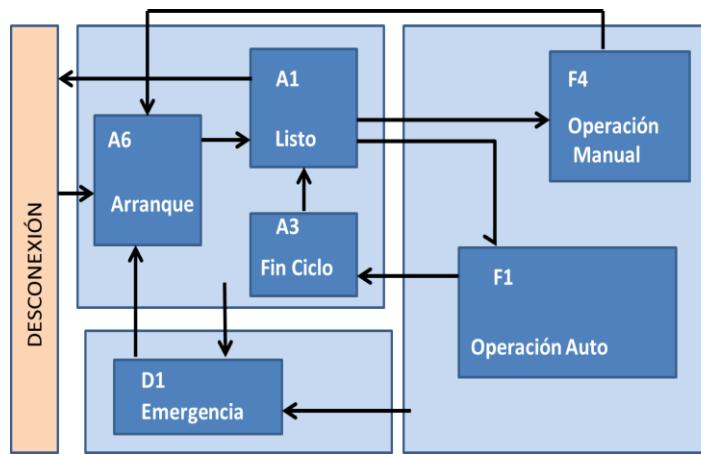


Figura 13-5 – Modos de funcionamiento

### 13.3.2.2. Modo Automático

Si el sistema está en el estado “Listo” y el selector en “AUTO”, al pulsar *M* entramos en el estado de “Operación Auto” (F1) con el funcionamiento anteriormente descrito. Al pulsar *P*, el sistema se detendrá pero no inmediatamente, sino que se terminará el movimiento de la pieza que en ese momento esté en el sistema (A3) y se volverá al estado “Listo”. La luz correspondiente a “Operación Automática” estará encendida cuando el sistema está en este modo.

### 13.3.2.3. Modo Automático

Si estamos en el estado “Listo” y el selector en “MANUAL”, al pulsar *M* entramos en el estado de “Operación Manual” (F4) en el que se podrán mover los dos cilindros con los botones correspondientes. Al pulsar *P*, se volverá al estado “Arranque” (A6) para asegurarnos que los cilindros vuelven a su posición original. La luz correspondiente a “Operación Automática” estará encendida.

### 13.3.2.4. Modo Emergencia

Si desde cualquier estado (menos el de desconexión) se pulsa la seta de emergencia,

el sistema inmediatamente pasará a un estado de Emergencia (D1) en el que se interrumpe toda actividad.

De este estado se saldrá mediante la pulsación del botón de rearme, que nos llevará al estado de Arranque.

### **13.3.3. Desarrollo del ejercicio**

El resultado final del ejercicio será un GRAFCET a nivel superior cuyas etapas sirvan para representar los estados de la guía GEMMA de la figura anterior. Dentro de cada una de esas etapas, y mediante acciones, se llamará a la actividad que se realiza en cada uno de esos estados, y que se implementarán en GRAFCET o cualquier otro lenguaje de los autómatas.

La secuencia que se sugiere para el desarrollo del ejercicio será la siguiente:

- Desarrollo de las acciones asociadas a las actividades principales (F1, F4 y A6)
- Diseño e implementación del GRAFCET principal (programa GEMMA)
- Diseño e implementación del sistema de gestión de alarmas.

#### **13.3.3.1. Diseño del modo automático (F1)**

Diseñe e implemente un programa en GRAFCET que denominaremos *OperaciónAuto*, que realice el funcionamiento automático del sistema. Para comprobar su funcionamiento haga la llamada en el programa principal (*OperacionAuto();*)

#### **13.3.3.2. Diseño del modo manual (F4)**

Diseñe e implemente un programa en GRAFCET que denominaremos *OperaciónManual*, que realice el funcionamiento manual del sistema. Para comprobar su funcionamiento haga la llamada en el programa principal (*OperacionManual();*)

#### **13.3.3.3. Diseño del modo manual (A3)**

Diseñe e implemente un programa en el lenguaje que se prefiera (se recomienda LD) que denominaremos *OperacionArranque*, que realice el arranque descrito en la

sección anterior.

#### 13.3.3.4. Diseño de GEMMA sin gestión de alarmas

Diseñe e implemente un programa en GRAFCET denominado GEMMA que modele los distintos estados de la guía GEMMA. Este será el programa al que habrá que llamar desde PLC\_PRG.

Para el diseño de este GRAFCET habrá que decidir cuáles son las condiciones lógicas que deben ir en las distintas transiciones.

Las distintas etapas deberán incluir como acciones las llamadas a los programas escritos en las tres secciones anteriores. Estas acciones se pueden escribir en cualquier lenguaje aunque se recomienda hacerlo en ST. Por ejemplo, el código asociado a la acción del estado de modo automático (F1) será simplemente:

```
OperacionAuto();
```

En los estados donde sea necesario activar alguna señal luminosa, se incluirán acciones adicionales que activen las correspondientes señales.

#### 13.3.3.5. Gestión de estado de alarma

La gestión del estado de alarmas presenta dificultades adicionales. Cuando se pulse la seta de emergencia, el sistema deberá parar su actividad y pasar al estado de emergencia. En definitiva, habría que prever que en todas y cada una de las etapas que diseñemos una alternativa (una bifurcación) cuando se pulse la seta. Esto no es complicado en el programa principal *GEMMA*, pero complica tremadamente el diseño del resto de los grafcets (*OperacionManual* y *OperacionAuto*).

Para simplificar esta gestión vamos a utilizar el flag *SFCReset* asociado a cada programa GRAFCET que ofrece CoDeSys (soluciones similares hay en otros entornos). Este flag lo que va a permitir es reiniciar el GRAFCET, y por tanto marcará sólo el estado inicial y dejará de ejecutar todas las acciones que estuviera realizando.

Para utilizar el flag es necesario definirlo como variable de entrada en la zona de declaración del programa en GRAFCET, es decir:

---

```
PROGRAM OperacionAuto
VAR_INPUT
    SFCReset:BOOL;
END_VAR
```

---

A continuación, la llamada se realizará con un operando, que puede ser TRUE o FALSE, esto es:

---

```
OperaciónAuto (SFCReset:=FALSE) ;
    (* Ejecuta el Grafset de la forma normal *)
OperaciónAuto (SFCReset:=TRUE) ;
    (* Reinicia el GRAFCET *)
```

---

De este modo, cuando se detecte una situación de emergencia se hará la llamada con la opción TRUE y en cualquier otro caso con FALSE.

Por ejemplo, si las acciones las escribimos en ST, el código que habrá que escribir en la acción del modo automático será algo similar a:

---

```
IF EMERGENCIA=FALSE THEN
    OperaciónAuto (SFCReset:=FALSE) ;
ELSE
    OperaciónAuto (SFCReset:=TRUE) ;
END_IF
```

---

Esto será necesario hacerlo en los GRAFCET de *OperacionAuto* y *OperacionManual*. Por otro lado en el programa GEMMA definiremos una etapa de Emergencia y transiciones hasta ésta desde todas las etapas desde las que fuera necesario.

### 13.3.4. Lista de señales

Las siguientes tablas definen todas las entradas y salidas del sistema completo.

Tabla 13-3. Variables de entrada de Automatización con guía GEMMA

Nombre	Tipo	Descripción
A1	BOOL	Fin de carrera cilindro 1 recogido
A2	BOOL	Fin de carrera cilindro 1 extendido
B1	BOOL	Fin de carrera cilindro 2 recogido
B2	BOOL	Fin de carrera cilindro 2 extendido
<i>SensorPieza</i>	BOOL	Presencia de pieza en la posición inicial
<i>PiezaEnSistema</i>	BOOL	Presencia de pieza en cualquier parte del sistema
CONEXION	BOOL	Señal del interruptor general de conexión
MANUAL	BOOL	Selector de modo. TRUE Manual, FALSE Automatico
M	BOOL	Señal del pulsador M. Para entrar en modo de operación manual o automático
P	BOOL	Señal del pulsador P. Para salir del modo de operación manual o automático
EMERGENCIA	BOOL	Seta de Emergencia. TRUE si está pulsado
<i>Rearme</i>	BOOL	Señal del pulsador de rearme
<i>BotonExt1</i>	BOOL	Interruptor para avanzar cilindro 1 (en manual)
<i>BotonRet1</i>	BOOL	Interruptor para retroceder cilindro 1 (en manual)
<i>BotonExt2</i>	BOOL	Interruptor para avanzar cilindro 2 (en manual)
<i>BotonRet2</i>	BOOL	Interruptor para retroceder cilindro 2 (en manual)

Tabla 13-4. Variables de salida de Automatización con guía GEMMA

Nombre	Tipo	Descripción
<i>Cil1Ext</i>	BOOL	Movimiento de extensión del cilindro 1
<i>Cil1Ret</i>	BOOL	Movimiento de retroceso del cilindro 1
<i>Cil2Ext</i>	BOOL	Movimiento de extensión del cilindro 2
<i>Cil2Ret</i>	BOOL	Movimiento de retroceso del cilindro 2
<i>EnServicio</i>	BOOL	Señal luminosa de sistema “En Servicio”
<i>EnOperacionManual</i>	BOOL	Señal luminosa de sistema “En Operación Manual”
<i>EnOperacionAuto</i>	BOOL	Señal luminosa de sistema “En Operación Auto”
<i>LuzEmergencia</i>	BOOL	Señal luminosa de sistema en estado de emergencia

### 13.4. Control de un ascensor

El objetivo es diseñar un sistema de control para un ascensor de cuatro plantas, tal como se ve en la Figura 13-6. El ejercicio se resolverá en el fichero *AscensorCompleto.pro*.

El funcionamiento será el de un ascensor convencional, quedando a criterio del programador desarrollar un algoritmo de control más o menos complejo.

El ascensor dispone de un pulsador de llamada colocado en cada una de las plantas (*P<sub>i</sub>*) y de pulsadores dentro de la cabina para indicar la planta a la que se desea ir (*C<sub>i</sub>*). Además, en cada planta hay un sensor de presencia que se activa cuando la cabina se encuentra en cada una de las plantas (*S<sub>i</sub>*).

El motor del ascensor se puede mover en las dos direcciones mediante las señales *Motor\_arriba* y *Motor\_Abajo*. Adicionalmente, si se desea implementar algún tipo de memoria de los pulsadores que se han pulsado, se dispone también de una señal luminosa asociada a cada pulsador (*LuzPi* y *LuzCi*). El encendido y apagado de

estas luces debe ser realizado por el sistema de control.

La programación se puede realizar en cualquier lenguaje.

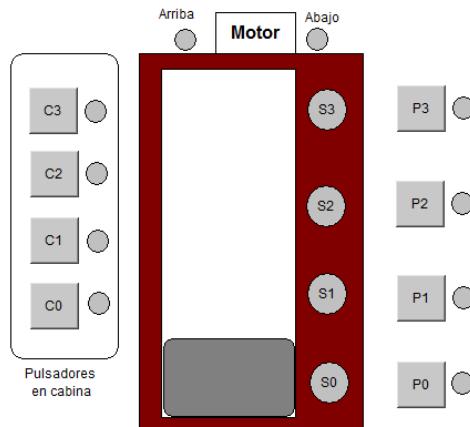


Figura 13-6 – Ascensor

#### 13.4.1. Lista de señales

Tabla 13-5. Variables de entrada del ascensor

Nombre	Tipo	Descripción
S0	BOOL	Sensor de posición Planta 0
S1	BOOL	Sensor de posición Planta 1
S2	BOOL	Sensor de posición Planta 2
S3	BOOL	Sensor de posición Planta 3
P0	BOOL	Pulsador exterior para llamada desde Planta 0
P1	BOOL	Pulsador exterior para llamada desde Planta 1
P2	BOOL	Pulsador exterior para llamada desde Planta 2
P3	BOOL	Pulsador exterior para llamada desde Planta 3
C0	BOOL	Pulsador en cabina para moverse a Planta 0

C1	BOOL	Pulsador en cabina para moverse a Planta 1
C2	BOOL	Pulsador en cabina para moverse a Planta 2
C3	BOOL	Pulsador en cabina para moverse a Planta 3

Tabla 13-6. Variables de salida del ascensor

Nombre	Tipo	Descripción
<i>Motor_arriba</i>	BOOL	Activa el movimiento del motor hacia arriba
<i>Motor_abajo</i>	BOOL	Activa el movimiento del motor hacia abajo
<i>LuzP0</i>	BOOL	Piloto luminoso junto a P0
<i>LuzP1</i>	BOOL	Piloto luminoso junto a P1
<i>LuzP2</i>	BOOL	Piloto luminoso junto a P2
<i>LuzP3</i>	BOOL	Piloto luminoso junto a P3
<i>LuzC0</i>	BOOL	Piloto luminoso junto a C0
<i>LuzC1</i>	BOOL	Piloto luminoso junto a C1
<i>LuzC2</i>	BOOL	Piloto luminoso junto a C2
<i>LuzC3</i>	BOOL	Piloto luminoso junto a C3

### 13.5. Control de dos ascensores simples

Se desea diseñar un sistema de control para un edificio con dos ascensores, de modo que la llamada es única, por lo que se deberá decidir cuál de los dos acude. El edificio es de cuatro plantas, tal como se muestra en la Figura 13-7. El ejercicio se realizará con el fichero *AscensorBasicoDoble.pro*.

Cada uno de los ascensores dispone de un sensor de presencia para cada una de las plantas (*Si\_1* para el ascensor 1 y *Si\_2* para el ascensor 2) y de un pulsador de llamada colocado en cada una de las plantas (*Pi*).

Cada uno de los ascensores dispone de un motor que se puede mover en las dos

direcciones mediante las señales *Motor\_arriba\_i* y *Motor\_Abajo\_i*.

El algoritmo para determinar el ascensor que acude en cada momento, lo determinará libremente el programador.

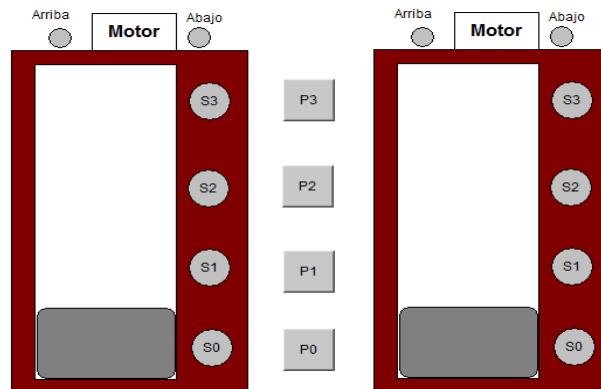


Figura 13-7 – Ascensor doble

### 13.5.1. Lista de señales

Tabla 13-7. Variables de salida del ascensor

Nombre	Tipo	Descripción
<i>Motor_arriba_1</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 1)
<i>Motor_abajo_1</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 1)
<i>Motor_arriba_2</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 2)
<i>Motor_abajo_2</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 2)

Tabla 13-8. Variables de entrada del ascensor doble

Nombre	Tipo	Descripción
<i>S0_1</i>	BOOL	Sensor de posición Planta 0 del ascensor 1
<i>S1_1</i>	BOOL	Sensor de posición Planta 1 del ascensor 1
<i>S2_1</i>	BOOL	Sensor de posición Planta 2 del ascensor 1
<i>S3_1</i>	BOOL	Sensor de posición Planta 3 del ascensor 1
<i>S0_2</i>	BOOL	Sensor de posición Planta 0 del ascensor 2
<i>S1_2</i>	BOOL	Sensor de posición Planta 1 del ascensor 2
<i>S2_2</i>	BOOL	Sensor de posición Planta 2 del ascensor 2
<i>S3_2</i>	BOOL	Sensor de posición Planta 3 del ascensor 2
<i>P0</i>	BOOL	Pulsador exterior para llamada desde Planta 0
<i>P1</i>	BOOL	Pulsador exterior para llamada desde Planta 1
<i>P2</i>	BOOL	Pulsador exterior para llamada desde Planta 2
<i>P3</i>	BOOL	Pulsador exterior para llamada desde Planta 3

## 13.6. Control de dos ascensores completos

Se trata de diseñar un sistema de control para dos ascensores de cuatro plantas con llamada exterior compartida para los dos ascensores y cada uno de ellos con una botonera en el interior de la cabina. El sistema se muestra en la Figura 13-8. Para realizar el ejercicio se usará el fichero *AscensorCompletoDoble.pro*.

Al igual que en los casos anteriores, el programador tendrá libertad para determinar las especificaciones de funcionamiento y la complejidad de los algoritmos.

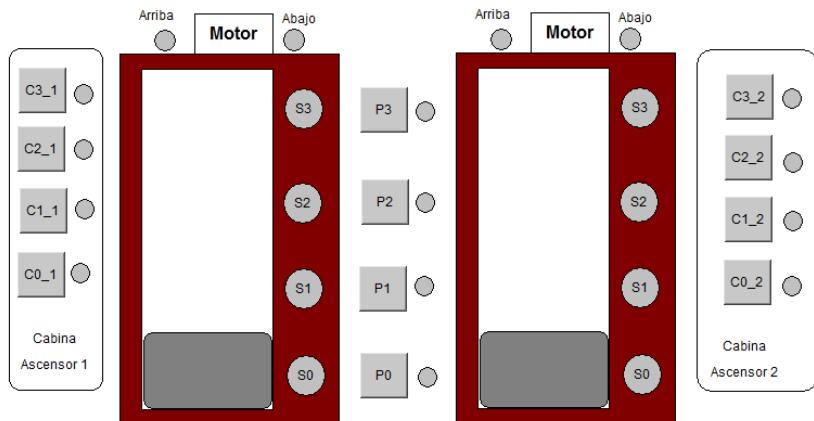


Figura 13-8 – Ascensor doble completo

### 13.6.1. Lista de señales

Tabla 13-9. Variables de salida del ascensor doble completo

Nombre	Tipo	Descripción
<i>Motor_arriba_1</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 1)
<i>Motor_abajo_1</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 1)
<i>Motor_arriba_2</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 2)
<i>Motor_abajo_2</i>	BOOL	Activa el movimiento del motor hacia arriba (ascensor 2)
<i>LuzP0</i>	BOOL	Piloto luminoso junto a P0
<i>LuzP1</i>	BOOL	Piloto luminoso junto a P1

Tabla 13-10. Variables de entrada del ascensor doble completo

Nombre	Tipo	Descripción
S0_1	BOOL	Sensor de posición Planta 0 del ascensor 1
S1_1	BOOL	Sensor de posición Planta 1 del ascensor 1
S2_1	BOOL	Sensor de posición Planta 2 del ascensor 1
S3_1	BOOL	Sensor de posición Planta 3 del ascensor 1
S0_2	BOOL	Sensor de posición Planta 0 del ascensor 2
S1_2	BOOL	Sensor de posición Planta 1 del ascensor 2
S2_2	BOOL	Sensor de posición Planta 2 del ascensor 2
S3_2	BOOL	Sensor de posición Planta 3 del ascensor 2
P0	BOOL	Pulsador exterior para llamada desde Planta 0
P1	BOOL	Pulsador exterior para llamada desde Planta 1
P2	BOOL	Pulsador exterior para llamada desde Planta 2
P3	BOOL	Pulsador exterior para llamada desde Planta 3
C0_1	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 0
C1_1	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 1
C2_1	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 2
C3_1	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 3
C0_2	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 0
C1_2	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 1
C2_2	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 2
C3_2	BOOL	Pulsador en cabina de ascensor 1 para moverse a Planta 3

### 13.7. Control de una mezcladora

La base del proceso es el realizado en *Cap11\_17.pro*. En este ejercicio, el objetivo

consiste en realizar una automatización completa usando la Guía GEMMA.

El proceso de la Figura 13-9 consiste en mezclar cantidades exactas de dos productos con agua y mezclarlas. El sistema consta de dos tanques superiores, respectivamente con los productos A y B. Un tanque intermedio en el que se mezclarán los dos productos anteriores y un tanque inferior de agua que dispone de un agitador.

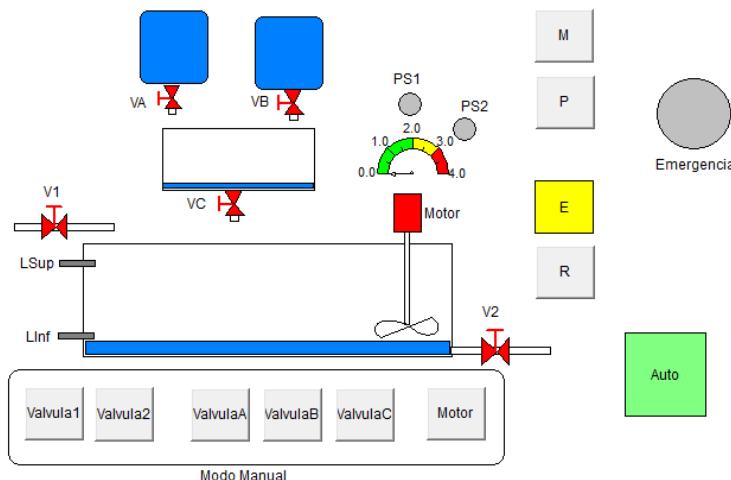


Figura 13-9 – Mezcladora con Guía GEMMA

El sistema puede estar en dos modos: MANUAL y AUTO, que se cambia con el selector Auto. Si está en modo MANUAL se podrán operar las válvulas y el motor mediante los interruptores del recuadro “Modo Manual” (Variables de entrada *MV1, MV2, MVA, MVB, MVC y MMotor*)

En modo AUTO deberá realizar el siguiente proceso básico:

- Llenar el tanque de agua hasta el nivel *LSup*
- En paralelo con lo anterior se llenará el depósito intermedio de A y B de la siguiente manera: En primer lugar se llenará de producto A abriendo *VA* hasta que el peso llegue a 2 (se activará la señal *PS1*). En ese momento se cierra la válvula A y se abre la B (señal *VB*) para continuar con el llenado hasta llegar a

un peso de 3 (Se activará la señal  $PS2$ ), momento en el que se cerrará la válvula B.

- En el momento en que las dos operaciones anteriores estén realizadas, se abrirá la válvula C para verter la mezcla en el depósito inferior. Esta válvula estará abierta un tiempo suficiente para vaciar el depósito. Dicho tiempo lo fijará el programador.
- Activar el agitador mediante la señal *Motor* durante 10 segundos.
- Vaciar el contenido del depósito inferior hasta llegar al nivel  $Linf$ .

En modo AUTO se pondrá en marcha con el pulsador  $M$ . La parada se realiza con el pulsador  $P$  que se podrá pulsar en cualquier momento pero se terminará el ciclo del proceso que se esté realizando antes de detenerse. Los pulsadores  $M$  y  $P$  solo se usan en modo AUTO.

Tanto en modo AUTO como MANUAL, en caso de emergencia se pulsará el interruptor *Emergencia*. En ese momento se deberán cerrar todas las válvulas y apagar el motor. También se encenderá la señal luminosa de alarma *LuzEm*. En el estado de emergencia, ninguna de las actuaciones estará habilitada, salvo el botón de rearme.

Para salir del estado de emergencia se deberá pulsar el rearme (*Rearme*). En ese momento se deberá llevar el sistema a la posición inicial, es decir, los depósitos vacíos y todas las válvulas cerradas. Una vez realizada esta operación el sistema volverá a operar con normalidad y se apagará la señal de emergencia.

### 13.7.1. Lista de señales

Tabla 13-11. Variables de entrada del proceso de mezcla

Nombre	Tipo	Descripción
<i>M</i>	BOOL	Pulsador de puesta en marcha
<i>LInf</i>	BOOL	Sensor nivel bajo en depósito inferior
<i>LSup</i>	BOOL	Sensor nivel alto en depósito inferior
<i>PS1</i>	BOOL	Señal de peso igual a 2 alcanzado en depósito intermedio
<i>PS2</i>	BOOL	Señal de peso igual a 3 alcanzado en depósito intermedio
<i>P</i>	BOOL	Pulsador de parada
<i>Auto</i>	BOOL	Modo AUTO=TRUE, modo MANUAL=FALSE
<i>Emergencia</i>	BOOL	Interruptor de Emergencia
<i>Rearme</i>	BOOL	Pulsador de rearme
<i>MV1</i>	BOOL	Interruptor de apertura en modo manual válvula 1
<i>MV2</i>	BOOL	Interruptor de apertura en modo manual válvula 2
<i>MVA</i>	BOOL	Interruptor de apertura en modo manual válvula A
<i>MVB</i>	BOOL	Interruptor de apertura en modo manual válvula B
<i>MVC</i>	BOOL	Interruptor de apertura en modo manual válvula C
<i>MMotor</i>	BOOL	Interruptor puesta en marcha en modo manual Motor

Tabla 13-12. Variables de salida del proceso de mezcla

Nombre	Tipo	Descripción
<i>Valvula1</i>	BOOL	Válvula de entrada a depósito inferior
<i>Valvula2</i>	BOOL	Válvula de salida de depósito inferior
<i>ValvulaA</i>	BOOL	Válvula de salida depósito de producto A
<i>ValvulaB</i>	BOOL	Válvula de salida depósito de producto B
<i>ValvulaC</i>	BOOL	Válvula de salida depósito intermedio
<i>Motor</i>	BOOL	Motor del agitador
<i>LuzEm</i>	BOOL	Luz de emergencia



# Apéndice A

## CONFIGURACIÓN DE CoDeSys

---

**E**n este apéndice se verán algunos aspectos básicos de configuración de CoDeSys. Evidentemente, el tema más importante de la configuración será la selección del tipo de hardware, tarjetas de entrada/salida, sistemas de comunicaciones, etc. Sin embargo, ya que el objetivo de este libro es la programación básica y CoDeSys se usará sólo en modo simulación, no se va a detallar la forma de realizar esta configuración. Para más información sobre esta configuración, se remite al lector al Manual de Usuario de CoDeSys [9]. Sin embargo, se describirán otros aspectos de configuración que pueden ser útiles para los ejemplos tratados en este libro.

### A.1. Gestión de librerías

Cada proyecto en CoDeSys puede tener asociado una serie de librerías donde están definidos un conjunto de POU (normalmente bloques funcionales o funciones), variables globales, tipos de datos o visualizaciones. La incorporación de nuevas librerías se realiza desde el Gestor de Librerías (*Library Manager*). El comando para abrir el Gestor de Librerías es:

Menú *Window -> Library Manager*

La ventana del gestor se muestra en la Figura A-1. En la parte superior izquierda aparecen las librerías incluidas en el proyecto, debajo, los bloques funcionales incluidos en la librería seleccionada, y a la derecha la cabecera y la representación del bloque funcional seleccionado.

Para incluir una nueva librería, con el Library Manager abierto, se seleccionará el menú *Insert*.

*Insert ->Additional Library*

A continuación se selecciona la librería requerida. Por defecto, estas librerías se encuentran en la carpeta *Codesys V2.3/Library*.

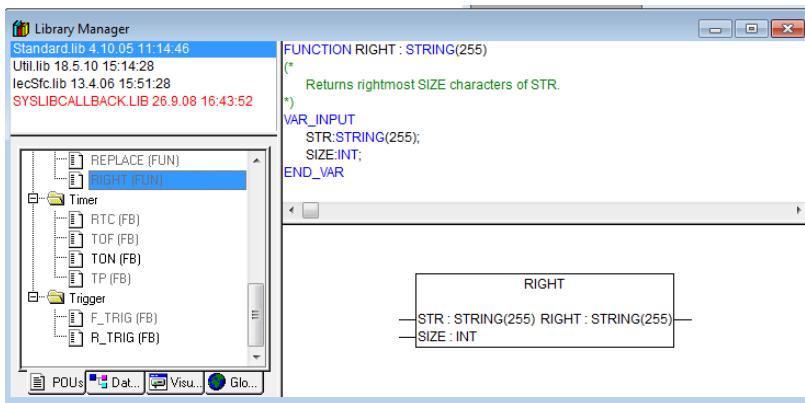


Figura A-1 – Ventana para Gestión de Librerías

Las librerías más utilizadas son:

- **Standard.lib:** Siempre está disponible y contiene los bloques funcionales y funciones básicos para la norma IEC-61131-3
- **Util.lib:** Entre otros contiene bloques funcionales para controladores PIDs, intermitentes (BLINK), cálculo de integral y derivada de una señal, monitorización de límites de alarmas,...
- **IecSfc.lib:** Necesaria para utilizar GRAFCET cumpliendo la norma IEC.

El usuario también puede crear sus propias librerías sin más que guardar su Proyecto como una librería (Con *Save as...*). Una vez hecho, la librería creada se puede añadir a cualquier otro proyecto y los bloques funcionales serán reutilizables.

## A.2. Configuración de tareas

Como se ha descrito en este texto, por defecto Codesys ejecuta en cada ciclo del autómata el programa denominado PLC\_PRG, de forma que todos los programas

que se deseen ejecutar en el ciclo se deberán llamar directa o indirectamente desde éste.

Sin embargo, la forma en que se ejecuten los programas puede ser mucho más compleja, pudiéndose ejecutar programas de forma periódica, cíclica (la opción por defecto en PLC\_PRG) o cuando ocurra un cierto evento. Esto se puede realizar mediante la configuración de las tareas de CoDeSys (*Tasks*).

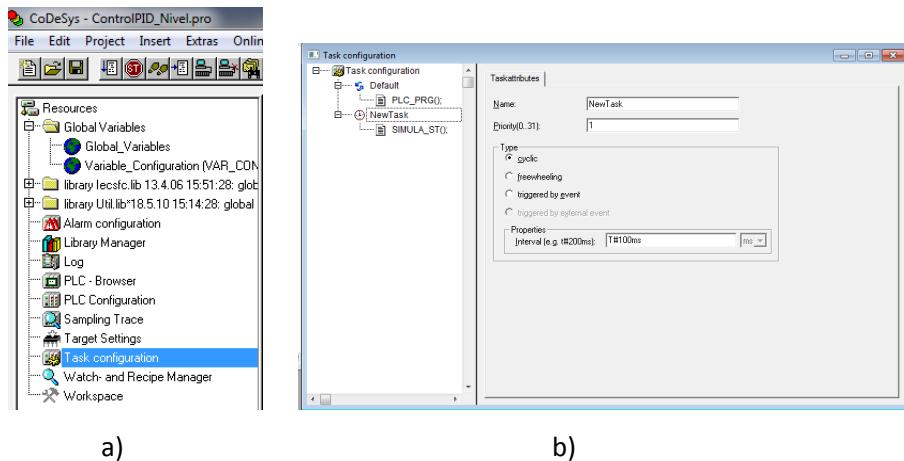


Figura A-2 – Ventanas de configuración de tareas

Una *Tarea* es una forma de especificar la forma de ejecución del programa o programas asociados, es decir, modo de ejecución, prioridad, etc. Cada tarea va a estar identificada por un identificador (nombre de la tarea).

Para acceder a la configuración de tareas, seleccionar la pestaña *Resources* del Organizador de Objetos y escoger la opción *Task Configuration*. (Figura A-2-a) Al hacerlo se abrirá una ventana de configuración (Figura A-2-b) donde a la izquierda aparece un árbol de las tareas existentes con el nombre, el tipo y los programas que tiene asignado y a la derecha los parámetros de configuración de la tarea que se seleccione en el lado izquierdo. En la figura aparecen dos tareas, la primera denominada *Default* que tiene asociado el programa *PLC\_PRG*, y la segunda denominada *New Task*, que tiene asociado el programa *SIMULA\_ST*. El ícono que tiene asociada la tarea indica el modo de ejecución de la tarea.

Los modos de ejecución de las tareas en CoDeSys son:

- **Cíclica:** La tarea se ejecutará cada cierto tiempo que se le indica como parámetro en el campo *Interval* (ver figura anterior). En otros entornos de programación a este tipo de tareas se le denomina periódica.
- **Freewheeling:** La tarea de ejecuta de forma cíclica, de forma que cuando termine la última línea de código, comenzará a ejecutarse de nuevo la primera. En este caso no se establece cada cuanto tiempo se ejecutará la tarea, ya que depende del número de líneas y de la complejidad del código. Es el tipo por defecto en CoDeSys.
- **Lanzada por evento:** Se ejecuta cada vez que en la variable que se le indique en el campo *Event* se detecte un flaco de subida. Esto se puede utilizar por ejemplo, cuando lanzar un programa cuando una alarma se active.
- **Lanzada por evento externo:** Similar a la anterior, pero en este caso, el evento es del sistema. Esta opción depende de la configuración del PLC y puede no estar disponible.

Además del tipo, cada tarea tiene los siguientes atributos:

- **Nombre:** Identificador asignado a la tarea.
- **Prioridad:** Valor entre 0 y 31, donde 0 es la prioridad más alta y 31 la más baja. Esta prioridad marcará el orden en que se ejecutarán las tareas cuando varias estén listas para ejecutarse.
- **Watchdog:** Sirva para asegurarse que una determinada tarea va a tardar en ejecutarse menos de un cierto tiempo que se puede preestablecer. Si la tarea tarda en ejecutarse más que dicho tiempo, se genera un mensaje de error. Los parámetros que se pueden asignar para establecer el Watchdog son:
  - *Tiempo:* Tiempo límite de ejecución
  - *Sensibilidad:* Es un número entero que indica el número de veces que se debe violar la restricción de tiempo para que se genere un error.

Nota: Esta última opción puede estar accesible o no dependiendo de la configuración que se haya establecido.

Para crear nuevas tareas, eliminarlas, añadir programas a las tareas, etc., se accederá al menú contextual (botón derecho del ratón).



# Apéndice B

## VISUALIZACIONES EN CoDeSys

---

**L**a herramienta de visualización en CoDeSys permite la monitorización de datos y la asignación de valores a entradas desde el teclado o el ratón, utilizando para ello un entorno gráfico. El editor de visualización está integrado en el entorno de programación de CoDeSys, resultando una herramienta potente y cómoda para observar el comportamiento de un programa en la fase de depuración, o incluso como un SCADA simple en la fase de ejecución usando las herramientas más avanzadas de la que dispone. En este libro, como ha podido ver el lector, se ha usado profusamente esta herramienta en la práctica totalidad de los ejemplos y ejercicios propuestos.

Este apéndice pretende dar unas nociones elementales del uso de las Visualizaciones, utilización de los elementos básicos, conexión con las variables de programa de control, etc., pero con la suficiente profundidad que permita al usuario establecer un entorno de monitorización y mando básico para los programas que esté desarrollando. Para una profundización en esta herramienta consulte [13].

### B.1. Editor de Visualizaciones

La edición de visualizaciones está basada en la colocación de elementos gráficos en una ventana de edición, cuyas características (forma, color, texto,...) pueden estar asociadas a las variables del programa del PLC que se ha realizado en CoDeSys. El editor permite sólo unos pocos tipos de elementos, pero suficientes para proporcionar una gran funcionalidad desde una notable simplicidad. Una de las características que tiene es la facilidad en la conexión de los elementos gráficos con las variables del programa del PLC, que no es habitual en otras implementaciones de este tipo. En definitiva, permite la realización de una aceptable interface de

usuario con un esfuerzo de programación relativamente pequeño.

La creación de una Visualización en un Proyecto, se realiza desde la pestaña *Visualizaciones* en el Organizador de Objetos de CoDeSys. Cada Proyecto puede tener tantas visualizaciones como se desee.

La creación de elementos de este tipo es la habitual en el entorno CoDeSys. Una vez seleccionada la pestaña de visualizaciones, y con el menú contextual (botón derecho del ratón) se selecciona *Add Object*. El único parámetro que es necesario introducir es el nombre de la Visualización. Aparecerá una ventana gráfica donde se podrán introducir los elementos. Nótese que la ventana aparecerá en la zona de edición de CoDeSys, donde están las ventanas con el código de los POU, etc. Es decir se trata de una herramienta totalmente integrada en el desarrollo de programas para PLC y con la que se podrá trabajar de forma simultánea al desarrollo de dichos programas. Con este mismo menú contextual se pueden realizar otras operaciones con la Visualizaciones, como borrarlas, copiarlas, etc.

Si se selecciona la ventana de Visualización aparecerá en la barra de herramientas, los botones con los que se podrán insertar los distintos elementos. Los elementos que se dispone están listados en la Tabla B-1.

Para insertar cualquiera de estos elementos, se pulsa el botón correspondiente y se coloca en la ventana de edición en la posición y con el tamaño que se desee. La edición es muy similar a la de cualquier programa de edición de gráficos por lo que no se entrará en detalles. En *Extras->ElemenList* aparecerá una tabla con todos los elementos de la visualización y sus coordenadas.

## B.2. Configuración básica de elementos

Haciendo doble-click sobre un objeto, se podrá acceder a sus parámetros de configuración, que permiten definir sus propiedades y asociarlos a variables del proceso.

Para hacer referencia a nombres de variables, hay que tener en cuenta que se puede acceder tanto a variables globales como locales, por lo que es necesario seguir la siguiente sintaxis:

- Variables globales: El formato será *nombre\_var* o *.nombre\_var*

- Variables locales: Si la variable está definida dentro de un POU denominado *MiPOU*, se referencia como *MiPOU.nombre\_var*.

Por ejemplo, si se dispone un temporizador tipo TON denominado *Tempo* definido en un POU con nombre *Control*, la forma de usar el parámetro de entrada IN será:

*Control.Tempo.IN*

Tabla B-1. Elementos gráficos en las Visualizaciones

Elementos gráficos	Botón	Elementos gráficos	Botón
Rectángulo		Rectángulo con borde redondeado	
Elipse		Polígono	
Polilínea		Curva	
Gráfico de tarta		Gráfico bitmap	
Visualizaciones		Botones	
Gráficos WMF		Tablas	
Gráficos de tendencia		Tablas de alarmas	
Elementos Active-X		Barra de desplazamiento	
Indicador de aguja		Diagrama de barras	
Histograma			

**Nota:** Es necesario tener en cuenta que las visualizaciones no se compilan como el resto del programa, así que, si se escribe mal el nombre de la variable o dicha variable no existe, no se producirá ningún error de compilación ni ningún tipo de aviso, simplemente la visualización no se comportará como se desea.

También, en lugar de variables se pueden indicar expresiones y constantes, pero siempre respetando la identificación de variables arriba indicada.

La Figura B-1 muestra la ventana de edición de parámetros. Como se observa, a la izquierda aparecen una serie de categorías de parámetros y a la derecha los parámetros específicos de dicha categoría.

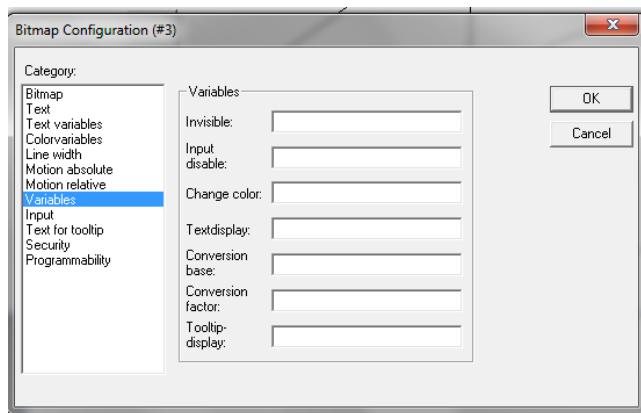


Figura B-1 – Ventana de edición de visualizaciones

Algunas de las categorías son comunes a muchos de los tipos de elementos. Otras son específicas de cada elemento. A continuación se describen algunas de las categorías más generales. No se van a describir todos y cada uno de los parámetros sino sólo los más útiles para un uso básico. Para más detalle, consulte el manual de usuario.

### B.2.1. Text

Permite incluir texto que se mostrará en el interior del elemento. Además, permite la alineación del texto y la elección de las fuentes. También es posible hacer que

aparezca en el texto el valor de una determinada variable. Para ello se utiliza una sintaxis similar a la que usa el lenguaje C en la función *printf*. De forma general se utilizara %s en el lugar donde se quiera que aparezca el valor de la variable. La variable cuyo valor se quiere que aparezca se indicará en la Categoría *Variables -> TextDisplay*. En general se puede utilizar cualquier tipo de variable, aunque también se permiten utilizar formato específico para tipos de datos concretos, por ejemplo %d para enteros, %f para reales, etc. (en el botón de ayuda de la ventana se encuentra una información más detallada de todos los posibles formatos).

## B.2.2. Text variables

Aunque dentro de la ventana *Text* es posible fijar las características del texto (color, fuente,...), éstas permanecen estáticas a lo largo del tiempo. Si se pretende que alguna de estas características cambie dinámicamente, se pueden asignar estos parámetros a nombres de variables. Algunos parámetros que se pueden modificar son:

- **Textcolor:** El color del texto
- **Textflags:** Alineación del texto (centrado, izquierda,...)
- **Fontheight:** Tamaño de letra
- **Fontname:** Nombre de la fuente
- **Fontflags:** Negrita, subrayado, itálica,...

Ejemplo: Se ha escrito una POU en leguaje ST en el fichero *CapB\_1.pro* denominado *CambiaLetra*. En el campo *Textcolor* introducimos *CambiaLetra.ColorLetra* y en el campo *Fontheight* la variable *CambiaLetra.AlturaLetra*. El código del programa se muestra a continuación.

Compruebe en el botón que aparece a la izquierda de la visualización, que dependiendo de si M está a TRUE o a FALSE (pulsando el botón) cambian el color y el tamaño de la letra. El color se introduce en una variable DWORD en formato RGB (dos dígitos para cada color).

---

```

AlturaLetra:INT;
ColorLetra: DWORD;
-----
IF M THEN
    AlturaLetra:=16;
    ColorLetra:=16#FF00FF;
ELSE
    AlturaLetra:=22;
    ColorLetra:=16#121212;
END_IF

```

---

### B.2.3. Line width

En esta categoría se puede seleccionar el grosor de la línea de la figura indicando el número de pixeles o bien indicar un nombre de variable si se quiere que cambie a lo largo de la simulación.

### B.2.4. Color

Permite seleccionar el color del objeto (rectángulos, círculos...) y de la línea del marco. Marcando las casillas correspondientes se puede eliminar el color del interior o del marco. A cada objeto se le pueden asignar dos colores tanto al interior como al marco, denominados *Color* y *Color Alarma*.

El objeto cambia el color entre los dos dependiendo de la variable que se introduce en la categoría *Variables -> Change Color*.

En *CapB\_1.pro* se muestra como usando *Color* y *Color Alarma* puede cambiar el color de la figura. La variable de cambio es *Alarma*, que se puede cambiar pulsando el interruptor de alarma.

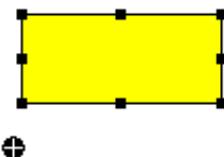
### B.2.5. Colorvariables

*Color* y *Color Alarma* son fijos si lo definimos en la categoría anterior. En esta categoría se pueden asignar a variables, por lo que todos esos colores pueden

cambiar a lo largo de la ejecución del programa sin más que cambiar el valor de las variables. Como se indicó anteriormente, las variables de color se definen del tipo DWORD y se expresan en formato RGB.

### B.2.6. Motion absolute

Permite el movimiento del objeto durante la ejecución del programa. Se permite desplazar el objeto en sentido vertical, horizontal, girarlo o desplazarlo. En *CapB\_1.pro* está implementado un pequeño programa en ST que permite mover todos estos parámetros. Nótese que en el giro, la rotación se realiza alrededor del punto negro que aparece cuando se selecciona el objeto. El objeto es un rectángulo, pero nótese que los lados no giran sino el objeto completo. Si se quiere un giro del objeto se deberá seleccionar necesariamente un polígono.



### B.2.7. Motion relative

Permite mover los límites laterales del objeto. En *CapB\_2.pro* se puede ver un ejemplo de movimiento de los laterales de una elipse. En la categoría *Shape* de los parámetros de la elipse se puede cambiar el tipo de objeto a un rectángulo, línea, etc., para ver el efecto sobre otras formas geométricas.

### B.2.8. Variables

Se introducen variables que realizan algunas acciones, algunas de las cuales ya se ha visto en la descripción de categorías anteriores.

- **Change Color:** Cambia entre Color y Color Alarma
- **TextDisplay:** Valor de la variable que se representa en el texto cuando incluye

%s.

- **Invisible:** La variable desaparece cuando la variable está a TRUE. Compruebe su efecto en *CapB\_3.pro*.
- **Disable Input:** Deshabilita todo lo que esté en la Categoría Input. Por ejemplo, en *CapB\_3.pro* el interruptor *Deshabilita botones* activa la variable *Deshabilita* que está puesta en *Variables->Disable Input* de los dos botones de al lado. Se comprueba que los botones dejan de funcionar.

### B.2.9. Input

Es la forma más simple de dar valores a una variable booleana. Cuando pulsamos sobre el objeto, cambia el valor de la variable que tiene asignada. Este es el uso más típico del objeto botón, pero se puede hacer con cualquier otro objeto.

- **Toggle Variable:** Funcionamiento como interruptor. Cada vez que pulsemos el objeto cambia el estado de la variable asociada. Por ejemplo, compruebe los parámetros y el funcionamiento el objeto *Desactiva Botones* en *CapB\_3.pro*.
- **Tap Variable:** Funcionamiento como pulsador. La variable solo cambia mientras esté pulsado el elemento. Por ejemplo, analice el objeto *Invisible* en *CapB\_3.pro*.
- **Tap False:** Determina el estado inicial del pulsador.
- **Text Input of Variable TextDisplay:** Opción muy útil para introducir valores a una variable no booleana. Se introduce valores a la variable definida en la categoría *Variables->Textdisplay*. Hay varias opciones que permiten introducir valores por teclado, con un teclado en pantalla, con los valores ocultos, etc. En *CapB\_3.pro* se muestra un ejemplo, que al pulsar el círculo se puede introducir valores, mediante un teclado numérico en pantalla, de la variable *Número*, cuyo valor se muestra en una ventana.

### B.2.10. Text for tooltip

Texto que aparece cuando pasa el ratón por encima del objeto cuando está en modo ejecución. En *CapB\_3.pro* cuando se pasa el ratón por encima del círculo amarillo,

aparece un texto.

## B.3. Configuración avanzada de elementos

En la sección anterior se describieron algunos de los atributos más comunes compartidos por los distintos elementos. Los elementos más comunes (rectángulos, círculos, líneas, botones, etc.) prácticamente se definen con los parámetros arriba indicados. En esta sección se verán algunos elementos gráficos más específicos y que tienen atributos distintos, aunque también pueden compartir alguno de los atributos anteriores.

### B.3.1. Table

Se utilizan para la representación de matrices. El ejemplo *CapB\_4.pro* muestra un ejemplo de representación de una tabla declarada y usada en el programa PLG\_PLC escrito en código ST. Cuando se pulsa el botón *Duplica* los valores de la tabla se duplican.

Las cuatro primeras categorías son específicas para tablas:

- **Table:** Define la tabla a representar, que tiene que estar declarada en algún programa.
- **Columns:** Se indica que columnas se quieren representar. En la zona izquierda están todas las columnas y a la izquierda las que se quieren representar. Si se hace doble-click en una columna de las seleccionadas a la derecha, se puede cambiar el nombre del encabezamiento de la columna, el ancho de cada columna, etc.
- **Rows:** Permite configurar el alto de cada fila.
- **Selection:** Sirve para determinar cómo se van a marcar las celdas seleccionadas con el ratón: color, recuadro, celdas, columnas o filas. También se pueden marcar las celdas determinadas por variables de los programas.

### B.3.2. Scrollbar

La configuración de las barras de desplazamiento no tiene ningún problema especial. Estudie el comportamiento de la barra de desplazamiento en *CapB\_4.pro*, donde la selección de una celda de la tabla se realiza desde dos barras de desplazamiento.

### B.3.3. Indicador de aguja

El indicador de aguja (*Meter*) presenta muchas opciones de diseño. En cualquier caso, en la propia ventana de configuración tiene una pre-visualización, por lo que el diseño se realiza de una forma muy simple. En *CapB\_4.pro* la variable de selección Y está indicada con uno de estos elementos gráficos. Modifique los parámetros de diseño y observe el resultado.

### B.3.4. Diagrama de barras

Al igual que en el caso anterior, tiene varias opciones de configuración, pero al disponer de una pre visualización es fácil de usar. En *CapB\_4.pro* el valor de la variable *SeleccionX* está representada por un diagrama de este tipo.

### B.3.5. Inclusión de archivos gráficos

Se pueden introducir en una visualización dibujos o fotografías en distintos formatos. Para esto hay dos posibilidades: el elemento bitmap permite introducir archivos en formato bmp, tif o jpg, mientras que el elemento WMF admite el formato WMF (Windows MetaFile). Los parámetros de configuración son los típicos que se vieron en la sección anterior.

### B.3.6. Graficas de variables frente al tiempo

Permite representar una o más variables frente al tiempo. De nuevo presenta muchas opciones, pero lo más recomendable es experimentar con él. Para esto, se proporciona en *CapB\_5.pro* un simulador de un depósito escrito en ST. Pulsando los

botones de *Entrada* y *Salida* se abren las válvulas de entrada y salida respectivamente. En la gráfica está representado el Nivel del depósito. De nuevo, practique con las distintas opciones que presenta este elemento para entender su funcionamiento.

Preste atención al simulador escrito en ST. Realiza una integración de la ecuación diferencial simple del depósito utilizando el método de Euler. En cada ciclo del autómata realiza un paso de integración de 0.1 segundos. Como el programa está incluido en una tarea cíclica que se ejecuta cada 100 ms., el comportamiento del simulador lo podemos considerar en tiempo real.



# REFERENCIAS

---

- [1] J. P. Romera Ramirez, J. A. Lorite Godoy y S. Montoro Tirado, Automatización Problemas resueltos con autómatas programables, Madrid: Thomson, 2003.
- [2] P. A. Daneri, PLC. Automatización y control industrial, Buenos Aires: Hispano Americana, 2008.
- [3] J. Balcells y J. L. Romeral, Autómatas Programables, Barcelona: Marcombo, 1997.
- [4] E. Mandado Pérez, J. Marcos Acevedo, C. Fernández Silva y J. I. Armesto Quiroga, Autómatas programables y sistemas de automatización, Barcelona: Marcombo, 2009.
- [5] A. Creus Solé, Instrumentación Industrial, Barcelona: Marcombo, 2011.
- [6] J. G. Webster, The Measurement, Instrumentation and Sensor Handbook, CRC Press - IEEE Press, 1999.
- [7] J. M. Rubio Calin, Buses Industriales y de Campo, Marcombo, 2009.
- [8] K.-H. John y M. Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems, Berlin: Springer-Verlag, 2001.
- [9] Smart Software Solutions, User Manual for PLC Programming with CoDeSys

- 2.3, Kempten: 3S - Smart Software Solutions GmbH, 2010.
- [10] R. David y H. Alla, Petri Nets and Gtafcet, Prentice-Hall, 1992.
- [11] M. Silva, Redes de Petri: En la Automática y la Informática, Alfa Centauro, 1985.
- [12] P. Ponsa Asencio y R. Vilanova Arbós, Automatización de procesos mediante la guía GEMMA, Barcelona: Universitat Politécnica de Catalunya, 2005.
- [13] Smart Software Solutions, CoDeSys Visualization, Kempten: 3S-Smart Software Solutions GmbH, 2007.

# ÍNDICE DE CONCEPTOS

---

*No se encuentran entradas de índice.*