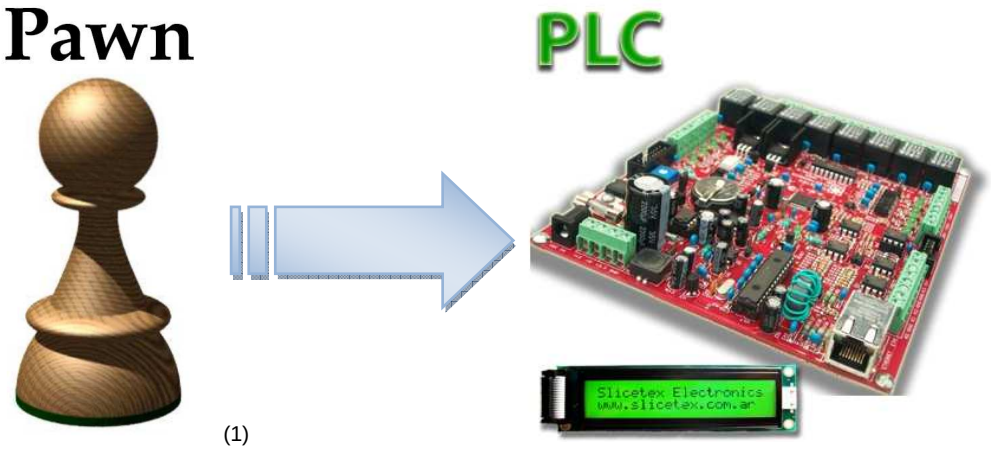
Slicetex Ladder Designer Studio

(StxLadder)

Introducción al Lenguaje Pawn

Autor: Ing. Boris Estudiez



(1)

1 Descripción General

*En este documento se expone una guía genérica de primeros pasos del Lenguaje **Pawn**.*

*El lenguaje **Pawn** puede ser utilizado para programar nuestros **PLC** (Controlador Lógico Programable) o para insertar código a través de componentes desde el lenguaje **Ladder** en el entorno **StxLadder**.*

*Aquí se introduce al lenguaje de programación Pawn. Se recomienda complementar esta guía con el **Manual de Programación Pawn del PLC** que incluye procedimientos completos para cargar programas en el PLC.*

(1) PAWN logo copyright by ITB Compuphase

2 Lecturas Recomendadas

Antes de leer este documento, recomendamos que se familiarice con el PLC y el entorno **StxLadder**. Para ello recomendamos leer los siguientes documentos, en el orden detallado a continuación:

1. Hoja de Datos del PLC
2. **STXLADDER-UM**: Manual de Usuario de **StxLadder**.

Es altamente recomendado complementar esta guía con el Manual de Programación Pawn del PLC que incluye procedimientos completos para cargar programas en el PLC, breve descripción del entorno **StxLadder** y una guía referencia completa de las funciones nativas soportadas por el PLC.

3 Alcance del Documento

Este documento está orientado a comprender el lenguaje Pawn. Si desea mezclar código Pawn con el lenguaje Ladder, remítase al documento **STXLADDER-PL**, pero note que en dicho documento no se explican las bases del lenguaje Pawn, por lo tanto este documento es de lectura obligada.

4 Requerimientos

Para programar el PLC, es necesario tener instalado, el siguiente software:

1. **StxLadder**: Slicetex Ladder Designer Studio. Disponible en nuestro sitio Web.

5 Lenguaje PAWN

5.1 Introducción

PAWN es un lenguaje script simple, de 32-bits y con una sintaxis similar al lenguaje de programación C.

Esta guía, pretende orientarlo rápidamente al lenguaje para que pueda programar nuestros dispositivos en el menor tiempo posible. No es un manual completo del lenguaje, para una guía completa remítase al documento "**Pawn_Language_Guide.pdf**", disponible en nuestro sitio Web.

Para entender correctamente esta guía, es necesario un mínimo conocimiento de programación, en cualquier lenguaje. Recomendamos complementar la lectura con el "**Manual de Programacion Pawn del PLC**", donde se detallan todas las funciones nativas soportadas por el PLC.

PAWN fue seleccionado para nuestra familia de PLC (Controladores Lógicos Programables) por ser simple y potente. La capacidad que se obtiene con PAWN, es muy superior a las posibilidades que normalmente se logran con los lenguajes gráficos tradicionales (Ladder) para PLC standards.

Esta guía, comienza desde lo básico e incrementa su complejidad a medida que avanza. Por ello, es conveniente leerla desde el comienzo. Creemos que es la forma más didáctica de aprender.

5.2 Primer Script

El primer script que crearemos solo conmuta un Relay del PLC cada un segundo, pero nos permitirá explicar los pasos iníciales de programación. En el entorno StxLadder, escribir el siguiente código en el archivo **PlcMain.p**:

```
PlcMain()
{
   while(true)
   {
      // Conmutar RELAY1.
      RelayToggle(RELAY1)

      // Esperar 1000 ms.
      DelayMS(1000)
   }
}
```

El código de arriba, nos muestra que todo script en PAWN, tiene una función principal llamada "PlcMain()" que es la primera en llamarse al activarse el PLC. El código de PlcMain(), empieza luego de "{" y termina en el ultimo "}". Cada bloque de código se delimita con los corchetes "{" y "}".

Dentro de PlcMain(), hay un ciclo "while", cuya condición es "true" o "1". Esto quiere decir, que el script nunca saldrá del ciclo "while", ya que es siempre "verdadero". Todo script que se realice para el PLC debe tener un ciclo "while" principal con la condición "true", esto implica que se repite infinitamente. De lo contrario el script llegaría al final de las instrucciones y el PLC no tendría más órdenes para ejecutar.

Dentro del ciclo “while”, hay dos llamadas a funciones:

- RelayToggle(RELAY1): Conmuta el RELAY1 del PLC cada vez que es llamada.
- DelayMS(1000): Cuando es llamada, detiene la ejecución del programa en 1000 mS. Si utiliza eventos, lea pagina 36 para el uso correcto de la función DelayMS().

Las funciones RelayToggle() y DelayMS(), se llaman funciones nativas, ya que están incorporadas en el PLC.

En el Manual de Programación Pawn del PLC, se explica cómo cargar el script generado al PLC y además se listan todas las funciones nativas disponibles, así como su argumentos y empleo adecuado.

La ejecución del script, es secuencial, es decir, la líneas superiores de la función PlcMain() se ejecutan en primer lugar.

Por último, los comentarios en el script (notas, palabras y/o información que no es código), se pueden escribir uno por línea anteponiendo dos barras invertidas “//”.

```
// Comentario de una sola línea.
```

Si escribimos un párrafo grande, con muchas líneas, utilizamos:

```
/*
   Esto es un comentario
   de varias líneas…

   El compilador, ignora los comentarios!.
*/
```

Note que cada línea del script esta TABULADA, es decir que esta en el mismo nivel de columna, respecto al bloque en que se encuentra. Para ello, se aprieta TAB cada vez que escribimos una nueva línea. Esto es opcional, pero facilita la lectura del script (muy importante para no cometer errores).

Finalmente, guardamos el script (archivo con extensión .p), luego lo compilamos para asi cargarlo al PLC desde el entorno StxLadder.

5.3 Variables en el Script

5.3.1 Variables enteras

Las variables en PAWN ocupan celdas de memoria, cada celda consume 4-bytes de memoria RAM. En nuestro PLC las variables pueden ser enteras o con decimal (punto flotante).

Una variable entera (cell o celda) tiene 32-bits y puede almacenar números negativos o positivos en el siguiente rango:

- Números positivos: 0 a 2.147.483.647.
- Números negativos: -1 a 2.147.483.648.

Para definir una variable entera, utilizamos la palabra clave “new”, seguida con el nombre de la variable:

```
new Botellas
```

La variable “Botellas” fue creada en memoria y ahora es posible asignarle valores o leerla, con el rango de valores anteriormente mencionado. Por ejemplo, le asignamos el valor 80 de la siguiente forma:

```
Botellas = 80
```

Es posible inicializar la variable al mismo tiempo de crearla, por ejemplo:

```
new Botellas = 80
```

El siguiente ejemplo, crea una variable “Pausa” para especificar un retardo en el primer script de la página 3:

```
PlcMain()
{
   new Pausa = 1000

   while(true)
   {
      // Conmutar RELAY1.
      RelayToggle(RELAY1)

      // Esperar 1000 ms.
      DelayMS(Pausa)
   }
}
```

La función DelayMS() esperara 1000 mS, ya que es el valor almacenado en “Pausa”.

Si una variable no va a ser modificada NUNCA, es posible definirla como constante, con “const”:

```
new const BotellasMaximas = 1000
```

Recomendamos ver también la pagina 38. Las reglas para nombrar variables se detallan en pag. 45.

5.3.2 Arreglos o Array de variables

Es posible definir un array o arreglo de celdas. Un array es un conjunto de variables que puede ser accedido mediante un índice, por ejemplo:

```
new Botellas[5]
```

La anterior definición, crea cinco variables tipo celdas, las cuales pueden ser accedidas de la siguiente forma:

```
Botellas[0] = 23
Botellas[1] = 56
Botellas[2] = 219832
Botellas[3] = 35
Botellas[4] = 5657
```

Como podemos observar, fue posible asignarle un valor a cada variable del array, mediante un índice que va desde 0 (primer elemento del array) a 4 (último elemento del array). En programación, es común empezar a contar desde 0, y no desde 1.

Podemos suponer a modo de ejemplo, que las “Botellas” están en cajas, y cada caja es el número que usaremos como índice.

Inclusive, podemos definir una variable llamada “Caja” y acceder al número de botellas por caja:

```
// Definimos la variable Caja.
new  Caja = 2

// En la Caja 2, no hay botellas.
Botellas[Caja] = 0
```

Tenga presente, que cada celda o variable creada, consume 4 bytes, por lo tanto el array Botellas[] de 5 celdas, consume 4 x 5 = 20 bytes de memoria RAM. Es conveniente, consultar la memoria RAM de nuestros PLC, para mantener nuestra cantidad de RAM utilizada por debajo del límite.

Si queremos inicializar el array, mientras lo definimos, podemos hacerlo de la siguiente forma:

```
new Botellas[5] = {23, 56, 219832, 35, 5657}
```

El código anterior, inicializa cada elemento del array con un valor inicial. Note que debe inicializar todos los elementos del array (en este caso 5 elementos).

Es posible también definir un array con un valor constante (que nunca se a cambiar), muy útil para crear tablas:

```
new const Tabla[7] = {0, 12, -34, 45, 111, 678, 1000}
```

5.3.3 Variables decimales o de punto flotante

Para definir una variable decimal o de punto flotante, hacemos:

```
// Crear variable para almacenar voltaje.
new Float: Voltaje
```

La variable creada, llamada “Voltaje” es del tipo “Float”. Notar que la declaración se escribe “Float:”, con dos puntos “:” al final, y no puede existir espacios en blanco.

Ahora le asignaremos un valor decimal:

```
Voltaje = 5.2345
```

También es posible crear un arreglo de variables “Float”, por ejemplo, definimos 10 celdas “Float”:

```
new Float: Voltaje[10]
```

Como aclaración final, si una función retorna un valor “Float” y queremos almacenarlo, debemos hacer lo siguiente:

```
Voltaje = Float: Vin1ReadVolt()
```

*Para que el valor sea correctamente almacenado en una variable del tipo “Float” es necesario decirle a PAWN que el valor retornado por la función, es del tipo “Float” mediante un **cast**. Esto se hace con la palabra “Float:” antecediendo el nombre de la función Vin1ReadVolt().*

La función Vin1ReadVolt() retorna el voltaje de la entrada analógica 1, y se explica en el manual de usuario.

5.3.4 Variables tipo booleanas

Una variable boleana, solo puede contener dos posibles valores “true” (verdadero) o “false” (falso). Se usan generalmente como banderas, para señalizar algún evento o condición.

A continuación se define una variable tipo “bool”, llamada “Entrada” y con un valor inicial “true”:

```
new bool: Entrada = true
```

La variable “Entrada” puede representar el estado de alguna entrada discreta. Las cuales, tienen dos posibles valores por naturaleza.

Las constantes “true” y “false” están definidas en el lenguaje, y su valore son 1 y 0 respectivamente.

5.4 Condicionales

5.4.1 Condicional “if”

Un condicional nos permite tomar decisiones evaluando si alguna condición es verdadera o falsa. Por ejemplo, “Si la entrada vale 1, activar rele. De lo contrario, apagar Rele”.

El condicional más importante se llama “if” (“si” en ingles). Tiene la siguiente sintaxis:

```
if(condición)
{
      Bloque de código
}
```

Por ejemplo, si alguna condición es verdadera (true o 1), se ejecuta el “Bloque de código”. De lo contrario el script sigue su ejecución sin ejecutar el código dentro del bloque.

En el siguiente ejemplo, se comprueba si la variable “Temperatura” vale “35”, y solo en ese caso, activa un rele:

```
PlcMain()
{
      new Temperatura = 35

      while(true)
      {
            // Comprobar si Temperatura es igual a 35.
            if(Temperatura == 35)
            {
               // Activar RELAY1.
               RelayClose(RELAY1)
            }

            // Esperar 1000 ms.
            DelayMS(Pausa)
      }
}
```

El código anterior, comprueba si la variable es igual a 35 con símbolo relacional “==”. Que retorna “true” si la variable es igual al valor representado a su derecha. De lo contrario es “false” y el código del “if” no se ejecuta.

Ahora, ¿Qué pasa si queremos apagar el Relé, solo en caso de que el valor de Temperatura no cumpla con la condición contenida dentro del “if” ?.

A continuación, respondemos la pregunta.

Luego del bloque “if”, se agrega la palabra “else”, que significa “sino”:

```
if(condición)
{
      Bloque de código si la condición se cumple (true)
}
else
{
      Bloque se código que se ejecuta solo si la condición anterior
      no se cumple (false).
}
```

En el ejemplo de Temperatura, podemos apagar el Rele si la condición no se cumple:

```
// Comprobar si Temperatura es igual a 35.
if(Temperatura == 35)
{
      // Activar RELAY1.
      RelayClose(RELAY1)
}
else
{
      // Desactivar RELAY1.
      RelayOpen(RELAY1)
}
```

Finalmente para hacer más “divertido” el asunto, podemos comprobar en un mismo bloque “if” varias veces a la variable Temperatura, con el “else if”:

```
// Comprobar si Temperatura es igual a 35.
if(Temperatura == 35)
{
      // Activar RELAY1.
      RelayClose(RELAY1)
}
else if(Temperatura == 40)
{
      // Activar RELAY2.
      RelayClose(RELAY2)
}
else
{
      // Ninguna expresión es verdadera, desactivar RELAY1.
      RelayOpen(RELAY1)
}
```

El código anterior comprueba si la Temperatura es igual a 35, si es verdadero, activa el RELAY1. Si no pasa al siguiente “else if” que comprueba si la Temperatura es igual a 40, si es verdadero, activa el RELAY2. Si ninguna de las condiciones se cumple, se ejecuta el bloque “else” que apaga el.

Dentro de un mismo “if” podemos anidar varios “if”, por ejemplo:

```
new Temperatura = 35
new Humedad = 50

// Comprobar si Temperatura es igual a 35.
if(Temperatura == 35)
{
      // Activar RELAY1.
      RelayClose(RELAY1)

      if(Humedad >= 50)
      {
            // Activar RELAY4.
            RelayClose(RELAY4)
      }
}
```

El código anterior, comprueba si la Temperatura es igual a 35, y en ese caso, activa el RELAY1 y también comprueba si la Humedad es “mayor o igual” a 50 con el símbolo “>=”. Si la Humedad es mayor o igual a 50, activa el RELAY4.

Tabla: Símbolos para comprobar relaciones (ver también página 46)

Símbolo	Descripción	Precedencia
<	Menor que	Izquierda a derecha
<=	Menor que o igual a	Izquierda a derecha
>	Mayor que	Izquierda a derecha
>=	Mayor o igual que	Izquierda a derecha
==	Igual que	Izquierda a derecha
!=	Distinto que	Izquierda a derecha

Otra forma de comprobar Temperatura y Humedad en el mismo “if”, es usando expresiones lógicas:

Por ejemplo:

```
// Comprobar si Temperatura es igual a 35 y Humedad es >= 50.
if( (Temperatura == 35) && (Humedad >= 50) )
{
      // Activar RELAY1.
      RelayClose(RELAY1)

      // Activar RELAY4.
      RelayClose(RELAY4)
}
```

El símbolo “&&” corresponde al símbolo lógico “AND” o “y”. Solo **SI** las expresiones a su izquierda y derecha son verdaderas, produce un “true” y el “if” se ejecuta.

Los paréntesis "(" y ")" le dicen al lenguaje que primero compruebe las expresiones dentro de ellos y luego, el resto de las expresiones. Entonces, el script comprobará primero si la Temperatura es igual a 35, luego si la Humedad es mayor o igual a 50, y finalmente aplicara un "AND" de ambas expresiones. Si ambas son verdaderas, el "if" se ejecutara.

Es posible emplear el símbolo lógico "||" que corresponde a un "OR" (o). Si **alguna** de las expresiones a su izquierda o derecha es verdadera, produce un "true".

Por ejemplo:

```
// Comprobar si Temperatura es igual a 35 o la Humedad es >= 50.
if( (Temperatura == 35) || (Humedad >= 50) )
{
      // Activar RELAY1.
      RelayClose(RELAY1)

      // Activar RELAY4.
      RelayClose(RELAY4)
}
```

Si alguna de las expresiones de "Temperatura igual a 35" o la "Humedad mayor igual a 50" se cumplen, el "if" se ejecuta.

Tabla: Símbolos para expresiones con relaciones lógicas (ver también página 46)

Símbolo	Descripción	Precedencia
&&	AND lógico	Izquierda a derecha
||	OR lógico	Izquierda a derecha

Como nota importante, en la tabla especificamos la "Precedencia", esto nos dice como evalúa el lenguaje las expresiones. Por ejemplo, al utilizar un "||", el script evaluara primero la expresión a la izquierda y luego la expresión de la derecha. Al final, efectuara un OR lógico entre ambas.

Ejercicios:

1. Realice un script que compruebe si el valor de Temperatura es menor a 35, y solo en ese caso active el RELAY1. De lo contrario desactivar el RELAY1.
2. Realice un script que en caso de subir la Temperatura por encima de los 40 grados, active el RELAY1. Si la Temperatura baja de los 20 grados apague el RELAY1.
3. Realice un script que compruebe si la Humedad es mayor a 35 y la Temperatura mayor a 30, y en ese caso active el RELAY1. De lo contrario, desactivar el RELAY1.
4. Realice un script que lea una entrada discreta (DIN1) del PLC, si esta activa, active el RELAY1. De lo contrario, debe apagarlo.

5.4.2 Condicional “switch”

El condicional “switch” nos permite evaluar una expresión y ejecutar una lista de casos de una forma ordenada. Es similar al condicional “if…else if…else”, pero no tan potente.

En general, se utiliza para adoptar acciones, de acuerdo al valor de una variable. Dependiendo del valor obtenido, se ejecuta una lista de casos (case) que coincidan.

Por ejemplo, supongamos que tenemos un motor, del cual obtenemos su RPM (revoluciones por minuto) a través de las entradas de contadores rápidos del PLC. El rango de RPM es de 0 a 7500. Para distintos valores de RPM, activamos o desactivamos relés, que activan lámparas indicadoras a un operario en un banco de prueba.

Con un switch(), resolvemos el ejemplo rápidamente:

```
// Variable para almacenar los RPM del motor.
new RPM

// Obtenemos RPM del contador rápido COUNT1 (DIN7).
RPM = Count1GetEventRPM()

// Procesar valor del RPM
switch(RPM)
{
	// RPM igual a 0 ?
	case 0:
	{
		// Desactivar todos los RELAYS
		RelayOpen(RELAY1)
		RelayOpen(RELAY2)
		RelayOpen(RELAY3)
	}
	// RPM igual a 2500 ?
	case 2500:
	{
		// Activar RELAY1.
		RelayClose(RELAY1)
	}
	// RPM igual a 5000 ?
	case 5000:
	{
		// Activar RELAY2.
		RelayClose(RELAY2)
	}
	// RPM igual a 7500 ?
	case 7500:
	{
		// Activar RELAY3.
		RelayClose(RELAY3)
	}
}
```

En el ejemplo anterior, obtenemos el valor de RPM desde la función Count1GetEventRPM(), que se explica en el Manual de Programación Pawn del PLC. Pero lo importante aquí, es que luego de obtener el valor y almacenarlo en la variable RPM, lo procesamos con el “switch”.

Notar que para los valores 0, 2500, 5000 y 7500 de RPM, hay un “case”, que se ejecuta si el valor del RPM coincide. Solo un “case” se ejecuta por cada vez que se ejecuta el “switch”. Esto es una diferencia con respecto al famoso leguaje “C”.

Las expresiones en los “case” deben ser constantes, es decir números. No es posible hacer:

```
case RPM >= 2500:
{
      // Activar RELAY1.
      RelayClose(RELAY1)
}
```

Debido a que RPM no es una constante, es una variable.

Otra palabra que se puede agregar a un “switch”, es la palabra “default”, que se ejecuta, solo si ningún caso coincide con el valor de la variable a comprobar:

```
// Procesar valor del RPM
switch(RPM)
{
      // RPM igual a 0 ?
      case 0:
      {
            // Desactivar todos los RELAYS
            RelayOpen(RELAY1)
            RelayOpen(RELAY2)
            RelayOpen(RELAY3)
      }
      // RPM igual a 2500 ?
      case 2500:
      {
            // Activar RELAY1.
            RelayClose(RELAY1)
      }
      // RPM igual a 5000 ?
      case 5000:
      {
            // Activar RELAY2.
            RelayClose(RELAY2)
      }

      // Código sigue en la siguiente pagina…
```

```
		// RPM igual a 7500 ?
		case 7500:
		{
			// Activar RELAY3.
			RelayClose(RELAY3)
		}

		// Ningún valor coincide con el valor de RPM ?.
		default:
		{
			// Activar todos los RELAYS
			RelayClose(RELAY1)
			RelayClose(RELAY2)
			RelayClose(RELAY3)
		}
	}
```

El caso “default” se ejecuta si ningún valor coincide con el valor de RPM, y activa todos los relés.

También es posible especificar un rango de valores en los “cases” a comprobar. Esto se hace con dos puntos seguidos “..” . A la izquierda se especifica el límite inferior del rango y a la derecha el límite superior. Por ejemplo, “case 0..2499:” se ejecuta si el valor está entre 0 y 2499.

Veamos el siguiente ejemplo del motor y sus RPM:

```
	// Procesar valor del RPM
	switch(RPM)
	{
		// RPM igual entre 0 y 2499 ?
		case 0..2499:
		{
			// Desactivar todos los RELAYS
			RelayOpen(RELAY1)
			RelayOpen(RELAY2)
			RelayOpen(RELAY3)
		}
		// RPM igual entre 2500 y 4999 ?
		case 2500..4999:
		{
			// Activar RELAY1.
			RelayClose(RELAY1)
		}
		// RPM igual entre 5000 y 7499 ?
		case 5000..7499:
		{
			// Activar RELAY2.
			RelayClose(RELAY2)
		}

		// Código sigue en la siguiente pagina…
```

```
        // RPM igual entre 7500 y 8000 ?
        case 7500..8000:
        {
              // Activar RELAY3.
              RelayClose(RELAY3)
        }

        // Ningún valor coincide con el valor de RPM ?.
        default:
        {
              // Activar todos los RELAYS
              RelayClose(RELAY1)
              RelayClose(RELAY2)
              RelayClose(RELAY3)
        }
}
```

El ejemplo anterior, activa o desactiva relays, si el RPM esta en algunos de los rangos:

- 0 a 2499
- 2500 a 4999
- 5000 a 7499
- 7500 a 8000

Utilizar “switch” puede ser muy útil para comprobar muchos valores al mismo tiempo de una variable, de una forma simple. Pero, no debe utilizarse como un reemplazo a un “if”. Si usted piensa que puede resolver un problema con un “if”, no use el “switch”.

Ejercicios:

1. Modifique el último ejemplo, para apagar el RELAY1 y el RELAY2, si el RPM está entre 8001 y 1000.
2. Realice un script que emplee el condicional “switch” para leer la variable RPM, y en vez de activar o desactivar un RELAY, muestre el valor de la variable RPM en el display LCD del PLC. Consulte con el Manual de Programación Pawn del PLC.

5.5 Ciclos o loops

Un ciclo o loop, es un bloque de código que se ejecuta de forma repetitiva, mientras la condición del ciclo sea verdadera.

5.5.1 Ciclo “while”

El ciclo “while” es el más simple de todos. Mientras la condición es verdadera (true), se ejecutará cíclicamente el bloque de código siempre:

```
while(condicion)
{
	// Bloque de código.
}
```

La “condición”, puede ser una expresión o un valor (ya sea constante o de una variable). Si la condición resulta ser verdadera (true), es decir 1 o mayor a 1, el ciclo se repite. Si es falsa (false), es decir 0, el ciclo termina.

Veamos un ejemplo:

```
PlcMain()
{
	while(true)
	{
		// Conmutar RELAY1.
		RelayToggle(RELAY1)

		// Esperar 1000 ms.
		DelayMS(1000)
	}
}
```

El anterior ejemplo, pertenece al primer script de la página 3, en este caso el cliclo “while” siempre se ejecuta, ya que su condición es verdadera. También podríamos utilizar:

```
	while(1)
	{
		// Conmutar RELAY1.
		RelayToggle(RELAY1)

		// Esperar 1000 ms.
		DelayMS(1000)
	}
```

La constante “true” fue reemplazada por “1”, que es equivalente.

Supongamos que deseamos repetir un bloque de código, mientras la entrada discreta DIN1 del PLC sea igual a “1”:

```
while(true)
{
      // La entrada DIN1 está en alto ?
      while(DinValue(DIN1) == 1)
      {
            // Si, cerrar RELAY1.
            RelayClose(RELAY1)
      }

      // Abrir RELAY1.
      RelayOpen(RELAY1)


      // Esperar 300 ms.
      DelayMS(300)
}
```

En el código anterior, la función DinValue(DIN1) retorna “1” si la entrada fue activada, de lo contrario retorna “0”. Ver Manual de Programación Pawn del PLC. Para nuestro ejemplo, esto significa que mientras la entrada DIN1 este activa, el RELAY1 estará cerrado. Y la ejecución del script estará constantemente repitiendo el código del “while” que cierra el RELAY1.
Cuando la entrada DIN1 valga “0”, el ciclo “while” lee una condición falsa (0), y el script sale del ciclo. Finalmente, el RELAY1 se abre.

Se puede simplificar mas el código de la siguiente forma que resulta implícito:

```
while(true)
{
      // La entrada DIN1 está en alto?
      while(DinValue(DIN1))
      {
            // Si, cerrar RELAY1.
            RelayClose(RELAY1)
      }

      // Abrir RELAY1.
      RelayOpen(RELAY1)


      // Esperar 300 ms.
      DelayMS(300)
}
```

Como la función DinValue() retorna 0 o 1, no es necesario comprobar que sea igual a 1 el valor retornado.

También es posible comprobar expresiones en los “while”, de la misma forma que lo hacíamos en los condicionales “if” (ver página 8):

```
// Mientras Temperatura es igual a 35 y Humedad es >= 50,
// Ejecutar el siguiente bloque de código:

while( (Temperatura == 35) && (Humedad >= 50) )
{
      // Activar RELAY1.
      RelayClose(RELAY1)

      // Activar RELAY4.
      RelayClose(RELAY4)
}
```

El siguiente ejemplo, conmuta 10 veces el RELAY1, y luego sale del ciclo:

```
PlcMain()
{
      new Contador = 0

      while(true)
      {
            while(Contador < 10)
            {
                  // Conmutar RELAY1.
                  RelayToggle(RELAY1)

                  // Esperar 1000 ms.
                  DelayMS(1000)

                  // Incrementar contador en 1.
                  Contador = Contador + 1
            }
      }
}
```

La expresión “Contador < 10”, solo es válida, mientras la variable “Contador” sea menor a 10. Por lo tanto el relé, se conmutara 10 veces. Notar que empezamos a contar desde 0.

¿ Cómo podemos salir del ciclo “while” (o cualquier otro tipo de ciclo) de forma inmediata ?. Simple, con la palabra “break”. Que interrumpe la ejecución del ciclo desde donde se llama.

Por ejemplo, supongamos que deseamos salir del ciclo “while” cuando la variable contador llega al valor 5. Para ello utilizamos un “break” de la siguiente forma:

```
PlcMain()
{
	new Contador = 0

	while(true)
	{
		while(Contador < 10)
		{
			// Conmutar RELAY1.
			RelayToggle(RELAY1)

			// Esperar 1000 ms.
			DelayMS(1000)

			// Incrementar contador en 1.
			Contador = Contador + 1

			// El contador llego a 5 ?
			if(Contador == 5)
			{
				// Si interrumpir ciclo.
				break
			}
		}
	}
}
```

Cuando el valor del Contador llegue a 5, el ciclo se interrumpe.

5.5.2 Ciclo “do - while”

El ciclo “do – while”, es igual al “while”, solo que la condición se comprueba luego de ejecutar el bloque de código. Si la condición es verdadera (true), se vuelve a repetir el código:

```
	do
	{
		// Conmutar RELAY1.
		RelayToggle(RELAY1)

		// Esperar 1000 ms.
		DelayMS(1000)

		// Incrementar contador en 1.
		Contador = Contador + 1
	}
	while(Contador < 10)
```

Puede resultar útil, cuando necesitamos que se ejecute un bloque de código al menos una vez.

5.5.3 Ciclo “for”

El ciclo “**for**”, es uno de los más útiles para operaciones que deben repetirse una cierta cantidad de veces. Tiene la siguiente sintaxis:

```
for(Expresión 1 ; Expresión 2 ; Expresión 3)
{
      // Bloque de código.
}
```

Donde:

- Expresión 1: Se evalúa solo una vez, antes de entrar al ciclo. Esta expresión puede ser utilizada para inicializar una variable por ejemplo. También es posible crear una variable con “new” que será validad solo en el ciclo “for”. No se puede combinar variables existentes con nuevas.

- Expresión 2: Evaluada antes de cada interacción del ciclo. Si resulta falsa (false), finaliza la ejecución del ciclo. Si se omite, se considera siempre verdadero la condición. Esta expresión es la condición que determina si se ejecuta o no el ciclo.

- Expresión 3: Evaluada luego de ejecutar cada bloque de código del ciclo.

El siguiente ejemplo, conmuta 10 veces un relé cada 1 segundo (similar al ejemplo con “while” en página 18):

```
PlcMain()
{
      for(new Contador = 0; Contador < 10; Contador++)
      {
            // Conmutar RELAY1.
            RelayToggle(RELAY1)

            // Esperar 1000 ms.
            DelayMS(1000)
      }

      while(true)
      {
            // Apagar y prender el led de DEBUG cada 500 mS.
            LedToggle()
            DelayMS(500)
      }
}
```

El script anterior, comienza con el ciclo “for”. Inicialmente, se crea una variable Contador, que se inicializa a 0 (expresión 1). Luego se comprueba que Contador sea menor a 10, si es verdadero, se ejecuta el código del ciclo (expresión 2).

En cada interacción, luego de que el bloque se ejecuta, se procede a incrementar en 1, la variable Contador, utilizando la expresión “Contador++”, que corresponde a la expresión 3.

El operador "++" se llama incremento, y es equivalente a:

Contador++ → Contador = Contador + 1

También existe el operador "--", que hace un decremento de uno en la variable:

Contador-- → Contador = Contador - 1

Finalmente, el ciclo "**while**" del final del script, se ejecuta infinitamente, activando y desactivando un led de prueba (DEBUG) de la placa.

La variable **Contador** solo existe dentro del ciclo "for", luego deja de existir. Es posible crearla e inicializarla afuera del ciclo también:

```
new  Contador = 0

for( ; Contador < 10; Contador++)
{
      // Conmutar RELAY1.
      RelayToggle(RELAY1)

      // Esperar 1000 ms.
      DelayMS(1000)
}
```

Con el separador "," (coma) es posible usar varias operaciones en las Expresiones 1 y 3. Por ejemplo, supongamos crear 2 variables **Contador:**

```
for(new Contador1=0, Contador2=10 ; Contador1 < 10; Contador1++, Contador2--)
{
      // Imprimir en el Display LCD los valores de ambos contadores.
      LcdPrintf(0, 0, "C1 = %02d, C2 = %02d", Contador1, Contador2)

      // Esperar 1000 ms.
      DelayMS(1000)
}
```

Dos variables fueron creadas, Contador1 y Contador2, la primera se inicializa con 0 y la segunda con 10. El ciclo se ejecutará 10 veces, mientras Contador1 sea menor a 10. La variable Contador1 se incrementa y la variable Contador2 se decrementa, al final de cada interacción del ciclo.

En el display LCD, se imprimirán los valores de ambas variables. Ver Manual de Programación Pawn del PLC, para saber más sobre la función "LcdPrintf()" o consulte la página 35.

Al igual que en los ciclos "**while**", aquí también es posible utilizar la palabra "break" para salir del ciclo en cualquier momento. Ver página 19.

Para hacer más “emocionante” el tema, existe la palabra “**continue**”, que al ser llamada, provoca que el ciclo vuelva a llamarse inmediatamente, sin llegar al final.

En el siguiente ejemplo, si **Contador** es mayor o igual a 5, se provoca una iteración forzada del ciclo, haciendo que se conmute el RELAY2 y no el RELAY1.

```
for(new Contador = 0; Contador < 10; Contador++)
{
	// Contador es mayor o igual 5 ?
	if(Contador >= 5)
	{
		// Conmutar RELAY2.
		RelayToggle(RELAY2)

		// Esperar 1000 ms.
		DelayMS(1000)

		continue
	}

	// Conmutar RELAY1.
	RelayToggle(RELAY1)

	// Esperar 1000 ms.
	DelayMS(1000)
}
```

El uso de la palabra clave “**continue**”, se aplica en pocos casos prácticos, pero es bueno conocerlo.

Finalmente, es posible dejar en blanco alguna expresión del ciclo “for”:

```
for( ; Contador < 10; Contador++)
{
	// Código
}

for( ; Contador < 10; )
{
	Contador++
}
```

La siguiente expresión:

```
for( ; ; )
{
	// Código
}
```

→

Equivale a:

```
while(true)
{
	// Código
}
```

Ejercicio: Cree un ciclo “for” para obtener la variable “a” elevada a la “5”. Donde “a” vale inicialmente “2”.

5.6 Expresiones

Las expresiones son declaraciones contenidas en una línea, similar a las conocidas en matemática. Son grupos de símbolos que devuelven una pieza de información

Las expresiones son normalmente comprendidas por operaciones con paréntesis, y son evaluadas en un cierto orden (primero los paréntesis, luego las multiplicaciones, divisiones, sumas, restas, etc).

```
// La siguiente expresión, es muy simple, devuelve solo el numero 0.
0

// Es posible también, ponerla entre paréntesis.
(0)
```

Una expresión que vale 0, se considera que devuelve un “false” o falso.

Si una expresión **no es cero** no es "false", no solo devuelve un valor, también se devuelve como "true". De otra manera, devolverá 0 como "false".

```
// Aquí se encuentran mas expresiones matemáticas.
// Los operadores matemáticos son:
//
// + para SUMA
// - para RESTA
// * para MULTIPLICACIÓN
// / para DIVISIÓN
// % para Módulos (encontrar el resto de un número dividido por otro,
//   por ejemplo, 5%2 es 1)


(5+6)                       // Devuelve 11
((5*6)+3)                   // Devuelve 33
(((5+3) /2) *4) -9)         // Devuelve 5
((5*6) % 7)                 // Devuelve 2

// Otras expresiones:

(true)                      // Devuelve el estado "true"
(5.0 + 2.3)                 // Devuelve 7.3 como un Float.
```

También hay extensiones en estos operadores para poder ser utilizados en variables.

```
new a = 5
new b = 6

// POST y PRE operadores de incremento.

a++   // Devuelve a+1, o 6. Esto es un POST incremento.
++a   // Devuelve a+1, o 6. Esto es un PRE Incremento.
```

La diferencia entre el PRE y POST incremento es simple pero importante: a++ es evaluado al final en una expresión, cuando ++a es evaluado primero.

Esta diferencia es práctica para los códigos que usan ciclos. También es importante señalar que los operadores de incremento/decremento no solo devolverán a+1, sino que alteran el valor de la variable incrementándola o decrementándola.

En operaciones que modifican variables, utilizando como valor de entrada la variable en cuestión, es posible hacer lo siguiente:

Expresiones:		Equivalente a:
`a += 5`	→	`a = a + 5`
`a -= b + 4`	→	`a = a – (b + 4)`
`a *= 7`	→	`a = a * 7`
`a /= 2`	→	`a = a / 2`
`a %= 2`	→	`a = a % 2`

Para tomar decisiones lógicas, existen los operadores && (AND) y || (OR), que fueron discutidos en la pagina 11.

El operador "AND" toma dos expresiones (izquierda y derecha), si las dos son "true", devuelve el estado "true".

```
// Esto es falso, ya que 1 devuelve "true" pero 0 devuelve "false",
// ya que ambos no son "true", devolverá "false".

(1 && 0)

// Ambos números son "true", entonces la expresión será "true"
(1 && 2)

(true && false)    // Resultado: "false"

(false && false)   // Resultado: "false"

(true && true)     // Resultado: "true"
```

“true” y “false” son constantes booleanas, definidas por el lenguaje.

El operador "OR" toma dos expresiones (izquierda y derecha), **solo si alguna** es "true", devuelve el estado "true".

```
(1 || 0)           // "true", ya que uno de los valores es "true"
(1 || 2)           // "true"
(true || false)    // "true"
(false || false)   // "false"
(true || true)     // "true"
```

Existe el operador “negación”, que tiene el símbolo “!”. Este operador transforma una expresión cuyo resultado es “true” en “false”, y viceversa:

```
!(false)            // Resultado “true”
!(true)             // Resultado “false”

!(1)                // Resultado “false”
!(0)                // Resultado “true”

!(true && true)     // Resultado: "false"

new a = 38782
!(a)                // Resultado: "false"

new a = 0
!(a)                // Resultado: "true"
```

Por último existen los operadores “bitwise” o “bit a bit”, que permiten manipular un bit o un grupo de bits de una expresión o variable. A continuación una lista de los mismos:

Tabla: Símbolos para manipular bits en expresiones (e1 y e2) - (ver también página 46)

Símbolo	Descripción	Precedencia
~	~e Produce el complemento a uno de e.	derecha a izquierda
>>	e1 >> e2 Produce un desplazamiento aritmético a la derecha en e1 por e2 bits. La operación tiene signo, es decir el bit mas a la izquierda de e1 que representa el signo, es copiado en los bits vacantes del resultado.	Izquierda a derecha
>>>	e1 >>> e2 Produce un desplazamiento lógico a la derecha en e1 por e2 bits. La operación no tiene signo, los bits vacantes del resultado, son rellenados con 0.	Izquierda a derecha
<<	e1 << e2 Produce un desplazamiento a la izquierda de e1 por e2 bits. La operación no tiene signo. No hay distinción entre un desplazamiento a la izquierda lógico o aritmético.	Izquierda a derecha
&	e1 & e2 Resulta en el AND lógico bit-a-bit de e1 y e2.	Izquierda a derecha
|	e1 | e2 Resulta en el OR lógico bit-a-bit de e1 y e2.	Izquierda a derecha
^	e1 ^ e2 Resulta en el OR exclusivo (XOR) bit-a-bit de e1 y e2.	Izquierda a derecha

Si necesita alterar algunos bits o leer específicamente algún bit, nuestro PLC ofrece funciones (macros) para simplificar el uso de operadores “bit-a-bit”. En el Manual de Programación Pawn del PLC, podrá encontrar dichas funciones: BitSet(), BitClr(), BitRead(), NBitToggle, NBitSet(), NBitClr(), NBitRead() y NBitToggle.

Los operadores “bit-a-bit”, realizan una operación lógica, entre todos los bits de dos expresiones.

Por ejemplo, tenemos dos números y aplicamos un & (AND):

```
(5 & 3)
```

Resulta en: “1”.

Esto es fácil de corroborar, si hacemos la cuenta con números binarios para ambos números:

```
 1   0   1    → 5 (decimal)
 &   &   &    → AND
 0   1   1    → 3 (decimal)
-------------
 0   0   1    → 1 (decimal)
```

En el caso de hacer un | (OR):

```
(5 | 3)
```

Resulta en: “7”.

```
 1   0   1    → 5 (decimal)
 |   |   |    → OR
 0   1   1    → 3 (decimal)
-------------
 1   1   1    → 7 (decimal)
```

Una operación útil del AND, es asegurarse que una variable, tenga todos los bits en cero, excepto un determinado grupo (para limitar el rango de la variable), por ejemplo:

```
new a = 255

a &= 15       // a = a & 15 -> 11111111 & 00001111 = 00001111 = 15 (decimal)
```

Resulta en 15.

Es posible utilizar notación hexadecimal para constantes anteponiendo “0x” al número, por ejemplo:

```
new a = 255

a &= 0x0F   // a = a & 0x0F -> 11111111 & 00001111 = 00001111 = 15 (decimal)
```

Si necesita solo modificar o leer un bit, recomendamos ver las funciones de manipulación de bits de datos del Manual de Programación Pawn del PLC.

Ejercicios:

1. Multiplique dos variables “a” y “b”. El resultado asígnelo a otra variable “c”.
2. Si una variable “a=20” se le aplica el siguiente operador “a = a << 1” cuál es el resultado que se obtiene ?. Esto equivale a multiplicar por 2 ?. Si efectuó el proceso inverso “a = a >> 1” ?. Divido por 2 ?.

5.7 Funciones

Las funciones agrupan código que se utiliza frecuentemente. Son muy importantes para optimizar la escritura de código.

Veamos el siguiente caso ejemplo:

Si queremos aplicar una ecuación a dos variables, para promediar su valor, hacemos:

```
new Edad1 = 10
new Edad2 = 6
new Promedio

Promedio = (Edad1 + Edad2) / 2
```

Si frecuentemente hacemos ese cálculo, debemos repetir el código:

```
Promedio = (Edad1 + Edad2) / 2
```

Una forma de mejorar el código, es encapsularlo en una función, llamada “Promediar”:

```
Promediar(a, b)
{
      new c

      c = (a + b) / 2

      return c
}
```

Finalmente, llamamos a la función “Promediar” y le pasamos los dos valores:

```
Promedio =  Promediar(Edad1, Edad2)
```

La función “Promediar()” retorna el promedio entre los dos valores que se pasan en sus argumento: a y b. El nombre de los argumentos puede ser cualquiera. El retorno de la función, ocurre cuando se llama a la palabra clave “return”.

El programa completo resulta con el siguiente código:

Ver página Siguiente.

```
PlcMain()
{
      new Edad1 = 10
      new Edad2 = 6
      new Promedio

      // Promediar valores.
      Promedio =  Promediar(Edad1, Edad2)

      // Imprimir resultado en display LCD.
      LcdPrintf(0, 0, “Promedio = %d”, Promedio)


      while(true)
      {
            // Apagar y prender el led de DEBUG cada 500 mS.
            DelayMS(500)
            LedToggle()
      }
}

// Función para promediar dos valores: a y b.

Promediar(a, b)
{
      new c

      c = (a + b) / 2

      return c
}
```

En el script anterior, se encuentra el código completo, que promedia valores mediante una función llamada “Promediar()”.

Notemos que la función, se define a un nivel global, es decir fuera de otras funciones. Por lo tanto es globalmente accesible.

La función, si retorna valor, lo hace mediante la palabra “**return**”.

Los argumentos de la función (en este caso “a” y “b”), son variables implícitas de la función. Las variables pasadas como argumento, así como las variables creadas dentro de la función, son del tipo local. Esto significa, que no son accesibles por otras funciones.

La función “PlcMain()” es la función principal del programa, representa otro ejemplo de función.

Si una función no retorna valor, no hace falta especificar la palabra “**return**”. Pero por buena práctica, nosotros recomendamos retornar el valor “0”, en caso de no ser necesario retornar ningún valor.

Por ejemplo, la siguiente función imprime la palabra “hola” en el display de la placa. No requiere argumentos, y el valor de retorno es opcional, pero como dijimos anteriormente, si no necesitamos retornar nada, utilizamos “0” como valor de retorno.

```
Hola()
{
      LcdPrintf(0, 0, “hola”, Promedio)

      return 0
}
```

Cuando llamemos a “Hola()”, en el display, se imprimirá “Hola”. Para más información sobre imprimir en el display del PLC, consulte la página 35.

Cuando una función, ejecuta “**return**”, la función retorna inmediatamente. Esto es útil, cuando tenemos varios puntos de retorno.

Supongamos que necesitamos retornar el valor 2, si las entradas DIN1 y DIN2 del PLC están activas, y retornar el valor “0” en caso de que ambas no estén activas. Podemos realizar la siguiente función:

```
ComprobarEntradas()
{
      // Comprobar si entrada DIN1 y DIN2 valen “1”.
      if(DinValue(DIN1) && DinValue(DIN2))
      {
            return 2
      }
      else
      {
            return 0
      }
}
```

Desde PlcMain() podemos llamarla, y en caso que el valor retornado sea “2”, sabemos que ambas entradas están activas. El siguiente código, solo activa y desactiva el led DEBUG, si la función retorna “2”.

```
PlcMain()
{
      while(true)
      {
            // Entradas DIN1 y DIN2 activadas ?.
            if(ComprobarEntradas() == 2)
            {
                  // Apagar y prender el led de DEBUG cada 500 mS.
                  DelayMS(500)
                  LedToggle()
            }
      }
}
```

5.7.1 Funciones – Argumentos por Referencia

Hasta ahora, solo vimos funciones cuyos argumentos se pasan por valor. Cuando pasamos por valor, la variable recibida en la función, es una copia de la variable que utilizamos para llamar a la función, por ejemplo:

```
Suma(a, b)
{
      return a+b
}
```

Al llamar a la función (desde PlcMain() por ejemplo):

```
new numero1 = 10
new numero2 = 15
new resultado

resultado = Suma(numero1, numero2)  // El valor de resultado será 25.
```

Las variables "**numero1**" y "**numero2**", no son visibles desde la función **Suma**(). En este caso, se crean dos variables nuevas en memoria RAM, llamadas "**a**" y "**b**", y los valores de "**numero1**" y "**numero2**" se copian respectivamente.

Supongamos que deseamos almacenar el resultado de la función, en otro argumento, llamado "**c**". Pero la función, no retorna la suma sino, "**0**". Para ello, podemos pasar un tercer argumento como referencia a una variable externa, anteponiendo el símbolo "**&**":

```
Suma(a, b, &c)
{
      c = a + b

      return 0
}
```

Al llamar a la función (desde PlcMain() por ejemplo):

```
new numero1 = 10
new numero2 = 15
new resultado

Suma(numero1, numero2, resultado)  // El valor de resultado será 25.
```

Notemos la diferencia: la variable "resultado" se pasa por referencia, esto quiere decir que "Suma()" accede directamente a la variable, pudiendo leer su valor y modificarlo. Al retornar la función, la variable "resultado" ya tiene el valor cargado y no es necesario asignarlo con el símbolo "=".

Dentro de suma, si modificamos "**c**", alteramos directamente a la variable "**resultado**". Si modificamos "**a**" o "**b**", no alteramos a "**numero1**" y "**numero2**", ya que ambos se pasaron como valor (copia en memoria).

5.7.2 Funciones – Argumentos por Referencia - Arrays

Los arreglos o arrays, siempre se pasan por referencia.

Por ejemplo, si tenemos un array de 3 números, es posible inicializarlo desde una función:

```
PlcMain()
{
      new Vector[3]

      // Obtener posición en el espacio.
      PosicionXYZ(Vector)

      // Imprimir posicion en display LCD.
      LcdPrintf(0, 0, “X = %d Y = %d Z = %d”, Vector[0], Vector[1], Vector[2])

      while(true)
      {
            // Apagar y prender el led de DEBUG cada 500 mS.
            DelayMS(500)
            LedToggle()
      }
}

// Función que inicializa un vector con la posición X Y Z.
PosicionXYZ(Vect[])
{
      Vect[0] = 23     // Posición X.
      Vect[1] = -34    // Posición Y.
      Vect[2] = 2      // Posición Z.

      return 0
}
```

En el código anterior, el array “**Vector**” es pasado por referencia a la función “**PosicionXYZ**()”, la cual accede directamente al array y modifica su valores. Note que la función específica un array como entrada con los “[“ “]” que se agregan a “**Vect**”.

El tamaño del array es opcional, pero para mejorar la comprobación de errores del compilador, si sabemos que solo un array de 3 elementos puede ser pasado a la función, podemos hacer:

```
PosicionXYZ(Vect[3])
{
      Vect[0] = 23     // Posición X.
      Vect[1] = -34    // Posición Y.
      Vect[2] = 2      // Posición Z.

      return 0
}
```

5.7.3 Funciones – Punto Flotante

A continuación, damos ejemplos para funciones que requieren argumentos del tipo “float” o retornan valores “float”.

La siguiente función retorna la suma de dos números “float”.

```
Float:SumaF(Float:a, Float:b)
{
	new Float: c

	c = a + b

	return c
}
```

Para llamarla de PlcMain(), hacemos:

```
	new Float: Resultado

	Resultado = Float: SumaF(1.5, 10.7)	// Debería resultar en 12.2
```

Código completo:

```
PlcMain()
{
	new Float: Resultado

	Resultado = Float: SumaF(1.5, 10.7)

	// Imprimir resultado en display LCD.
	LcdPrintf(0, 0, “R = %f”, Resultado)

	while(true)
	{
		// Apagar y prender el led de DEBUG cada 500 mS.
		DelayMS(500)
		LedToggle()
	}
}

Float:SumaF(Float:a, Float:b)
{
	new Float: c

	c = a + b

	return c
}
```

5.7.4 Funciones – Variables Globales

Es posible compartir una variable entre varias funciones, si la variable es declarada globalmente, fuera de cualquier función.

Por ejemplo, la variable “Compartida”, es visible desde todo el script:

```
new Compartida = 0      // Variable global.

PlcMain()
{
      // Incrementar la variable compartida, desde 3 funciones.
      IncrementarCompartida1()
      IncrementarCompartida2()
      IncrementarCompartida3()

      // Imprimir valor de variable en display LCD.
      LcdPrintf(0, 0, “C = %d”, Compartida)

      while(true)
      {
            // Apagar y prender el led de DEBUG cada 500 mS.
            DelayMS(500)
            LedToggle()
      }
}

IncrementarCompartida1()
{
      Compartida++
      return 0
}

IncrementarCompartida2()
{
      Compartida++
      return 0
}

IncrementarCompartida3()
{
      Compartida++
      return 0
}
```

Inicialmente, la variable “**Compartida**” se define con el valor “0” fuera de todas las funciones, haciéndola globalmente accesible desde todas las funciones. Las funciones **IncrementarCompartida1()**, **IncrementarCompartida2()** e **IncrementarCompartida3()**, la incrementan en 1 cada vez que son llamadas desde la función **PlcMain**(). El resultado final, debería ser “**Compartida** = 3”.

5.8 Strings – Cadenas de Caracteres y Caracteres

Los “strings” o cadenas de caracteres, es una sucesión de de celdas que contiene letras o caracteres. En PAWN, los “strings” funcionan igual que los arrays (ver página 6), con la diferencia que la ultima celda, termina en 0 (cero).

Los “strings” se aplican en nuestros PLC generalmente para imprimir mensajes cortos en el display LCD. La función nativa “LcdPrintf()” es especialmente útil para este propósito. Recomendamos ver el Manual de Programación Pawn del PLC para más información.

Un string se puede definir de la siguiente forma:

```
new Saludo[] = “Hola Slicetex”
```

El array Saludo[] se inicializa con el string “Hola Slicetex”, que está encerrado por comillas “ ”.

Dicho string, contiene un carácter por celda, mas el terminador 0 (cero), la estructura es:

Celda 0	Celda 1	Celda 2	Celda 3	Celda 4	Celda 5	Celda 6	Celda 7	Celda 8	Celda 9	Celda 10	Celda 11	Celda 12	Celda 13
H	o	l	a		S	l	i	c	e	t	e	x	0

Notemos que el string consume 14 celdas (caracteres + terminador). Es conveniente racionalizar el uso de strings, ya que este mensaje consume 14 x 4 bytes = 56 bytes de memoria RAM, y solo es utilizado para caracteres.

Si queremos obtener el carácter “S”, podemos indexar el array, como sigue:

```
new Caracter
new Saludo[] = “Hola Slicetex”

Caracter = Saludo[5]    // Leer celda número 5.
```

Es posible inicializar el array como un string, de la siguiente forma (atención al tamaño del array):

```
new Saludo[5] = {‘H‘, ‘o‘, ‘l‘, ‘a‘, 0}  // 4 caracteres + terminador.
```

En el inicializador, debemos especificar el tamaño del array de antemano. Como cada celda tiene un valor, debemos asignarle el valor ASCII de cada carácter que le corresponde, esto se logra encerrando un carácter con comillas simples ‘ ‘.

Otra posibilidad para inicializar el array es:

```
new Saludo[5]

Saludo[0] = ‘H‘
Saludo[1] = ‘o‘
Saludo[2] = ‘l‘
Saludo[3] = ‘a‘
Saludo[4] = 0
```

Como pudimos observar los “strings” se manejan como arrays, contienen caracteres y terminan en 0.

Para imprimir los “strings” en el display LCD podemos utilizar la función “LcdPrintf()”, cuyos detalles de operación completos, se detallan en el Manual de Programación Pawn del PLC.

Básicamente:

```
LcdPrintf(0, 0, “Hola”)
```

Imprime la cadena o string “Hola” (recordar que esta encerradas por comillas dobles, y eso la convierte en una cadena de caracteres) en la columna 0 y línea 0 del display LCD.

La función LcdPrinf() puede imprimir una cadena con un formato, esto quiere decir que si la cadena contiene un código de formato precedido por el símbolo “%”, remplazara dicho código por el argumento mas próximo de la función. Por ejemplo, para números utilizamos “%d”:

```
LcdPrintf(0, 0, “Numero = %d”, 55)
```

Imprime “Numero = 55” (sin las comillas) en el display LCD, ya que el %d le dice a la función que hay un numero en la lista de argumentos a su derecha.

Para dos números:

```
LcdPrintf(0, 0, “%d  %d”, 55, 66)  // Imprime “55 66” en el display LCD
```

Para imprimir un string, utilizamos el código “%s” que representa un string:

```
new Saludo[] = “Hola Slicetex”

LcdPrintf(0, 0, “%s”, Saludo)
```

Tambien es posible:

```
LcdPrintf(0, 0, “%s”, “Hola Slicetex”)
```

Finalmente, si queremos imprimir un número punto flotante, lo hacemos con %f:

```
LcdPrintf(0, 0, “Numero %f”, 5.5)
```

De nuevo, consulte con el Manual de Programación Pawn del PLC, allí se explica en profundidad.

Puede ser útil utilizar la función “LcdClear()” para borrar todo el display LCD.

Ejercicios:

1. Cree un script que borre el display LCD e imprima su nombre.
2. Realice un script que cree un string e inicialícelo con su apellido. Luego imprímalo en la segunda línea del display LCD.

5.9 Eventos

Los eventos son funciones que son llamadas asincrónicamente (es decir en cualquier momento) por el PLC cuando se produce un evento determinado (por ejemplo llego un dato al puerto Ethernet, una entrada cambio de valor, etc.). En el Manual de Programación Pawn del PLC se explican todos los eventos disponibles, con mayor detalle.

Una función que maneja un evento en el script, empieza con el carácter "@". En PAWN este tipo de funciones se llaman "funciones públicas". Los nombres de dichas funciones están definidos en el Manual de Programación Pawn del PLC.

Veamos el siguiente ejemplo:

```
// Variable contador que se incrementa desde el evento timer.
new Contador = 0

PlcMain()
{
   // Configurar Timer1 generar un evento cada 1000 mS.
   Timer1SetEvent(1000, true)

   while(true)
   {
      // Imprimir variable Contador.
      LcdPrintf(0, 0, "C = %04d", Contador)

      // Apagar/Encender Led.
      DelayMS(150)
      LedToggle()
   }
}

@OnTimer1()
{
   // Incrementar contador.
   Contador++

   // Limitar "Contador" al valor 255.
   if(Contador > 255)
   {
      Contador = 0
   }
}
```

En el código anterior, se activa desde "**PlcMain**()" al Timer1, que genera un evento "**@OnTimer1**" cada 1000 mS. La función "**PlcMain()**" solo imprime cíclicamente el valor de la variable global "**Contador**" en el display LCD.
Cuando transcurren 1000 mS, se interrumpe la ejecución de "**PlcMain()**" y la función "**@OnTimer1()**" es llamada. La variable "**Contador**" se incrementa, se comprueba que no sea mayor a 255, y luego retorna.
Como "**@OnTimer1**" interrumpe la ejecución del programa principal, es aconsejable realizar el menor número de operaciones (o evitar llamar a funciones lentas) de tal forma de no consumir tiempo de procesador que podría emplearse en la función **PlcMain()**. Es posible tener múltiples eventos también.

Algunas consideraciones:

- Nunca dos eventos se ejecutan al mismo tiempo. El PLC procesa los eventos pendientes en orden secuencial.
- Un evento es una condición especial, recuerde que otros eventos pueden estar pendientes y necesitan ejecutarse. Por lo tanto (según la aplicación) intente ejecutar la menor cantidad de código posible para aumentar la velocidad de procesamiento.
- Las funciones "**DelayMS**() y **DelayS**()" **no pueden llamarse desde un evento**. Además, no es conveniente generar un retardo desde un evento, ya que es tiempo perdido que puede utilizarse para procesar otro evento. Si desea igualmente generar un retardo desde un evento, utilice las funciones **PauseXX**().Remitirse al Manual de Programación Pawn del PLC.
- La instrucción "**sleep**" puede utilizarse de la misma forma que las funciones "Delay", y tampoco puede llamarse desde un evento. Remitirse al Manual de Programación Pawn del PLC.

5.10 Directivas de Pre-Procesador, Macros y Constantes

Las directivas de "Pre-Procesador" le indican al compilador de PAWN directivas que no deben compilarse como código, sino que deben evaluarse antes de compilar y realizar una sustitución en el código o alterar una opción de compilación.

Las directiva más utilizada es la llamada "#define", que define un macro o sustitución.

Por ejemplo, si tenemos una constante numérica, que define la cantidad de muestras que debemos tomar desde una entrada analógica, podemos definirla como:

```
#define MUESTRAS_TOTAL        (10)
```

En nuestro código, cuando hagamos una referencia a "MUESTRAS_TOTAL", se reemplazara por el numero "10". Muy útil para hacer legible el código. Como consejo, siempre que nombremos a constantes, utilicemos "MAYUSCULAS".

Una posible aplicación, puede ser para dimensionar un "array" que contendrá las muestras:

```
new Muestras[MUESTRAS_TOTAL]
```

El array "**Muestras**[]", tendrá un tamaño de 10. Como vemos, el código es más legible y fácil de modificar.

El código completo seria:

```
#include <stx8081.inc>

#define MUESTRAS_TOTAL        (10)

PlcMain()
{
   new Muestras[MUESTRAS_TOTAL]

   while(true)
   {
      // Apagar/Encender Led.
      DelayMS(150)
      LedToggle()
   }
}
```

Notemos que el "define" debe realizarse al principio del código del script. **Todas las directivas deben estar en las primeras líneas de código.**

La directiva "#include <stx8081.inc>" es otra directiva de pre-procesador. Su función consiste en llamar un archivo stx8081.inc (un archivo encabezado), que contiene declaraciones y también otras muchas directivas "#define" y macros, para hacerlas visibles en su script. Esta directiva ya la incluye automáticamente **StxLadder**, pero es posible explicitar su uso llamándola.

Usted puede crear sus propios archivos .inc, con constantes y declaraciones.

Parta incluir un archivo (por ejemplo: archivo.inc) desde el actual directorio del script:

```
#include “archivo.inc” // Note las comillas que encierran al archivo.inc
```

5.10.1 Macros

Un macro es similar a una función, solo que contiene “literalmente” una porción de código. Se crea con la directiva “#define”.

Supongamos realizar la siguiente operación suma con un macro:

```
#define Suma(%1,%2)           ((%1) + (%2))
```

Desde nuestro script podemos llamarla de la siguiente forma:

```
new a = 5
new b = 2
new c

c = Suma(a, b)                // Suma con macro.
```

Como un **macro** es una sustitución literal de código, la línea “c = Suma(a, b)” es vista por el compilador (una vez que se procesan las directivas del pre-procesador) de la siguiente manera:

```
c = ((a) + (b))               // Suma con macro (expandido)
```

Es por ello que en los macros es **muy importante utilizar los paréntesis** “(“ “)”, para envolver sus argumentos y expresiones. De tal forma, que una vez expandidos se evalúen correctamente.

Un macro tiene muchas limitaciones, debería utilizarse para operaciones básicas y **NO reemplazan a las funciones**.

Pueden utilizarse para renombrar funciones de una forma fácil. Por ejemplo, si la función LcdPrintf() la utilizamos seguido para imprimir en la primera línea del display, con solo mensajes, podemos simplificarla de la siguiente manera con un macro:

```
#define Imprimir(%1)          (LcdPrintf(0, 0, (%1))
```

Ahora desde nuestro código, podemos llamar al macro:

```
Imprimir(“Hola!”)             // Imprime en display, posición 0,0.
```

El compilador, cuando encuentre Imprimir(“Hola”) lo expandirá como sigue:

```
(LcdPrintf(0, 0, (“Hola!”))   // Imprimir con macro (expandido)
```

Los macros son una poderosa herramienta. Puede escribirlos en un archivo separado (ej: macros.inc) y luego incluirlos con la directiva #include “macros.inc”.

Información: Las funciones **DelayMS**() y **DelayS**() son macros que utilizan la instrucción **sleep** y están definidos en **delay.inc**. Dejamos como tarea examinarlos para comprender aun más el sistema.

5.10.2 Directivas #if

Las directivas #if, #elseif, #else y #endif, permiten excluir ciertas partes del código de un script de acuerdo a una condición. Su uso es:

```
#if expresión_constante
      // código1
#else
      // código2
#endif
```

Si "**expresión_constante**" es distinta a "0", se compila el "**código1**" de lo contrario, se compila el "**codigo2**".

Por ejemplo:

```
#define PRUEBA    (1)

PlcMain()
{
      while(true)
      {
            #if PRUEBA

            LedToggle()
            DelayMS(500)

            #else

            RelayToggle(RELAY1)
            DelayMS(1000)

            #endif
      }
}
```

En el ejemplo anterior, si PRUEBA es igual a 1, se compila el código para conmutar un Led. De lo contrario se compila el código para conmutar el RELAY1. Esto es muy útil cuando hacemos pruebas o queremos distintos comportamientos del programa.

Notemos que al final siempre se termina con un #endif. El campo #else, es opcional.

Para varios condicionales, se puede utilizar el **#elseif** como a continuación:

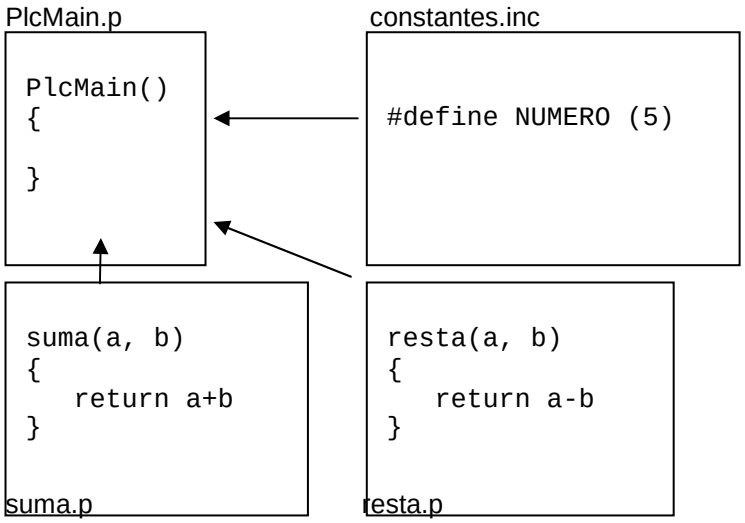
```
#if expresión_constante1
      // código1
#elseif expresión_constante2
      // código2
#else
      // codigo3
#endif
```

5.11 Múltiple Scripts – Organización del Código

Si su script crece rápidamente y como consecuencia se acumulan demasiadas líneas de código en su archivo ".p", es posible que empiece a confundirse y el código se vuelva desordenado.

Para ello, podemos crear varios archivos o scripts ".p" en el entorno **StxLadder**, en donde almacenamos funciones que son comunes a ciertos procesos y las llamamos desde un script principal.

Para ilustrar la idea, veamos el siguiente grafico:



Tenemos 3 archivos script y uno de encabezado:

- Archivo PlcMain.p : Contiene la función principal.
- Archivo suma.p: Contiene función para sumar.
- Archivo resta.p: Contiene función para restar.
- Archivo constantes.inc: Contiene constantes #define.

El objetivo es desagrupar código para no escribirlo todo junto en el archivo PlcMain.p, de tal forma que el archivo se mantenga fácil de leer. Entonces escribimos funciones en otros archivos y luego las accedemos desde PlcMain() como se ejemplifica a continuación:

Ver código en siguiente pagina

```
#include “constantes.inc”     // Incluimos constantes del archivo.

PlcMain()
{
      new c
      new d

      // Sumamos 10 + NUMERO.
      c = suma(10, NUMERO)

      // Restamos 10 - NUMERO.
      d = resta(10, NUMERO)

      // Limpiar LCD.
      LcdClear()

      // Imprimir resultado en display LCD.
      LcdPrintf(0, 0, “s = %d r = %d”, c, d)

      // Ciclo principal del programa.
      while(true)
      {
            // Hacer nada…
      }
}
```

El código anterior muestra cómo es posible acceder desde el archivo PlcMain.p a constantes y funciones definidas en otros archivos.

En el entorno StxLadder es posible especificar la inclusión de archivos de forma grafica, a través de las propiedades de cada archivo desde el Explorador de Proyecto. Para ello, seleccionar dentro de propiedades del archivo la casilla “**Incluir en proyecto al compilar**”.

5.12 El Próximo Paso

Si leyó y comprendió toda la guía solo resta dejar volar la imaginación, crear poderosos scripts, cargarlos al PLC y empezar a automatizar el mundo!.

Bueno…calma, antes de lanzarse a conquistar el mundo le sugerimos:

- Leer la hoja de datos del PLC. Allí esta toda la información para evitar dañar eléctricamente al PLC.
- Leer el Manual de Programación Pawn del PLC. En este manual se describen todas las funciones nativas soportadas.
- **Leer la documentación oficial de PAWN (ver páginas** ¡Error! Marcador no definido. **y 3). Allí encontrara como exprimir al máximo el lenguaje. Es altamente recomendado.**
- Ingresar periódicamente a nuestro sitio web (www.slicetex.com) para estar al tanto de las actualizaciones y novedades. También brindamos un FORO de discusión para soporte.

6 Anexo

6.1 Palabras reservadas

Las siguientes son palabras que están reservadas uso del compilador y el lenguaje:

Sentencias	Operadores	Directivas	Otros
assert	char	#assert	const
break	defined	#define	enum
case	sizeof	#else	forward
continue	state	#elseif	native
default	tagof	#endif	new
do		#endinput	operator
else		#error	public
exit		#file	static
for		#if	stock
goto		#include	
if		#line	
return		#pragma	
sleep		#section	
state		#tryinclude	
switch		#undef	
while			

6.2 Constantes predefinidas

Las siguientes son constantes reservadas, definidas por el compilador y no pueden re-definirse. Puede usarlas en su código si le son de utilidad.

Constante	Descripción
cellbits	El tamaño de una celda, usualmente 32.
cellmax	El mayor valor valido de un número positivo para una celda. Usualmente 2147483647.
cellmin	El mayor valor valido de un número negativo para una celda. Usualmente - 2147483648.
charbits	El tamaño de un carácter packed en bits. Usualmente 8.
false	Constante con valor 0 del tipo bool:
true	Constante con valor 1 del tipo bool:
__Pawn	La versión del compilador PAWN en formato BCD (binary coded decimals).
charmin, charmax, debug, __line, ucharmax	Otras constantes, ver Pawn_Language_Guide.pdf.

6.3 Identificadores

Un identificador es el nombre de una variable, función o constante en nuestro script. Los caracteres que PAWN permite para crear los nombres son:

- Caracteres a…z
- Caracteres A…Z
- Caracteres 0…9
- Carácter _ (no puede estar solo)
- El primer carácter de un identificador no puede ser un numero.

PAWN diferencia entre palabras o caracteres que contengan minúsculas y mayúsculas, por ejemplo, las siguientes variables son diferentes:

```
new hola    // Todo en minúscula.

new Hola    // Contiene la primer letra en mayúscula.
```

El compilador de PAWN solo permite un identificador con una extensión máxima de 16 caracteres.

6.4 Errores y mensajes de advertencias

Cuando el compilador Pawn encuentra un error en un archivo, produce un mensaje de salida en el siguiente orden:

- Nombre del archivo.
- El numero de línea donde el compilador detecto el error entre paréntesis, directamente al lado del nombre del archivo.
- La clase de error (“error”, “fatal error” o “warning”).
- Un número con código de error.
- Un mensaje descriptivo con el error.

Por ejemplo:

PlcMain.p(3) : error 001: expected token: ";", but found "{"

Nota 1: el número de línea que contiene el error, puede encontrarse en la siguiente línea a la cual indica el compilador.
Nota 2: un warning es una advertencia, pero no un error.

6.4.1 Categoría de errores

Los errores son separados en tres categorías:

- **Error**: Situación donde es imposible generar el código apropiado. Códigos entre 1 y 99.
- **Fatal error**: Errores del cual el compilador no puede recuperarse. Códigos entre 100 y 199.
- **Warnings**: Advertencias son mostradas con la intención que el usuario preste atención en el código y evite errores. Códigos entre 200 y 299.

Los códigos de errores pueden obtenerse (junto a sus descripción) en el documento **Pawn_Language_Guide.pdf** (ver página 3).

6.5 Precedencia de operadores y Expresiones

La importancia de la precedencia de las expresiones y operadores fue explicada en la pagina 11. Si no se agregan paréntesis “( )“ a una expresión, la expresión sigue las reglas de asociación de la siguiente tabla:

`()`	function call	left-to-right
`[]`	array index (cell)	
`{}`	array index (character)	
`!`	logical not	right-to-left
`~`	one’s complement	
`-`	two’s complement (unary minus)	
`++`	increment	
`--`	decrement	
`:`	tag override	
`char`	convert number of packed characters to cells	
`defined`	symbol definition status	
`sizeof`	symbol size in “elements”	
`state`	automaton/state condition	
`tagof`	unique number for the tag	
`*`	multiplication	left-to-right
`/`	division	
`%`	remainder	
`+`	addition	left-to-right
`-`	subtraction	
`>>`	arithmetic shift right	left-to-right
`>>>`	logical shift right	
`<<`	shift left	
`&`	bitwise and	left-to-right
`^`	bitwise exclusive or	left-to-right
`|`	bitwise or	left-to-right
`<`	smaller than	left-to-right
`<=`	smaller than or equal to	
`>`	greater than	
`>=`	greater than or equal to	
`==`	equality	left-to-right
`!=`	inequality	
`&&`	logical and	left-to-right
`||`	logical or	left-to-right
`? :`	conditional	right-to-left
`=`	assignment	right-to-left
`*= /= %= += -= >>= >>>= <<= &= ^= |=`		
`,`	comma	left-to-right

7 Abreviaciones y Términos Empleados

- **PLC**: Programable Logic Controller (Controlador Lógico Programable).
- **DAQ**: Data Aquisition (Adquisición de Datos).
- **Modo** PLC: Permite programar la STX8081 mediante script PAWN y ejecutarlos autónomamente para realizar algún tipo de control.
- **Modo DAQ**: Permite controlar la STX8081 a través de una computadora conectada a la interfaz Ethernet, ya sea para adquirir datos o controlar las salidas de la placa.
- **UDP**: User Datagram Protocol. Protocolo orientado a la transmisión/recepción de datos. En el PLC se utiliza para intercambiar datos mediante la interfaz Ethernet.
- **IP**: Dirección Internet, conformada por cuatro octetos, por ejemplo 192.168.1.81.
- **Ethernet**: Red de computadoras, que generalmente se utilizan el protocolo de internet TCP/IP o UDP/IP.
- **Firmware**: Software embebido que controla una placa electrónica, y es ejecutado por el procesador.
- **Modo Bootloader**: Modo de funcionamiento del PLC, en el cual se ejecuta un pequeño programa (bootloader) que es el encargado de actualizar el firmware de la placa.
- **Script**: Secuencia de sentencias y funciones que se escriben en un archivo.

8 Historial de Revisiones

Tabla: Historia de Revisiones del Documento

Revisión	Cambios	Descripción	Estado
04 03/Sep/2012	1	1. Se cambia el nombre del documento de “Guía Básica del Lenguaje Pawn” a “Introducción al Lenguaje Pawn”. 2. Se adapta el documento al nuevo entorno **StxLadder**.	Preliminar
03 25/Ene/2012	1	1. Se remplazan todas las llamadas a LedOnOff(). 2. Se agregan consideraciones y notas en “**Eventos**” pag. 36.	Preliminar
02 18/Mar/2011	1	1. Agrega introducción al lenguaje PAWN.	Preliminar
01 19/Oct/2010	1	1. Versión preliminar liberada.	Desarrollo

9 Referencias

1. **Pawn "The Language",** August 2007, ITB CompuPhase (www.compuphase.com). Nota: Este documento puede descargarse de nuestro sitio Web también.
2. **Manual de Usuario de StxLadder**, Slicetex Electronics, http://www.slicetex.com/ladder
3. **Manual de Programacion Pawn del PLC**, placa STX8081, Slicetex Electronics: http://www.slicetex.com/hw/stx8081/docs.html
4. **Hoja de Datos de la placa STX8081** (datasheet), Slicetex Electronics: http://www.slicetex.com/hw/stx8081/docs.html

10 Información Legal

10.1 Aviso de exención de responsabilidad

General: La información de este documento se da en buena fe, y se considera precisa y confiable. Sin embargo, Slicetex Electronics no da ninguna representación ni garantía, expresa o implícita, en cuanto a la exactitud o integridad de dicha información y no tendrá ninguna responsabilidad por las consecuencias del uso de la información proporcionada.

El derecho a realizar cambios: Slicetex Electronics se reserva el derecho de hacer cambios en la información publicada en este documento, incluyendo, especificaciones y descripciones de los productos, en cualquier momento y sin previo aviso. Este documento anula y sustituye toda la información proporcionada con anterioridad a la publicación de este documento.

Idoneidad para el uso: Los productos de Slicetex Electronics no están diseñados, autorizados o garantizados para su uso en aeronaves, área médica, entorno militar, entorno espacial o equipo de apoyo de vida, ni en las aplicaciones donde el fallo o mal funcionamiento de un producto de Slicetex Electronics pueda resultar en lesiones personales, muerte o daños materiales o ambientales graves. Slicetex Electronics no acepta ninguna responsabilidad por la inclusión y / o el uso de productos de Slicetex Electronics en tales equipos o aplicaciones (mencionados con anterioridad) y por lo tanto dicha inclusión y / o uso es exclusiva responsabilidad del cliente.

Aplicaciones: Las aplicaciones que aquí se describen o por cualquiera de estos productos son para fines ilustrativos. Slicetex Electronics no ofrece representación o garantía de que dichas aplicaciones serán adecuadas para el uso especificado, sin haber realizado más pruebas o modificaciones.

Los valores límites o máximos: Estrés por encima de uno o más valores límites (como se define en los valores absolutos máximos de la norma IEC 60134) puede causar daño permanente al dispositivo. Los valores límite son calificaciones de estrés solamente y el funcionamiento del dispositivo en esta o cualquier otra condición por encima de las indicadas en las secciones de Características de este documento, no está previsto ni garantizado. La exposición a los valores limitantes por períodos prolongados puede afectar la fiabilidad del dispositivo.

Documento: Prohibida la modificación de este documento en cualquier medio electrónico o impreso, sin autorización previa de Slicetex Electronics por escrito.

11 Información de Contacto

Para mayor información, visítenos en www.slicetex.com

Para información técnica, envíe un mail a: devel@slicetex.com

Para información general, envíe un mail a: info@slicetex.com

Para ventas, envíe un mail a: ventas@slicetex.com

Slicetex Electronics
Córdoba, Argentina

12 Contenido

www.slicetex.com