ROBOTICS
MAN AND MACHINE

MANUAL DE REFERENCIA VAL3

Versión 6

D28065205B - 01/2008

STÄUBLI

Encontrará adiciones y "erratas" en el documento "readme.pdf" suministrado con el CD-ROM del controlador.

ÍNDICE

CAPÍTULO 1

INTRODUCCIÓN

VAL3 es un lenguaje de programación diseñado para controlar **Stäubli** robots en todo tipo de aplicaciones.

El lenguaje **VAL3** combina las características básicas de un lenguaje informático de alto nivel en tiempo real estándar y de las funcionalidades específicas del control de una célula de robot industrial:

- herramientas de control del robot
- herramientas de modelización geométrica
- herramientas de control de entradas salidas

El presente manual de referencia explica las nociones indispensables para la programación de un robot, y detalla las instrucciones del lenguaje **VAL3**, clasificadas según las siguientes categorías:

- Elementos del lenguaje
- Tipos sencillos
- Interfaz de usuario
- Tareas
- Bibliotecas
- Control del robot
- Posición del brazo
- Control de los movimientos

Cada instrucción, con su sintaxis, figura en el índice general para facilitar su rápida consulta.

CAPÍTULO 2

ELEMENTOS DEL LENGUAJE VAL3

Los elementos constitutivos del lenguaje **VAL3** son:

- las aplicaciones
- los programas
- las bibliotecas
- los tipos de datos
- las constantes
- variables (variables globales, variables locales y parámetros)
- las tareas

2.1. APLICACIONES

2.1.1. DEFINICIÓN

Una aplicación **VAL3** es un paquete de software autónomo diseñado para controlar robots y entradas/salidas asociadas a un controlador.
Una aplicación **VAL3** está constituida por los siguientes elementos:

- un conjunto de **programas**: las instrucciones **VAL3** que deben ejecutarse
- un conjunto de **Variables globales**: las variables utilizadas por todos los programas
- un conjunto de **bibliotecas**: las instrucciones y variables externas utilizadas por la aplicación

Cuando una aplicación está siendo ejecutada, contiene asimismo:

- un conjunto de **tareas**: los programas ejecutados en paralelo

2.1.2. CONTENIDO PREDETERMINADO

Una aplicación **VAL3** siempre contiene los programas **start()** y **stop()**, un sistema de referencia (tipo **frame**) **world** y una herramienta (tipo **tool**) **flange**.
En el momento de crearse, una aplicación **VAL3** contiene además las instrucciones y datos del modelo elegido.
Estos elementos se describen con mayor precisión en el capítulo correspondiente a su tipo.

2.1.3. PUESTA EN MARCHA Y PARADA

Las instrucciones **VAL3** no permiten manipular aplicaciones: La carga, descarga, puesta en marcha y parada de las aplicaciones se hace únicamente a través de la interfaz de usuario del **MCP**.
Cuando una aplicación **VAL3** está en funcionamiento, se ejecuta su programa **start()**.
Una aplicación **VAL3** se para por sí misma cuando se termina la última de sus tareas: entonces, se ejecuta el programa **stop()**. Todas las tareas creadas por bibliotecas, si quedan, son destruidas en el orden inverso de su creación.
Si la parada de una aplicación **VAL3** se provoca desde la interfaz de usuario del **MCP**, la tarea de arranque, si ésta existe todavía, se destruye inmediatamente. Se ejecuta después del programa **stop()**, y, a continuación, todas las tareas de la aplicación, si quedan por hacer, se destruyen en el orden inverso del orden en que fueron creadas.

2.1.4. PARÁMETROS DE APLICACIÓN

Una aplicación **VAL3** puede ser configurada por los siguientes parámetros:

- la unidad de longitud
- cantidad de memoria de ejecución

Estos parámetros no son accesibles con ayuda de una instrucción **VAL3** y sólo pueden modificarse en la interfaz del usuario **MCP** o con la ayuda del editor **SRS**.

2.1.4.1. UNIDAD DE LONGITUD

En aplicaciones **VAL3**, la unidad de longitud es el milímetro o la pulgada. Ésta es utilizada por los tipos de datos geométricos del **VAL3**: sistema de referencia, punto, transformación, herramienta, alisado de trayectoria.
La unidad de longitud de una aplicación se define en el momento de su creación por la unidad de longitud en curso del sistema, y no puede ser ya modificada a continuación.

2.1.4.2. CANTIDAD DE MEMORIA DE EJECUCIÓN

Se necesita alguna memoria para el almacenamiento de cada tarea **VAL3**:

- La pila de ejecución (lista de las llamadas del programa ejecutadas durante esta tarea)
- Los parámetros para cada programa de la pila de llamada
- Las variables locales para cada programa de la pila de llamada

Por defecto, cada tarea incluye **5000** bytes para la memoria de ejecución.
Este nivel puede no ser suficiente para las aplicaciones que contengan grandes tablas de variables locales o de algoritmos recursivos: en este caso, debe aumentarse mediante la interfaz del usuario **MCP** o el editor **SRS**, u bien la aplicación debe optimizarse reduciendo el número de programas en la pila de ejecución o utilizando variables globales en lugar de las variables locales.

2.2. PROGRAMAS

2.2.1. DEFINICIÓN

Un programa es una secuencia de instrucciones **VAL3** por ejecutar.
Un programa está constituido por los siguientes elementos:

- La secuencia de **instrucciones**: las instrucciones **VAL3** que deben ejecutarse
- Un conjunto de **variables locales**: los datos internos del programa
- Un conjunto de **parámetros**: los datos facilitados al programa cuando se le llama

Los programas permiten agrupar secuencias de instrucciones suceptibles de ser utilizadas en varios lugares en una aplicación. Además de economizar tiempo de programación, también simplifican la estructura de las aplicaciones, facilitan la programación y el mantenimiento, y mejoran la legibilidad.
El número de instrucciones de un programa está únicamente limitado por el espacio disponible en la memoria del sistema.
El número de variables locales y de parámetros sólo está limitado por el tamaño de la memoria de ejecución para la aplicación.

2.2.2. REENTRADA

Los programas son reentrantes, lo que quiere decir que un programa puede iniciarse él mismo de manera recursiva (instrucción **call**), o ser llamado simultáneamente por varias tareas. Cada instancia de un programa tiene sus propias variables locales y parámetros.

2.2.3. PROGRAMA START()

El programa **start()** es el programa llamado cuando se ejecuta la aplicación **VAL3**. No puede tener parámetros.
Típicamente, este programa incluye todas las operaciones requeridas para ejecutar la aplicación: inicialización de las variables globales y de las salidas, creación de las tareas de aplicación, etc.
La aplicación no se termina al final del programa **start()**, si continúan ejecutándose otras tareas de aplicación.
Es posible iniciar el programa **start()** al interior de un programa (instrucción **call**) como cualquier otro programa.

2.2.4. PROGRAMA STOP()

El programa **stop()** es el programa llamado durante la parada de la aplicación **VAL3**. No puede tener parámetros.
En este programa se encontrarán especialmente todas las operaciones necesarias para terminar propiamente la aplicación: reactualizar las salidas y detener las tareas de la aplicación, según la secuencia apropiada, etc.
El programa **stop()** puede llamarse desde un programa (instrucción **call**), de la misma manera que cualquier otro programa, pero, el inicio del programa **stop()** no activa la parada de la aplicación.

2.3. TIPOS DE DATOS

2.3.1. DEFINICIÓN

El tipo de una constante o de una variable **VAL3** es una característica que permite que el sistema controle las instrucciones y programas a su disposición.
Todas las constantes y variables **VAL3** tienen un tipo. Esto permite un control inicial por el sistema en el momento de editar un programa y, por lo tanto, la detección inmediata de ciertos errores de programación.

2.3.2. TIPOS SENCILLOS

El lenguaje **VAL3** soporta los siguientes tipos sencillos:

- el tipo **bool**: para los valores booleanos (verdadero / falso)
- el tipo **num**: para los valores numéricos
- el tipo **string**: para las cadenas de caracteres
- el tipo **dio**: para entradas/salidas digitales
- el tipo **aio**: para las entradas-salidas numéricas (analógicas o digitales)
- el tipo **sio**: para las entradas-salidas en conexión serie y socket Ethernet

2.3.3. TIPOS ESTRUCTURADOS

Un tipo estructurado es un tipo que reúne varios datos caracterizados, los campos del tipo estructurado. Los campos de tipo estructurado son accesibles individualmente por su nombre.
El lenguaje **VAL3** soporta los siguientes tipos estructurados:

- el tipo **trsf**: para las transformaciones geométricas cartesianas
- el tipo **frame**: para los planos geométricos cartesianos
- el tipo **tool**: para las herramientas ajustadas en un robot
- el tipo **point**: para las posiciones cartesianas de una herramienta
- el tipo **joint**: para las posiciones articulares del robot
- el tipo **config**: para las configuraciones del robot
- el tipo **mdesc**: para los parámetros de desplazamiento del robot

2.4. INICIALIZACIÓN DE VARIABLES

2.4.1. VARIABLES DE TIPO SIMPLE

La sintaxis precisa para la inicialización de una variable de tipo simple se especifica en el capítulo de descripción de cada tipo simple. Todas las variables deben declararse en la sección de los datos globales o la sección de los datos locales para poder utilizarse.

Ejemplo

En este ejemplo, bBool es una variable booleana, nPi una variable numérica y sCadena un variable de cadena.

```
bBool = true
nPi = 3.141592653
sString = "esto es una cadena constante"
```

2.4.2. DATOS DE TIPO ESTRUCTURADO

El valor de un dato de tipo estructurado está definido por la secuencia de valores en estos campos. El orden de la secuencia está especificado en el capítulo que detalla cada tipo estructurado. Los símbolos '{ }' pueden utilizarse para indicar una estructura.

Ejemplo

En este ejemplo, p es una variable de punto y dummy un programa que utiliza un dato trsf y un dio como parámetros.

```
p = {{100, -50, 200, 0, 0, 0}, {sfree, efree, wfree}}
call dummy({a+b, 2* c, 120, limit(c, 0, 90), 0, 0}, io:válvula1)
```

2.4.3. CONJUNTO DE CONSTANTES

Un conjunto debe inicializarse, entrada por entrada.

Ejemplo

En este ejemplo, j es un cuadro de articulaciones de 5 elementos.

```
// Para los brazos de 6 ejes
j[0] = {0, 0, 0, 0, 0, 0}
j[1] = {90, 0, 90, 0, 0, 0}
j[2] = {-90, 0, 90, 0, 0, 0}
j[3] = {90, 0, 0, -90, 0, 0}
j[4] = {-90, 0, 0, -90, 0, 0}
```

2.5. VARIABLES

2.5.1. DEFINICIÓN

Una variable es un dato referenciado en un programa por su nombre.
Una variable se caracteriza por:

- su nombre: una cadena de caracteres
- su tipo: uno de los tipos **VAL3** descritos anteriormente
- su tamaño: para un conjunto, el número de elementos que contiene
- su alcance: el o los programas que pueden utilizar la variable

El nombre de una variable es una cadena de **1** a **15** caracteres entre **"a..zA..Z0..9_"**, y que comienza con una letra.
Todas las variables pueden utilizarse como cuadros. Las variables simples tienen un tamaño de **1**. La instrucción **size()** define el tamaño de una variable.

2.5.2. ALCANCE DE UNA VARIABLE

El alcance de la variables puede ser:

- global: todos los programas de la aplicación pueden utilizar la variable, o
- local: puede accederse a la variable únicamente mediante el programa en el cual está declarado

Cuando una variable global y una variable local tienen el mismo nombre, el programa donde está declarada la variable local utilizará la variable local, y no tendrá acceso a la variable global.
Los parámetros de un programa son variables locales, accesibles únicamente en el programa en el que están declarados.

2.5.3. ACCESO AL VALOR DE UNA VARIABLE

Los elementos de un cuadro son accesibles con un índice entre los corchetes **'['** y **']'**. El índice debe estar comprendido entre **0** y (tamaño-**1**), si no se genera un error de ejecución.
Si ningún índice está especificado, se utiliza el índice **0**: **var[0]** es equivalente a **var**.
Los campos de las variables de tipo estructurado son accesibles por un **'.'** seguido del nombre del campo.

Ejemplo

```
num a                  // a es una variable de tipo num y de tamaño 1
num b[10]              // b es una variable de tipo num y de tamaño 10
trsf t
point p

a = 0                  //  Inicialización de una variable sencilla
a[0] = 0               //  Correcto: equivalente a a = 0
b[0] = 0               // Inicialización del primer elemento en un conjunto b
b = 0                  //  Correcto: equivalente a b[0] = 0
b[5] = 5               // Inicialización del sexto elemento en un conjunto b
b[5.13] = 7            //  Correcto: equivalente a b[5] = 7 (sólo se utiliza la parte entera)

b[-1] = 0              //  error: índice inferior a 0
b[10] = 0              //  error: índice demasiado grande (un conjunto de los tamaños 10 tiene índices en
                       0..9)

t = p.trsf             //  Inicialización de t
p.trsf.x = 100         // Inicialización del campo x del campo trsf de la variable p
```

2.5.4. PARÁMETRO PASADO "POR VALOR"

Cuando se pasa un parámetro "por valor", el sistema crea una variable local y la inicializa con el valor de la variable o de la expresión suministrada por el programa solicitante.
Las variables del programa solicitante utilizadas como parámetros "por valor" no cambian, aun si el programa llamado modifica el valor del parámetro.
Un cuadro de datos no puede ser pasado por valor.

Ejemplo:

```
program dummy(num x)       // x ha pasado por valor
begin
  x=0
  putln(x)                 //  muestra 0
end

num a
a=10
putln(a)                   //  muestra 10
call dummy(a)              //  muestra 0
putln(a)                   //  muestra 10: a no es modificado por dummy()
```

2.5.5. PARÁMETRO PASADO "POR REFERENCIA"

Cuando se pasa un parámetro "por referencia", el programa deja de trabajar sobre una copia del dato pasada por el solicitante, haciéndolo sobre el propio dato, al que sencillamente se cambia de nombre localmente.
Las variables del programa solicitante utilizadas como parámetros "por referencia" cambian de valor en cuanto el programa llamado modifica el valor del parámetro.
Puede utilizarse o modificarse todos los componentes de un conjunto pasados por referencia. Si se pasa un elemento de un cuadro por referencia, este elemento y todos los elementos siguientes pueden ser utilizados y modificados. En este caso, el parámetro se considera como un conjunto que se inicia con el componente pasado por la llamada. La instrucción **size()** permite conocer el tamaño efectivo de un parámetro.
Cuando se pasa "por referencia" una constante o una expresión, una asignación del parámetro correspondiente no tendrá efecto alguno: el parámetro conservará el valor de la constante o de la expresión.

Ejemplo:

```
program dummy(num& x)          // x es pasado por referencia
begin
  x=0
  putln(x)                     //  muestra 0
end

program element(num& x)
begin
  x[3] = 0
  putln(size(x))
end

num a
num b[10]
a=10
putln(a)                       //  muestra 10
call dummy(a)                  //  muestra 0
putln(a)                       //  muestra 0: a es modificado por dummy()
b[2] = 2
b[5] = 5
call element(b[2])             //  muestra 8, los elementos 0 y 1 de b no son pasados
putln(b[5])                    //  Muestra 0: b[5] fue vinculado a x[3] y modificado en el programa
                               element()
```

2.6. INSTRUCCIONES DE CONTROL DE SECUENCIA

Comentario *//*

Sintaxis
***//* <Cadena>**

Función
Una línea que comienza con **« *//* »** no se evalúa y la ejecución continúa en la línea siguiente. No puede utilizarse **« *//* »** en la mitad de una línea; estos deben ser los primeros caracteres en la línea.

Ejemplo
```
//  Esto es un ejemplo de comentario
```

Llamada de subprograma **call**

Sintaxis
call programa([parámetro1][,parámetro2])

Función
Ejecuta el programa especificado con los parámetros especificados.

Ejemplo
```
//  Llama los programas pick() y place() para i,j entre 1 y 10
for i = 1 to 10
  for j = 1 to 10
    call pick (i, j)
    call place (i, j)
  endFor
endFor
```

Retorno de subprograma **return**

Sintaxis
return

Función
Abandona inmediatamente el programa en curso. Si este programa ha sido iniciado por un **call**, la ejecución se reanuda en el programa solicitante después del **call**. En caso contrario (si el programa es el programa **start()** o el punto de lanzamiento de una tarea), la tarea en curso se termina. Un retorno se efectúa automáticamente al final de un programa**.**

Instrucción de control **if**

Sintaxis

if <bool bCondición>
<instrucciones>
S6.4 **[elseIf <bool bCondiciónAlterna1>**
<instrucciones>]
../..
S6.4 **[elseIf <bool bCondiciónAlternaN>**
<instrucciones>]
[else
<instrucciones>]
endIf

Función

La secuencia **if...elseIf...else...endIf** evalúa sucesivamente las expresiones booleanas marcadas por las palabras clave **if** ó **elseIf** hasta que una expresión sea verdadera. Las instrucciones según la expresión booleana luego son evaluadas, hasta la siguiente palabra clave **elseIf**, **else** o **endIf**. El programa se reanuda finalmente después de la palabra clave **endIf**.
Si todas las expresiones booleanas señaladas por **if** o **elseIf** son falsas, se evalúan las instrucciones comprendidas entre las palabras clave **else** y **endIf** (si la palabra clave **else** está presente). El programa se reanuda a continuación después de la palabra clave **endIf**.

No hay límite al número de expresiones **elseIf** en una secuencia **if...endIf**.

Es mejor reemplazar la secuencia **if...elseIf...else...endIf** por la secuencia **switch...case...default...endSwitch** cuando se prueban los distintos valores posibles de una misma expresión.

Ejemplo

Este programa convierte un **día** escrito en un **string** (**sDía**) en un **num** (**nDía**).

```
put("Enter a day: ")
get(sDía)
if sDía=="Lunes"
  nDía=1
elseIf sDía=="Martes"
  nDía=2
elseIf sDía=="Miércoles"
  nDía=3
elseIf sDía=="Jueves"
  nDía=4
elseIf sDía=="Viernes"
  nDía=5
else
  // Fin de semana !
  nDía=0
endIf
```

Véase también

Instrucción de control switch

Instrucción de control **while**

Sintaxis

while <bool bCondición>
<instrucciones>
endWhile

Función

Las instrucciones entre **while** y **endWhile** se ejecutan mientras que la expresión booleana **Condición** sea verdadera **(true)**.
Si la expresión booleana **Condición** no es verdadera en el momento de la primera evaluación, las instrucciones entre **while** y **endWhile** no se ejecutan.

Parámetro

bool bCondición	expresión booleana por evaluar

Ejemplo

```
dio diLámpara
// Hace parpadear una señal mientras que el robot no esté parado
diLámpara = false
while (isSettled()==false)
  diLámpara = !diLámpara        // Invierte el valor de diLámpara: true false
  delay(0.5)                    // Espera ½ s
endWhile
diLámpara = false
```

Instrucción de control **do ... until**

Sintaxis

do
<instrucciones>
until <bool bCondición>

Función

Las instrucciones entre **do** y **until** se ejecutan hasta que la expresión booleana **bCondición** sea verdadera **(true)**.
Las instrucciones entre **do** y **until** se ejecutan una vez si la expresión booleana **bCondición** es verdadera al hacerse su primera evaluación.

Parámetro

bool bCondición	expresión booleana por evaluar

Ejemplo

```
num a
// Espera que se pulse la tecla Enter
do
  a = get()                     // Espera que se pulse una tecla
until (a == 270)                // Prueba el código de la tecla Enter
```

Instrucción de control **for**

Sintaxis

for <num nContador> = <num nComienzo> to <num nFin> [step <num nPaso>]
<instrucciones>
endFor

Función

Las instrucciones entre **for** y **endFor** se ejecutan hasta que el **nContador** exceda el valor de **nFin** especificado.
El **nContador** se inicializa mediante el valor **nComienzo**. Si **nComienzo** excede **nFin** las instrucciones entre **for** y **endFor** no se ejecutan. Con cada interacción, el **nContador** se incrementa por el valor **nPaso**, y las instrucciones entre **for** y **endFor** se ejecutan de nuevo si el **nContador** no excede **nFin**.
Si **nPaso** es positivo, el **nContador** excede **nFin** si es estrictamente superior a **nFin**. Si **nPaso** es negativo, el **nContador** excede **nFin** si es estrictamente inferior a **nFin**.

Parámetros

num nContador	variable de tipo **num** utilizada como contador
num nComienzo	expresión numérica de inicialización del contador
num nFin	expresión numérica de prueba de fin de bucle
[num nPaso]	expresión numérica de incremento del contador

Ejemplo

```
num i
joint jDest
jDest = {0,0,0,0,0,0}
//  Hace girar el eje 1, 90° grados en -90° grados
for i = 90 to -90 step -10
  jDest.j1 = i
  movej(jDest, flange, mNomSpeed)
  waitEndMove()
endFor
```

Instrucción de control **switch**

Sintaxis

switch <expresión>
case <valor1> [, <valor2>]
<instrucciones1-2>
break
[case <valor3> [, <valor4>]
<instrucciones3-4>
break]
[default
<instrucciones predeterminadas>
break]
endSwitch

Función

La secuencia **switch...case...default...endSwitch** evalúa sucesivamente las expresiones indicadas por la palabra clave **case** hasta que una expresión sea igual a la expresión inicial después de la palabra clave **switch**. Las instrucciones según la expresión luego se evalúan, hasta la palabra clave **break**. El programa se reanuda finalmente después de la palabra clave **endSwitch**.
Si ninguna expresión **case** es igual a la expresión **switch** inicial, se evalúan las instrucciones comprendidas entre las palabras clave **default** y **endSwitch** (si la palabra clave **default** está presente).

No hay límite al número de expresiones **case** en una secuencia **switch...endSwitch**. Las expresiones según la palabra clave **case** deben ser del mismo tipo que las que siguen a la palabra clave **switch**.

La secuencia **switch...case...default...endSwitch** es muy similar a la secuencia **if...elself...else...endIf**.
S6.4 Acepta no solamente las expresiones booleanas, sino cualquier tipo de expresión que acepte al operador estándar "is equal to""==".

Ejemplo

Este programa lee un **num** (**nmenú**) correspondiente a un **key strike** y modifica un **string s** en consecuencia.

```
nmenú = get()
switch nmenú
  case 271
    s = "menú 1"
  break
  case 272
    s = "menú 2"
  break
  case 273, 274, 275, 276, 277, 278
    s = "Menú 3 en 8"
  break
  default
    s = "esta tecla no está en el menú"
  break
endSwitch
```

Este programa convierte un **día** escrito en un **string** (**sDía**) en un **num** (**nDía**).

```
put("Enter a day: ")
get(sDía)
switch sDía
  case "Lunes"
  nDía=1
  break
  case "Martes"
  nDía=2
  break
  case "Miércoles"
```

```
  nDía=3
  break
 case "Jueves"
  nDía=4
  break
  case "Viernes"
  nDía=5
  break
  default
  // Fin de semana !
  nDía=0
  break
endIf
```

CAPÍTULO 3

TIPOS SENCILLOS

3.1. INSTRUCCIONES

num **tamaño**(variable)

Sintaxis
num tamaño(<variable>)

Función
Devuelve el tamaño de **variable**.
Si **variable** es un parámetro de programa pasado por referencia, el tamaño depende del índice especificado al hacer el llamado de programa. Si el **variable** es un ² individual y no un conjunto, el tamaño será 1.

Parámetro

variable	variable de cualquier tipo

Ejemplo

```
num nMatriz[10]
program printSize(num& nparámetro)
begin
  putln(size(nparámetro))
end
call printSize(nMatriz)              // Muestra 10
call printSize(nMatriz[6])           // Muestra 4
```

S6.4

num **getData**(string sNombreDatos, & variable)

Sintaxis

num getData(<string sNombreDatos>, <& variable>)

Función

Esta instrucción copia el valor de los datos, especificado por su nombre **sNombreDatos**, en la variable especificada. Si los datos y la variable son cuadros, la instrucción **getData** copia todas las entradas del cuadro hasta el final de uno de los cuadros. La instrucción devuelve el número de entradas copiadas en la variable.

El nombre de los datos debe tener el siguiente formato: "library:name[index]", donde "library:" y "[index]" son facultativos:

- "name" es el nombre del dato
- "library" es el nombre del identificante de librería donde se define el dato
- "index" es el valor numérico del índice al cual acceder cuando el dato está en un cuadro

La instrucción devuelve un código de error cuando los datos no se han podido copiar:

Valor devuelto	Descripción
n > 0	La variable se ha actualizado con n entradas copiadas
-1	El dato no existe
-2	El identificante de librería no existe
-3	El índice está fuera de los límites
-4	El tipo de dato no corresponde al tipo de variable

Ejemplo

Este programa fusiona 2 cuadros de puntos pAcercamiento y pTrayectoria a partir de una librería especificada por una cadena sParte para dar un cuadro local un único pCamino.

```
i = getData(sParte+":pAcercamiento", pCamino)
if(i > 0)
  nPuntos = i
  i = getData(sParte+":pTrayectoria", pCamino[nPuntos])
  if(i >0)
    nPuntos=nPuntos+i
  endIf
endIf
if (i<0)
  putln("Missing data in part  "+sParte)
endIf
```

3.2. TIPO BOOL

3.2.1. DEFINICIÓN

Los valores posibles de las variables o constantes de tipo bool son:

- **true**: valor verdadero
- **false**: valor falso

De manera predeterminada, una variable de tipo **bool** se inicializa en el valor **false**.

3.2.2. OPERADORES

Por orden de prioridad creciente:

bool <bool& bVariable> = <bool bCondición>	Asigna el valor de **bCondición** a la variable **bVariable** y reenvía el valor de **bCondición**
bool <bool bCondición1> or <bool bCondición2>	Reenvía el valor de la 0 lógica entre **bCondición1** y **bCondición2**. **bCondición2** es evaluado únicamente si **bCondición1** es **false**.
bool <bool bCondición1> and <bool bCondición2>	Reenvía el valor de la Y lógica entre **bCondición1** y **bCondición2**. **bCondición2** es evaluado únicamente si **bCondición1** es **true**.
bool <bool bCondición1> xor bool <bCondición2>	Reenvía el valor de la O exclusiva entre **bCondición1** y **bCondición2**
bool <bool bCondición1> != <bool bCondición2>	Prueba la desigualdad de los valores de **bCondición1** y **bCondición2**. Reenvía **true** si los valores son diferentes, **false** en el caso contrario.
bool <bool bCondición1> == <bool bCondición2>	Prueba la desigualdad de los valores de **bCondición1** y **bCondición2**. Reenvía **true** si los valores son idénticos, **false** en el caso contrario.
bool ! <bool bCondición>	Reenvía la negación del valor de **bCondición**

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones VAL3 utilizadas como parámetro de instrucción.

3.3. TIPO NUM

3.3.1. DEFINICIÓN

El tipo **num** representa un valor numérico, con aproximadamente **14** dígitos significativos.
Cada cálculo numérico se hace por tanto con una precisión limitada por esas **14** cifras significativas.
Hay que tenerlo en cuenta cuando se quiere probar la igualdad de dos valores numéricos: la mayoría de las veces, es necesario probar en un intervalo.

Ejemplo

```
putln(sel(cos(90)==0,1,-1))                          // Muestra -1
putln(sel(abs(cos(90))<0.000000000000001,1,-1))      // Muestra 1
```

Las constantes de tipo numérico tienen el siguiente formato:

```
[-] <cifras>[.<cifras>]
```

Ejemplo

```
1
0.2
-3.141592653
```

Las variables de tipo **num** se reinicializan de manera predeterminada en el valor **0**.

3.3.2. OPERADORES

Por orden de prioridad creciente:

num <num& nVariable> = <num nValor>	Asigna **nValor** a la variable **nVariable** y reenvía **nValor**.
bool <num nValor1> != <num nValor2>	Reenvía **true** si **nValor1** no es igual a **nValor2**, **false** en el caso contrario.
bool <num nValor1> == <num nValor2>	Reenvía **true** si **nValor1** es igual a **nValor2**, **false** en el caso contrario.
bool <num nValor1> >= <num nValor2>	Reenvía **true** si **nValor1** es superior o igual a **nValor2**, **false** en el caso contrario.
bool <num nValor1> > <num nValor2>	Reenvía **true** si **nValor1** es estrictamente superior a **nValor2**, **false** en el caso contrario.
bool <num nValor1> <= <num nValor2>	Reenvía **true** si **nValor1** es inferior o igual a **nValor2**, **false** en el caso contrario.
bool <num nValor1> < <num nValor2>	Reenvía **true** si **nValor1** es estrictamente inferior a **nValor2**, **false** en el caso contrario.
num <num nValor1> - <num nValor2>	Reenvía la diferencia entre **nValor1** y **nValor2**.
num <num nValor1> + <num nValor2>	Reenvía la suma de **nValor1** y **nValor2**.
num <num nValor1> % <num nValor2>	Reenvía el resto de la división entera de **nValor1** por **nValor2**. Se genera un error de ejecución si **nValor2** y **0**. El signo del resto es el signo de **nValor1**.
num <num nValor1> / <num nValor2>	Reenvía el cociente de **nValor1** por **nValor2**. Se genera un error de ejecución si **nValor2** y **0**.
num <num nValor1> * <num nValor2>	Reenvía el producto de **nValor1** y **nValor2**.
num - <num nValor>	Reenvía el opuesto de **nValor**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones VAL3 utilizadas como parámetro de instrucción.

3.3.3. INSTRUCCIONES

num **sin**(num nAngulo)

Sintaxis

num sin(<num nAngulo>)

Función

Retorna el seno de **nAngulo**.

Parámetro

num nAngulo	ángulo en grados

Ejemplo

```
putln(sin(30))                    // muestra 0.5
```

num **asin**(num nValor)

Sintaxis

num asin(<num nValor>)

Función

Retorna el seno inverso de **nValor**, en grados. El resultado está comprendido entre **-90** y **+90** grados.

Se genera un error de ejecución si **nValor** es superior a **1** o inferior a **-1**.

Parámetro

num nValor	expresión numérica

Ejemplo

```
putln(asin(0.5))          // muestra 30
```

num **cos**(num nAngulo)

Sintaxis

num cos(<num nAngulo>)

Función

Retorna el coseno de **Angulo**.

Parámetro

num nAngulo	ángulo en grados

Ejemplo

```
putln(cos(60))            // muestra 0.5
```

num **acos**(num nValor)

Sintaxis

num acos(<num nValor>)

Función

Retorna el coseno inverso de **nValor**, en grados. El resultado está comprendido entre **0** y **180** grados.

Se genera un error de ejecución si **nValor** es superior a **1** o inferior a **-1**.

Parámetro

num nValor	expresión numérica

Ejemplo

```
putln(acos(0.5))          // muestra 60
```

num **tan**(num nAngulo)

Sintaxis

num tan(<num nAngulo>)

Función

Retorna la tangente de **Angulo**.

Parámetro

num nAngulo	ángulo en grados

Ejemplo

```
putln(tan(45))          // muestra 1.0
```

num **atan**(num nValor)

Sintaxis

num atan(<num nValor>)

Función

Retorna la tangente inversa de **nValor**, en grados. El resultado está comprendido entre **-90** y **+90** grados.

Parámetro

num nValor	expresión numérica

Ejemplo

```
putln(atan(1))          // muestra 45
```

num **abs**(num nValor)

Sintaxis

num abs(<num nValor>)

Función

Retorna el valor absoluto de **nValor**.

Parámetro

num nValor	expresión numérica

Ejemplo

```
putln(sel(abs(45)==abs(-45),1,-1))   // muestra 1
```

num **sqrt**(num nValor)

Sintaxis

num sqrt(<num nValor>)

Función

Retorna la raíz cuadrada de **nValor**.

Se genera un error de ejecución si **nValor** es negativo.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(sqrt(9))          // muestra 3
```

num **exp**(num nValor)

Sintaxis

num exp(<num nValor>)

Función

Regresa la exponencial de **nValor**.

Se genera un error de ejecución si **nValor** es demasiado grande.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(exp(1))           // muestra 2.718282
```

num **power**(num nX, num nY)

Sintaxis

num power(<num nX>, <num nY>)

Función

Devuelve nX a la potencia nY: nX^{nY}

Se genera un error de ejecución si nX es negativo o nulo o si el resultado es demasiado grande.

Ejemplo

Este programa calcula de 2 maneras distintas 5 a la potencia 7.

```
// Primera manera:  instrucción power
nResult = power(5,7)

// Segunda manera: power(x,y)=exp(y*ln(x)) (con imprecisión numérica)
nResult = exp(7*ln(5))
```

num **ln**(num nValor)

Sintaxis

num ln(<num nValor>)

Función

Retorna el logaritmo neperiano de **nValor**.

Se genera un error de ejecución si **nValor** es negativo o nulo.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(ln(2.718281828))          // muestra 1
```

num **log**(num nValor)

Sintaxis

num log(<num nValor>)

Función

Retorna el logaritmo decimal de **nValor**.

Se genera un error de ejecución si **nValor** es negativo o nulo.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(log(10))          // muestra 1
```

num **roundUp**(num nValor)

Sintaxis

num roundUp(<num nValor>)

Función

Retorna **nValor** redondeado al entero inmediatamente superior.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(roundUp(7.8))          // Muestra el valor 8
putln(roundUp(-7.8))         // Muestra el valor -7
```

num **roundDown**(num nValor)

Sintaxis

num roundDown(<num nValor>)

Función

Retorna **nValor** redondeado al entero inmediatamente inferior.

Parámetro

num nValor	Expresión numérica

Ejemplo

```
putln(roundDown(7.8))        // Muestra el valor 7
putln(roundDown(-7.8))       // Muestra el valor -8
```

num **round**(num nValor)

Sintaxis

num round(<num nValor>)

Función

Retorna **nValor** redondeado al entero más cercano.

Parámetros

num nValor	Expresión numérica

Ejemplo

```
putln(round(7.8))            // Muestra el valor 8
putln(round(-7.8))           // Muestra el valor -8
```

num **min**(num nX, num nY)

Sintaxis

num min(<num nX>, <num nY>)

Función

Retorna el mínimo de **nX** e **nY**.

Parámetros

num nX	Expresión numérica
num nY	Expresión numérica

Ejemplo

```
putln(min(-1,10))          // Muestra el valor -1
```

num **max**(num nX, num nY)

Sintaxis

num max(<num nX>, <num nY>)

Función

Retorna el máximo de **nX** e **nY**.

Parámetros

num nX	Expresión numérica
num nY	Expresión numérica

Ejemplo

```
putln(max(-1,10))          // Muestra el valor 10
```

num **limit**(num nValor, num nMín, num nMáx)

Sintaxis

num limit(<num nValor>, <num nMín>, <num nMáx>)

Función

Reenvía **nValor** limitado por **nMín** y **nMáx**.

Parámetros

num nValor	Expresión numérica
num nMín	Expresión numérica
num nMáx	Expresión numérica

Ejemplo

```
putln(limit(30,-90,90))    // Muestra 30
putln(limit(100,90,-90))   // Muestra 90
putln(limit(-100,-90,90))  // Muestra -90
```

num **sel**(bool bCondición, num nValor1, num nValor2)

Sintaxis

num sel(<bool bCondición>, <num nValor1>, <num nValor2>)

Función

Reenvía **nValor1** si **Condición** es **true**, en caso contrario **nValor2**.

Parámetros

bool bCondición	Expresión booleana
num nValor1	Expresión numérica
num nValor2	Expresión numérica

Ejemplo

```
putln(sel(bFlag,a,b))
//  es equivalente a
if bFlag==true
  putln(a)
else
  putln(b)
endIf
```

S6.4 ## 3.4. TIPO DE CAMPO DE BITS

3.4.1. DEFINICIÓN

Un campo de bits es un medio de almacenar e intercambiar bajo forma compacta una serie de bits (valores booleanos o entradas/salidas numéricas). **VAL3** no da un tipo de datos específico para administrar los campos de bits pero reutiliza el tipo num para almacenar un campo de bits de 32 bits en forma de un número entero positivo en el margen [0, 2^32].

Todo valor numérico de **VAL3** puede considerarse como un campo de bits de 32 bits ; las instrucciones de tratamiento del campo de bits redondean automáticamente un valor numérico a un número entero positivo de 32 bits, que es entonces tratado como un campo de bits de 32 bits.

3.4.2. OPERADORES

Los operadores estándar del tipo num se aplican en un campo de bits: '=', '==', '!='.

3.4.3. INSTRUCCIONES

num **bAnd**(num nCampoBit1, num nCampoBit2)

Sintaxis

num bAnd(<num nCampoBit1>, <num nCampoBit2>)

Función

Esta instrucción reenvía la operación lógica binaria "y" sobre los dos campos de bits de 32 bits. (El i ésimo bit del resultado se inicializa en 1 si los i ésimos bits de las dos entradas se inicializan en 1). Se obtiene así un número entero positivo en el margen [0, 2^32].

Las entradas numéricas primero se redondean a un número entero positivo en el margen [0, 2^32] antes de que la operación binaria se aplique.

Ejemplo

Este programa muestra un campo de bits nCampoBit de 32 bits en la pantalla probando sucesivamente cada bit:

```
for i=31 to 0 step -1
  // Calcular la máscara para el i ésimo bit
  nMáscara=power(2,i)
  if (nCampoBit and nMáscara)==nMáscara
    put("1")
  else
    put("0")
  endIf
endFor
putln("")
```

num **bOr**(num nCampoBit1, num nCampoBit2)

Sintaxis

num bOr(<num nCampoBit1>, <num nCampoBit2>)

Función

Esta instrucción reenvía la operación lógica binaria "o" sobre los dos campos de bits de 32 bits. (El i ésimo bit del resultado se inicializa en 1 si el i ésimo bit de al menos una entrada se inicializa en 1). Se obtiene así un número entero positivo en el margen [0, 2^32].

Las entradas numéricas primero se redondean a un número entero positivo en el margen [0, 2^32] antes de que la operación binaria se aplique.

Ejemplo

Este programa calcula de dos maneras diferentes una máscara de campo de bits en la cual se activa los i ésimos a j ésimos bits.

```
// Primera manera: lógica 'or' sobre los bits i a j
nCampoBit=0
for k=i to j
  nCampoBit=bOr(nCampoBit, power(2,k))
endFor

// Segunda manera: calcular una máscara de bits de (j-i) bits
nCampoBit=(power(2,j-i+1)-1)
// Desplazar luego la máscara de bits de i bits
nCampoBit=nCampoBit*power(2,i)
```

num **bXor**(num nCampoBit1, num nCampoBit2)

Sintaxis

num bXor(<num nCampoBit1>, <num nCampoBit2>)

Función

Esta instrucción reenvía la operación lógica binaria "xor" (o exclusivo) a los dos campos de bits de 32 bits. (El i ésimo bit del resultado se inicializa en 1 si los i ésimos bits de las dos entradas son diferentes). Se obtiene así un número entero positivo en el margen [0, 2^32].

Las entradas numéricas primero se redondean a un número entero positivo en el margen [0, 2^32] antes de que la operación binaria se aplique.

Ejemplo

Este programa invierte los bits i a j del campo de bits nCampoBit:

```
// Calcular la máscara para los bits i a j (ver el ejemplo bOr)
nMáscara=(power(2,j-i+1)-1)*power(2,i)
// Invertir los bits i a j con ayuda de la máscara
nCampoBit=bXor(nCampoBit,nMáscara)
```

num **bNot**(num nCampoBit)

Sintaxis

num bNot(<num nCampoBit>)

Función

Esta instrucción reenvía la operación lógica binaria "no" (negación) a un campo de bits de 32 bits. (El i ésimo bit del resultado se inicializa en 1 si el i ésimo bit de la entrada está en 0). Se obtiene así un número entero positivo en el margen [0, 2^32].

La entrada numérica se redondea primero a un número entero positivo en el margen [0, 2^32] antes de que la operación binaria se aplique.

Ejemplo

Este programa reinicializa los bits i a j de un campo de bits nCampoBit con la ayuda de una máscara nMask.

```
// Calcular una máscara de bits con los bits i a j inicializados en 1 (ver el ejemplo bOr)
nMáscara=(power(2,j-i+1)-1)*power(2,i)
// Invertir la máscara para que todos los bits estén en 1 excepto los bits i a j
nMáscara=bNot(nMáscara)
// Reinicializar los bits i a j con la ayuda de la operación binaria 'and'
nCampoBit=bAnd(nCampoBit, nMáscara)
```

num **toBinary**(num nValor, num nTamañoValor, cadena sFormatoDatos, num& nByteDatos)

num **fromBinary**(num nByteDatos, num nTamañoDatos, cadena sFormatoDatos, num& nValor)

Sintaxis

num toBinary(<num nValor>, <num nTamañoValor>, <cadena sFormatoDatos>, <num& nByteDatos>)

num fromBinary(<num nByteDatos>, <num nTamañoDatos>, <cadena sFormatoDatos>, <num& nValor>)

Función

El propósito de las instrucciones **toBinary**/**fromBinary** es permitir el intercambio de valores numéricos entre dos dispositivos, utilizando una línea serial o una conexión de red. Los valores numéricos primero se codifican en una corriente de bytes. Los bytes se envían entonces al dispositivo paritario. Finalmente el dispositivo paritario descifra los bytes para recuperar los valores numéricos iniciales. Diversas codificaciones binarias de valores numéricos son posibles.

La instrucción **toBinary** codifica valores numéricos en una serie de bytes (campo de bits de 8-bits, número entero positivo en el rango [0, 255]) según lo especificado por el formato de datos **sFormatoDatos**. El número de valores numéricos **nValor** a codificar es proporcionado por el parámetro **nTamañoValor**. El resultado se almacena en la serie **nByteDatos**, y la instrucción devuelve el número de bytes codificados en esta serie.

Se genera un error de ejecución si el número de valores a codificar **nTamañoValor** es mayor que el tamaño de **nValor**, si el formato especificado no es soportado o si la serie **nByteDatos** del resultado no es lo bastante grande para codificar todos los datos de entrada.

La instrucción **fromBinary** decodifica una serie de bytes en valores numéricos **nValor**, según lo especificado por el formato de datos **sFormatoDatos**. El número de bytes a decodificar es proporcionado por el parámetro **nTamañoDatos**. El resultado se almacena en la serie **nValor** y la instrucción devuelve el número de valores en esta serie. Si algunos datos binarios están corrompidos (bytes fuera del rango [0, 255] o codificación inválida de punto flotante), la instrucción devuelve el opuesto del número de valores correctamente decodificados (valor negativo).

Se genera un error de ejecución si el número de bytes a decodificar **nTamañoDatos** es mayor que el tamaño de **nByteDatos**, si el formato especificado no es soportado o si la serie **nValor** del resultado no es lo bastante grande para decodificar todos los datos de entrada.

Las codificaciones binarias soportadas se proporcionan en el cuadro de abajo:

- El signo "-" indica la codificación de un entero con signo (el último bit del campo de bits codifica el signo del valor).
- El dígito proporciona el número de bytes para la codificación de cada valor numérico.
- La extensión ".0" marca los valores del punto flotante codificando (se soportan las codificaciones simples y de precisión doble de IEEE 754).
- La letra final especifica el orden de los bytes: "l" para 'little endian' (el byte menos significativo se codifica primero), "b" para 'big endian' (el byte más significativo se codifica primero). La codificación 'big endian' es el estándar para las aplicaciones de red (TCP/IP).

"-1"	Byte con signo
"1"	Byte sin signo
"-2l"	Palabra con signo, little endian
"-2b"	Palabra con signo, big endian
"2l"	Palabras sin signo, little endian
"2b"	Palabra sin signo, big endian
"-4l"	Palabra con doble signo, little endian
"-4b"	Palabra con doble signo, big endian
"4l"	Doble palabra sin signo, little endian
"4b"	Doble palabra sin signo, big endian
"4.0l"	Valor del punto flotante de precisión simple, little endian
"4.0b"	Valor del punto flotante de precisión simple, big endian
"8.0l"	Valor del punto flotante de precisión doble, little endian
"8.0b"	Valor del punto fotante de precisión doble, big endian

El formato nativo **VAL3** para los datos numéricos es la codificación de precisión doble. Este formato se debe utilizar para intercambiar valores numéricos sin pérdida de exactitud.

Ejemplo

El primer programa codifica un **tShiftOut** de dato **trsf** en una serie **nByteOut** del byte y lo envía con la conexión serie **tcpClient**. El segundo programa lee los bytes de la conexión serie **tcpServer** y los convierte nuevamente en un **trsf tShiftIn**.

```
// ---- Programa para enviar un trsf ----
// Copiar las coordenadas trsf en un buffer numérico
nTrsfOut[0]=tShiftOut.x
nTrsfOut[1]=tShiftOut.y
nTrsfOut[2]=tShiftOut.z
nTrsfOut[3]=tShiftOut.rx
nTrsfOut[4]=tShiftOut.ry
nTrsfOut[5]=tShiftOut.rz
// Codificar los valores numéricos 6 (punto flotante de precisión doble, por lo tanto 8 bytes) en 6*8=48 bytes en la serie nByteOut[48]
toBinary(nTrsfOut, 6, "8.0b", nByteOut)
// Enviar serie nByte (bytes 48) a través de tcpClient
sioSet(io:tcpClient, nByteOut)


// ---- Programa para leer un trsf ----
nb=i=0
while (nb<48)
  nb=sioGet(io:tcpServer, nByteIn[i])
  if(nb>0)
    i=i+nb
  else
    // Error de comunicación
    return
  endIf
endWhile
if (fromBinary(nByteIn, 48, "8.0b", nTrsfIn) != 6)
  // Datos corrompidos
  return
else
  tShiftIn.x=nTrsfIn[0]
  tShiftIn.y=nTrsfIn[1]
  tShiftIn.z=nTrsfIn[2]
  tShiftIn.rx=nTrsfIn[3]
  tShiftIn.ry=nTrsfIn[4]
  tShiftIn.rz=nTrsfIn[5]
endIf
```

3.5. TIPO STRING

3.5.1. DEFINICIÓN

Las variables de tipo cadena de caracteres permiten almacenar textos. El tipo de cadena de caracteres soporta el grupo de caracteres Unicode. Cabe notar que la visualización correcta de un carácter Unicode depende de las fuentes del carácter instaladas en el dispositivo de visualización.
Una cadena de caracteres se memoriza en 128 octetos; la cantidad máxima de caracteres en una cadena depende de los caracteres utilizados, debido a que la codificación interna de caracteres (Unicode UTF8) utiliza de 1 byte (para ASCII caracteres) a 4 bytes (3 para los caracteres chinos).
Por consiguiente, la longitud máxima de una cadena de caracteres ASCII es de 128 caracteres; la longitud máxima de una cadena de caracteres en chino es de 42 caracteres.

3.5.2. OPERADORES

Por orden de prioridad creciente:

string <string& sVariable> = <string sCadena>	Asigna **sCadena** a la variable **sVariable** y reenvía **sCadena**.
bool <string sCadena1> != <string sCadena2>	Reenvía **true** si **sCadena1** y **sCadena2** no son idénticas, **false** en caso contrario.
bool <string sCadena1> == <string sCadena2>	Reenvía **true** si **sCadena1** y **sCadena2** son idénticas, **false** en caso contrario.
string <string sCadena1> + <string <sCadena2>	Retorna los primeros caracteres (limitado a **128** bytes) de **sCadena1** concatenados con **sCadena2**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

3.5.3. INSTRUCCIONES

string **toString**(string sFormato, num nValor)

Sintaxis
string toString(<string sFormato>, <num nValor>)

Función
Reenvía una cadena de caracteres que representa **nValor** según el formato de visualización **sFormato**.
El formato es **"tamaño.precisión"**, en dónde **tamaño** es el tamaño mínimo del resultado (se añaden espacios al comienzo de la cadena si fuera necesario), y **precisión** es la cantidad de cifras significativas después de la coma (los **0** al final de la cadena son sustituidos por espacios). De manera predeterminada, **tamaño** y **precisión** valen **0**. La parte entera del valor no se trunca jamás, aun si su longitud de visualización es más grande que **tamaño**.

Parámetros

string sFormato	Expresión de tipo cadena de caracteres
num nValor	Expresión numérica

Ejemplo

```
num nPi
nPi = 3.141592654
putln(toString(".4", nPi))          // muestra «3.1416»
putln(toString("8", nPi))           // muestra «       3»
putln(toString("8.4", nPi))         // muestra «  3.1416»
putln(toString("8.4", 2.70001))     // muestra «  2.7   »
putln(toString("", nPi))            // muestra «3»
putln(toString("1.2", 1234.1234))   // muestra «1234.12»
```

Véase también
string chr(num nPuntoCódigo)
string toNum(string sCadena, num& nValor, bool& bRelación)

string **toNum**(string sCadena, num& nValor, bool& bRelación)

Sintaxis

string toNum(<string sCadena>, <num& nValor>, bool& bRelación)

Función

Calcula el **nValor** numérico representado al comienzo de la **sCadena** especificada, y retorna **sCadena** en la cual todos los caracteres han sido suprimidos hasta la representación siguiente de un valor numérico.

Si el inicio de **sCadena** no representa un valor numérico, **bRelación** se configura en **false** y **nValor** no se modifica; en caso contrario, **bRelación** se configura en **true**.

Parámetros

string sCadena	Expresión de tipo cadena de caracteres
num& nValor	Variable de tipo **num**
bool& bRelación	Variable de tipo **bool**

Ejemplo

```
num nVal
bool bOk
putln(toNum("10 20 30", nVal, bOk))   // %Muestra «20 30», nVal vale 10, bOk vale true
putln(toNum("a10 20 30", nVal, bOk))  // Muestra «a10 20 30», nVal no cambia, bOk vale false
putln(toNum("10 end", nVal, bOk))     // %Muestra «», nVal vale 10, bOk vale true
buffer = "+90 0.0 -7.6 17.3"
do
  buffer = toNum(buffer, nVal, bOk)
  putln(nVal)                         // Muestra sucesivamente 90, 0, -7.6, 17.3
until (bOk != true)
```

Véase también

string toString(string sFormato, num nValor)

string **chr**(num nPuntoCódigo)

Sintaxis

string chr(<num nPuntoCódigo>)

Función

Retorna la cadena de caracteres constituida por el carácter de punto de código Unicode, si ésta es un punto de código Unicode válido. En caso contrario, retorna una cadena de caracteres vacía.
El siguiente cuadro proporciona los puntos de código Unicode debajo de **128** (corresponde al cuadro de caracteres **ASCII**). Los carácteres en las celdas grises son los códigos de control que se pueden substituir por un signo de interrogación cuando se visualiza la secuencia.
Todos los puntos de código Unicode válidos son soportados por el tipo de secuencia **VAL3**. Sin embargo, la visualización del carácter depende de las fuentes de carácter instaladas en el dispositivo de visualización. La lista completa de carácteres Unicode se puede encontrar en http://www.unicode.org (buscar 'Tablas de códigos').

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NUL	SOH	STX	ETX	EOT	ENQ	ACQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
" "	!	"	#"	$	%	&	'	(	)	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	(	|	)	~	DEL

Parámetros

num nPuntoCódigo	Expresión de tipo **num**

Ejemplo

```
putln(chr(65))                    // Muestra «A»
```

Véase también

num asc(string sTexto, num nPosición)

num **asc**(string sTexto, num nPosición)

Sintaxis

num asc(<string sTexto>, <num nPosición>)

Función

Retorna el punto de código Unicode del carácter del índice **nPosición**.
Retorna -1 si **nPosición** es negativo o superior a la longitud de texto especificada.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nPosición	Expresión numérica

Ejemplo

```
putln(asc("A",0))          // Muestra 65
```

Véase también

string chr(num nPuntoCódigo)

string **left**(string sTexto, num nTamaño)

Sintaxis

string left(<string sTexto>, <num nTamaño>)

Función

Reenvía **nTamaño** de los primeros caracteres de **sTexto**. Si **nTamaño** es más grande que la longitud de la cadena **sTexto**, la instrucción reenvía la cadena **sTexto**.
Se genera un error de ejecución si **nTamaño** es negativo.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nTamaño	Expresión numérica

Ejemplo

```
putln(left("hello world",5))    // Muestra «hello»
```

string **right**(string sTexto, num nTamaño)

Sintaxis

string right(<string sTexto>, <num nTamaño>)

Función

Reenvía **nTamaño** de los últimos caracteres de **sTexto**. Si el número especificado es más grande que la longitud de **sTexto**, la instrucción reenvía **sTexto**.

Se genera un error de ejecución si **nTamaño** es negativo.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nTamaño	Expresión numérica

Ejemplo

```
putln(right("hello world",5))   // Muestra «world»
```

string **mid**(string sTexto, num nTamaño, num nPosición)

Sintaxis

string mid(<string sTexto>, <num nTamaño>, <num nPosición>)

Función

Reenvía **nTamaño** de los caracteres de **sTexto** a partir del carácter de índice **nPosición**, parándose al final de **sTexto**.

Se genera un error de ejecución si **nTamaño** o **nPosición** son negativos.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nTamaño	Expresión numérica
num nPosición	Índice en la cadena (de **0** a **127**)

Ejemplo

```
putln(mid(«hello wild world»,4,6))   // Muestra «wild»
```

string **insert**(string sTexto, string sInserción, num nPosición)

Sintaxis

string insert(<string sTexto>, <string sInserción>, <num nPosición>)

Función

Reenvía **sTexto** en la cual **sInserción** está insertada después del carácter de índice **nPosición**. Si **nPosición** es más grande que el tamaño de **sTexto**, se añade **sInserción** al final de **sTexto**. El resultado es truncado, si excede los 128 bytes.

Se genera un error de ejecución si **nPosición** es negativo.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
string sInserción	Expresión de tipo cadena de caracteres
num nPosición	Índice en la cadena (de 0 a 127)

Ejemplo

```
putln(insert("hello world","wild ",6))   // Muestra «hello wild world»
```

string **delete**(string sTexto, num nTamaño, num nPosición)

Sintaxis

string delete(<string sTexto>, <num nTamaño>, <num nPosición>)

Función

Reenvía **sTexto** en la cual **nTamaño** de caracteres han sido suprimidos a partir del carácter de índice **nPosición**. Si **nPosición** es superior a la longitud de **sTexto**, la instrucción reenvía **sTexto**.

Se genera un error de ejecución si **nTamaño** o **nPosición** son negativos.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nTamaño	Expresión numérica
num nPosición	Índice en la cadena (de **0** a **127**)

Ejemplo

```
string sSource
sSource = "hello wild world"
putln(delete(sSource,5,6))          // muestra «hello world»
putln(sSource)                      // muestra «hello wild world»
```

string **replace**(string sTexto, string sSustitución, num nTamaño, num nPosición)

Sintaxis

string replace(<string sTexto>, <string sSustitución>, <num nTamaño>, <num nPosición>)

Función

Reenvía **sTexto** en la cual **nTamaño** de los caracteres han sido sustituidos a partir del carácter del índice de **nPosición** por **sSustitución**. Si **nPosición** es superior a la longitud de **sTexto**, la instrucción reenvía **sTexto**.

Se genera un error de ejecución si **nTamaño** o **nPosición** son negativos.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
string sSustitución	Expresión de tipo cadena de caracteres
num nTamaño	Expresión numérica
num nPosición	Índice en la cadena (de **0** a **127**)

Ejemplo

```
putln(replace("hello ? world","wild",1,6))   // muestra «hello wild world»
```

num **find**(string sTexto1, string sTexto2)

Sintaxis

num find(<string sTexto1>, <string sTexto2>)

Función

Reenvía el índice (entre **0** y **127**) del primer carácter de la primera ocurrencia de **sTexto2** en **sTexto1**. Si **sTexto2** no figura en **sTexto1**, la instrucción reenvía **-1**.

Parámetros

string sTexto1	Expresión de tipo cadena de caracteres
string sTexto2	Expresión de tipo cadena de caracteres

Ejemplo

```
putln(find("hello wild world","wild"))   // Muestra 6
```

num **len**(string sTexto)

Sintaxis

num len(<string sTexto>)

Función

Retorna el número de caracteres en **sTexto**.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres

Ejemplo

```
putln(len("hello wild world"))        // Muestra 16
```

Véase también

num getDisplayLen(string sTexto)

3.6. TIPO DIO

3.6.1. DEFINICIÓN

El tipo **dio** se utiliza para vincular una variable **VAL3** a una entrada/salida digital de sistema.

Las entradas/salidas declaradas en el sistema pueden utilizarse en una aplicación **VAL3**, a partir de la biblioteca io, sin tener que ser declaradas en la aplicación como variable global o local. Por consiguiente, el tipo **dio** se utiliza para una entrada/salida de sistema o como parámetro, cuando se llama un programa.

Todas las instrucciones que utilizan un tipo variable **dio** no vinculado a una entrada/salida declarada en el sistema, generan un error de ejecución.

Por defecto, un tipo variable **dio** no está vinculada a una entrada/salida del sistema y por lo tanto no genera un error de ejecución si se utiliza como tal en un programa antes de ser vinculado.

3.6.2. OPERADORES

Por orden de prioridad creciente:

bool <dio diSalida> = <dio diEntrada>	Asigna el estado de **diEntrada** a salida **diSalida**, y reenvía este estado. Se genera un error de ejecución si **diSalida** no está vinculado a una salida de sistema.
bool <dio diSalida> = <bool bCondición>	Asigna **bCondición** al estado de **diSalida**, y reenvía **bCondición**. Se genera un error de ejecución si **diSalida** no está vinculado a una salida de sistema.
bool <dio diEntrada1> != <bool bEntrada2>	Reenvía **true** si **diEntrada1** y **bEntrada2** no están en el mismo estado, en caso contrario reenvía **false**.
bool <dio diEntrada> != <bool bCondición>	Reenvía **true** si el estado de **diEntrada** no es igual a **bCondición**, en caso contrario reenvía **false**.
bool <dio diEntrada> == <bool bCondición>	Reenvía **true** si el estado de **diEntrada** es igual a **bCondición**, en caso contrario reenvía **false**.
bool <dio diEntrada1> == <dio diEntrada2>	Reenvía **true** si **diEntrada1** y **diEntrada2** están en el mismo estado, en caso contrario reenvía **false**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

3.6.3. INSTRUCCIONES

void **dioLink**(dio& diVariable, dio diOrigen)

Sintaxis

void dioLink(<dio& diVariable>, <dio diOrigen>)

Función

Conecta **diVariable** a la entrada-salida del sistema a cuyo **diOrigen** está enlazada.

Se genera un error de ejecución si **diVariable** es una entrada/salida de la biblioteca io.

Parámetros

dio& diVariable	Variable de tipo entrada/salida digital
dio diOrigen	Expresión de tipo **dio**

Ejemplo

```
dio dpinza1
dio dpinza2
dioLink(diPinza1,                  // Conecta diPinza1 a la entrada-salida sistema válvula1
io:válvula1)
dioLink(diPinza2, diPinza1)        // Conecta diPinza2 a la entrada-salida de diPinza1, y por tanto, al
                                   válvula1
dioLink(diPinza1,                  // diPinza2 está ahora unida a válvula2, y diPinza1 siempre a
io:válvula2)                       válvula1
```

num **dioGet**(dio diMatriz)

Sintaxis

num dioGet(<dio diMatriz>)

Función

Retorna el valor numérico a partir de la lectura **diMatriz**, como un número entero en código binario, es decir: **diMatriz[0]+2 diMatriz[1]+4 * diMatriz[2]+...+2^k * diMatriz[k]**, en dónde **diMatriz[i] = 1** si **diMatriz[i]** es **true**, **0** en el caso contrario.

Se genera un error de ejecución si no se vincula un miembro de **diMatriz** a una entrada/salida del sistema.

Parámetros

dio diMatriz	Expresión de tipo **dio**

Ejemplo

```
dio diCode[4]
diCode[0] = false
diCode[1] = true
diCode[2] = false
diCode[3] = true
putln(dioGet(diCode))              // Muestra 10 = 0 + 2 * 1 + 4 * 0 + 8 * 1
```

Véase también

num dioSet(dio diMatriz, num nValor)

num **dioSet**(dio diMatriz, num nValor)

Sintaxis

num dioSet(<dio diMatriz>, <num nValor>)

Función

Asigna la parte entera de **nValor** escrita en binario a las salidas todo o nada de **diMatriz**, y reenvía el valor efectivamente asignado, es decir:

diMatriz[0]+2 * **diMatriz[1]+4** * **diMatriz[2]+...+2^k** * **diMatriz[k]**, en dónde **diMatriz[i] = 1** si **diMatriz[i]** es **true**, **0** en el caso contrario.

Se genera un error de ejecución si no se vincula un miembro de **dcuadro** a una salida del sistema.

Parámetros

dio diMatriz	Expresión de tipo **dio**
num nValor	Expresión de tipo **num**

Ejemplo

```
dio dCode[4]
putln(dioSet(diCode, 10)      // Muestra 10 = 0 + 2 * 1 + 4 * 0 + 8 * 1
putln(dioSet(diCode, 26)      // Muestra 10, código no es suficientemente grande para codificar por entero 26
```

Véase también

num dioGet(dio diMatriz)

3.7. TIPO AIO

3.7.1. DEFINICIÓN

El tipo **aio** permite enlazar una variable **VAL3** a una entrada-salida numérica del sistema (digital o analógico).
Las entradas/salidas declaradas en el sistema pueden utilizarse en una aplicación **VAL3**, a partir de la aplicación io, sin tener que declararlas en la biblioteca como variable global o local. Por consiguiente, se utiliza el tipo **aio** como un alias, para una entrada/salida analógica de sistema, o como un parámetro, cuando se llama un programa.
Todas las instrucciones que utilizan un tipo variable **aio** no vinculado a una entrada/salida declarada en el sistema, generan un error de ejecución.
Por defecto, un tipo variable **aio** no está vinculada a una entrada/salida del sistema y por lo tanto no genera un error de ejecución si se utiliza como tal en un programa antes de ser vinculado

3.7.2. INSTRUCCIONES

void **aioLink**(aio& aiVariable, aio aiOrigen)

Sintaxis
void aioLink(<aio& aiVariable>, <aio aiOrigen>)

Función
Conecta **aiVariable** a la entrada-salida del sistema a cuyo **aiOrigen** está enlazada.
Se genera un error de ejecución si **aiVariable** es una entrada/salida de la biblioteca io.

Parámetros

aio& aiVariable	Variable de tipo **aio**
aio aiOrigen	Expresión de tipo **aio**

Ejemplo

```
aio aiSensor1
aio aiSensor2
aioLink(aiSensor1, io:system1)   // Conecta aiSensor1 a la entrada-salida sistema system1
aioLink(aiSensor2, aiSensor1)    // Conecta aiSensor2 a la entrada-salida de aiSensor1, y por
                                 tanto, al system1
aioLink(aiSensor1, io:system2)   // aiSensor2 está ahora unido al system2, y aiSensor1
                                 siempre a system1
```

num **aioGet**(aio aiEntrada)

Sintaxis
num aioGet(<aio aiEntrada>)

Función
Reenvía el valor numérico de **aiEntrada**.
Se genera un error de ejecución si **aiEntrada** no está vinculado a una entrada/salida del sistema.

Parámetros

aio& aiEntrada	Expresión de tipo **aio**

Ejemplo

```
aio aiSensor
putln(aioGet(aiSensor))   // Muestra el valor actual del sensor
```

Véase también
num aioSet(aio aiSalida, num nValor)

num **aioSet**(aio aiSalida, num nValor)

Sintaxis

num aioSet(<aio aiSalida>, <num nValor>)

Función

Asigna **nValor** a **aiSalida**, y reenvía **nValor**. Si el valor configurado se encuentra fuera de la gama de aio, el número retornado será el valor actual de la salida aio.

Se genera un error de ejecución si **aiSalida** no está vinculado a una salida de sistema.

Parámetros

aio& aiSalida	Expresión de tipo **aio**
num nValor	Expresión de tipo **num**

Ejemplo

```
aio aiCommand
putln(aioSet(aiCommand, -12.3))      // Muestra -12.3
```

Véase también

num aioGet(aio aiEntrada)

3.8. TIPO SIO

3.8.1. DEFINICIÓN

El tipo **sio** permite vincular una variable **VAL3** a una entrada-salida serie del sistema o una conexión mediante socket Ethernet. Una entrada-salida **sio** se caracteriza por:

- Parámetros propios al tipo de comunicación, definidos en el sistema
- Un caracter de fin de cadena, para hacer posible la utilización del tipo **string**
- Un plazo de espera de comunicación

Las entradas-salidas serie del sistema están siempre activas. Las conexiones mediante socket Ethernet se activan en el primer acceso en lectura o escritura por medio de un programa **VAL3**. Las conexiones mediante socket Ethernet se desactivan automáticamente cuando se termina la aplicación **VAL3**.

Las entradas/salidas declaradas en el sistema pueden utilizarse en una aplicación **VAL3**, a partir de la biblioteca io, sin tener que declararlas en la aplicación, como variable global o local. Por consiguiente, el tipo **sio** se utiliza como un alias, para una entrada/salida sio de sistema, o como un parámetro, al llamar un programa.

Todas las instrucciones que utilizan un tipo variable **sio** no vinculado a una entrada/salida declarada en el sistema, generan un error de ejecución.

Por defecto, un tipo variable **sio** no está vinculada a una entrada/salida del sistema y por lo tanto no genera un error de ejecución si se utiliza como tal en un programa antes de ser vinculado.

Operadores

Cuando se alcanza el plazo de tiempo de espera de la comunicación en lectura o escritura de la entrada-salida serie, se genera un error de ejecución.

string <sio siSalida> = <string sTexto>	Escribe sucesivamente en **siSalida** los códigos **UTF8** de caracteres **sTexto**, seguidos del fin del carácter de cadena, y retorna **sTexto**.
num <sio siSalida> = <num nDato>	Escribe sobre **siSalida** el entero más próximo de **nDato**, módulo **256**, y reenvía el valor efectivamente enviado.
num <num nDatos> = <sio siEntrada>	Lee un byte en **siEntrada** y asigna **nDatos** con el valor de byte.
string <string sTexto> = <sio siEntrada>	Lee en **siEntrada** una cadena de caracteres UFT8 Unicode y asigna a **sTexto** esta cadena. Los caracteres no soportados por el tipo **string** son ignorados. La cadena se completa cuando se lee el carácter de fin de cadena, o cuando **sTexto** alcanza el tamaño máximo de una **string** (**128** bytes). El carácter de final de cadena no se vuelve a copiar en **sTexto**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

3.8.2. INSTRUCCIONES

void **sioLink**(sio& siVariable, sio siOrigen)

Sintaxis

void sioLink(<sio& siVariable>, <sio siOrigen>)

Función

Enlaza **siVariable** a la entrada-salida serie del sistema sobre el cual está enlazado dicho **siOrigen**.

Se genera un error de ejecución si **siVariable** es una entrada/salida de la biblioteca io.

Parámetros

sio& siVariable	Variable de tipo **sio**
sio siOrigen	Expresión de tipo **sio**

Ejemplo

```
sio siSensor1
sio siSensor2
sioLink(siSensor1, io:portSerial1)   // Conecta siSensor1 a la entrada-salida sistema portSerial1
sioLink(siSensor2, siSensor1)        // Conecta siSensor2 a la entrada-salida de siSensor1, y por tanto, al portSerial1
sioLink(siSensor2, io:portSerial1)   // Conecta siSensor2 a portSerial1, siSensor1 está siempre unido a portSerial1
```

num **clearBuffer**(sio siEntrada)

Sintaxis

num clearBuffer(<sio siEntrada>)

Función

Vacía el buffer de lectura de **siEntrada** y retorna la cantidad de caracteres así borrados.

Para una conexión socket Ethernet, **clearBuffer** desactiva (cierra) el socket, **clearBuffer** retorna **-1** si el socket estaba ya desactivado.

Se genera un error de ejecución si **siEntrada** no está conectado a un enlace serie del sistema o a una base Ethernet.

num **sioGet**(sio siEntrada, num& nDatos)

Sintaxis

num sioGet(<sio siEntrada>,<num& nDatos>)

Función

Lee una carácter individual sobre un conjunto de caracteres, a partir de **siEntrada**, y retorna el número de caracteres leídos. La secuencia de lectura se detiene cuando el **nDatos** está lleno o cuando la memoria intermedia de lectura está llena.
Para una conexión socket Ethernet, **sioGet** trata primero de establecer la conexión si ésta no está ya activa. Cuando se alcanza el plazo de espera de comunicación de entrada, **sioGet** retorna **-1**. Si la conexión está activa pero no hay datos en el buffer de lectura de entrada, **sioGet** espera hasta que los datos sean recibidos o que el plazo de espera se cumpla.
Se genera un error de ejecución si **siEntrada** no está enlazado a un puerto serie del sistema o a una base Ethernet, o si **nDatos** no es una variable **VAL3**.

num **sioSet**(sio siSalida, num& nDatos)

Sintaxis

num sioSet(<sio siSalida>,<num& nDatos>)

Función

Escribe un carácter o un conjunto de caracteres en **siSalida** y retorna el número de caracteres escritos. Los valores numéricos son convertidos antes de transmisión en entero entre **0** y**255**, tomando el entero más cercano a módulo **256**.
Para una conexión socket Ethernet, **sioSet** trata primero de establecer la conexión si ésta no está ya activa. Cuando se alcanza el plazo de espera de comunicación de salida **sioSet** retorna **-1**. La cantidad de caracteres escritos puede ser inferior al tamaño de **nDatos** si se detecta un error de comunicación.
Se genera un error de ejecución si **siSalida** no está enlazado a un puerto serie del sistema o a una base Ethernet.

S6.4 num **sioCtrl**(sio siCanal, string nParámetro, valor)

Sintaxis

num sioCtrl(<sio siCanal>, <string nParámetro>, <valor>)

Función

Esta instrucción modifica un parámetro de comunicación de la siCanal de entrada/salida serie especificada.

(!) Para las líneas serie, algunos parámetros o valores de parámetro pueden no ser soportados por el hardware: remitirse al manual del controlador.

La instrucción devuelve:

0	El parámetro se ha modificado con éxito
-1	El parámetro no se ha definido
-2	El valor de parámetro no tiene el tipo esperado
-3	El valor de parámetro no es soportado
-4	El canal serie no está listo para aplicar el cambio del parámetro (primero pararlo)
-5	El parámetro no está definido para este tipo de canal

Los parámetros soportados se proporcionan en el cuadro de abajo:

Nombre del parámetro	Tipo de parámetro	Descripción
"port"	num	(Para el cliente o el servidor TCP) puerto TCP
"target"	string	(Para el cliente TCP) Dirección IP del servidor TCP por alcanzar, como "192.168.0.254"
"clients"	num	(Para el servidor TCP) Número máximo de clientes simultáneos en el servidor
"endOfString"	num	(Para la línea serie, el cliente y el servidor TCP) Código ASCII para el extremo del carácter de la secuencia a utilizar con los operadores sio '=' (en el rango [0, 255])
"timeout"	num	(Para la línea serie, el cliente y el servidor TCP) Tiempo de reacción máximo para el canal de comunicaciones. 0 significa sin tiempo de espera.
"baudRate"	num	(Para la línea serie) Velocidad de comunicación
"parity"	string	(Para la línea serie) Control de paridad: "none", "even" o "odd"
"bits"	num	(Para la línea serie) Número de bits por byte (5, 6, 7 o 8)
"stopBits"	num	(Para la línea serie) Número de bits de parada por byte (1 o 2)
"mode"	string	(Para la línea serie) Modo de comunicación: "RS232" ó "RS422"
"flowControl"	string	(Para la línea serie) Control de flujo: "none" ó "hardware"

Ejemplo

Este programa fija los parámetros para una línea serie.

```
sioCtrl(io:portSerial1, "baudRate", 115200)
sioCtrl(io:portSerial1, "bits", 8)
sioCtrl(io:portSerial1, "parity", "none")
sioCtrl(io:portSerial1, "stopBits", 1)
sioCtrl(io:portSerial1, "timeout", 0)
sioCtrl(io:portSerial1, "endOfString", 13)
```

CAPÍTULO 4

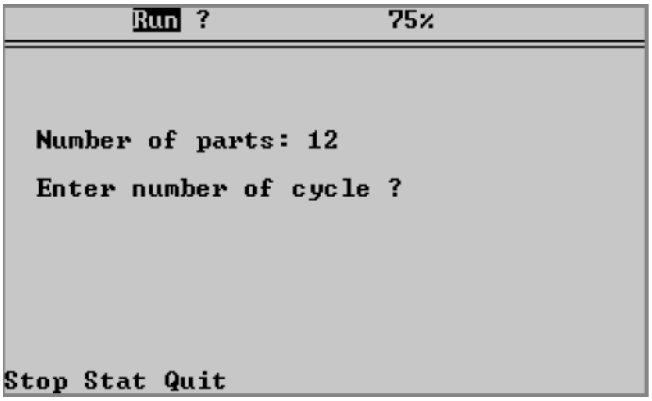
INTERFAZ DE USUARIO

4.1. PÁGINA DE USUARIO

Las instrucciones de interfaz de usuario del lenguaje **VAL3** permiten:

- la visualización de mensajes en una página del MCP (mando manual) reservada a la aplicación
- la adquisición de las presiones-teclas en el teclado del **MCP**

Página de usuario



La página de usuario tiene **14** líneas de **40** columnas. La última línea puede ser utilizada para realizar menús con tecla asociada. Se encuentra disponible una línea adicional para la visualización de un título.

4.2. INSTRUCCIONES

void **userPage**(), void **userPage**(bool bFijo)

Sintaxis

void userPage ()
void userPage (<bool bFijo>)

Función

Hace que se presente la página de usuario en la pantalla del **MCP**.
Si el parámetro **bFijo** es **true**, sólo la página de usuario será accesible al operador, a excepción de la página de cambio de perfil accesible por la tecla de acceso rápido "Shift User". Cuando esta página se muestra, es posible detener la aplicación con la tecla "Stop" si el perfil de usuario en curso lo autoriza.

Si el parámetro es **false**, las otras páginas de la interfaz de usuario **MCP** vuelven a ser accesibles.

void **gotoxy**(num nX, num nY)

Sintaxis

void gotoxy(<num nX>, <num nY>)

Función

Coloca el cursor de la página de usuario en las coordenadas **(nX, nY)**. La esquina de arriba a la izquierda tiene por coordenadas **(0,0)** y la esquina de abajo a la derecha **(39, 13)**.

El número de la columna **nX** es tomado del módulo **40**. El número de líneas **nY** es tomado del módulo **14**.

Parámetros

num nX	Columna de cursor (**0** a **39**)
num nY	Fila de cursor (**0** a **13**)

Véase también

void cls()

void **cls**()

Sintaxis

void cls()

Función

Borra la página de usuario y coloca el cursor en **(0,0)**.

Véase también

void gotoxy(num nX, num nY)

num **getDisplayLen**(string sTexto)

Sintaxis

void getDisplayLen(string sTexto)

Función

Retorna la longitud de **sTexto** sobre la visualización **MCP** (número de columnas necesarias para visualizar **sTexto**).

Para las cadenas de caracteres **ASCII**, la longitud sobre la visualización es el número de caracteres en la cadena de caracteres; por consiguiente **getDisplayLen()** es idéntico a la instrucción **len()**.

Algunos caracteres (chinos) se visualizan sobre dos columnas de visualización adyacentes; por consiguiente, **getDisplayLen()** es mayor que la longitud **sTexto** y puede utilizarse para controlar el alineamiento **sTexto** en la pantalla.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres

Véase también

num len(string sTexto)

void **put**() void **putln**()

Sintaxis

void put(<string sTexto>)
void put(<num nValor>)
void putln(<string sTexto>)
void putln(<num nValor>)

Función

Muestra, en la posición del cursor de la página de usuario, la **sTexto** o el **nValor** especificado (con **3** cifras después de la coma). A continuación, el cursor se coloca sobre el carácter que sigue el último carácter del mensaje visualizado (instrucción **put**), o sobre el primer carácter de la línea siguiente (instrucción **putln**).
Al final de la línea, la visualización continúa en la línea siguiente.
Al final de la página, la visualización de la página de usuario se desfasa una línea hacia arriba.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres
num nValor	Expresión numérica

Véase también

void popUpMsg(string sTexto)
void logMsg(string sTexto)
void title(string sTexto)

void **title**(string sTexto)

Sintaxis

void title(<string sTexto>)

Función

Cambia el título de la página de usuario.

La posición actual del cursor no se modifica por la instrucción **title()**.

Parámetros

string sTexto	Expresión de tipo cadena de caracteres

num **get**()

Sintaxis

num get(<string& sCadena>)
num get(<num& nValor>)
num get()

Función

Hace la adquisición de una cadena, un número o una tecla en el teclado de la consola.

El parámetro **sCadena** o **nValor** se visualiza en la posición actual del cursor y puede ser cambiado por el usuario. La introducción se completa pulsando una tecla de menú o las teclas **Return** o **Esc**.

La instrucción reenvía el código de la tecla que ha terminado la entrada de datos.

Cuando se pulsa **Return** o un menú, la variable **sCadena** o **nValor** se actualiza. Cuando se pulsa **Esc**,ésta no cambia.

Si no ha pasado ningún parámetro, la instrucción **get()** espera que se pulse una tecla cualquiera y retorne su código. La tecla pulsada no aparece en la pantalla.

En todos los casos, la posición en curso del cursor no se modifica por la instrucción **get()**.

Sin **Shift**

3	Caps	Space			Move
283	-	32			-
2	Shift	Esc	Help	Ret.	Run
282	-	255	-	270	-
	Menu	Tab	Up	Bksp	Stop
	-	259	261	263	-
1	User	Left	Down	Right	
281	-	264	266	268	

Con **Shift**

3	Caps	Space			Move
283	-	32			-
2	Shift	Esc	Help	Ret.	Run
282	-	255	-	270	-
	Menu	UnTab	PgUp	Bksp	Stop
	-	260	262	263	-
1	User	Home	PgDn	End	
281	-	265	267	269	

Menús (con o sin **Shift**)

F1	F2	F3	F4	F5	F6	F7	F8
271	272	273	274	275	276	277	278

Para las teclas estándares, el código retornado es el código **ASCII** del carácter correspondiente:

Sin **Shift**									
q	w	e	r	t	y	u	i	o	p
113	119	101	114	116	121	117	105	111	112
a	s	d	f	g	h	j	k	l	<
97	115	100	102	103	104	106	107	108	60
z	x	c	v	b	n	m	.	,	=
122	120	99	118	98	110	109	46	44	61

Con **Shift**									
7	8	9	+	*	;	(	)	[	]
55	56	57	43	42	59	40	41	91	93
4	5	6	-	/	?	:	!	{	}
52	53	54	45	47	63	58	33	123	125
1	2	3	0	"	%	-	.	,	>
49	50	51	48	34	37	95	46	44	62

Con doble **Shift**									
Q	W	E	R	T	Y	U	I	O	P
81	87	69	82	84	89	85	73	79	80
A	S	D	F	G	H	J	K	L	}
65	83	68	70	71	72	74	75	76	125
Z	X	C	V	B	N	M	$	\	=
90	88	67	86	66	78	77	36	92	61

Parámetros

string& sCadena	Variable de tipo **string**
num& nValor	Variable de tipo **num**

Ejemplo

```
num nValor
num nKey
// Espera la presión de Return para validar la entrada
do
 nKey = get (nValor)
until (nKey == 270)
```

Véase también

num getKey()

num **getKey**()

Sintaxis

num getKey()

Función

Hace la adquisición de una tecla en el teclado de la consola. Retorna el código de la última tecla pulsada después de la última llamada a **getKey()**, **-1** si no se ha pulsado desde entonces ninguna tecla. Un golpe de tecla puede detectarse únicamente cuando se visualiza la página del usuario.

Contrariamente a la instrucción **get()**, **getKey()** retorna de inmediato.

La tecla pulsada no se muestra, y la posición en curso del cursor no se modifica.

Ejemplo

```
// Muestra el reloj sistema hasta que se pulse una tecla cualquiera
getKey()                              // Reinicializa el código de la última tecla pulsada
while (getKey()== -1)
  gotoxy(0,0)
  put(toString(«», clock()* 10))
endWhile
```

Véase también

num get(), void userPage(), void userPage(bool bFijo)

bool isKeyPressed(num nCódigo)

bool **isKeyPressed**(num nCódigo)

Sintaxis

bool isKeyPressed(<num nCódigo>)

Función

Retorna el estado de la tecla especificada por su código (véase **get()**), **true** si se pulsa la tecla, **false** en caso contrario. Un golpe de tecla puede detectarse únicamente cuando se visualiza la página del usuario, excepto para las teclas (1), (2) y (3), las cuales son siempre detectadas.

Véase también

num get(), void userPage(), void userPage(bool bFijo)

void **popUpMsg**(string sTexto)

Sintaxis

void popUpMsg(<string sTexto>)

Función

Presenta **sTexto** en una ventana **"popup"** por encima de la ventana en curso del **MCP**. Esta ventana se mantiene visualizada hasta que sea validada por el menú **Ok** o la tecla **Esc**.

Véase también

void userPage(), void userPage(bool bFijo)
void put() void putln()

void **logMsg**(string sTexto)

Sintaxis

void logMsg(<string sTexto>)

Función

Escribe **sTexto** en el historial del sistema (mensaje de error). El mensaje será grabado con la fecha y la hora en curso. "USR" se añade al inicio de la cadena de caracteres, para etiquetarlo como mensaje de usuario.

Véase también

void popUpMsg(string sTexto)

string **getProfile**()

Sintaxis

string getProfile()

Función

Retorna el nombre del perfil de usuario en curso.

Véase también

num setProfile(string sLoginUsuario, string sContraseñaUsuario)

num **setProfile**(string sLoginUsuario, string sContraseñaUsuario)

Sintaxis

num setProfile(<string sLoginUsuario>, <string sContraseñaUsuario>)

Función

Cambia el perfil del usuario actual (efecto inmediato).

La función reenvía:

0: El perfil del usuario especificado es ahora efectivo

-1: El perfil del usuario definido no está especificado

-2: La contraseña del usuario especificado no es correcta

-3: **'staubli'** no está autorizado como perfil del usuario con esta instrucción

-4: El perfil de usuario actual es **'staubli'** y no puede cambiarse con esta instrucción

Véase también

string getProfile()

S6.4

string **getLanguage**()

Sintaxis

string getLanguage()

Función

Esta instrucción devuelve el lenguaje actual del controlador del robot.

Ejemplo

```
switch(getLanguage())
  case "francais"
    sMessage="Attention!"
  break
  case "english"
    sMessage="Warning!"
  break
  case "deutsch"
    sMessage="Achtung!"
  break
  case "italiano"
    sMessage="Avviso!"
  break
  case "espanol"
    sMessage="¡Advertencia!"
  break
  case "chinese"
    sMessage="注意!"
  break
  default
    sMessage="Warning!"
  break
endSwitch
```

Véase también

bool setLanguage(string sLenguaje)

S6.4

bool **setLanguage**(string sLenguaje)

Sintaxis

bool setLanguage(<string sLenguaje>)

Función

Esta instrucción modifica el lenguaje actual del controlador del robot: el nombre del lenguaje especificado sLenguaje debe corresponder al nombre de un archivo de traducción en el controlador. Remitirse al manual del controlador para eliminar, o instalar lenguajes adicionales en el controlador del robot.

Ejemplo

Este programa cambia el lenguaje del robot al chino:

```
if(setLanguage("chinese")==false)
  putln("The Chinese language is not available on the robot controller")
endIf
```

Véase también

string getLanguage()

S6.4

string **getDate**(string sFormato)

Sintaxis

string getDate(<string sFormato>)

Función

Esta instrucción devuelve la fecha actual y/o la hora del controlador del robot. El parámetro sFormato especifica el formato para la fecha devuelta. En esta secuencia, cada ocurrencia de algunas palabras claves se substituye por el valor correspondiente de la fecha o la hora. Las palabras claves de formato soportadas se enumeran en el cuadro de abajo:

Palabra clave	Descripción
%y	Año de 2 dígitos (00-99), sin siglo
%Y	Año de 4 dígitos como 2007
%m	Mes (00-12)
%d	Día (00-31)
%H	Hora en el formato 24 horas (00-23)
%I	Hora en el formato 12 horas (01-12)
%p	Indicador A.M./P.M. para el reloj de 12 horas
%M	Minutos (00-59)
%S	Secundos (00-59)

Ejemplo

Este programa muestra la fecha y la hora en el formato "January 01, 2007 13:45:23"

```
switch (getDate("%m"))
  case "01"
    sMes="Enero"
  break
  case "02"
    sMes="Febrero"
  break
  case "03"
    sMes="Marzo"
  break
  case "04"
    sMes="Abril"
```

```
  break
  case "05"
    sMes="Mayo"
  break
  case "06"
    sMes="Junio"
  break
  case "07"
    sMes="Julio"
  break
  case "08"
    sMes="Agosto"
  break
  case "09"
    sMes="Septiembre"
  break
  case "10"
    sMes="Octubre"
  break
  case "11"
    sMes="Noviembre"
  break
  case "12"
    sMes="Diciembre"
  break
  default
    sMes="???"
  break
endSwitch
// Muestra la fecha y la fecha en la forma: "January 01, 2007  13:45:23"
putln(getDate(sMes+" %d, %Y  %H:%M:%S"))
```

CAPÍTULO 5

TAREAS

5.1. DEFINICIÓN

Una tarea es un programa que está siendo ejecutado. Una aplicación puede (y generalmente tiene) diversas tareas en ejecución.

En una aplicación, se encontrará especialmente una tarea para los desplazamientos del brazo, una tarea autómata, una tarea para la interfaz de usuario, una tarea para el seguimiento de las señales de seguridad, tareas de comunicación...

Una tarea se caracteriza por los siguientes elementos:

- un nombre: un único identificador de tarea dentro de la biblioteca o la aplicación
- una prioridad o un período: parámetro para la secuenciación de las tareas
- un programa: punto de entrada (y salida) de la tarea
- un estado: activo o parado
- la próxima instrucción a ejecutar (y su contexto)

5.2. REANUDACIÓN TRAS UN ERROR DE EJECUCIÓN

Cuando una instrucción causa un error de ejecución, se para la tarea. Se utiliza la instrucción **taskStatus()** para diagnosticar el error de ejecución. Entonces es posible volver a arrancar la tarea con la instrucción **taskResume()**. Si el error de ejecución puede corregirse, la tarea puede reanudarse desde la línea de instrucción donde se paró. En caso contrario, es necesario volver a partir sea antes, sea después de esta línea de instrucción.

Puesta en marcha y parada de aplicación

Cuando se pone en marcha una aplicación, se ejecuta su programa **start()** en una tareal del nombre de la aplicación seguido de **'~'**, y de prioridad **10**.

Cuando una aplicación se termina, su programa **stop()** se ejecuta en una tarea del nombre de la aplicación precedido de **'~'**, y de prioridad **10**.

Si la parada de una aplicación **VAL3** se provoca desde la interfaz de usuario del **MCP**, la tarea de arranque, si ésta existe todavía, se destruye inmediatamente. Se ejecuta después del programa stop(), y, a continuación, todas las tareas de la aplicación, si quedan por hacer, se destruyen en el orden inverso del orden en que fueron creadas.

5.3. VISIBILIDAD

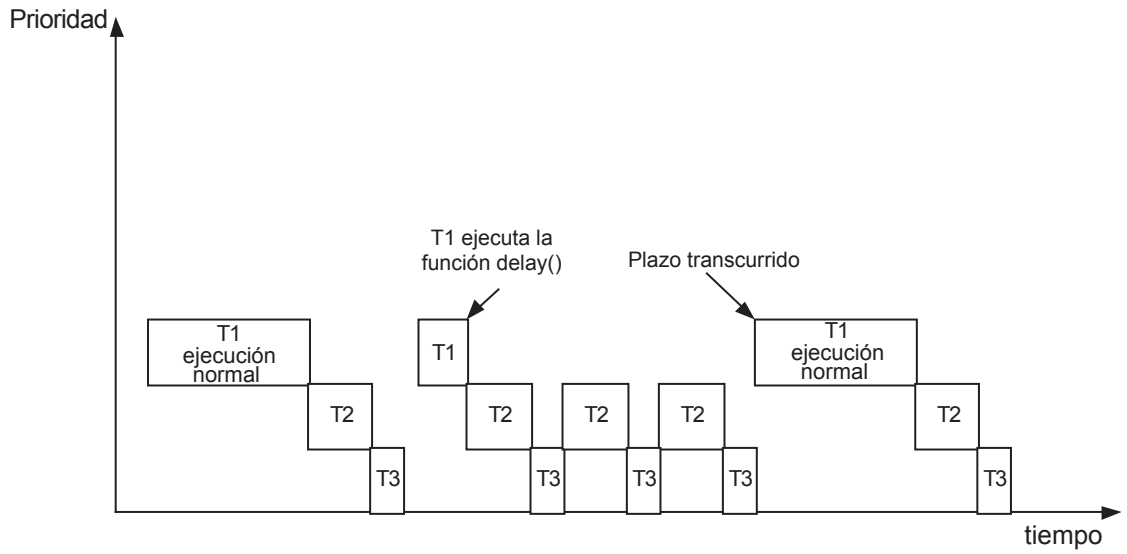
Las tareas sólo son visibles desde el interior del programa o de la biblioteca que las creó. Las instrucciones taskSuspend(), taskResume(), taskKill() y taskStatus() actúan sobre una tarea creada por otra biblioteca, como si no se hubiese creado la tarea. Esto significa que dos bibliotecas diferentes pueden crear tareas que tienen el mismo nombre.

5.4. SECUENCIACIÓN

Cuando una aplicación tiene varias tareas en curso de ejecución, éstas parecen ejecutarse simultánea e independientemente. Esto es cierto si se observa la aplicación globalmente en intérvalos de tiempo suficientemente largos (del orden del segundo), pero no es exacto cuando uno se interesa en el comportamiento preciso en un intervalo de tiempo reducido.

En efecto, como el sistema tiene un solo procesador, sólo puede ejecutar una tarea a la vez. La simultaneidad de la ejecución se simula a través de una secuenciación rápida de las tareas, que ejecutan una después de otras unas instrucciones, antes que el sistema pase a la tarea siguiente.

Secuenciación



La secuenciación de las tareas **VAL3** se hace según las siguientes reglas:

1. Las tareas se secuencian según su orden de creación
2. En cada secuencia, el sistema trata de ejecutar un número de instrucciones **VAL3** igual a la prioridad de la tarea.
3. Cuando una línea de instrucción no se puede terminar (error de ejecución, esperando una señal, tarea parada, etc.) el sistema se mueve a la siguiente tarea **VAL3**.
4. Cuando se terminan todas las tareas **VAL3**, el sistema conserva tiempo disponible para las tareas del sistema de prioridad inferior (por ej. comunicación de red, refresco de la pantalla del usuario, acceso a los ficheros) antes de iniciar un nuevo ciclo.
   La espera máxima entre dos ciclos secuenciales es igual a la duración del último ciclo de secuencia; pero generalmente, esta espera es nula ya que el sistema no la necesita.

Las instrucciones **VAL3** que pueden provocar la secuenciación inmediata de la tarea siguiente son:

- **watch()** (espera temporizada de una condición)
- **delay()** (temporización)
- **wait()** (espera de una condición)
- **waitEndMove()** (espera de la parada del brazo)
- **open()** y **close()** (espera de la parada del brazo y, a continuación, temporización)
- **get()** (espera de una presión de tecla)
- **taskResume()** (espera que la tarea esté lista para ser reiniciada)
- **taskKill()** (espera que la tarea esté efectivamente terminada)
- **disablePower()** (espera que la potencia se haya cortado efectivamente)
- Las instrucciones de acceso al contenido del disco (**libLoad, libSave, libDelete, libList, setProfile**)
- Las instrucciones de lectura/escritura de sio (operador =, **sioGet(), sioSet()**)
- **setMutex()** (espera que el mutex booleano sea falso)

5.5. TAREAS SÍNCRONAS

La secuencia anteriormente descrita es la secuencia de las tareas normales, llamadas tareas asíncronas, cuya ejecución programa el sistema lo más rápido posible. A veces es necesario programar las tareas a intervalos regulares, tanto para la adquisición de datos como para el control de periféricos: se habla entonces de tareas síncronas.

Estas son ejecutadas en ciclo en secuencia, mediante la interrupción de la tarea asíncrona actual entre dos líneas **VAL3**. Cuando las tareas síncronas se terminan, la tarea asíncrona se reanuda.

La secuenciación de las tareas síncronas **VAL3** obedece a las siguientes reglas:

1. Cada tarea síncrona es secuenciada exactamente una vez para cada período definido en el momento de la creación de la tarea (por ejemplo cada 4 ms).
2. Durante cada secuenciación, el sistema ejecuta hasta 3000 líneas de instrucciones **VAL3**. Pasa a la siguiente tarea cuando una línea de instrucciones no puede terminarse inmediatamente (error en la ejecución (runtime), espera de una señal, tarea parada, etc.).
   En la práctica, una tarea síncrona se termina a menudo explícitamente por la instrucción delay(0) que obliga el sistema a pasar a la secuenciación de la siguiente tarea.
3. Las tareas síncronas de la misma duración son secuenciadas según su orden de creación.

5.6. ERROR DE CADENCIA

Si la duración de ejecución de una tarea síncrona **VAL3** es superior al período definido, el ciclo en curso se termina normalmente pero se anula el ciclo siguiente. Este error de cadencia se indica a la aplicación **VAL3** regulando la variable booleana especialmente prevista a tal efecto en el momento de la creación de la tarea en "true". Así pues, esta variable booleana indica al principio de cada ciclo si la secuencia anterior se realizó completamente o no.

5.7. ACTUALIZACIÓN DE LAS ENTRADAS/SALIDAS

Las entradas son refrescadas antes de la ejecución de las tareas síncronas y asíncronas. De la misma manera, las salidas son refrescadas tras la ejecución de las tareas síncronas y asíncronas.

ATENCIÓN:
No es posible definir cuáles son las entradas/salidas que debe utilizar una tarea determinada. Cada actualización se refiere, por lo tanto, al conjunto de las entradas/salidas.
La actualización de las entradas/salidas en Modbus, BIO board, MIO board, CIO board o AS-i bus no es controlado por el programador VAL3. Se pueden actualizar en cualquier momento durante la secuenciación de una tarea VAL3.

5.8. SINCRONIZACIÓN

A veces es necesario sincronizar varias tareas antes que estas prosigan su ejecución.

Si se conoce a priori la duración de ejecución de cada una de estas tareas, esta sincronización puede hacerse sencillamente por la espera de una señal emitida por la tarea más lenta. Pero cuando no se sabe cuál de las tareas será la más lenta, hay que utilizar un mecanismo de sincronización más complejo, del cual se da a continuación un ejemplo de programación **VAL3**.

Ejemplo

```
// variables globales de gestión de sincronización
num n
bool bSynch
n=0                                          // Inicialización de las variables globales
bSynch=false
program Task1()
begin
  while(true)
    call synchro(n, bSynch, 2)               // Sincronización con la tarea 2
    <instructionsTask1>
  endWhile
end
program Task2()
begin
  while(true)
  call synchro(n, bSynch, 2)                 // Sincronización con la tarea 1
  <instructionsTask2>
  endWhile
end
// Programa de sincronización de N tareas
program synchro(num& n, bool& bSynch, num N)
begin
  n = n + 1
  wait((n==N) or (bSynch==true))             // espera de sincronización de las tareas
  bSynch = true
  n = n - 1
  wait((n==0) or (bSynch == false))          // espera de liberación de las tareas
  bSynch = false
end
```

5.9. REPARTICIÓN DE RECURSO

Cuando varias tareas utilizan un mismo recurso del sistema o de la célula (variables globales, pantalla, teclado, robot, etc.), es indispensable asegurarse que no van a perturbarse mutuamente.

Para esto puede utilizarse un mecanismo de exclusión mutua **('mutex')** que proteje un recurso autorizando su acceso a una sola tarea a la vez. Un ejemplo de programación de mutex en **VAL3** se presenta a continuación.

Ejemplo

```
bool bScreen
bScreen= false                          // Inicialización: el recurso de pantalla está
                                        libre


program Task1()
begin
  while(true)
    setMutex(bScreen)                   // Solicitud del recurso pantalla
    call fillScreen(1)
    bScreen = false                     // Liberación del recurso pantalla
    delay(0)                            // Da la mano a la tarea siguiente
  endWhile
end

program Task2()
begin
  while(true)
    setMutex(bScreen)                   // Solicitud del recurso pantalla
    call fillScreen(2)
    bScreen = false                     // Liberación del recurso pantalla
    delay(0)                            // Da la mano a la tarea siguiente
  endWhile
end

// programa para rellenar la pantalla con la cifra i
program fillScreen(num i)
num x
num y
begin
  i = i % 10
  for x = 0 to 39
    for y = 0 to 13
      gotoxy(x, y)
      put(i);
    endFor
  endFor
end
```

5.10. INSTRUCCIONES

void **taskSuspend**(string sNombre)

Sintaxis

void taskSuspend(<string sNombre>)

Función

Suspende la ejecución de la tarea **sNombre**.

Si la tarea está ya en el estado **STOPPED**, la instrucción no tiene efecto.

Un error de ejecución se genera si **sNombre** no corresponde a ninguna tarea **VAL3**, o corresponde a una tarea **VAL3** creada por otra biblioteca.

Parámetros

string sNombre	Expresión de tipo cadena de caracteres

Véase también

void taskResume(string sNombre, num nSalto)
void taskKill(string sNombre)

void **taskResume**(string sNombre, num nSalto)

Sintaxis

void taskResume (<string sNombre>, <num nSalto>)

Función

Reanuda la ejecución de la tarea **sNombre** en la línea situada **nSalto** líneas de instrucciones antes o después de la línea en curso.

Si **nSalto** es negativo, la ejecución se reanuda antes de la línea en curso. Si la tarea no está en el estado **STOPPED**, la instrucción no tiene efecto.

Un error de ejecución se genera si **sNombre** no corresponde a una tarea **VAL3**, corresponde a una tarea **VAL3** creada por otra biblioteca, o si no hay ninguna línea de instrucción en el **nSalto** especificado.

Parámetros

string sNombre	Expresión de tipo cadena de caracteres
num nSalto	Expresión numérica

Véase también

void taskSuspend(string sNombre)
void taskKill(string sNombre)

void **taskKill**(string sNombre)

Sintaxis

void taskKill (<string sNombre>)

Función

Suspende y después destruye la tarea **sNombre**. Después de ejecutarse esta instrucción, la tarea **sNombre** deja de estar presente en el sistema.

Si no existe tarea **sNombre** o si la tarea **sNombre** fue creada por otra biblioteca, la instrucción no tiene ningún efecto.

Parámetros

string sNombre	Expresión de tipo cadena de caracteres

Véase también

void taskSuspend(string sNombre)
void taskCreate string sNombre, num nPrioridad, programa(...)

void **setMutex**(bool& bMutex)

Sintaxis

void setMutex(<bool& bMutex>)

Función

Espera que la variable **bMutex** sea falsa, y luego la ajusta en true.
Esta función exige que se utilice una variable booleana como mecanismo de exclusión mutua con el fin de proteger los recursos compartidos (véase capítulo 5.9).

string **help**(num nCódigoError)

Sintaxis

string help(<num nCódigoError>)

Función

Esta instrucción devuelve la descripción del código de error de ejecución de la tarea especificada con el parámetro nCódigoError. La descripción se proporciona en el lenguaje del controlador actual.

Ejemplo

Este programa verifica si la tarea del "robot" está en error, y muestra el código de error al operador si existe.

```
nCódigoError=taskStatus("robot")
if (nCódigoError > 1)
  gotoxy(0,12)
  put(help(nCódigoError))
endIf
```

num **taskStatus**(string sNombre)

Sintaxis

num taskStatus (<string sNombre>)

Función

Devuelve el estado actual de la tarea **sNombre**, o el código de error de ejecución de la tarea si este último está en la condición de error:

Código	Descripción
-1	Ninguna tarea **sNombre** ha sido creada por la biblioteca o la aplicación actuales
0	Ningún error de ejecución
1	La tarea **sNombre** creada por la aplicación o la biblioteca actual se está ejecutando
10	Cálculo numérico no válido (división por cero).
11	Cálculo numérico no válido (por ejemplo **ln(-1)**)
20	Acceso a un conjunto con un índice mayor que el tamaño del conjunto.
21	Acceso a un conjunto con un índice negativo.
29	Nombre de tarea no válido. Véase instrucción **taskCreate()**.
30	El nombre especificado no corresponde a ninguna tarea **VAL3**.
31	Existe ya una tarea del mismo nombre. Véase instrucción **taskCreate**.
32	Sólo se soportan 2 períodos diferentes para las tareas síncronas. Modificar el período de programación.
40	No hay suficiente espacio de memoria sistema para los datos.
41	No hay suficiente espacio de memoria de ejecución para la tarea. Véase Tamaño de memoria de ejecución.
60	Tiempo máximo de ejecución de la instrucción excedido.
61	Error interno al interpretador **VAL3**
70	Valor de parámetro de instrucción no válido. Véase instrucción correspondiente.
80	Utilización de un dato o un programa de una biblioteca no cargada en la memoria.
81	Cinemática incompatible: Utilización de un punto/campo/config incompatible con el brazo cinemático.
82	El plano o herramienta de referencia de una variable pertenece a una biblioteca y no es accesible a partir del campo de la variable (biblioteca no declarada en el proyecto de la variable o la variable de referencia es privada).
90	La tarea no puede reanudarse en el lugar especificado. Véase instrucción **taskResume()**.
100	La velocidad especificada en el descriptor de movimiento no es válida (negativa o demasiado alta).
101	La aceleración especificada en el descriptor de movimiento no es válida (negativa o demasiado alta).
102	La deceleración especificada en el descriptor de movimiento no es válida (negativa, demasiado alta o inferior a la velocidad).
103	La velocidad de traslación especificada en el descriptor de movimiento es inválida (negativa o demasiado grande).
104	La velocidad de rotación especificada en el descriptor de movimiento es inválida (negativa o demasiado grande).
105	El parámetro **reach** especificado en el descriptor de movimiento no es válido (negativo).
106	El parámetro **leave** especificado en el descriptor de movimiento no es válido (negativo).
122	Tentativa de escritura sobre una entrada del sistema.
123	Utilización de una entrada-salida dio, aio o sio no enlazada a una entrada-salida del sistema.
124	Tentativa de acceso a una entrada-salida protegida del sistema
125	Error de lectura o de escritura en una **dio**, **aio** o **sio** (error en un bus de campo)
150	Imposible ejecutar esta instrucción de movimiento: un movimiento solicitado anteriormente no ha podido ser terminado (punto fuera de alcance, singularidad, problema de configuración...)
153	Comando de movimiento no soportado
154	Instrucción de movimiento no válido: compruebe el descriptor de movimiento.
160	Coordenadas de la herramienta **flange** no válidas
161	Coordenadas del sistema de referencia **world** no válidas
162	Utilización de un **point** sin plano. Véase Definición.
163	Utilización de un plano sin sistema de referencia. Véase Definición.
164	Utilización de una herramienta sin herramienta de referencia. Véase Definición.
165	Sistema de referencia o herramienta de referencia no válido (variable global enlazada a una variable local)
250	No hay licencia de duración de ejecución (runtime) para esta instrucción, o se ha terminado la validez de la licencia de demo.

Parámetros

string sNombre	Expresión de tipo cadena de caracteres

Véase también

void taskResume(string sNombre, num nSalto)
void taskKill(string sNombre)

void **taskCreate** string sNombre, num nPrioridad, programa(...)

Sintaxis

void taskCreate <string sNombre>, <num nPrioridad>, programa([p1] [,p2])

Función

Crea y pone en marcha la tarea **sNombre**.
sNombre debe tener de **1** a **15** caracteres entre **"a..zA..Z0..9_"**. Ninguna otra tarea creada por la misma biblioteca debe llevar el mismo nombre.
La ejecución de **sNombre** comienza con la llamada de **programa** con los parámetros especificados. No es posible utilizar una variable local para un parámetro pasado por referencia.
La tarea se terminará de manera predeterminada con la última línea de instrucciones de **programa**, o más pronto si es explícitamente destruida.
nPrioridad debe estar comprendido entre **1** y **100**. En cada secuenciación de la tarea, el sistema ejecutará un número de líneas de instrucciones igual a **nPrioridad**, o menos si se encuentra una instrucción bloqueante.
Se genera un error de ejecución si el sistema no tiene suficiente memoria para crear la tarea, si **sNombre** no es válido o ya se está utilizando en la misma biblioteca, o si **nPrioridad** no es válido.

Parámetros

string sNombre	Expresión de tipo cadena de caracteres
num nPrioridad	Expresión numérica
programa	Nombre de un programa de la aplicación
p1	Expresión de tipo especificada por programa

Ejemplo

```
program display(string& sText)
begin
  putln(sText)
  sText = "stop"
end
string sMessage
program start()

begin
  sMessage = "start"
  taskCreate "t1", 10, display(sMessage)  // Muestra « start »
  wait(taskStatus("t1") == -1)            // Espera el final de t1
  putln(sMessage)                         // Muestra "stop"
end
```

Véase también

void taskSuspend(string sNombre)
void taskKill(string sNombre)
num taskStatus(string sNombre)

void **taskCreateSync** string sNombre, num nPeriodo, bool& bSobreejecución, programa(...)

Sintaxis

void taskCreateSync <string sNombre>, <num nPeriodo>, <bool& bSobreejecución>, programa(...)

Función

Crea y ejecuta una tarea síncrona.
La ejecución de la tarea exige ejecutar el programa especificado con los parámetros especificados.
Se genera un error de runtime si el sistema no posee la cantidad de memoria necesaria para poder crear la tarea o si uno o más parámetros no son válidos.
Para una presentación detallada de las tareas síncronas (véase capítulo 5.5).

Parámetros

string sNombre	Nombre de la tarea a crear. Debe contener 1 a 15 caracteres elegidos en la siguiente gama "_a..zA..Z0..9". Dos tareas que pertenecen a la misma aplicación o biblioteca no pueden llevar el mismo nombre.
num nPeriodo	Duración de la tarea por crear (s). El valor indicado se redondea al múltiplo de 4 ms inmediatamente inferior (0.004 segundos). El sistema acepta todas las duraciones positivas, pero no más de dos duraciones diferentes de tareas síncronas a la vez.
bool& bSobreejecución	Variable booleana que indica los errores de cadencia. Sólo son soportadas las variables globales, con el fin de garantizar que la variable no se suprima antes de la tarea.
programa	Nombre del programa **VAL3** por llamar cuando se ejecuta la tarea, los parámetros correspondientes aparecen entre paréntesis. Las variables locales no pueden utilizarse como parámetro si éste ha pasado por referencia con el fin de garantizar que la variable no se suprima antes de la tarea.

Ejemplo

```
// Crear una tarea de supervisión cada 20 ms
taskCreateSync "supervisor", 0.02, bSupervisor, supervisor()
```

void **wait**(bool bCondición)

Sintaxis
void wait(<bool bCondición>)

Función
Pone la tarea en curso en espera hasta que **bCondición** sea **true**.
La tarea se mantiene **RUNNING** durante el tiempo de espera. Si **bCondición** es **true** al hacerse la primera evaluación, la ejecución continúa inmediatamente en la misma tarea (no hay secuenciación de la tarea siguiente).

Parámetros

bool bCondición	Expresión booleana

Véase también
void delay(num nSegundos)
bool watch(bool bCondición, num nSegundos)

void **delay**(num nSegundos)

Sintaxis
void delay(<num nSegundos>)

Función
Pone la tarea en curso en espera durante **nSegundos** segundos.
La tarea se mantiene **RUNNING** durante el tiempo de espera. Si **nSegundos** es negativo o nulo, el sistema procede inmediatamente a la secuenciación de la siguiente tarea **VAL3**.

Parámetros

num nSegundos	Expresión numérica

Véase también
num clock()
bool watch(bool bCondición, num nSegundos)

num **clock**()

Sintaxis

num clock()

Función

Reenvía el valor en curso del reloj interno del sistema, expresado en segundos.

La precisión del reloj interno del sistema es el milisegundo. Este se inicializa en **0** cuando se pone en marcha el controlador, y, por tanto, no tiene ninguna relación con la hora civil.

Ejemplo

```
num nStart
nStart=clock()
<instructions>
put("duración de la operación= " )
putln(clock()-nStart)
```

Véase también

void delay(num nSegundos)
bool watch(bool bCondición, num nSegundos)

bool **watch**(bool bCondición, num nSegundos)

Sintaxis

bool watch (<bool bCondición>, <num nSegundos>)

Función

Coloca la tarea actual en espera, hasta que **bCondición** es **true** o han transcurrido **nSegundos** segundos.

Reenvía **true** si la espera se termina cuando la **bCondición** es **true**,en caso contrario **false** cuando la espera se termina porque el plazo ha transcurrido.

La tarea se mantiene **RUNNING** durante el tiempo de espera. Si **bCondición** es **true** al hacerse la primera evaluación, la ejecución continúa inmediatamente en la misma tarea; en caso contrario el sistema secuenciará las otras tareas **VAL3** (aún si **nSegundos** es inferior o igual a **0**).

Parámetros

bool bCondición	Expresión booleana
num nSegundos	Expresión numérica

Ejemplo

```
while (watch (diSensor==true, 20)) == false
  popUpMsg("Espera de la pieza")
  wait(diSensor==true)
endWhile
```

Véase también

void delay(num nSegundos)
void wait(bool bCondición)
num clock()

CAPÍTULO 6

BIBLIOTECAS

6.1. DEFINICIÓN

Una biblioteca **VAL3** es una aplicación **VAL3** con variables o programas que pueden volver a ser utilizados por otra aplicación u otros bibliotecas **VAL3**.
Al ser una aplicación **VAL3**, una biblioteca **VAL3** consta de los siguientes componentes:

- un conjunto de **programas**: las instrucciones **VAL3** que deben ejecutarse
- Un conjunto de **variables globales**: los datos de la biblioteca
- un conjunto de **bibliotecas**: las instrucciones y variables externas utilizadas por la biblioteca

Cuando una biblioteca se está ejecutando, puede contener asimismo:

- un conjunto de **tareas**: los programas propios de la biblioteca que está ejecutándose

Cualquier aplicación puede utilizarse como una biblioteca, y cualquier biblioteca puede utilizarse como una aplicación, si los programas **start()** y **stop()** están definidos en la misma.

6.2. INTERFAZ

Los programas y variables globales de una biblioteca son públicos o privados. Sólo los programas y variables globales públicos son accesibles fuera de la biblioteca. Los programas privados y variables globales pueden ser utilizados únicamente por los programas de la biblioteca.
El conjunto de los programas y variables globales públicos de una biblioteca forman su interfaz: un cierto número de bibliotecas diferentes pueden tener la misma interfaz, en la medida en que sus programas y variables públicos utilicen los mismos nombres.
Las tareas creadas por un programa de una biblioteca son siempre privadas, es decir que son únicamente accesibles por esta biblioteca.

6.3. IDENTIFICADOR DE INTERFAZ

Para poder utilizar una biblioteca, una aplicación debe en primer lugar declarar un identificante que se le asigne, y luego solicitar, en un programa, cargar la biblioteca en la memoria utilizando este identificador.
El identificador se asigna a la interfaz de la librería y no a la propia librería. Toda librería que posea la misma interfaz se puede a continuación cargar utilizando este identificante. Este mecanismo puede utilizarse, por ejemplo, para definir una biblioteca para cada parte posible de una aplicación, y luego carga únicamente la parte actualmente procesada por cada ciclo.

6.4. CONTENIDO

Una biblioteca no tiene un contenido obligatorio: esta puede contener únicamente programas, o únicamente variables, o ambos.

Para acceder al contenido de una biblioteca, se escribe el nombre del identificador seguido de **':'** antes del nombre del programa o del dato de la biblioteca, por ejemplo:

```
pieza:libLoad("pieza_7")      // Carga la biblioteca "pieza_7" con el identificador 'pieza'
title(pieza:Nombre)           // Muestra como título el contenido de la variable nombre de la biblioteca
                              "pieza_7"
call pieza:init()             // Llama el programa init() de la pieza en curso
```

El acceso al contenido de una biblioteca que todavía no se ha cargado en la memoria causa un error de ejecución.

S6.1

6.5. CODIFICACIÓN

VAL3 soporta bibliotecas codificadas, basadas en las herramientas de compresión y codificación ZIP, de uso extendido.

Una biblioteca codificada es un archivo ZIP estándar del contenido de la carpeta de biblioteca (atención: no está soportada la codificación avanzada de 128-bit y 256-bit). El nombre del archivo zip debe tener la extensión '.zip' y menos de 15 caracteres (incluida la extensión). Para lograr una codificación sólida, la contraseña ZIP debe tener más de 10 caracteres y no debe poder encontrarse en un diccionario.

Contraseña secreta, contraseña pública

El intérprete **VAL3** debe tener acceso a la contraseña ZIP secreta, para cargar una biblioteca codificada; para ello, se suministra una herramienta PC, con Stäubli Robotics Studio(*), para codificar la contraseña ZIP secreta en una contraseña **VAL3** pública. La contraseña **VAL3** pública hace posible utilizar la biblioteca codificada en un programa **VAL3**. Pero el contenido de la biblioteca continúa siendo secreto, debido a que la contraseña ZIP no puede calcularse a partir de la contraseña **VAL3**.

(*) Esta herramienta, un passwordZip.exe ejecutable, suministrada con el emulador **VAL3**, requiere la utilización de una licencia **SRS** específica.

Codificación del proyecto

No se puede codificar directamente el proyecto de inicio sobre el controlador. Un proyecto completo puede codificarse:

- Declarando su programa de inicio() como público.

- Creando otro proyecto, el cual simplemente carga el proyecto codificado y llama al programa inicio().

6.6. CARGA Y DESCARGA

Cuando está abierta una aplicación **VAL3**, todas las bibliotecas declaradas son analizadas para construir las interfaces correspondientes. Esta etapa no carga las bibliotecas en la memoria.

> **ATENCIÓN:**
> No están soportadas las referencias circulares entre bibliotecas. Si la biblioteca A utiliza la biblioteca B, la biblioteca B no puede utilizar la biblioteca A**.**

Cuando se carga una biblioteca, se inicializan sus variables globales y se comprueban sus programas para detectar eventuales errores de sintaxis.

No es necesaria la descarga de una biblioteca; ésta se hace automáticamente cuando se termina la aplicación o cuando se carga una nueva biblioteca en lugar de otra.

Cuando se para una aplicación **VAL3** desde la interfaz de usuario del **MCP**, se ejecuta en primer lugar el programa **stop()** y, a continuación, se destruyen todas las tareas de la aplicación y de sus bibliotecas, si quedan algunas.

Ruta de acceso

Las instrucciones **libLoad(), libSave()** y **libDelete()** utilizan una ruta de acceso a una biblioteca, especificado en forma de cadena de caracteres. Una ruta de acceso contiene una raíz (facultativa), una ruta (facultativa), y un nombre de biblioteca, según el formato:

raíz://Camino/Nombre

La raíz especifica el soporte de fichero: **"Floppy"** para un disquete, **"USB0"** para un dispositivo sobre un puerto **USB** (stick, lector de disquetes), **"Disk"** para el disco flash del controlador, o el nombre de una conexión **Ftp** definida sobre el controlador para un acceso de red.
De forma predeterminada, la raíz es **"Disk"** y la ruta queda vacía.

Ejemplos

```
pieza:libLoad("pieza_1")
pieza:libSave("USB0://pieza")
pieza:libSave("Disk://pieza_x/pieza_1")
```

Códigos de error

La biblioteca **VAL3** que maneja funciones nunca genera errores de ejecución pero devuelven un código de error utilizado para comprobar el resultado de la instrucción y para localizar cualquier problema que pueda presentarse.

Código	Descripción
0	Sin error
10	El identificador de biblioteca no ha sido inicializado por **libLoad()**.
11	Biblioteca cargada, pero la interfaz pública no corresponde. Se producirá un error 80 de tiempo de ejecución, si el programa **VAL3** intenta acceder a elementos faltantes. Véase instrucción **libExist**.
12	Imposible cargar biblioteca: la biblioteca contiene datos o programas no válidos, o, para una biblioteca codificada, la contraseña especificada no es correcta.
13	Descarga de biblioteca imposible: La biblioteca está siendo utilizada por otra tarea.
14	Descarga de biblioteca imposible: La biblioteca posee una tarea **VAL3** de ejecución. Todas las tareas creadas por los programas **VAL3**, a partir de la biblioteca, deberán haber sido completadas, antes de descargar la biblioteca.
20	Error de acceso fichero: la raíz de la ruta no es válida.
21	Error de acceso fichero: la ruta no es válida.
22	Error de acceso fichero: el nombre no es válido.
23	Biblioteca codificada esperada. La biblioteca no está codificada, o está mal codificada.
>=30	Error de lectura/escritura en un fichero.

S6.1

Error de lectura/escritura en un fichero

Código	Descripción
31	No se puede guardar la biblioteca: la ruta especificada contiene ya una librería. Para reemplazar la biblioteca en un disco, borrarla primero, con **libdelete()**.
32	Informes de variador/amplificador "Dispositivo no encontrado"
33	Informes de variador/amplificador "Error de dispositivo"
34	Informes de variador/amplificador "Temporización de dispositivo agotada"
35	Informes de variador/amplificador "Dispositivo protegido contra escritura"
36	Informes de variador/amplificador "Disco no presente"
37	Informes de variador/amplificador "Disco no formateado"
38	Informes de variador/amplificador "Disco lleno"
39	Informes de variador/amplificador "Archivo no encontrado"
40	Informes de variador/amplificador "Archivo de sólo lectura"
41	Informes de variador/amplificador "Conexión rechazada"
42	Informes de variador/amplificador "El servidor Ftp no responde"
43	Informes de variador/amplificador "Error ftp kernel"
44	Informes de variador/amplificador "Error de parámetros ftp"
45	Informes de variador/amplificador "Error de acceso ftp"
46	Informes de variador/amplificador "Disco ftp lleno"
47	Informes de variador/amplificador "Login de usuario ftp inválido"
48	Informes de variador/amplificador "Conexíon ftp no definida"

S6.2

6.7. INSTRUCCIONES

num identificante:**libLoad**(string sCamino)

S6.1 num identificante:**libLoad**(string sCamino, string sContraseña)

Sintaxis

num identificante:libLoad(string sCamino)

S6.1 **num identificante:libLoad(string sCamino, string sContraseña)**

Función

Incializa el identificador de biblioteca, cargando el programa y variables de biblioteca en la memoria, tras los **sCamino** especificados. El parámetro **sContraseña** (opcional) especificado se utiliza como clave de descodificación para bibliotecas codificadas. El **sContraseña** especificado debe ser la contraseña **VAL3** calculada a partir de la contraseña ZIP secreta de la biblioteca codificada (véase el capítulo 6.5, página 94).

Retorna **0** tras una carga exitosa, un código de error de carga de biblioteca, si persisten tareas en ejecución creadas por la biblioteca, si el camino de acceso a la biblioteca es inválido, si la biblioteca contiene errores de sintaxis o si la biblioteca especificada no corresponde a la interfaz declarada por el identificador.

Véase también

num identificante:libSave(), num libSave()

num identificante:**libSave**(), num **libSave()**

Sintaxis

num identificante:libSave()
num identificante:libSave(string sCamino)

Función

Guarda las variables y programas asignados al identificador de la biblioteca. Si se llama **libSave()** sin identificador, se guarda la aplicación de la biblioteca que inicia. Si está especificado un parámetro, la salvaguarda se hace en el **sCamino** indicado. En caso contrario, la salvaguarda se hace en la ruta especificada al proceder a la carga.

Devuelve **0** si se ha guardado el contenido, un código de error si el identificador no se ha inicializado, si el camino no es válido, si se produce un error de la escritura o si la trayectoria especificada ya contiene una biblioteca.

Véase también

num libDelete(string sCamino)

num **libDelete**(string sCamino)

Sintaxis

num libDelete(string sCamino)

Función

Suprime la biblioteca situada en el **sCamino** indicado.

Devuelve **0** si la biblioteca especificada no existe o se ha suprimido, y un código de error si el identificador no se ha inicializado, si el camino es inválido o si ocurre un error de la escritura.

Véase también

num identificante:libSave(), num libSave()

string identificante:libPath(), string libPath()

string identificante:**libPath**(), string **libPath**()

Sintaxis
string identificante:libPath()

Función
Esta instrucción devuelve el camino de acceso de la biblioteca asociada al identificador, o el de la aplicación de llamada si no se especifica ningún identificador.

Véase también
bool libList(string sCamino, string& scontenido)

bool **libList**(string sCamino, string& scontenido)

Sintaxis
bool libList(string sCamino, string& scontenido)

Función
Lista el contenido de acceso de **sCamino** especificado en el conjunto **scontenido**. Retorna **true** si el conjunto **scontenido** lista el resultado completo, y **false** si el conjunto es demasiado pequeño para contener toda la lista.
Todos los elementos del conjunto **scontenido** son primero inicializados en "" (cadena de caracteres vacía). Por consiguiente, tras la ejecución de libList(), se encuentra el fin de la lista, mediante la búsqueda de la primera cadena de caracteres vacía en el conjunto **scontenido**.
Si **scontenido** es una variable global, el tamaño del conjunto es automáticamente extendido, según lo requerido para permitir el almacenamiento del resultado total.

Véase también
string identificante:libPath(), string libPath()

S6.4

bool identifier:**libExist**(string sNombreSímbolo)

Sintaxis
bool identifier:**libExist**(string sNombreSímbolo)

Función
La instrucción **libExist** prueba si un símbolo (datos globales o programa) está definido en una biblioteca. Devuelve verdadero si el símbolo existe y es accesible (público), de lo contrario, falso.

El nombre del símbolo para un programa se debe añadir con "()": "mySymbol" denota un nombre de datos, mientras que "mySymbol()" denota un nombre de programa.

Resulta útil probar la instrucción **libExist** si se define una entrada/salida en un controlador ; también resulta provechoso manejar la evolución de la interfaz de una biblioteca, y adaptar su uso dependiendo de si es una versión más nueva o más vieja de la interfaz.

Ejemplo
Este programa envía un mensaje de registro **sMensajeRegistro** a la línea serie COM1 si existe, de lo contrario lo registra.

```
if(io:libExist("COM1")==true)
  io:COM1=sMensajeRegistro
else
  logMsg(sMensajeRegistro)
endIf
```

Este programa llama al programa "init" de la biblioteca del 'protocolo', si existe:

```
if(protocol:libExist("init()")==true)
  call protocol:init()
endIf
```

CAPÍTULO 7

CONTROL DEL ROBOT

Este capítulo enumera las instrucciones que dan acceso al estado de las diferentes partes del robot.

7.1. INSTRUCCIONES

void **disablePower**()

Sintaxis

void disablePower()

Función

Corta la potencia del brazo y espera que la potencia esté efectivamente cortada.

Si el brazo está en movimiento, se efectúa una rápida parada en trayectoria antes del corte de potencia.

Véase también

void enablePower()
bool isPowered()

void **enablePower**()

Sintaxis

void enablePower()

Función

En modo desplazado, aplica la potencia al brazo.

Esta instrucción no tiene ningún efecto en los modos local, manual o prueba, o cuando la potencia está siendo cortada.

Ejemplo

```
// Aplica la potencia y espera que esté instalada
enablePower()
if(watch(isPowered(), 5) == false)
    putln("No puede ponerse la potencia")
endIf
```

Véase también

void disablePower()
bool isPowered()

bool **isPowered**()

Sintaxis

bool isPowered()

Función

Reenvía el estado de la potencia del brazo:

true: el brazo está con potencia

false: La potencia de brazo ha sido desactivada, o está siendo habilitada

bool **isCalibrated**()

Sintaxis

bool isCalibrated()

Función

Devuelve el estado de la calibración del robot:

true: todos los ejes del robot están calibrados

false: por lo menos un eje de robot no está calibrado

num **workingMode**(), num **workingMode**(num& nEstado)

Sintaxis

num workingMode (num& nEstado)
num workingMode()

Función

Reenvía el modo de funcionamiento en curso del robot:

Modo	Estado	Modo de funcionamento	Estado
0	**0**	**No válido o en transición**	-
1	**0**	**Manual**	Movimiento programado
	1		Movimiento de conexión
	2		Articular **(Joint)**
	3		Cartesiano **(Frame)**
	4		Herramienta **(Tool)**
	5		Hacia punto **(Point)**
	6		Hold
2	**0**	**Prueba**	Movimiento programado **(< 250 mm/s)**
	1		Movimiento de conexión **(< 250 mm/s)**
	2		Movimiento rápido programado **(> 250 mm/s)**
	3		Hold
3	**0**	**Local**	**Move** (movimiento programado)
	1		**Move** (movimiento de conexión)
	2		Hold
4	**0**	**Desplazado**	**Move** (movimiento programado)
	1		**Move** (movimiento de conexión)
	2		Hold

Parámetros

num& nEstado Variable de tipo numérico.

num **esStatus**()

Sintaxis

num esStatus()

Función

Reenvía el estado del circuito de señales de seguridad:

Código	Estado
0	**Todas las señales de seguridad están inactivas.**
1	**Espera de validación después de parada de emergencia.**
2	**Señal de seguridad activada.**

Véase también

num workingMode(), num workingMode(num& nEstado)

S6.4

num **getMonitorSpeed**()

Sintaxis

num getMonitorSpeed()

Función

Esta instrucción devuelve la velocidad del monitor actual del robot (en el rango [0, 100]).

Ejemplo

Este programa, a llamarse en una tarea específica, comprueba que el primer ciclo del robot se haya efectuado a baja velocidad:

```
while true
  if(nCiclo < 2)
    if (getMonitorSpeed()> 10)
      stopMove()
      gotoxy(0,0)
      putln("For the first cycle the monitor speed must remain at 10%")
      wait(getMonitorSpeed()> 10)
    endIf
    restartMove()
  endIf
endWhile
```

Véase también

num setMonitorSpeed(num nVelocidad)

S6.4

num **setMonitorSpeed**(num nVelocidad)

Sintaxis

num setMonitorSpeed(<num nVelocidad>)

Función

Esta instrucción modifica la velocidad del monitor actual del robot. Es eficaz solamente si el robot está en modo de funcionamiento remoto y si el operador no tiene acceso a la velocidad del monitor.

Devuelve 0 si la velocidad del monitor se ha modificado con éxito, de lo contrario un código de error negativo:

Código	Descripción
-1	El robot no está en modo de funcionamiento remoto
-2	La velocidad del monitor está bajo el control del operador: cambiar el perfil de usuario actual para quitar el acceso del operador a la velocidad del monitor
-3	La velocidad especificada no es soportada: debe estar en el rango [0, 100]

Véase también

num getMonitorSpeed()

S6.4

string **getVersion**(string nComponente)

Sintaxis

num getVersion(<string nComponente>)

Función

Esta instrucción devuelve la versión de los diversos componentes de hardware y de software del controlador del robot. El cuadro de abajo lista los componentes soportados y, para cada uno, el formato del valor devuelto:

Componente	Descripción
"VAL3"	Versión del controlador **VAL3**, como "s6.4 - Sep 27 2007 - 16:01:17"
"ArmType"	Tipo de brazo conectado al controlador, "tx90-S1" o "rs60-S1-D20-L200"
"Tuning"	Versión de la sintonización del brazo, por ejemplo "R3"
"Mounting"	Montaje del brazo, como "piso", "pared" o "techo"
"ControllerSN"	Número de serie del controlador, como "F07_12R3A1_C_01"
"ArmSN"	Número de serie del brazo, como "F07_12R3A1_A_01"
"Starc"	La versión del paquete de firmware Starc (CS8C), como "1.16.3 - Sep 27 2007 - 16:01:17"
Nombre de la licencia	Estado de la licencia del software del controlador: "" (no instalado o plazo de demostración vencido), "demo" o "enabled" Los nombres de las licencias de controlador instaladas (por ejemplo "alter", "compliance", "remoteMcp", "oemLicence", "plc", "testMode", "mcpMode", etc.) se enumeran en el panel de control del robot colgante

Ejemplo

```
if getVersion("compliance")!="enabled"
  putln("The compliance license is missing on the controller")
endIf
```

Véase también

string getLicence(string sOemNombreLicencia, string sOemContraseña)

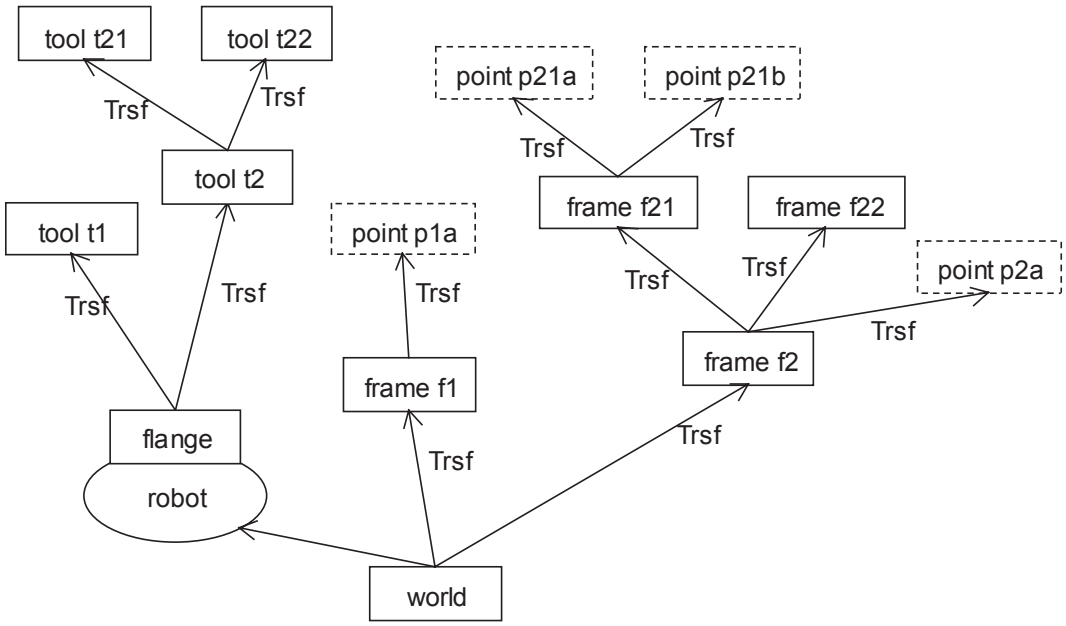
CAPÍTULO 8

POSICIONES DEL BRAZO

8.1. INTRODUCCIÓN

Este capítulo describe los tipos de datos **VAL3** que permiten programar las posiciones del brazo ocupadas en una aplicación **VAL3**.
Se definen dos tipos de posiciones en **VAL3**: posiciones joint (tipo **joint**) que proporcionan la posición angular de cada eje angular y la posición lineal para cada eje lineal, y puntos cartesianos (tipo **point**) que proporcionan la posición cartesiana del punto central de la herramienta en el extremo del brazo relativo a un plano de referencia.
El tipo **tool** describe una herramienta y su geometría, utilizadas para posicionar y controlar la velocidad del brazo; describe igualmente cómo activar la herramienta (salida digital, retardo).
El tipo **frame** describe un plano de referencia geométrico. La ventaja de utilizar sistemas de referencia es que hace más sencillas e intuitivas las manipulaciones geométricas, especialmente en los puntos.
El tipo **trsf** describe una transformación geométrica. Es utilizado por los tipos **tool**, **point** y **frame**.
Por último, el tipo **config** describe la noción más avanzada de configuración de brazo.
Las relaciones entre estos diferentes tipos pueden resumirse de la manera siguiente:

Organigrama: frame / point / tool / trsf



8.2. TIPO JOINT

8.2.1. DEFINICIÓN

Un emplazamiento de articulación (de tipo **joint**) define la posición angular de cada articulación y la posición lineal de cada eje de robot de eje lineal.
El tipo **joint** es un tipo estructurado cuyos campos, presentados en orden, son:

num j1	Posición articular del eje **1**
num j2	Posición articular del eje **2**
num j3	Posición articular del eje **3**
num j...	Posición articular del eje **...** (un campo por eje)

Estos campos se expresan en grados para los ejes rotativos, y en milímetros para los ejes lineales. El origen de cada eje se define según el tipo de brazo utilizado.
De manera predeterminada, cada campo de una variable de tipo **joint** se inicializa en el valor **0**.

8.2.2. OPERADORES

Por orden de prioridad creciente:

joint <joint& jPosición1> = <joint jPosición2>	Asigna **jPosición2** a la variable **jPosición1**, campo por campo, y reenvía **jPosición2**.
bool <joint jPosición1> != <joint jPosición2>	Reenvía **true** si un campo de **jPosición1** no es igual al campo correspondiente de **jPosición2** según la precisión del robot, **false** en caso contrario.
bool <joint jPosición1> == <joint jPosición2>	Reenvía **true** si cada campo de **jPosición1** es igual al campo correspondiente de **jPosición2** según la precisión del robot,**false** en caso contrario.
bool <joint jPosición1> > <joint jPosición2>	Reenvía **true** si cada campo de **jPosición1** es estrictamente superior al campo correspondiente de **jPosición2**, **false** en caso contrario.
bool <joint jPosición1> < <joint jPosición2>	Reenvía **true** si cada campo de **jPosición1** es estrictamente inferior al campo correspondiente de **jPosición2**, **false** en caso contrario. **Atención: jPosición1 > jPosición2 no es estrictamente idéntico a !(jPosición1 < jPosición2)**
joint <joint jPosición1> - <joint jPosición2>	Reenvía la diferencia campo por campo de **jPosición1** y **jPosición2**.
joint <joint jPosición1> + <joint jPosición2>	Reenvía la suma campo por campo de **jPosición1** y **jPosición2**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

8.2.3. INSTRUCCIONES

joint **abs**(joint jposición)

Sintaxis

joint abs(joint jposición)

Función

Devuelve el valor absoluto de un campo **Posición**, campo por campo.

Parámetros

jPosition	Expresión del campo

Detalles

El valor absoluto de un campo, con los operadores de campo ">" o "<", permite calcular fácilmente la distancia que separa la posición de un campo de la posición de referencia.

Ejemplo

```
jMarca de referencia = {90, 45, 45, 0, 30, 0}
jMaxDistance = {5, 5, 5, 5, 5, 5}
j = herej()
// Verifica que todos los ejes tengan menos de 5 grados con relación a la referencia
if(!(abs(j - jMarca de referencia) < jMaxDistance))
  popUpMsg("Acérquese de las marcas")
endIf
```

Véase también

Operator < (joint)
Operator > (joint)

joint **herej**()

Sintaxis

joint herej()

Función

Reenvía la posición actual del campo del brazo.

Detalles

Con la potencia de brazo activada, el valor retornado es la posición enviada a los amplificadores por el controlador y no la posición leíada a partir de los codificadores de eje.

Con la potencia de brazo desactivada, el valor retornado es la posición leída a partir de los codificadores de eje; debido al ruido en las medidas del codificador, la posición puede variar un poco, incluso con el brazo detenido.

La posición del campo del controlador se reactualiza cada 4 ms.

Ejemplo

```
//Esperar hasta que el brazo esté cerca de la posición de la referencia, con un tiempo de espera de 60 s
bStart = watch(abs(herej() - jMarca de referencia) < jMaxDistance, 60)
if bStart==false
  popUpMsg("Move closer to the start position")
endIf
```

Véase también

point here(tool therramienta, frame fMarca de referencia)
bool getLatch(joint& jposición) (CS8C only)
bool isInRange(joint jposición)

bool **isInRange**(joint jposición)

Sintaxis

bool isInRange(joint jposición)

Función

Prueba si la posición de un campo se ajusta a los límites de software de campo del brazo.

Parámetros

jPosition	Expresión de campo por probar

Detalles

Cuando el brazo está fuera de los límites de articulación del software (tras una operación de mantenimiento), no puede moverse el brazo con una aplicación **VAL3**; sólo son posibles los movimientos manuales (con la dirección de movimiento limitada al movimiento hacia los límites).

Ejemplo

```
// Verifica que la posición actual se sitúe dentro de los límites de campo
if isInRange(herej())==false
  putln("Please place the arm within its workspace")
endIf
```

Véase también

joint herej()

void **setLatch**(dio diEntrada) (CS8C only)

Sintaxis

void setLatch(dio diEntrada)

Función

Activa el bloqueo de la posición del robot sobre el próximo frente ascendente de la señal de entrada.

Parámetros

diEntrada	Expresión Dio que define la entrada numérica que debe utilizarse para el bloqueo

Detalles

El bloqueo de la posición del robot es una función material que sólo es soportada por las entradas rápidas del controlador CS8C(io:fIn0, io:fIn1).

Sólo se garantiza la detección del frente ascendente de la señal de entrada si la señal permanece baja al menos durante 0.2 ms antes del frente ascendente, y alta durante al menos 0.2 ms después del frente ascendente.

ATENCIÓN:
El cerrojo sólo se activa al cabo de un determinado tiempo (entre 0 y 0.2 ms) después de la ejecución de la instrucción setLatch. Puede añadir una instrucción delay(0) después de setLatch para cerciorarse de que el cerrojo sea efectivo antes de la ejecución de la siguente instrucción VAL3.

Se genera el error 70 de tiempo de ejecución (valor de parámetro inválido), si la entrada digital especificada no soporta el seguro de posición de robot.

Ejemplo

```
// activa el cerrojo durante la primera entrada rápida (CS8C)
setLatch(io:fIn0)
```

Véase también

bool getLatch(joint& jposición) (CS8C only)

bool **getLatch**(joint& jposición) (CS8C only)

Sintaxis

bool getLatch(joint& jposición)

Función

Lee la última posición bloqueada del robot.

Parámetros

jposición	Expresión de campo que define la variable que debe ponerse al día en función de la posición cerrada

Detalles

La función envía un mensaje si se puede leer una posición bloqueada válida. Si un cerrojo está pendiente, o si el bloqueo nunca se activó, la función envía un mensaje false y la posición no se actualiza.
getLatch reenvía la misma posición bloqueada mientras un nuevo cerrojo no sea activado por la instrucción setLatch.
La posición del brazo se reactualiza en el controlador CS8C cada 0.2 ms; la posición bloqueada corresponde a la posición del brazo entre 0 y 0.2 ms después del frente ascendente de la entrada rápida.

Ejemplo

```
// Espera una posición bloqueada durante 5 segundos.
bLatch = watch(getLatch(jposición)==true, 5)
if bLatch==true
  putln("Posición anclaje alcanzada")
else
  putln("'No se ha detectado ninguna señal anclaje")
endif
```

Véase también

void setLatch(dio diEntrada) (CS8C only)
joint herej()

8.3. TIPO TRSF

8.3.1. DEFINICIÓN

Una transformación (de tipo **trsf**) define un cambio de posición y/o orientación. Es la composición matemática de una traslación y una rotación.

Una transformación no representa por sí misma una posición en el espacio, pero puede interpretarse como la posición y orientación de un punto cartesiano o plano relativo a otro plano.

El tipo **trsf** es un tipo estructurado, cuyos campos, presentados en orden, son:

num x	Traslación a lo largo del eje **x**
num y	Traslación a lo largo del eje **y**
num z	Traslación a lo largo del eje **z**
num rx	Rotación alrededor del eje **x**
num ry	Rotación alrededor del eje **y**
num rz	Rotación alrededor del eje **z**

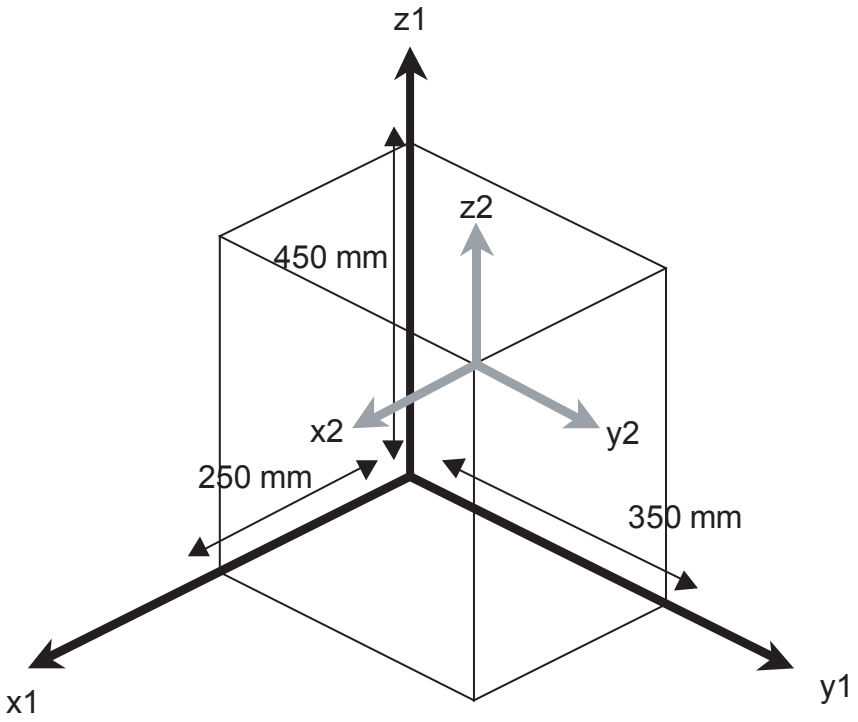
Los campos **x, y** y **z** se expresan en la unidad de longitud de la aplicación (milímetro o pulgada, véase capítulo Unidad de longitud). Los campos **rx, ry** y **rz** se expresan en grados.

Las coordenadas **x, y** y **z** son las coordenadas cartesianas de la traslación (o la posición de un punto o plano en el plano de referencia). Cuando **rx, ry** y **rz** son cero, la transformación es una traslación sin cambio de orientación.

De manera predeterminada, una variable de tipo **trsf** se inicializa en el valor **{0,0,0,0,0,0}**.

8.3.2. ORIENTACIÓN

Orientación

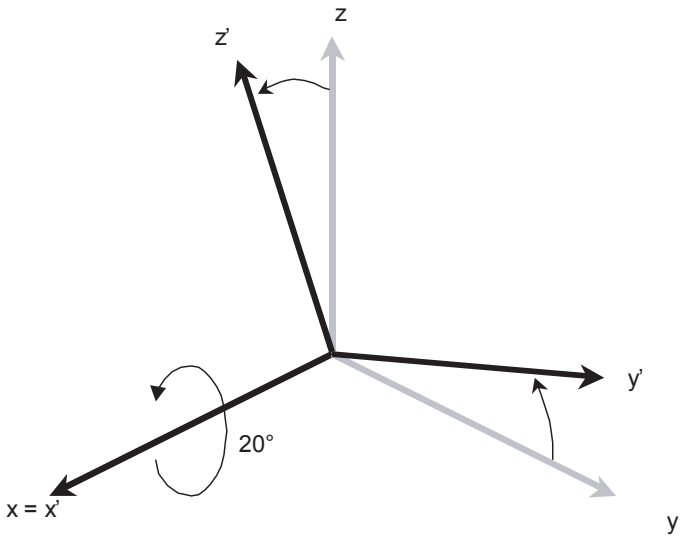


La posición del plano **R2** (gris) respecto a **R1** (negro) es:
x = 250mm, y = 350 mm, z = 450mm, rx = 0°, ry = 0°, rz = 0°

Las tres coordenadas **rx, ry** y **rz** corresponden a los ángulos de las rotaciones que deben aplicarse sucesivamente alrededor de los ejes **x, y** y **z** para obtener la orientación del sistema de referencia.

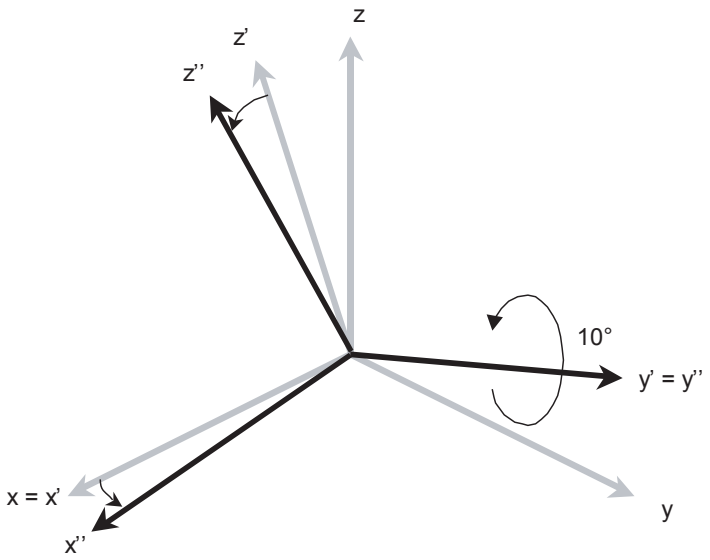
Por ejemplo, la orientación **rx = 20°, ry = 10°, rz = 30°** se obtiene de la manera siguiente. Para comenzar, el plano **(x,y,z)** está girado **20°** alrededor del eje **x**. Se obtiene un nuevo plano **(x',y',z')**. Los ejes **x** y **x'** se confunden.

Rotación del plano respecto al eje: X



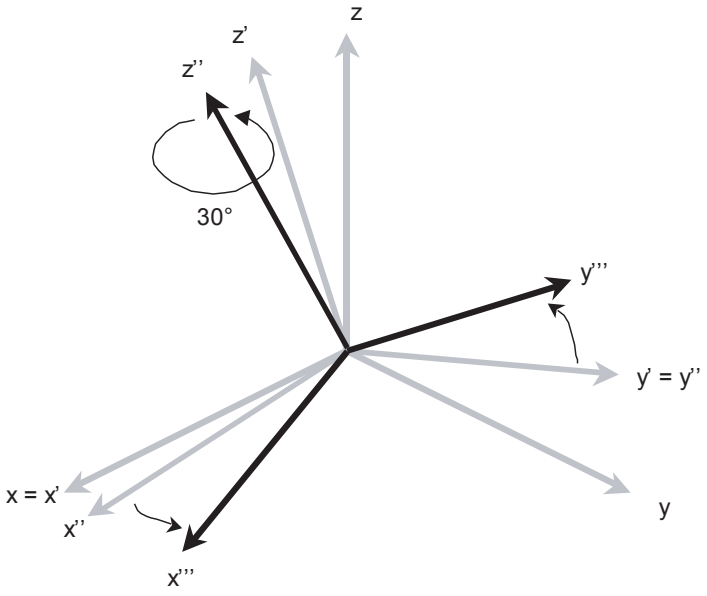
A continuación, el plano está girado **20°** alrededor del eje **y'** del sistema de referencia que se obtuvo en la etapa anterior. Se obtiene un nuevo plano **(x'',y'',z'')**. Los ejes **y'** y **y''** se confunden.

Rotación del plano respecto al eje: Y'



Por último, el sistema de referencia está girado **20°** alrededor del eje **z''** del sistema de referencia que se obtuvo en la etapa anterior. El nuevo plano **(x''',y''',z''')** obtenido es aquel cuya orientación está definida por **rx, ry, rz**. Los ejes **z''** y **z'''** se confunden.

Rotación del plano respecto al eje: Z"



La posición del plano **R2** (gris) respecto a **R1** (negro) es:
x = 250mm, y = 350 mm, z = 450mm, rx = 20°, ry = 10°, rz = 30°

Los valores de **rx, ry** y **rz** son definidos módulo **360** grados. Cuando el sistema calcula **rx, ry** y **rz**, sus valores están siempre comprendidos entre **-180** y **+180** grados. Quedan entonces todavía varios valores posibles para **rx, ry,** y **rz**: El sistema garantiza que al menos dos coordenadas se sitúen entre **-90** y **90** grados (salvo si **rx** vale **+180** y si **ry** vale **0**). Cuando **ry** vale **90** grados **(modulo 180)**, **rx** se elige nulo.

8.3.3. OPERADORES

Por orden de prioridad creciente:

trsf <trsf& trPosición1> = <trsf trPosición2>	Asigna **trPosición2** a la variable **trPosición1**, campo por campo, y reenvía **trPosición2**.
bool <trsf trPosición1> != <trsf trPosición2>	Reenvía **true** si un campo de **trPosición1** no es igual al campo correspondiente de **trPosición2**, **false** en caso contrario.
bool <trsf trPosición1> == <trsf trPosición2>	Reenvía **true** si cada campo de **trPosición1** es igual al campo correspondiente de **trPosición2**, **false** en caso contrario.
trsf <trsf trPosición1> * **<trsf trPosición2>**	Reenvía la composición geométrica de las transformaciones **trPosición1** y **trPosición2**. ¡Atención! **trPosición1** * **trPosición2 != trPosición2** * **trPosición1** en el caso general**!**
trsf ! <trsf trPosición>	Reenvía la transformación invertida de **trPosición**.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

8.3.4. INSTRUCCIONES

num **distance**(trsf trPosición1, trsf trPosición2)

Sintaxis

num distance(<trsf trPosición1>, <trsf trPosición2>)

Función

Reenvía la distancia entre **trPosición1** y **trPosición2**.

> **ATENCIÓN:**
> **Para que la distancia sea válida, es necesario que posición 1 y posición 2 sean definidas respecto al mismo plano de referencia.**

Parámetros

trsf trPosición1	Expresión de tipo transformación
trsf trPosición2	Expresión de tipo transformación

Ejemplo

```
// Muestra la distancia entre dos puntos, cualquiera que sea sus sistemas de referencia
putln(distance(position(point1, world), position(point2, world)))
```

Véase también

point appro(point pPosición, trsf trTransformación)
point compose(point pPosición, frame fMarca de referencia, trsf trTransformación)
trsf position(point pPosición, frame fMarca de referencia)
num distance(point pPosición1, point pPosición2)

S6.4 trsf **interpolateL**(trsf trInicio, trsf trFin, num nPosición)

Sintaxis

trsf interpolateL(<trsf trInicio>,<trsf trFin>,<num nPosición>)

Función

Esta instrucción devuelve una posición intermedia alineada con una posición de inicio **trInicio** y una posición de objetivo **trFin**. El parámetro **nPosición** especifica la interpolación lineal a aplicar según la ecuación, para la coordenada x: trsf.x0 = trInicio.x + (trFin.x-trInicio.x)*nPosición. La misma ecuación sostiene las coordenadas Y y Z.

La orientación rx, ry, rz se calcula con una ecuación similar, pero más compleja. El algoritmo utilizado por interpolateL es igual al algoritmo utilizado por el generador de movimiento para calcular posiciones intermedias sobre una instrucción de movimiento.

interpolateL(trInicio, trFin, 0) devuelve **trInicio** ; interpolateL(trInicio, trFin, 1) devuelve **trFin** ; interpolateL(trInicio, trFin, 0.5) devuelve la posición media entre **trInicio** y **trFin**. Un valor negativo del parámetro **nPosición** da lugar a una posición "antes" de **trInicio**. Un valor superior a 1 da lugar a una posición "después" de **trFin**.

Se genera un error de ejecución si el parámetro nPosición no está en el rango]-1, 2[.

Véase también

trsf position(point pPosición, frame fMarca de referencia)

trsf position(frame tPlano, frame fMarca de referencia)

trsf position(tool tHerramienta, tool tMarca de referencia)

trsf interpolateC(trsf trInicio, trsf trIntermediario, trsf trFin, num nPosición)

trsf align(trsf trPosición, trsf Marca de referencia)

S6.4 trsf **interpolateC**(trsf trInicio, trsf trIntermediario, trsf trFin, num nPosición)

Sintaxis

trsf interpolateC(<trsf trInicio>, <trsf trIntermediario>, <trsf trFin>, <num nPosición>)

Función

Esta instrucción devuelve una posición intermedia respecto al arco de un círculo definido por las posiciones **trInicio**, **trIntermediario** y **trFin**. El parámetro **nPosición** especifica la interpolación circular a aplicarse. El algoritmo utilizado por **interpolateC** es igual al algoritmo utilizado por el generador del movimiento para calcular posiciones intermedias respecto a una instrucción **movec**.

interpolateC(trInicio, trIntermediario, trFin, 0) devuelve **trInicio** ;
interpolateC(trInicio, trIntermediario, trFin, 1) devuelve **trFin** ;
interpolateC(trInicio, trIntermediario, trFin, 0.5) devuelve la posición media respecto al arco entre **trInicio** y **trFin**. Un valor negativo del parámetro **nPosición** da lugar a una posición "antes" de **trInicio**. Un valor superior a 1 da lugar a una posición "después" de **trFin**.

Se genera un error de ejecución si el arco no está correctamente definido (posiciones demasiado cercanas), o si la interpolación de la rotación sigue siendo indeterminada (ver capítulo "Control del movimiento - interpolación de la orientación").

Véase también

trsf position(point pPosición, frame fMarca de referencia)

trsf position(frame tPlano, frame fMarca de referencia)

trsf position(tool tHerramienta, tool tMarca de referencia)

trsf interpolateL(trsf trInicio, trsf trFin, num nPosición)

trsf align(trsf trPosición, trsf Marca de referencia)

S6.4 trsf **align**(trsf trPosición, trsf Marca de referencia)

Sintaxis

trsf align(<trsf trPosición>, <trsf Marca de referencia>)

Función

Esta instrucción devuelve la entrada **trPosición** con la orientación modificada de tal modo que el eje Z de la orientación devuelta esté alineada con el eje más cercano X, Y o Z de la orientación de la referencia de **trMarca de referencia**. Las coordenadas X, Y, Z de **trPosición** y de **trMarca de referencia** no se utilizan: las coordenadas x, y, z del valor devuelto son iguales a las coordenadas x, y, z de **trPosición**.

Véase también

trsf position(point pPosición, frame fMarca de referencia)

trsf position(frame tPlano, frame fMarca de referencia)

trsf position(tool tHerramienta, tool tMarca de referencia)

trsf interpolateL(trsf trInicio, trsf trFin, num nPosición)

trsf interpolateC(trsf trInicio, trsf trIntermediario, trsf trFin, num nPosición)

8.4. TIPO FRAME

8.4.1. DEFINICIÓN

El tipo frame permite definir la posición de sistemas de referencia en la célula.

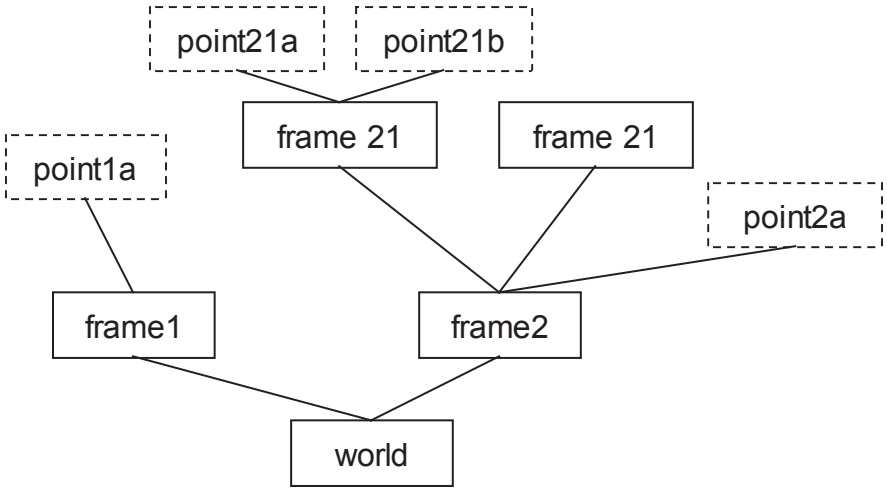
El tipo frame es un tipo estructurado con un solo campo accesible:

trsf trsf	posición del plano en su sistema de referencia

El **sistema de referencia** de una variable de tipo **frame** se define en el momento de su inicialización (desde la interfaz de usuario, o por el operador **=**). El sistema de referencia **world**, de tipo **frame**, está siempre definido en una aplicación **VAL3**: todo plano está, directamente o a través de otros sistemas de referencia, vinculado al sistema **world**.

Un error de ejecución se genera durante un cálculo geométrico si se han modificado las coordenadas del plano **world**.

Vínculos entre sistemas de referencia



De manera predeterminada, una variable de tipo **frame** utiliza **world** como sistema de referencia.

8.4.2. UTILIZACIÓN

Se recomienda fuertemente la utilización de sistemas de referencia en una aplicación robótica:

- **Para proporcionar una vista más intuitiva de los puntos de la aplicación**
  La visualización de los puntos aprendidos de la célula se estructura según la arborescencia de los sistemas de referencia.
- **Para actualizar rápidamente la posición de un conjunto de puntos**
  En cuanto un punto de la aplicación está vinculado a un objeto, es deseable definir un sistema de referencia para este objeto, y vincular los puntos **VAL3** a dicho sistema de referencia. Si el objeto se desplaza, basta con volver a aprender el sistema de referencia para que todos los puntos que están vinculados a éste sean corregidos en una sola vez.
- **Para reproducir una trayectoria a varios lugares de la célula**
  Para esto, se pueden definir los puntos de la trayectoria respecto a un sistema de referencia de trabajo, y aprender un sistema de referencia para cada punto donde deba reproducirse la trayectoria. Mediante la asignación del valor de un plano aprendido en el sistema de referencia de trabajo, toda la trayectoria se "desplaza" sobre el sistema de referencia aprendido.
- **Para calcular fácilmente los desplazamientos geométricos**
  La instrucción **compose()** permite efectuar en cualquier punto desplazamientos geométricos expresados en un sistema de referencia cualquiera. La instrucción **position()** permite calcular la posición de un punto en cualquier sistema de referencia.

8.4.3. OPERADORES

Por orden de prioridad creciente:

frame <frame& fMarca de referencia1> = <frame fMarca de referencia2>	Asigna la posición y el plano de **fMarca de referencia2** a la variable **fMarca de referencia1**.
bool <frame fMarca de referencia1> != <frame fMarca de referencia2>	Reenvía **true** si **fMarca de referencia1** y **fMarca de referencia2** no tienen el mismo sistema de referencia o no tienen la misma posición en su sistema de referencia.
bool <frame fMarca de referencia1> == <frame fMarca de referencia2>	Reenvía **true** si **fMarca de referencia1** y **fMarca de referencia2** tienen la misma posición en el mismo sistema de referencia.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

8.4.4. INSTRUCCIONES

num **setFrame**(point pOrigen, point pEjeOx, point pPlanoOxy, frame& fResult)

Sintaxis

num setFrame(point pOrigen, point pEjeOx, point pPlanoOxy, frame& fResult)

Función

Calculas las coordenadas de **fResult** a partir de su punto de origen **pOrigen**, de un punto **pEjeOx** sobre el eje **(Ox)**, y un punto **pPlanoOxy** sobre el plano **(Oxy)**.

El punto **pEjeOx** debe estar del lado de las **x** positivas. El punto **pPlanoOxy** debe estar del lado de las **y** positivas.

La función reenvía:

0	Sin error.
-1	El punto **pEjeOx** está demasiado cerca de **pOrigen**.
-2	El punto **pPlanoOxy** está demasiado cerca del eje **(Ox)**.

Se genera un error de ejecución si uno de los puntos no tiene ningún plano de referencia.

S6.4 trsf **position**(frame tPlano, frame fMarca de referencia)

Sintaxis

trsf position(<frame tPlano>, <frame fMarca de referencia>)

Función

Esta instrucción devuelve las coordenadas del plano **tPlano** en el plano de referencia **fMarca de referencia**.

Se genera un error de ejecución si **tPlano** o **fMarca de referencia** no tienen ningún plano de referencia.

Véase también

trsf position(point pPosición, frame fMarca de referencia)

trsf position(tool tHerramienta, tool tMarca de referencia)

8.5. TIPO TOOL

8.5.1. DEFINICIÓN

El tipo **tool** permite definir la geometría y la acción de una herramienta.

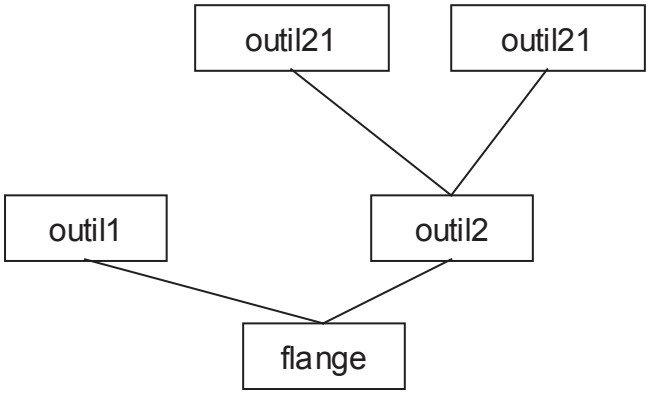
El tipo **tool** es un tipo estructurado con, como campos, en el orden siguiente:

trsf trsf	posición del punto del centro de herramienta (TCP) en su herramienta de base
dio gripper	Salida todo o nada que sirve para accionar la herramienta
num otime	Plazo de apertura de la herramienta (segundos)
num ctime	Plazo de cierre de la herramienta (segundos)

La **herramienta básica** de una variable de tipo **tool** queda definida al inicializarla (desde la interfaz de usuario, o por el operador **=**). La herramienta básica **flange**, de tipo **tool**, está siempre definida en una aplicación **VAL3**: toda herramienta está, directamente o a través de otras herramientas, vinculada a la herramienta **flange**.

Un error de ejecución se genera durante un cálculo geométrico si se han modificado las coordenadas de la herramienta **flange**.

Vínculos entre herramientas



De manera predeterminada, la salida de una herramienta es la salida **io:válvula1** del sistema, los tiempos de apertura y cierre están en **0**, y la herramienta básica es **flange**.

8.5.2. UTILIZACIÓN

Se recomienda fuertemente la utilización de herramientas en una aplicación robótica:

- **Para controlar la velocidad de desplazamiento**
  En el caso de desplazamientos manuales o programados, el sistema controla la velocidad cartesiana en el extremo de la herramienta.
- **Para ir a los mismos puntos con diferentes herramientas**
  Basta con seleccionar la herramienta **VAL3** correspondiente a la herramienta física en el extremo del brazo.
- **Para manejar un desgaste o un cambio de herramienta**
  Basta con actualizar las coordenadas geométricas de la herramienta para que el posicionamiento del brazo se ponga al día.

8.5.3. OPERADORES

Por orden de prioridad creciente:

tool <tool& tHerramienta1> = <tool tHerramienta2>	Asigna la posición y la herramienta básica de **tHerramienta2** a la variable **tHerramienta1**.
bool <tool tHerramienta1> != <tool tHerramienta2>	Reenvía **true** si **tHerramienta1** y **tHerramienta2** no tienen la misma herramienta básica, la misma posición en su herramienta básica, la salida digital, o los mismos tiempos de apertura y cierre.
bool <tool tHerramienta1> == <tool tHerramienta2>	Reenvía **true** si **tHerramienta1** y **tHerramienta2** tienen la misma posición en la misma herramienta básica, utilizan la misma salida digital con los mismos tiempos de apertura y cierre.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

8.5.4. INSTRUCCIONES

void **open**(tool tHerramienta)

Sintaxis

void open (tool tHerramienta)

Función

Acciona la herramienta (apertura) poniendo la salida digital de la herramienta en **true**.
Antes de accionar la herramienta, **open()** espera que el robot esté en el punto al hacer el equivalente de un **waitEndMove()**. Una vez activado, el sistema espera **otime** segundos antes de ejecutar la instrucción siguiente.
Con esta instrucción, no es seguro que el robot sea estabilizado en su posición final antes que la herramienta se active. Cuando la espera de estabilización completa del movimiento es necesaria, se debe utilizar la instrucción **isSettled()**.
Se genera un error de ejecución si el **tHerramienta dio** no está definido o no es una salida, o si no se puede ejecutar un comando previamente grabado del movimiento.

Parámetros

tool tHerramienta	Expresión de tipo herramienta

Ejemplo

```
// la instrucción open() es equivalente a:
waitEndMove()
therramienta.gripper=true
delay(therramienta.otime)
```

Véase también

void close(tool tHerramienta)
void waitEndMove()

void **close**(tool tHerramienta)

Sintaxis

void close (tool tHerramienta)

Función

Acciona la herramienta (cierre) poniendo la salida digital de la herramienta en **false**.
Antes de accionar la herramienta, **open()** espera que el robot esté parado en el punto al hacer el equivalente de un **waitEndMove()**. Una vez activado, el sistema espera ctime segundos antes de ejecutar la instrucción siguiente.
Con esta instrucción, no es seguro que el robot sea estabilizado en su posición final antes que la herramienta se active. Cuando la espera de estabilización completa del movimiento es necesaria, se debe utilizar la instrucción **isSettled()**.
Se genera un error de ejecución si el **tHerramienta dio** no está definido o no es una salida, o si no se puede ejecutar un comando previamente grabado del movimiento.

Parámetros

tool tHerramienta	Expresión de tipo herramienta

Ejemplo

```
// la instrucción close es equivalente a:
waitEndMove()
therramienta.gripper = false
delay(therramienta.ctime)
```

Véase también

Type tool
void open(tool tHerramienta)
void waitEndMove()

S6.4 trsf **position**(tool tHerramienta, tool tMarca de referencia)

Sintaxis

trsf position(<tool tHerramienta>, <tool tMarca de referencia>)

Función

Esta instrucción devuelve las coordenadas de la herramienta **tHerramienta** en el plano de referencia de la herramienta **tMarca de referencia**.

Se genera un error de ejecución si **tHerramienta** o **tMarca de referencia** no tienen ninguna herramienta de referencia.

Véase también

trsf position(point pPosición, frame fMarca de referencia)

trsf position(frame tPlano, frame fMarca de referencia)

8.6. TIPO POINT

8.6.1. DEFINICIÓN

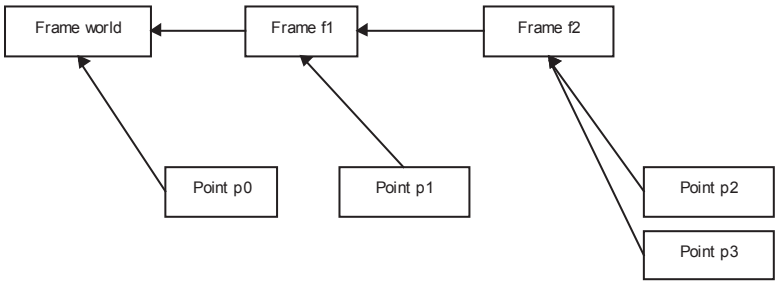
El tipo **point** permite definir la posición y la orientación de la herramienta del robot en la célula.

El tipo **point** es un tipo estructurado con, como campos, en el orden siguiente:

trsf trTrsf	posición del punto en su sistema de referencia
config config	configuración del brazo para alcanzar la posición

El sistema de referencia de un **point** es una variable de tipo **frame**, que se define al iniciarla (desde la interfaz de usuario, por el operador **=** y las instrucciones **here()**, **appro()** y **compose()**).

Definición punto



Se genera un error de ejecución si se utiliza un tipo variable **point** sin plano de referencia definido.

ATENCIÓN:
De manera predeterminada, una variable local de tipo point no tiene un sistema de referencia. Antes de utilizarla, debe inicializarse a partir de otro point, con el operador '=', o mediante una de las instrucciones here(), appro() y compose().

8.6.2. OPERADORES

Por orden de prioridad creciente:

point <point& pPunto1> = <point pPunto2>	Asigna la posición, la configuración y el sistema de referencia de **pPunto2** a la variable **pPunto1**.
bool <point pPunto1> ! = <point pPunto2>	**Reenvía true si pPunto1 y pPunto2 no tienen el mismo sistema de referencia o no tienen la misma posición en su sistema de referencia.**
bool <point pPunto1> == <point pPunto2>	Reenvía **true** si **pPunto1** y **pPunto2** tienen la misma posición en el mismo sistema de referencia.

Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

num **distance**(point pPosición1, point pPosición2)

Sintaxis

num distance(point pPosición1, point pPosición2)

Función

Reenvía la distancia entre **pPosición1** y **pPosición2**.

Se genera un error de ejecución si **pPosición1** o **pPosición2** no tiene un plano de referencia definido.

Parámetros

point pPosición1	Expresión de tipo punto
point pPosición2	Expresión de tipo punto

Ejemplo

```
// Muestra la distancia entre dos puntos, cualquiera que sea sus sistemas de referencia
putln(distance(pPoint1, pPoint2))
```

Véase también

point appro(point pPosición, trsf trTransformación)
point compose(point pPosición, frame fMarca de referencia, trsf trTransformación)
trsf position(point pPosición, frame fMarca de referencia)
num distance(trsf trPosición1, trsf trPosición2)

point **compose**(point pPosición, frame fMarca de referencia, trsf trTransformación)

Sintaxis

point compose(point pPosición, frame fMarca de referencia, trsf trTransformación)

Función

Reenvía **pPosición** a la cual está aplicada la transformación geométrica **trTransformación** definida respecto a **fMarca de referencia**.

> **ATENCIÓN:**
> **El componente de rotación de tTransformación modifica en general no sólo la orientación de pPosición, sino también sus coordenadas cartesianas (salvo si pPosición se sitúa en el origen de fMarca de referencia).**
> **Si se desea que tTransformación sólo modifique la orientación de pPosición, hay que actualizar el resultado con las coordenadas cartesianas de pPosición (véase ejemplo).**

El sistema de referencia y la configuración del punto reenviado son los de **pPosición**.

Se genera un error de ejecución si **pPosición** no tiene ningún plano de referencia definido.

Parámetros

point pPosición	Expresión de tipo punto
frame fMarca de referencia	Expresión de tipo sistema de referencia
trsf trTransformación	Expresión de tipo transformación

Ejemplo

```
point pResult
// modificación de la orientación sin modificación de Posición
pResult = compose (pPosición,fMarca de referencia,trTransformación)
pResult.trsf.x = pPosición.trsf.x
pResult.trsf.y = pPosición.trsf.y
pResult.trsf.z = pPosición.trsf.z
// modificación de Posición sin modificación de orientación
trTransformación.rx = trTransformación.ry =trTransformación.rz = 0
pResult = compose (pResult,fMarca de referencia,trTransformación)
```

Véase también

Operator trsf <trsf pPosición1> * <trsf pPosición2>
point appro(point pPosición, trsf trTransformación)

point **appro**(point pPosición, trsf trTransformación)

Sintaxis

point appro(point pPosición, trsf trTransformación)

Función

Esta instrucción devuelve un punto modificado por una transformación geométrica. La transformación se define relativamente al mismo plano de referencia que el punto de entrada.
El plano de referencia y la configuración del punto devuelto son los del punto de entrada.
Se genera un error de ejecución si **pPosición** no tiene ningún plano de referencia definido.

Parámetros

point pPosición	Expresión de tipo punto
trsf trTransformación	Expresión de tipo transformación

Ejemplo

```
// acérquese a 100 mm por encima del punto (eje z)
point p
movej(appro(p,{0,0,-100,0,0,0}), flange, mNomDesc)   // Aproximación
movel(p, flange, mNomDesc)                           // Vaya al punto
```

Véase también

Operator trsf <trsf trPosición1> * <trsf trPosición2>
point compose(point pPosición, frame fMarca de referencia, trsf trTransformación)

point **here**(tool therramienta, frame fMarca de referencia)

Sintaxis

point here(tool tHerramienta, frame fMarca de referencia)

Función

Reenvía la posición actual de la herramienta **tHerramienta** en **fMarca de referencia** (posición ordenada y no la posición medida).

El sistema de referencia del punto reenviado es **fMarca de referencia**. La configuración del punto reenviado es la configuración en curso del brazo.

Véase también

joint herej()
config config(joint jPosición)
point jointToPoint(tool tHerramienta, frame fMarca de referencia, joint jPosición)

point **jointToPoint**(tool tHerramienta, frame fMarca de referencia, joint jPosición)

Sintaxis

point jointToPoint (tool tHerramienta, frame fMarca de referencia, joint jPosición)

Función

Retorna la posición de **tHerramienta** en el plano **fMarca de referencia**, cuando el brazo se encuentra en la posición angular **jPosición**.
El sistema de referencia del punto reenviado es **fMarca de referencia**. La configuración del punto reenviado es la configuración del brazo en la posición articular **jPosición**.

Parámetros

tool tHerramienta	Expresión de tipo herramienta
frame fMarca de referencia	Expresión de tipo sistema de referencia
joint jPosición	Expresión de tipo posición articular

Véase también

point here(tool therramienta, frame fMarca de referencia)
bool pointToJoint(tool tHerramienta, joint jInicial, point pPosición,joint& jResult)

bool **pointToJoint**(tool tHerramienta, joint jInicial, point pPosición,joint& jResult)

Sintaxis

bool pointToJoint(tool tHerramienta, joint jInicial, point pPosición, joint& jResult)

Función

Esta instrucción calcula la posición angular **jResult** que corresponde al punto especificado **pPosición**. Devuelve **true** si **jResult** se actualiza, **false** si no se ha encontrado ninguna solución.

La posición angular a localizar corresponde a la configuración de **pPosición**. Los campos de valor **free** no imponen la configuración. Los campos de valor **same** imponen la misma configuración que **jInicial**.

Para los ejes que pueden dar más de una vuelta, hay varias soluciones articulares que tienen exactamente la misma configuración: en ese caso se adopta la solución más cercana a la **jInicial**.

No puede haber una solución si **pPosición** está fuera de alcance (brazo demasiado corto) o fuera de los topes de los softwares. Si **pPosición** especifica una configuración, puede estar fuera de los topes para dicha configuración, pero dentro de los topes para otra configuración.

Se genera un error de ejecución si **pPosición** no tiene ningún plano de referencia definido.

Parámetros

tool tHerramienta	Expresión de tipo herramienta
joint jInicial	Expresión de tipo posicíon angular
point pPosición	Expresión de tipo punto
joint& jResult	Variable de tipo posición angular

Véase también

joint herej()
point jointToPoint(tool tHerramienta, frame fMarca de referencia, joint jPosición)

trsf **position**(point pPosición, frame fMarca de referencia)

Sintaxis

trsf position(point pPosición, frame fMarca de referencia)

Función

Reenvía las coordenadas de **pPosición** en **fMarca de referencia**.

Se genera un error de ejeución si **pPosición** no tiene ningún plano de referencia.

Ejemplo

```
//  Muestra la distancia entre dos puntos, cualquiera que sea sus sistemas de referencia
putln(distance(position(pPoint1, world), position(pPoint2, world)))
```

Véase también

num distance(point pPosición1, point pPosición2)

trsf position(tool tHerramienta, tool tMarca de referencia)

trsf position(frame tPlano, frame fMarca de referencia)

8.7. TIPO CONFIG

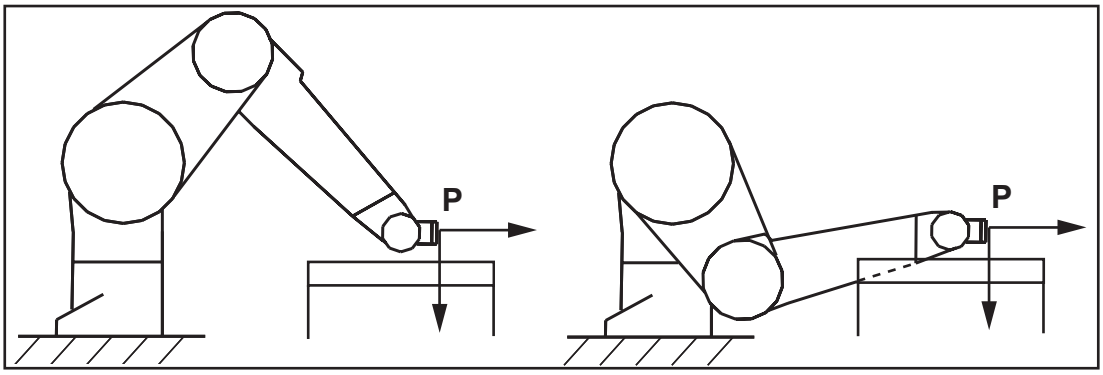
El concepto de configuración del punto cartesiano es un concepto "avanzado"; si se lee este documento por la primera vez, puede saltarse este capítulo.

8.7.1. INTRODUCCIÓN

En general, el robot tiene varias posibilidades para alcanzar una posición cartesiana dada.

Estas diferentes posibilidades se llaman " configuraciones ". En la figura siguiente se representan dos configuraciones diferentes:

Dos configuraciones posibles para alcanzar el mismo punto: P



En algunos casos, es importante especificar, entre todas las configuraciones posibles, las que son válidas y las que se quiere prohibir. Para resolver este problema, el tipo **point** permite especificar las configuraciones admitidas para el robot gracias a su campo de tipo **config** que se define a continuación.

8.7.2. DEFINICIÓN

El tipo **config** permite definir las configuraciones autorizadas para una posición cartesiana dada.

Depende del tipo de brazo utilizado.

Para un brazo **Stäubli RX/TX** el tipo **config** es un tipo estructurado cuyos campos son, en este orden:

shoulder	configuración del hombro
elbow	configuración del codo
wrist	configuración de la muñeca

Para un brazo **Stäubli RS**, el tipo **config** se limita al campo **Shoulder**:

shoulder	configuración del hombro

Los campos **shoulder**, **elbow** y **wrist** pueden tomar los valores siguientes:

shoulder	**righty**	Configuración del hombro **righty** impuesto
	lefty	Configuración del hombro **lefty** impuesto
	ssame	Cambio de configuración del hombro prohibido
	sfree	Configuración del hombro libre

elbow	**epositive**	Configuración del codo **epositive** impuesta
	enegative	Configuración del codo **enegative** impuesta
	esame	Cambio de configuración del codo prohibido
	efree	Configuración del codo libre

wrist	**wpositive**	Configuración de la muñeca **wpositive** impuesta
	wnegative	Configuración de la muñeca **wnegative** impuesta
	wsame	Cambio de configuración de la muñeca prohibido
	wfree	Configuración de la muñeca libre

8.7.3. OPERADORES

Por orden de prioridad creciente:

config <config& configuración1> = <config configuración2>	Asigna los campos **shoulder**, **elbow** y **wrist** de **configuración2** a la variable **configuración1**.
bool <config configuración1> != <config configuración2>	Reenvía **true** si **configuración1** y **configuración2** no tienen el mismo valor de campo **shoulder**, **elbow** o **wrist**.
bool <config configuración1> == <config configuración2>	Reenvía **true** si **configuración1** y **configuración2** tienen los mismos valores de campo **shoulder**, **elbow** y **wrist**.

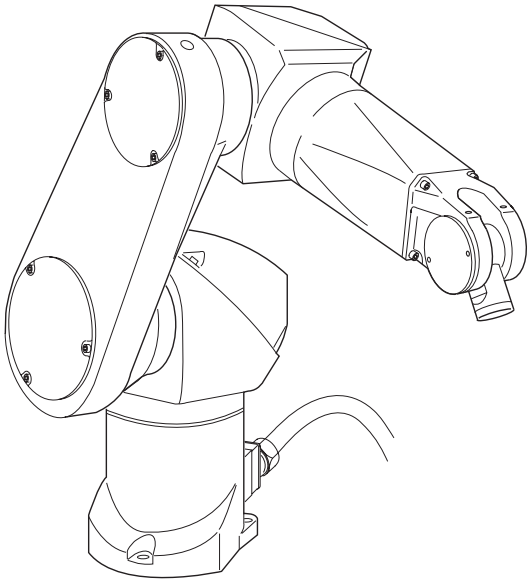
Para evitar confusiones entre los operadores = y ==, el operador = no está autorizado al interior de expresiones **VAL3** utilizadas como parámetro de instrucción.

8.7.4. CONFIGURACIÓN (BRAZO RX/TX)

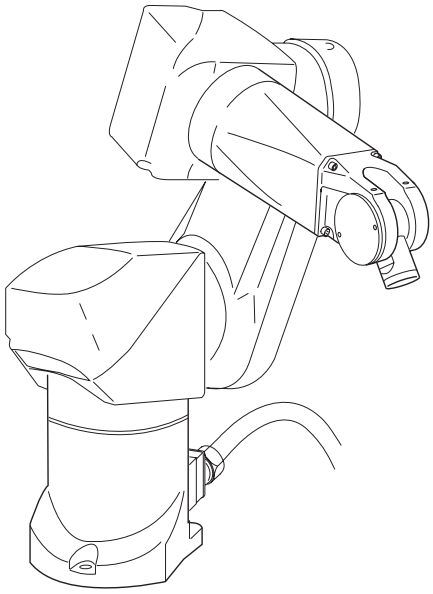
8.7.4.1. CONFIGURACIÓN DEL HOMBRO

Para alcanzar un punto cartesiano, el brazo del robot puede estar a la derecha o a la izquierda del punto: estas dos configuraciones se llaman respectivamente **righty** y **lefty**.

Configuración: righty



Configuración: lefty

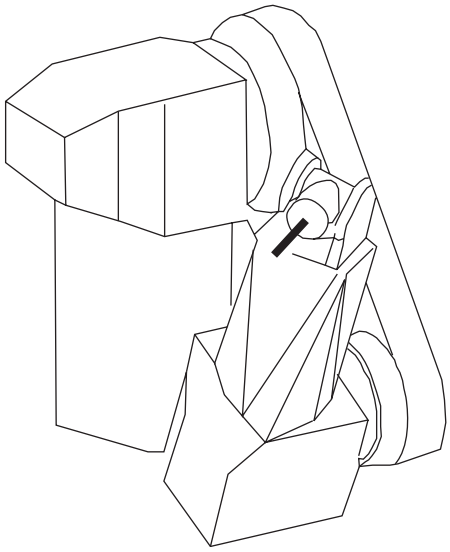


La configuración righty se define mediante (d1 * sin(j2) + d2 * sin(j2+j3) + δ) < 0, la configuración lefty se define mediante (d1 * sin(j2) + d2 * sin(j2+j3) + δ) >= 0, donde d1 es la longitud del brazo del robot, d2 es la longitud del antebrazo, y δ es la distancia entre el eje 1 y el eje 2, en la dirección rightyO.

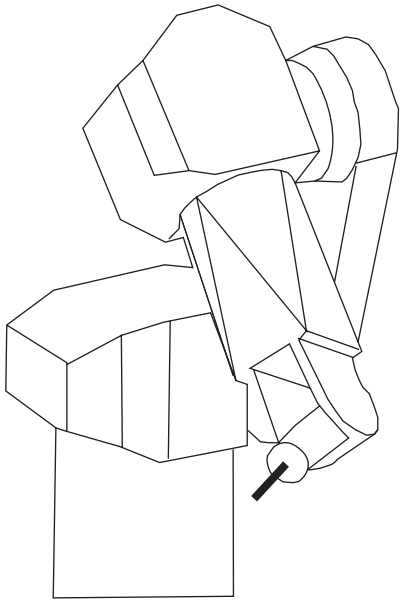
8.7.4.2. CONFIGURACIÓN DEL CODO

Además de la configuración del hombro, hay dos posibilidades para el codo del robot: Las configuraciones del codo se llaman **epositive** y **enegative**.

Configuración: enegative



Configuración: epositive



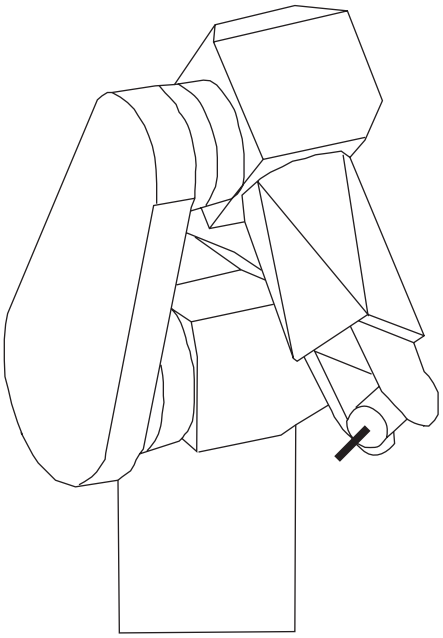
La configuración **epositive** queda definida por **j3 >= 0**.

La configuración **enegative** queda definida por **j3 < 0**.

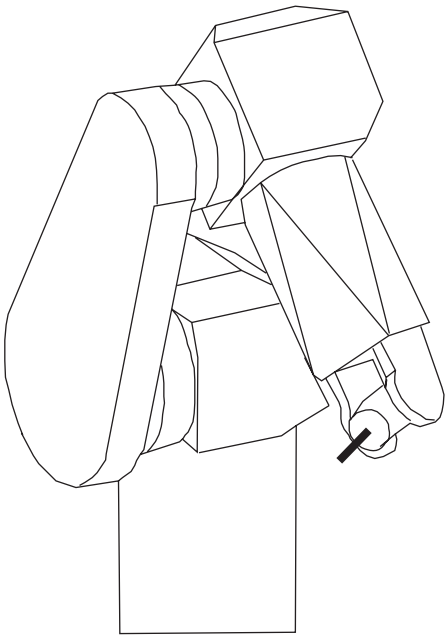
8.7.4.3. CONFIGURACIÓN DE LA MUÑECA

Además de la configuración del hombro y del codo, hay dos posibilidades para posicionar la muñeca del robot. Las dos configuraciones de la muñeca se llaman **wpositive** y **wnegative**.

Configuración: wnegative



Configuración: wpositive



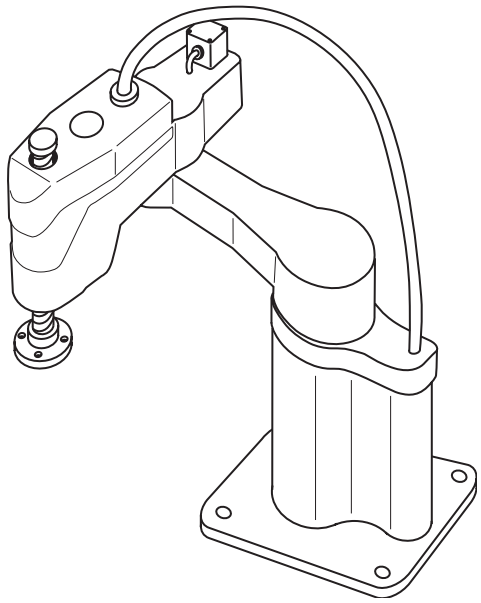
La configuración **wpositive** queda definida por **j5 >= 0**.

La configuración **wnegative** queda definida por **j5 < 0**.

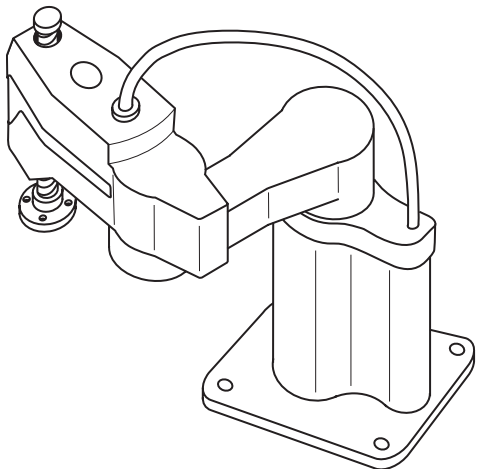
8.7.5. CONFIGURACIÓN (BRAZO RS)

Para alcanzar un punto cartesiano, el brazo del robot puede estar a la derecha o a la izquierda del punto: estas dos configuraciones se llaman respectivamente **righty** y **lefty**.

Configuración: righty



Configuración: lefty



La configuración **righty** se define mediante **sin(j2) > 0**, la configuración **lefty** se define mediante **sin(j2) ≤ 0**.

8.7.6. INSTRUCCIONES

config **config**(joint jPosición)

Sintaxis

config config(joint jPosición)

Función

Retorna la configuración del robot para la posición de articulación **jPosición**.

Parámetros

joint jPosición	Expresión de tipo posicíon angular

Véase también

point here(tool therramienta, frame fMarca de referencia)
joint herej()

CAPÍTULO 9

CONTROL DE LOS MOVIMIENTOS

9.1. CONTROL DE TRAYECTORIA

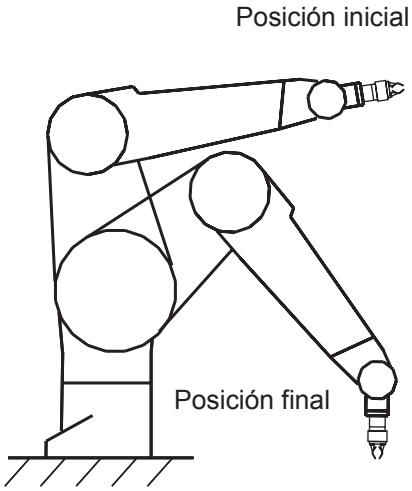
No basta con dar una sucesión de puntos para definir la trayectoria de un robot. Hay también que indicar el tipo de trayectoria utilizado entre los puntos (curva o línea recta), especificar cómo esas trayectorias se conectan entre sí, y por último definir los parámetros relativos a la velocidad del movimiento. Por consiguiente, esta sección presenta los diferentes tipos de movimientos (instrucciones **movej**, **movel** y **movec**) y describe cómo utilizar los parámetros de descriptor (de tipo **mdesc**).

9.1.1. TIPOS DE MOVIMIENTO: PUNTO A PUNTO, LÍNEA RECTA, CÍRCULO

Los movimientos del robot se programan básicamente con las instrucciones **movej, movel** y **movec**. La instrucción **movej** permite hacer movimientos punto a punto, **movel** se utiliza para los movimientos en línea y **movec** en los movimientos circulares.

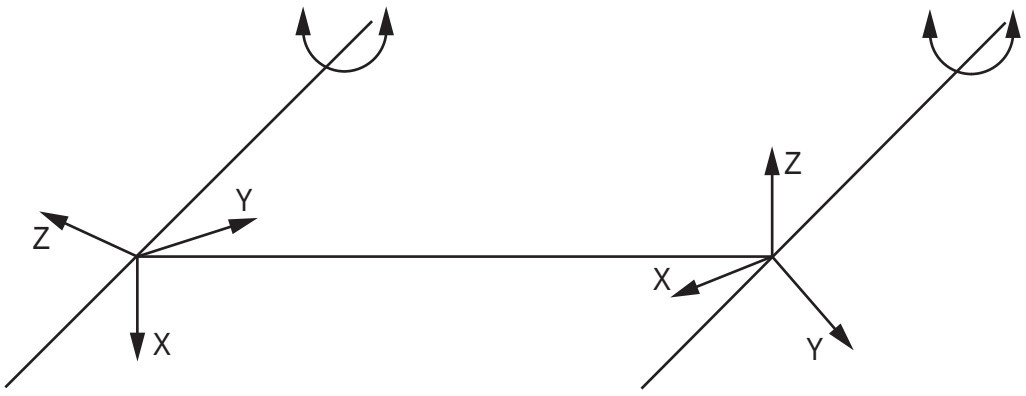
Un movimiento punto a punto es un movimiento donde sólo importa el destino final (punto cartesiano o articular). Entre el punto de partida y el punto de llegada, el centro de la herramienta sigue una curva definida por el sistema de manera que se optimice la velocidad del movimiento.

Posición inicial y final



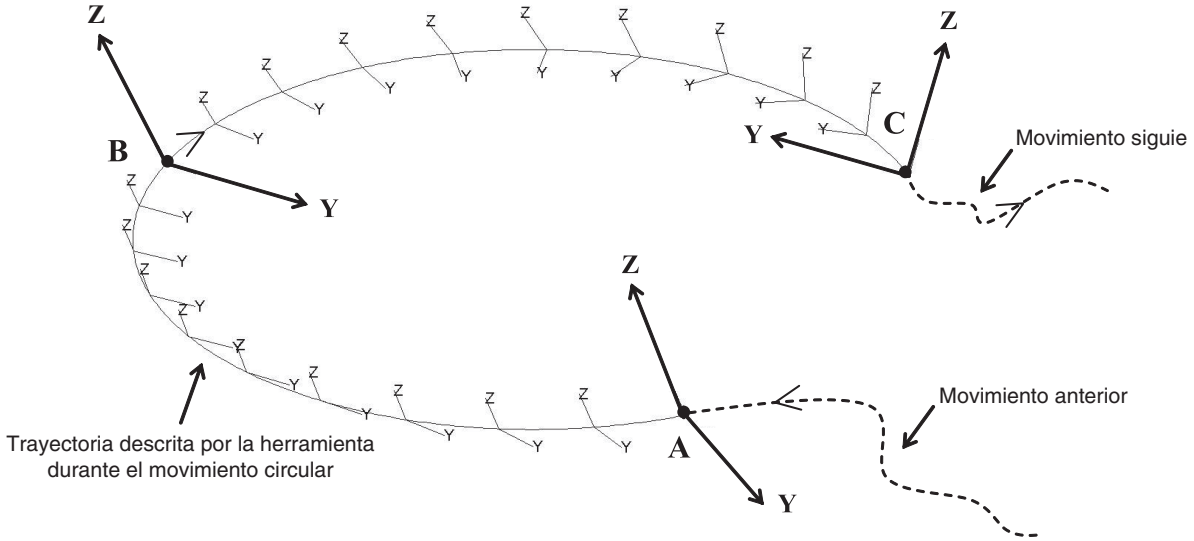
Por el contrario, en un movimiento en línea recta, el centro de la herramienta se desplaza a lo largo de una línea recta. La orientación se interpola linealmente entre la orientación de partida y la orientación final de la herramienta.

Movimiento en línea recta



En un movimiento circular, el centro de la herramienta se desplaza a lo largo de un arco de círculo definido por **3** puntos, y la orientación de la herramienta se interpola entre la orientación de partida, la orientación intermedia y la orientación final.

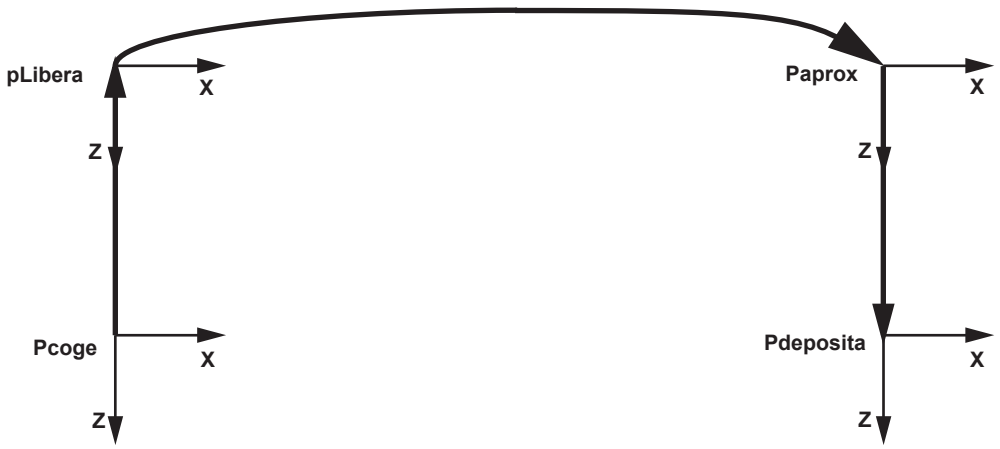
Movimiento circular



Ejemplo:

Una tarea de manipulación típica consiste en coger piezas en un lugar dado y depositarlas en otro lugar. Se supone que deben tomarse las piezas en el punto de toma **pPick**, y depositarlas en el punto de déposito **pPlace**. Para ir del punto **pPick** al punto **pPlace**, el robot debe pasar por un punto de liberación **pDepart** y un punto de aproximación **pAppro**.

Ciclo en: U



Se supone que el robot está inicialmente en el punto **pPick**. El programa para ejecutar el movimiento puede escribirse:

movel(pDepart, tHerramienta, mDesc)
movej(pAppro, tHerramienta, mDesc)
movel(pPlace, tHerramienta, mDesc)

Se utilizan movimientos en línea recta para la liberación y la aproximación. En cambio, el movimiento principal es un movimiento punto a punto ya que no es necesario controlar con precisión la geometría de esta porción de la trayectoria, siendo el objetivo avanzar lo más rápido posible.

Nota:
Para ambos tipos de movimientos, la geometría de la trayectoria no depende de la velocidad a la cual se ejecutan los movimientos. El robot pasa siempre por el mismo sitio. Esto es especialmente importante en el momento del desarrollo de las aplicaciones. Se puede comenzar ejecutando los movimientos a baja velocidad y, a continuación, aumentarla progresivamente sin deformar la trayectoria del robot.

9.1.2. ENCADENAMIENTO DE MOVIMIENTOS

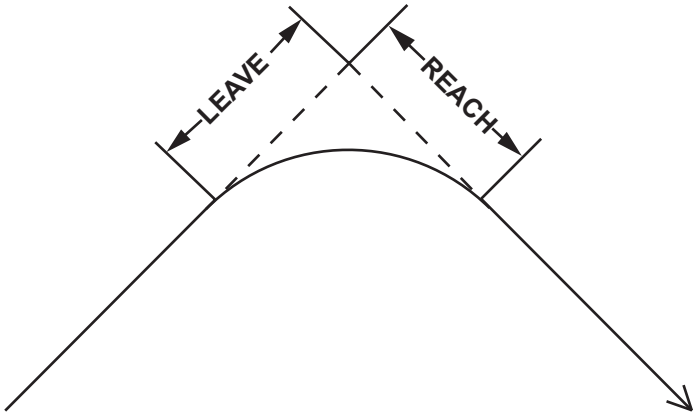
9.1.2.1. ALISADO

Volvamos a examinar el ejemplo de ciclo en **U** del capítulo anterior. Sin una gestión especial del encadenamiento de los movimientos, el robot va a pararse en los puntos **pDepart** y **pAppro**, ya que la trayectoria presenta ángulos en esos puntos. Esto aumenta inútilmente la duración de la operación, y no presenta ninguna utilidad pasar exactamente por esos puntos.

Es posible reducir considerablemente la duración del movimiento "alisando" la trayectoria en las cercanías de los puntos **pDepart** y **pAppro**. Para esto, se utiliza el campo **blend** del descriptor de movimiento. Cuando el valor de este campo es **off**, la trayectoria del robot se para en cada punto. En cambio, cuando este parámetro se pone en **joint**, la trayectoria se alisa en las cercanías de cada uno de los puntos y el robot no se para ya en los puntos de paso.

Cuando el campo **blend** toma el valor **joint**, deben especificarse otros dos parámetros: **leave** y **reach**. Estos parámetros determinan a qué distancia del punto de llegada se abandona la trayectoria nominal (inicio del alisado), y a qué distancia del punto de llegada se vuelve unirse a ella (fin del alisado).

Definición de las distancias: 'leave' / 'reach'



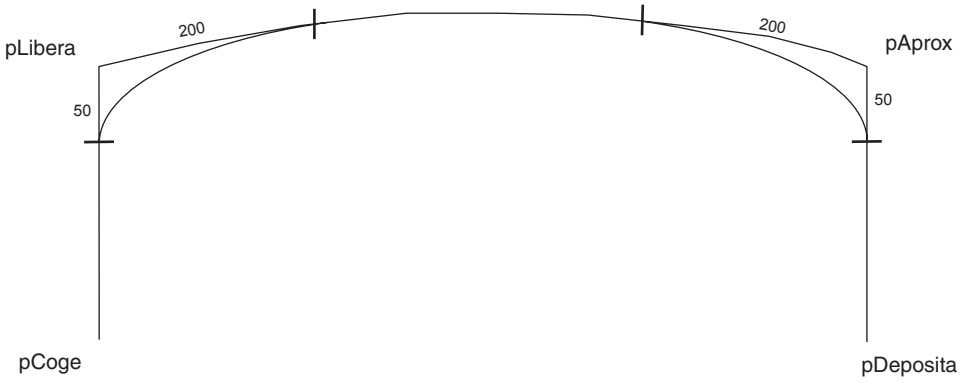
Ejemplo:

Volvamos a examinar el programa del apartado "Tipos de movimiento: punto a punto o línea recta". Es posible modificar el programa de movimiento anterior:

```
mDesc.blend = joint
mDesc.leave = 50
mDesc.reach = 200
movel(pDepart, tHerramienta, mDesc)
mDesc.leave = 200
mDesc.reach = 50
movej(pAppro, tHerramienta, mDesc)
mDesc.blend = OFF
movel(pPlace, tHerramienta, mDesc)
```

En ese caso se obtiene la trayectoria siguiente:

Ciclo alisado



El robot ya no se para en los puntos **pDepart** y **pAppro**. El movimiento resulta así más rápido. Y no sólo eso, sino que será tanto más rápido cuanto más grandes sean las distancias **leave** y **reach**.

9.1.2.2. ANULACIÓN DEL ALISADO

La instrucción **waitEndMove()** permite, entre otras cosas, anular el efecto del alisado. El robot termina entonces completamente el último movimiento programado hasta su punto de llegada, como si el campo **blend** del descriptor de movimiento hubiera sido puesto en el valor **off**.

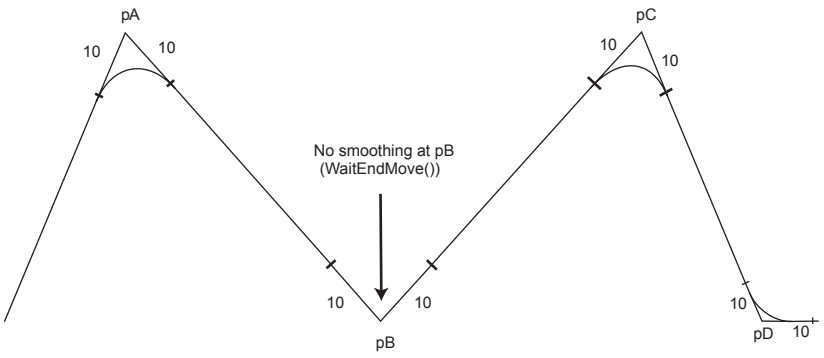
Por ejemplo, consideremos el programa siguiente:

```
mDesc.blend = joint
mDesc.leave = 10
mDesc.reach = 10
movej(pA, tHerramienta, mDesc)
movej(pB, tHerramienta, mDesc)
waitEndMove()
movej(pC, tHerramienta, mDesc)
movej(pD, tHerramienta, mDesc)
```

etc.

La trayectoria seguida por el robot es en este caso la siguiente:

Ciclo con interrupción del alisado



9.1.3. REANUDACIÓN DE MOVIMIENTO

Cuando la potencia del brazo se corta mientras que el robot no ha terminado sus movimientos, debido por ejemplo a una parada de emergencia, debe hacerse una reanudación de movimiento al volverse a aplicar la potencia. Si se ha desplazado manualmente el brazo durante la parada, puede encontrarse en una posición alejada de su trayectoria normal. En ese caso hay que asegurarse que la reanudación de movimiento pueda hacerse sin colisión. El control de trayectoria del **VAL3** proporciona la posibilidad de gestionar la reanudación de movimiento por medio de un "movimiento de conexión".

Al reanudarse el movimiento, el sistema se asegura que el robot esté efectivamente en la trayectoria programada: en caso de desvío, aun mínimo, registra automáticamente un comando de movimiento punto a punto hacia la posición exacta en la que el robot ha abandonado su trayectoria: es el llamado "movimiento de conexión". Este movimiento se hace a velocidad reducida. Debe ser validado por el operador, salvo en el modo automático, en el que puede hacerse sin intervención humana. La instrucción **autoConnectMove()** permite precisar el comportamiento en modo automático.

La instrucción **resetMotion()** hace posible anular el movimiento en curso, y eventualmente programar un movimiento de conexión que permita alcanzar una posición a velocidad reducida y bajo el control del operador.

9.1.4. PARTICULARIDADES DE LOS MOVIMIENTOS CARTESIANOS (LÍNEA RECTA, CÍRCULO)

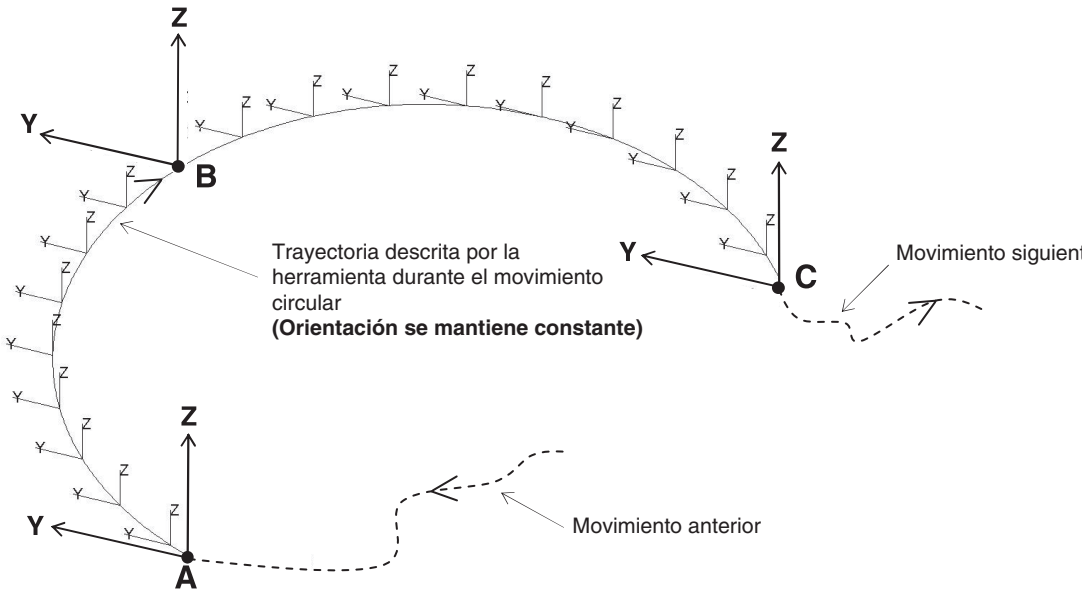
9.1.4.1. INTERPOLACIÓN DE LA ORIENTACIÓN

El generador de trayectoria del **VAL3** minimiza siempre la amplitud de las rotaciones de la herramienta para pasar de una orientación a otra.

Esto permite, como caso particular, programar una orientación constante, o respecto a la trayectoria, en todos los movimientos en línea recta o circulares.

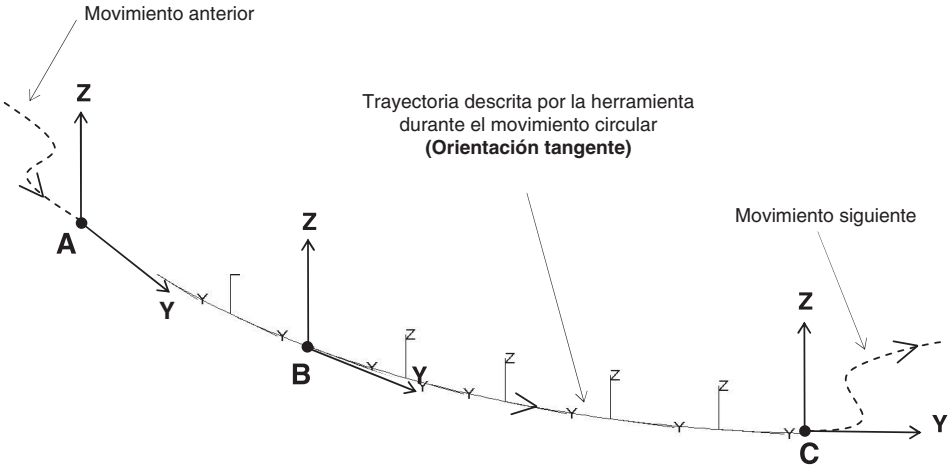
- Para obtener una orientación constante, las posiciones de partida, de llegada y la posición intermedia para un círculo, deben tener la misma orientación.

Orientación constante en absoluto



- Para una orientación constante respecto a la trayectoria (por ejemplo, dirección **Y** del punto de referencia herramienta tangente a la trayectoria), las posiciones de partida, de llegada y la posición intermedia para un círculo, deben tener la misma orientación respecto a la trayectoria.

Orientación constante respecto a la trayectoria

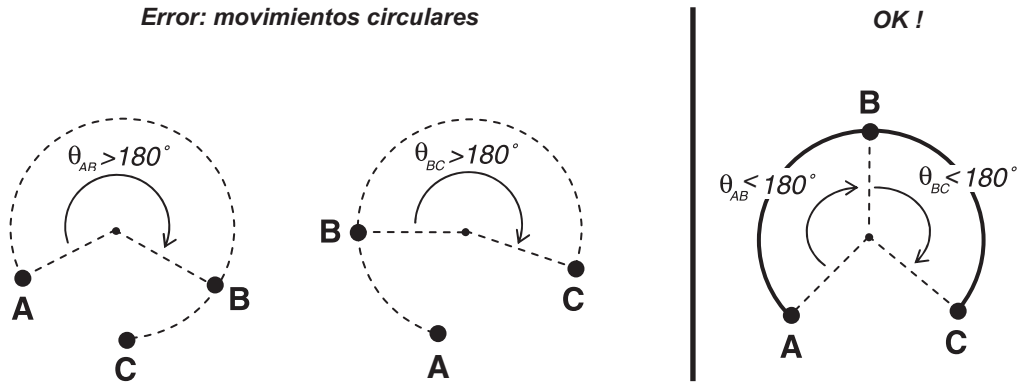


Como resultado se tiene una limitación para los movimientos circulares:

Si el punto intermedio hace un ángulo de **180°** o más con el punto de partida o el punto de llegada, hay varias soluciones de interpolación de la orientación, y se genera un error.

Entonces hay que modificar la posión del punto intermedio para disipar la ambigüedad sobre las orientaciones intermedias.

Ambigüedad sobre la orientación intermedia

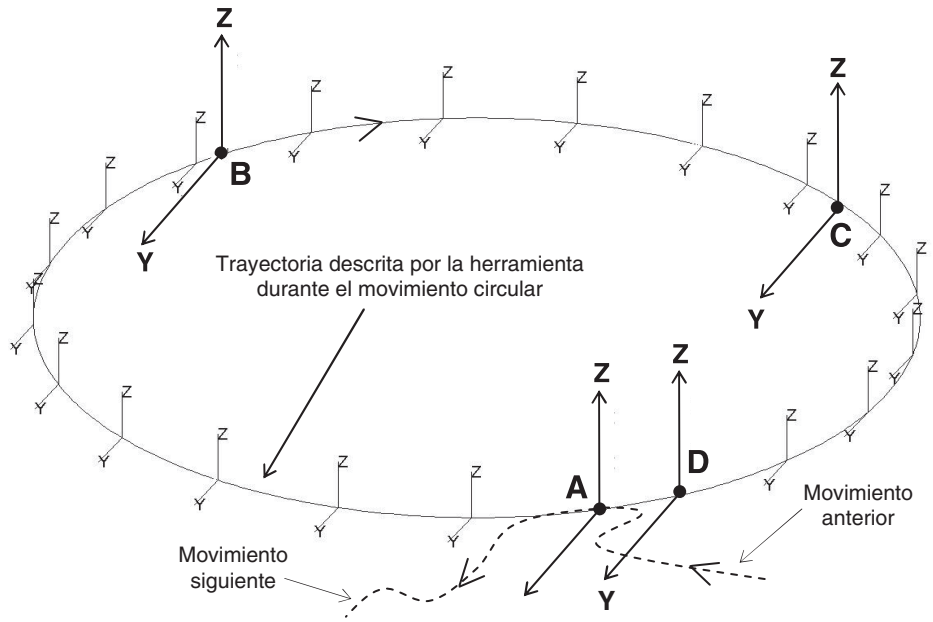


En particular, la programación de un círculo completo requiere **2** instrucciones **movec**:

movec (B, C, Herramienta, mDesc)

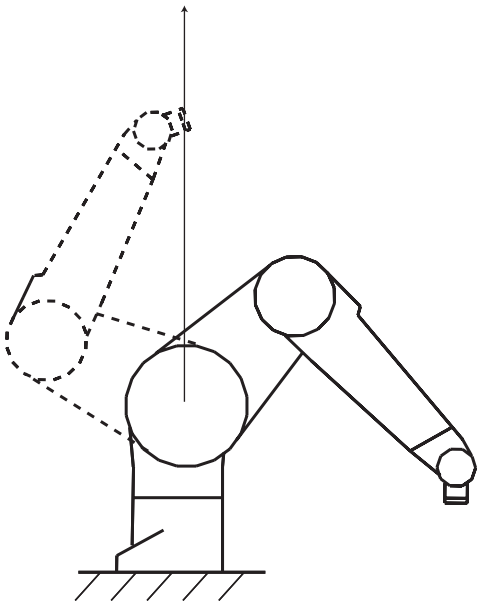
movec (D, A, Herramienta, mDesc)

Círculo completo



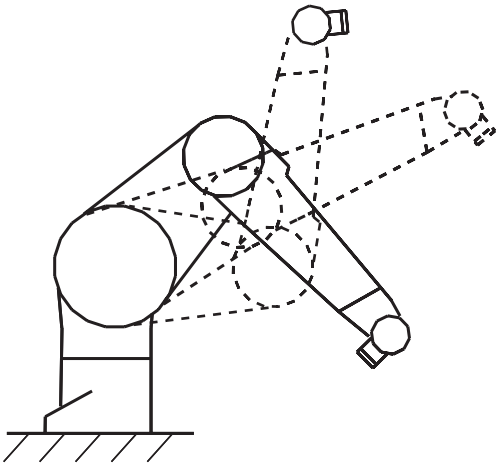
9.1.4.2. CAMBIO DE CONFIGURACIÓN (BRAZO RX/TX)

Cambio: righty / lefty



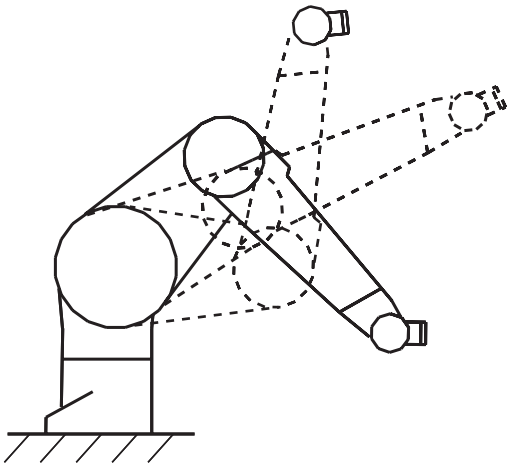
Cuando se produce un cambio de la configuración del hombro, el centro de la muñeca del robot pasa necesariamente por la vertical del eje **1** (no exactamente en el caso de robots con desfase).

Cambio positive/negative del codo



En el caso de un cambio de la configuración del codo, el brazo pasa necesariamente por la posición brazo tenso **(j3 = 0°)**.

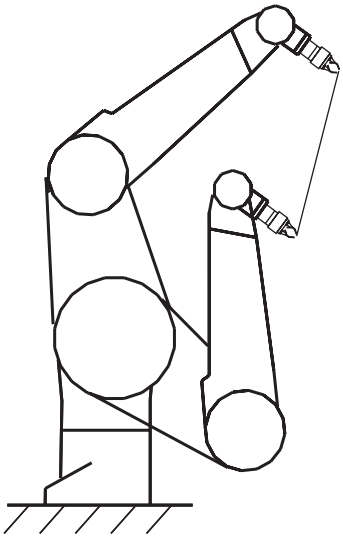
Cambio positive/negative de la muñeca



En el caso de un cambio de la configuración de la muñeca, el brazo pasa necesariamente por la posición muñeca tensa **(j5 = 0°)**.

Así, para cambiar de configuración, el robot debe obligatoriamente pasar por posiciones particulares. ¡Pero no puede imponerse que un movimiento en línea recta o circular pase por estas posiciones si éstas no se encuentran en la trayectoria deseada! En consecuencia, **no puede imponerse un cambio de configuración en un movimiento en línea recta o circular**.

Cambio de configuración del codo imposible

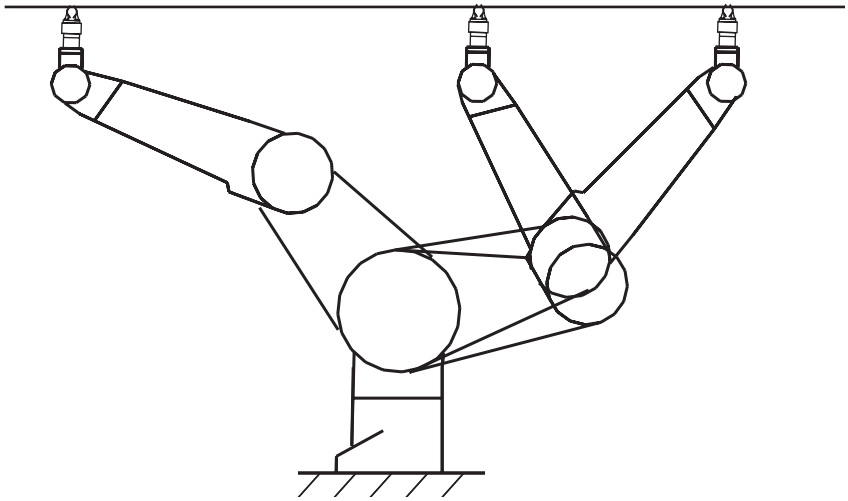


Dicho de otro modo, en un movimiento en línea recta o circular, sólo puede imponerse una configuración si esta es compatible con la posición de partida: entonces siempre puede especificarse una configuración libre, o bien idéntica a la inicial.

En ciertos casos excepcionales, la línea recta o el arco de círculo pasa efectivamente por una posición donde es posible un cambio de configuración. En este caso, si la configuración ha sido dejada libre, el sistema puede decidir cambiar de configuración en un movimiento en línea recta o circular.

Para un movimiento circular, la configuración del punto intermedio no se toma en cuenta. Sólo cuentan las configuraciones de las posiciones de partida y llegada.

Cambio de configuración del hombro posible



9.1.4.3. SINGULARIDADES (BRAZO RX/TX)

Las singularidaridades son una característica inherente a todos los robots de **6** ejes. Las singularidades pueden definirse como los puntos en los cuales el robot cambia de configuración. Entonces, ciertos ejes se encuentran alineados: dos ejes alineados se comportan como un solo eje, y el robot de **6** ejes se comporta entonces localmente como un robot de **5** ejes. Ciertos movimientos del actuador son entonces imposibles de realizar. Esto no es molesto en un movimiento punto a punto: los movimientos generados por el sistema son siempre posibles. En cambio, en un movimiento en línea recta o circular, se impone la geometría del movimiento. Si el movimiento es imposible, se genera un error al poner en movimiento el robot.

9.2. ANTICIPACIÓN DE LOS MOVIMIENTOS

9.2.1. PRINCIPIO

El sistema controla los movimientos del robot un poco como un piloto que conduce un coche. Por un lado, adapta la velocidad del robot a la geometría de la trayectoria. Por otra parte, cuanto mejor se conoce la trayectoria de antemano, más efectivo puede mostrarse el sistema para optimizar la velocidad del movimiento. Es por esta razón que el sistema no espera que el movimiento en curso del robot se haya terminado para tomar en cuenta las próximas instrucciones de movimiento.

Consideremos las siguientes líneas de programa:

```
movej (pA, tHerramienta, mDesc)
movej (pB, tHerramienta, mDesc)
movej (pC, tHerramienta, mDesc)
movej (pD, tHerramienta, mDesc)
```

Supongamos que el robot está parado cuando la ejecución del programa alcanza estas líneas. Cuando se ejecuta la primera instrucción, el robot comienza su movimiento hacia el punto **pA**. La ejecución del programa prosigue entonces inmediatamente con la segunda línea, mucho antes que el robot alcance el punto **pA**.

Cuando el sistema ejecuta la segunda línea, el robot está comenzando su movimiento hacia **pA** y el sistema registra que, después del punto **pA**, el robot deberá ir al punto **pB**. La ejecución del programa continúa entonces con la línea siguiente: mientras que el robot continúa su movimiento hacia **pA**, el sistema registra que, después del movimiento hacia **pB**, el robot deberá dirigirse a **pC**. Puesto que el programa se ejecuta mucho más rápido que los movimientos reales del robot, al ejecutar la línea siguiente, el robot probablemente continúa desplazándose hacia **pA**. El sistema registra de esta forma los puntos sucesivos siguientes.

Así, en el momento en que el robot comienza su movimiento hacia **pA**, ya 'sabe' que, después de **pA** deberá ir sucesivamente a **pB**, **pC** y finalmente **pD**. Si el alisado ha sido activado, el sistema sabe que el robot no se parará antes del punto **pD**. Puede entonces acelerar mucho más que si debiera a prepararse a pararse en **pB** o **pC**.

La ejecución de las líneas de comando no hace sino registrar los comandos de movimiento sucesivos. El robot los efectúa a continuación según sus posibilidades. La memoria en la que se registran los movimientos es importante, con objeto de permitir que el sistema optimice la trayectoria al máximo. No obstante, es limitada. Cuando está llena, la ejecución del programa se para en la próxima instrucción de movimiento. La ejecución del programa se reanuda en cuanto el robot ha terminado el movimiento en curso, liberando así espacio en la memoria del sistema.

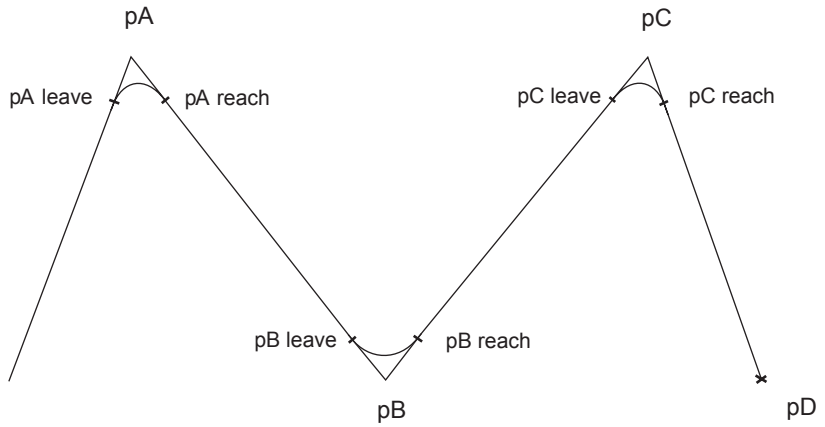
9.2.2. ANTICIPACIÓN Y ALISADO

Volvemos en este apartado a lo que sucede en detalle cuando se encadenan los movimientos. Tomemos el ejemplo anterior:

```
movej (pA, tHerramienta, mDesc)
movej (pB, tHerramienta, mDesc)
movej (pC, tHerramienta, mDesc)
movej (pD, tHerramienta, mDesc)
```

Se supone que el alisado está activado en el descriptor de movimiento **mDesc**. Cuando se ejecuta la primera línea, el sistema no sabe todavía cuál será el movimiento siguiente. Sólo el movimiento entre el punto inicial y el punto **pA leave** está completamente determinado; estando el punto **pA leave** determinado por el sistema a partir del dato **leave** del descriptor de movimientos (véase figura siguiente).

Ciclo alisado



Mientras que no haya sido ejecutada la segunda línea, la porción de trayectoria de alisado cercana al punto **pA** no está completamente determinada, ya que el sistema no ha tomado en cuenta todavía el movimiento siguiente. En el modo paso a paso, mientras que el usuario no pase a la instrucción siguiente, el robot no irá más lejos que el punto **pA leave**. En el momento en que se ejecuta la instrucción siguiente, la trayectoria de alisado cercana al punto **pA** (entre **pA leave** y **pA reach**) puede ser definida, así como el movimiento hasta el punto **pB leave**. El robot puede entonces avanzar hasta el punto **pB leave**. En el modo paso a paso, no excederá este punto mientras que el usuario no ejecute la tercera instrucción, y así sucesivamente.

Este modo de funcionamiento es interesante porque el robot pasa exactamente por el mismo sitio en el modo paso a paso y en ejecución normal del programa.

9.2.3. SINCRONIZACIÓN

El mecanismo de anticipación causa una desincronización entre las líneas de instrucciones **VAL3** y los movimientos correspondientes del robot: el programa **VAL3** está adelantado respecto al robot.

Cuando se quiere efectuar una acción en una posición dada del robot, es necesario que el programa espere que el robot haya terminado sus movimientos: la instrucción **waitEndMove()** permite esta sincronización.

Así, en el programa siguiente:
movej(A, tool, mDesc)
movej(B, tool, mDesc)
waitEndMove()
movej(C, tool, mDesc)
movej(D, tool, mDesc)
etc.
Las dos primeras líneas serán ejecutadas mientras que el robot comienza su movimiento hacia **A**. A continuación, la ejecución del programa será bloqueada en la tercera línea hasta que el robot se haya estabilizado en el punto **B**. Una vez estabilizado el movimiento del robot en **B**, el programa reanudará su ejecución.

Las instrucciones **open()** y **close()** esperan asimismo el final de los movimientos del robot antes de accionar la herramienta.

9.3. CONTROL DE VELOCIDAD

9.3.1. PRINCIPIO

El principio del control de velocidad en una trayectoria es el siguiente:
En cada momento, el robot se desplaza y acelera al máximo de sus capacidades, sin dejar de respetar las restricciones de velocidad y aceleración proporcionadas en el comando de movimiento.
Los comandos de movimiento contienen dos tipos de restricciones de velocidad, definidas en una variable de tipo **mdesc**:

1. Las limitaciones de velocidad (velocidades de articulación), aceleración y deceleración
2. las restricciones de velocidades cartesianas del centro de la herramienta

La aceleración determina la rapidez con la cual la velocidad aumenta al comienzo de una trayectoria. Inversamente, la deceleración define la rapidez con la cual la velocidad disminuye al final de la trayectoria. Cuando se utilizan grandes valores de aceleración y deceleración, los movimientos son más rápidos, pero más bruscos. Con valores reducidos, los movimientos toman un poco más de tiempo, pero se hacen con mayor suavidad.

9.3.2. AJUSTE SENCILLO

Cuando la herramienta y el objeto transportados por el robot no requieren precauciones especiales de manipulación, las restricciones sobre las velocidades cartesianas son inútiles. La puesta a punto de la velocidad en la trayectoria se hará típicamente de la forma siguiente:

1. Ponga las restricciones de velocidad cartesiana en valores muy grandes, por ejemplo los valores predeterminados, para que estas no intervengan en el resto del ajuste.
2. Inicializa la velocidad, aceleración y deceleración, utilizando los valores nominales **(100%)**.
3. Luego ajusta la velocidad a lo largo de la trayectoria, utilizando el parámetro velocidad.
4. Si no llega a alcanzarse una velocidad suficiente, aumente los parámetros de aceleración y deceleración

9.3.3. AJUSTE AVANZADO

Cuando se quiere controlar la velocidad cartesiana a nivel de la herramienta, por ejemplo para obtener una trayectoria efectuada a velocidad constante, se procederá más bien de la manera siguiente:

1. Ponga las restricciones de velocidades cartesianas en los valores que a priori se deseen.
2. Inicializa la velocidad, aceleración y deceleración, utilizando los valores nominales **(100%)**.
3. Luego ajusta la velocidad a lo largo de la trayectoria, utilizando únicamente los parámetros de velocidad cartesiana.
4. Si no llega a alcanzarse la velocidad deseada, aumente los parámetros de aceleración y deceleración.
   Si se desea frenar automáticamente en las partes de fuerte curvatura, disminuya los parámetros de aceleración y deceleración.

9.3.4. ERROR DE ARRASTRE

Los valores nominales para velocidad y aceleración de articulación son los valores de carga nominales soportados por el robot, al margen de la trayectoria.

Pero el robot puede con frecuencia ir más rápido: las velocidades máximas alcanzables por el robot dependen de su carga y de la trayectoria. En casos favorables (carga reducida, impacto favorable de la gravedad) el robot podrá exceder sus valores nominales sin ningún perjuicio.

Si el robot transporta una carga superior a su carga nominal, o si los valores de velocidad y aceleración de articulación son demasiado elevados, el robot no puede obedecer a sus mandos de movimiento y se detiene cuando se produce un error de envoltura. Estos errores pueden evitarse, especificando parámetros menores de velocidades y aceleración.

ATENCIÓN:
En el caso de movimientos en línea recta cerca a una singularidad, el movimiento de una herramienta pequeña requiere amplios movimientos de articulación. Si la velocidad se configura demasiado elevada, el robot no puede obedecer el mando y se detiene, cuando se produce un error de envoltura.

9.4. CONTROL DEL MOVIMIENTO EN TIEMPO REAL

Los comandos de movimiento que se han abordado hasta ahora no tienen un efecto inmediato: en el momento en que cada uno de ellos se ejecuta, el sistema registra una orden de movimiento. El robot ejecuta a continuación los movimientos registrados.

Es posible intervenir inmediatamente en el movimiento del robot de las siguientes maneras:

- La velocidad del monitor modifica la velocidad de todos los desplazamientos. Esta puede ajustarse únicamente mediante el mando manual del robot, y no en una aplicación **VAL3**. Sin embargo, la instrucción **speedScale()** permite que una aplicación conozca la velocidad del monitor corriente y, por lo tanto, si fuera necesario, solicitar al usuario su reducción durante una reanudación de ciclo, o ponerla al **100%** en producción.
- Las instrucciones **stopMove()** y **restartMove()** permiten la parada y la reanudación de movimiento en la trayectoria.
- La instrucción **resetMotion()** permite parar el movimiento en curso y anular los comandos de movimientos registrados.
- La instrucción Alter (opción) se refiere a la trayectoria de una transformación geométrica (translación, rotación, rotación en el punto del centro de herramienta) que es inmediatamente efectiva.

9.5. TIPO MDESC

9.5.1. DEFINICIÓN

El tipo **mdesc** permite definir los parámetros de un movimiento (velocidad, aceleración, alisado).

El tipo **mdesc** es un tipo estructurado cuyos campos, presentados en orden, son:

num accel	Aceleración de articulación máxima permitida, como **%** de la aceleración nominal del robot.
num vel	Velocidad de articulación máxima permitida como **%** de la velocidad nominal del robot.
num decel	Deceleración de articulación máxima permitida como **%** de la deceleración nominal del robot.
num tvel	Velocidad máxima autorizada de traslación del centro de la herramienta, en mm/s o pulgada/s, según la unidad de longitud de la aplicación.
num rvel	Velocidad máxima autorizada de rotación de la herramienta, en grados por segundo.
blend **blend**	Modo de alisado: **off** (sin alisado), o **joint** (alisado).
num leave	En el modo de alisado **joint**, la distancia al punto objetivo donde comienza el alisado hacia el punto siguiente, en mm o pulgadas según la unidad de longitud de la aplicación.
num reach	En el modo de alisado **joint**, la distancia al punto objetivo donde se termina el alisado hacia el punto siguiente, en mm o pulgadas según la unidad de longitud de la aplicación.

La explicación detallada de estos diferentes parámetros se da al comienzo del capítulo "Control de los movimientos".

Por defecto, una variable de tipo **mdesc** se inicializa en **{100,100,100,9999,9999,off,50,50}**.

9.5.2. OPERADORES

Por orden de prioridad creciente:

mdesc <mdesc& desc1> = <mdesc desc2>	Asigna cada campo de **desc2** al campo correspondiente de la variable **desc1**.
bool <mdesc desc1> != <mdesc desc2>	Reenvía **true** si **desc1** y **desc2** difieren por el valor de al menos un campo.
bool <mdesc desc1> == <mdesc desc2>	Reenvía **true** si **desc1** y **desc2** tienen los mismos valores de campos.

9.6. INSTRUCCIONES DE MOVIMIENTO

void **movej**(joint jPosición, tool tHerramienta, mdesc mDesc)

void **movej**(point pPosición, tool tHerramienta, mdesc mDesc)

Sintaxis

void movej(joint jPosición, tool tHerramienta, mdesc mDesc)
void movej(point pPosición, tool tHerramienta, mdesc mDesc)

Función

Registra un mando para el movimiento de una articulación hacia las posiciones **pPosición** o **jPosición**, utilizando los parámetros de movimiento **tHerramienta** y **mDesc**.

> **ATENCIÓN:**
> **El sistema no espera el final del movimiento para pasar a la siguiente instrucción VAL3: varios comandos de movimientos pueden haber sido registrados de antemano. Cuando al sistema no le queda más memoria disponible para registrar un nuevo comando, la instrucción espera hasta que pueda hacerse el registro.**

La explicación detallada de los diferentes parámetros de movimiento se da al comienzo del capítulo "Control de los movimientos".

Se genera un error de ejecución si **mDesc** contiene valores inválidos, si **jPosición** se sitúa fuera de los topes de software, si **pPosición** no puede alcanzarse o si una solicitud de movimiento anteriormente registrada no puede realizarse (posición fuera de alcance).

Parámetros

tool tHerramienta	Expresión de tipo herramienta
mdesc mDesc	Expresión de tipo descriptor de movimiento
joint jPosición	Expresión de tipo campo
point pPosición	Expresión de tipo punto

Véase también

void movel(point pPosición, tool tHerramienta, mdesc mDesc)
bool isInRange(joint jposición)
void waitEndMove()
void movec(point pIntermediario, point pObjetivo, tool tHerramienta, mdesc mDesc)

void **movel**(point pPosición, tool tHerramienta, mdesc mDesc)

Sintaxis

void movel(point pPosición, tool tHerramienta, mdesc mDesc)

Función

Registra un mando para un movimiento lineal hacia el punto **pPosición**, utilizando la herramienta **tHerramienta** y los parámetros de movimiento **mDesc**.

> **ATENCIÓN:**
> **El sistema no espera el final del movimiento para pasar a la siguiente instrucción VAL3: varios comandos de movimientos pueden haber sido registrados de antemano. Cuando al sistema no le queda más memoria disponible para registrar un nuevo comando, la instrucción espera hasta que pueda hacerse el registro.**

La explicación detallada de los diferentes parámetros de movimiento se da al comienzo del capítulo "Control de los movimientos".

Se genera un error de ejecución si **mDesc** contiene valores inválidos, si **pPosición** no puede alcanzarse, si un movimiento en línea recta hacia **pPosición** no es posible o si una solicitud de movimiento anteriormente registrada no puede realizarse (destino fuera de alcance).

Parámetros

point pPosición	Expresión de tipo punto.
tool tHerramienta	Expresión de tipo herramienta.
mdesc mDesc	Expresión de tipo descriptor de movimiento.

Véase también

void movej(joint jPosición, tool tHerramienta, mdesc mDesc)
void waitEndMove()
void movec(point pIntermediario, point pObjetivo, tool tHerramienta, mdesc mDesc)

void **movec**(point pIntermediario, point pObjetivo, tool tHerramienta, mdesc **mDesc**)

Sintaxis

void movec(point pIntermediario, point pObjetivo, tool tHerramienta, mdesc mDesc)

Función

Graba un comando de movimiento circular partiendo del destino del movimiento anterior haste el point **pObjetivo** y pasando por el point **pIntermediario**.

La orientación de la herramienta se interpola de modo que es posible programar una orientación constante en absoluto, o respecto a la trayectoria.

> **ATENCIÓN:**
> **El sistema no espera el final del movimiento para pasar a la siguiente instrucción VAL3: varios comandos de movimientos pueden haber sido registrados de antemano. Cuando al sistema no le queda más memoria disponible para registrar un nuevo comando, la instrucción espera hasta que pueda hacerse el registro.**

La explicación detallada de los diferentes parámetros de movimiento e interpolación de la orientación se proporciona al comienzo del capítulo "Control de los movimientos".

Se genera un error de ejecución si **mDesc** tiene valores inválidos, si point **pIntermediario** (o point **pObjetivo**) no puede alcanzarse, si el movimiento circular no es posible (ver el capítulo "Control de los movimientos - Interpolación de la orientación") o si una solicitud de movimiento anteriormente registrada no puede realizarse (destino fuera de alcance).

Parámetros

point pIntermediario	Expresión de tipo punto.
point pObjetivo	Expresión de tipo punto.
tool tHerramienta	Expresión de tipo herramienta.
mdesc mDesc	Expresión de tipo descriptor de movimiento.

Véase también

void movej(joint jPosición, tool tHerramienta, mdesc mDesc)
void movel(point pPosición, tool tHerramienta, mdesc mDesc)
void waitEndMove()

void **stopMove**()

Sintaxis

void stopMove()

Función

Para el brazo en la trayectoria y suspende la autorización de movimiento programada.

> **ATENCIÓN:**
> **Esta instrucción devuelve inmediatamente: la tarea VAL3 no espera la parada del brazo para ejecutar la instrucción siguiente.**

Los parámetros cinemáticos utilizados para efectuar la parada son aquellos del movimiento en curso.

Los movimientos no pueden reanudarse sino después de ejecutar una instrucción **restartMove()** o **resetMotion()**.

Los movimientos no programados (desplazamientos manuales) son posibles.

Ejemplo

```
wait (diSignal==true)          // espera de una señal
stopMove()                     // parada de los movimientos en la trayectoria
<instructions>
restartMove()                  // reanudación de los movimientos en la trayectoria
```

Véase también

void restartMove()
void resetMotion(), void resetMotion(joint jpartida)

void **resetMotion**(), void **resetMotion**(joint jpartida)

Sintaxis

void resetMotion()
void resetMotion(joint jpartida)

Función

Para el brazo en la trayectoria y anula todos los comandos de movimientos registrados.

> **ATENCIÓN:**
> **Esta instrucción devuelve inmediatamente: la tarea VAL3 no espera la parada del brazo para ejecutar la instrucción siguiente.**

La autorización de movimiento programado puede restaurarse si había sido suspendida por la instrucción **stopMove()**.

Si se especifica la posición articular **jpartida**, el próximo comando de movimiento sólo podrá ejecutarse a partir de esta posición: deberá efectuarse previamente un movimiento de conexión para alcanzar **jpartida**.

Si no se especifica ninguna posición articular, el próximo comando de movimiento se ejecutará a partir de la posición en curso del brazo, fuera la que fuera.

Véase también

bool isEmpty()
void stopMove()
void autoConnectMove(bool bActivo), bool autoConnectMove()

void **restartMove**()

Sintaxis

void restartMove()

Función

Restaura la autorización de movimiento programado, y reanuda la trayectoria interrumpida por la instrucción **stopMove()**.

Si la autorización de movimiento programado no ha sido interrumpida por la instrucción **stopMove()**, esta instrucción no tiene ningún efecto.

Véase también

void stopMove()
void resetMotion(), void resetMotion(joint jpartida)

void **waitEndMove**()

Sintaxis

void waitEndMove()

Función

Anula el alisado del último comando de movimiento registrado, y espera que este comando sea ejecutado.

Esta instrucción no espera que el robot se estabilice en su posición final, solamente que la consigna de la posición enviada a los variadores corresponda a la posición final deseada. Cuando la espera de estabilización completa del movimiento es necesaria, se debe utilizar la instrucción **isSettled()**.

Se genera un error de ejecución si un movimiento registrado anteriormente no puede realizarse (destino fuera de alcance).

Ejemplo

```
waitEndMove()
putln(sel(isEmpty(),1,-1))       // Muestra 1, no hay ya comandos en curso
putln(sel(isSettled(),1,-1))     // Puede mostrar -1, el robot no está ya necesariamente estabilizado
watch(isSettled(), 1)            // Espera la estabilización del robot 1 s segundo como máximo
```

Véase también

bool isSettled()
bool isEmpty()
void stopMove()
void resetMotion(), void resetMotion(joint jpartida)

bool **isEmpty**()

Sintaxis
bool isEmpty()

Función
Reenvía **true** si todos los comandos de movimiento han sido ejecutados, **false** si queda por lo menos un comando en curso.

Ejemplo
```
//  Si hay comandos en curso
if ! isEmpty()
  // Parar el robot y anular los comandos
  resetMotion()
endIf
```

Véase también
void waitEndMove()
void resetMotion(), void resetMotion(joint jpartida)

bool **isSettled**()

Sintaxis
bool isSettled()

Función
Reenvía **true** si el robot está parado, **false** si su posición no está ya estabilizada.

La posición se considera estabilizada cuando el error de posición en cada eje permanece, durante 50 ms, inferior al 1% del error de posición máxima autorizada.

Véase también
bool isEmpty()
void waitEndMove()

void **autoConnectMove**(bool bActivo), bool **autoConnectMove**()

Sintaxis
void autoConnectMove(bool bActivo)
bool autoConnectMove()

Función
En modo desalineado, el movimiento de conexión es automático si el brazo está muy cerca de su trayectoria (distancia inferior al error de arrastre máximo autorizado). Si el brazo está demasiado alejado de su trayectoria, el movimiento de conexión será automático o con control manual según el modo definido por la instrucción **autoConnectMove**: automático si **bActivo** vale **true**, bajo control manual si **bActivo** vale **false**. De ser llamado sin parámetro, **autoConnectMove** voltea el modo de movimiento de conexión en curso.

De manera predeterminada, el movimiento de conexión en modo desplazado está bajo control manual.

> **ATENCIÓN:**
> **En condiciones normales de uso, el brazo se detiene en su trayectoria cuando se da una parada de emergencia. Por consiguiente, en modo desalineado, el brazo podrá accionarse de nuevo automáticamente en cualquier modo de movimiento de conexión definido por la instrucción autoConnectMove.**

Véase también
void resetMotion(), void resetMotion(joint jpartida)

S6.4

num **getSpeed**(tool tHerramienta)

Sintaxis

num getSpeed(<tool tHerramienta>)

Función

Esta instrucción reenvía la velocidad de translación cartesiana actual al extremo de la herramienta tTool especificada. La velocidad se calcula a partir de la solicitud de velocidad de la articulación y no del retorno de velocidad de la articulación.

Véase también

point here(tool therramienta, frame fMarca de referencia)

S6.4

num **getPositionErr**()

Sintaxis

num getPositionErr()

Función

Esta instrucción reenvía el error de posición de articulación actual del brazo. El error de posición de la articulación es la diferencia entre el mando de posición de la articulación enviada a los motores y el retorno de posición de la articulación medida por los codificadores.

Véase también

void getJointForce(num& nFuerza)

S6.4

void **getJointForce**(num& nFuerza)

Sintaxis

void getJointForce(<num& nFuerza>)

Función

Esta instrucción reenvía el par actual (N.m para el eje de revolución) o lo fuerza (N para el eje lineal) sobre la articulación calculados a partir de las intensidades del motor.

La fuerza sobre la articulación no es una estimación directa de los esfuerzos externos. También incluye la gravedad, la fricción, la viscosidad, la inercia, el ruido y la precisión de los captadores de intensidad, la relación entre la intensidad del motor y el par. Sólo puede utilizarse para estimar los esfuerzos externos registrando las fuerzas en condiciones de referencia y comparándolas con las fuerzas medidas en condiciones similares con esfuerzos externos suplementarios

Sólo proporciona una magnitud de las fuerzas. Su precisión no está garantizada y se debe evaluar para cada aplicación.

Se genera un error de ejecución si el parámetro no es un cuadro de valores numéricos de un tamaño suficiente.

Véase también

num getPositionErr()

CAPÍTULO 10

OPCIONES

10.1. MOVIMIENTOS CONSECUENTES CON CONTROL DE ESFUERZO

10.1.1. PRINCIPIO

En un comando de movimiento estándar, el robot se desplaza para alcanzar la posición solicitada con una aceleración y una velocidad programada. Si el brazo no llega a cumplir el comando, se solicitará un esfuerzo cada vez mayor a los motores para tratar de alcanzar la posición deseada. Cuando la diferencia entre la posición solicitada y la posición real es demasiado grande, o cuando el esfuerzo solicitado se vuelve demasiado grande, se genera un error sistema que corta la potencia en el brazo.

Se dice que un robot es 'consecuente' cuando acepta ciertas desviaciones entre la posición comandada y la posición efectiva. El controlador puede programarse para adaptarse a la trayectoria, es decir, para aceptar un retardo o avance, en relación con la trayectoria programada, mediante el control de la fuerza aplicada por el brazo. En cambio, no se tolera ninguna desviación respecto a la trayectoria.

Concretamente, los movimientos consecuente del **VAL3** pueden permitir que el brazo siga una trayectoria estando empujado o tirado por una fuerza exterior, o bien ir en contacto con un objeto controlando el esfuerzo ejercido por el brazo sobre este objeto.

10.1.2. PROGRAMACIÓN

Los movimientos consecuentes se programan como movimientos estándar con las instrucciones **movelf()** y **movejf()**, permitiendo un parámetro adicional para controlar el esfuerzo ejercido por el brazo. Durante el movimiento consecuente, se efectúan limitaciones de velocidad, como en un movimiento estándar, a partir del descriptor de movimiento. El movimiento puede hacerse en la trayectoria en un sentido como en el otro.

Es posible encadenar movimientos consecuentes, o encadenar movimientos consecuentes y movimientos estándar: en cuanto se alcanza la posición de destino, el robot prosigue con el comando de movimiento siguiente. La instrucción **waitEndMove()** permite esperar el final de un movimiento consecuente.

La instrucción **resetMotion()** anula todos los movimientos, sean estos consecuentes o no. Después de la instrucción **resetMotion()**, el robot deja de ser consecuente.

Las instrucciones **stopMove()** y **restartMove()** se aplican también a los movimientos consecuentes:

stopMove() fuerza la velocidad del movimiento en curso a cero. Si se trata de un movimiento consecuente, éste será así detenido, y el robot dejará de ser consecuente hasta la ejecución del **restartMove()**.

Por último, la instrucción **isCompliant()** permite asegurarse que el robot está efectivamente en el modo consecuente antes, por ejemplo, de autorizar un esfuerzo exterior sobre el brazo.

10.1.3. CONTROL DEL ESFUERZO

Cuando el parámetro de esfuerzo especificado es nulo, el brazo está pasivo, es decir que sólo se desplaza bajo la acción de un esfuerzo exterior.

Cuando el parámetro de esfuerzo es positivo, todo sucede como si un esfuerzo exterior empujara el brazo hacia la posición comandada: el brazo se desplaza entonces completamente solo, pero puede ser retenido o acelerado por una acción exterior que se añade al esfuerzo comandado.

Cuando el parámetro de esfuerzo es negativo, todo sucede como si un esfuerzo exterior tirara el brazo hacia su posición inicial: para desplazar el brazo hacia la posición comandada, hay entonces que suministrar un esfuerzo exterior más fuerte que el esfuerzo comandado.

El parámetro fuerza se expresa como porcentaje de la carga nominal del brazo. **100%** significa que el brazo aplica una fuerza hacia la posición solicitada, que es equivalente a la carga nominal. En rotación, **100%** corresponde al par nominal autorizado sobre el brazo.

Cuando la velocidad o la aceleración del brazo alcanzan los valores especificados en el descriptor de movimiento, el robot se opone con toda su potencia a cualquier tentativa de aumento de velocidad o de aceleración.

10.1.4. LIMITACIONES

Los movimientos consecuentes presentan las siguientes limitaciones:

- No es posible utilizar el alisado al comienzo o al final de un movimiento consecuente: el brazo marcará necesariamente una parada al comienzo y al final de cada movimiento consecuente.
- Cuando se produce un movimiento consecuente, el brazo puede regresar hacia atrás hacia su punto de inicio, pero no puede excederlo: el brazo se para entonces brutalmente en el punto de inicio.
- El parámetro de esfuerzo en el brazo no puede exceder **1000%**. La precisión obtenida sobre el esfuerzo está limitada por los rozamientos internos. Esta depende en gran parte de la posición del brazo y de la trayectoria pedida.
- Los movimientos consecuentes largos requieren mucha memoria interna. Se genera un error de ejecución si el sistema no tiene suficiente memoria para realizar completamente el movimiento.

10.1.5. INSTRUCCIONES

void **movejf**(joint jPosición, tool tHerramienta, mdesc mDesc, num nFuerza)

Sintaxis

void movejf(joint jPosición, tool tHerramienta, mdesc mDesc, num nFuerza)

Función

Registra un mando de movimiento de articulación conforme, hacia la posición **jPosición**, utilizando la herramienta **tHerramienta**, los parámetros de movimiento **mDesc**, y un mando de fuerza **nFuerza**.

El mando de fuerza **nFuerza** es un valor numérico que representa la fuerza del brazo, que no puede sobrepasar **±1000**. Un valor de 100 corresponde aproximadamente a la masa nominal del brazo.

> **ATENCIÓN:**
> **El sistema no espera el final del movimiento para pasar a la siguiente instrucción VAL3: varios comandos de movimientos pueden haber sido registrados de antemano. Cuando al sistema no le queda más memoria disponible para registrar un nuevo comando, la instrucción espera hasta que pueda hacerse el registro.**

La explicación detallada de los diferentes parámetros de movimiento se proporciona al comienzo del capítulo.

Se genera un error de ejecución si **mDesc** o **nFuerza** tiene valores inválidos, si **jPosición** está fuera de los topes de software, si el movimiento anterior requiere un alisado, o si un mando de movimiento anteriormente registrado no puede realizarse (destino fuera de alcance).

Parámetros

tool tHerramienta	Expresión de tipo herramienta.
mdesc mDesc	Expresión de tipo descriptor de movimiento
joint jPosición	Expresión de tipo posición articular
num nFuerza	Expresión de tipo numérico

Véase también

void movelf(point pPosición, tool tHerramienta, mdesc mDesc, num nFuerza)
bool isCompliant()

void **movelf**(point pPosición, tool tHerramienta, mdesc mDesc, num nFuerza)

Sintaxis

void movelf(point pPosición, tool tHerramienta, mdesc mDesc, num nFuerza)

Función

Registra un comando de movimiento consintiente hacia la posición **pPosición**, con la herramienta **tHerramienta**, los parámetros de movimiento **mDesc**, y un comando de esfuerzo **nFuerza**.

El mando de fuerza **nFuerza** es un valor numérico que representa la fuerza del brazo, que no puede sobrepasar **±1000**. Un valor de 100 corresponde aproximadamente a la masa nominal del brazo.

> **ATENCIÓN:**
> **El sistema no espera el final del movimiento para pasar a la siguiente instrucción VAL3: varios comandos de movimientos pueden haber sido registrados de antemano. Cuando al sistema no le queda más memoria disponible para registrar un nuevo comando, la instrucción espera hasta que pueda hacerse el registro.**

La explicación detallada de los diferentes parámetros de movimiento se proporciona al comienzo del capítulo.

Se genera un error de ejecución si **mDesc** o **nFuerza** tiene valores inválidos, si **pPosición** no es alcanzable, si el movimiento hacia **pPosición** es imposible en línea recta, si el movimiento anterior requiere un alisado, o si un mando de movimiento anteriormente registrado no puede realizarse (destino fuera de alcance).

Parámetros

point pPosición	Expresión de tipo punto.
tool tHerramienta	Expresión de tipo herramienta
mdesc mDesc	Expresión de tipo descriptor de movimiento
num nFuerza	Expresión de tipo numérico

Véase también

void movejf(joint jPosición, tool tHerramienta, mdesc mDesc, num nFuerza)
bool isCompliant()

bool **isCompliant**()

Sintaxis

bool isCompliant()

Función

Reenvía **true** si el robot está en modo consecuente; en caso contrario, **false**.

Ejemplo

```
movelf(pPosición, tHerramienta,
mDesc, 0)
wait(isCompliant())                  // Espera que el robot esté efectivamente en modo consecuente
diEjection = true                    // Comando de la eyección de la prensa
waitEndMove()                        // Espera el final del movimiento consecuente
movej(jDepart, tHerramienta, mDesc)  // Encadena con un movimiento estándar
```

Véase también

void movelf(point pPosición, tool tHerramienta, mdesc mDesc, num nFuerza)
void movejf(joint jPosición, tool tHerramienta, mdesc mDesc, num nFuerza)

10.2. ALTER: CONTROL EN TIEMPO REAL DE UNA TRAYECTORIA

Cartesiano Alter

10.2.1. PRINCIPIO

La alteración cartesiana de una trayectoria permite aplicar al camino una transformación geométrica (translación, rotación, rotación en el punto del centro de herramienta) que es inmediatamente efectiva.
Esta función permite modificar una trayectoria nominal por medio de un sensor externo, con el fin, por ejemplo, de garantizar un seguimiento preciso de la forma de una pieza, o intervenir sobre una pieza en movimiento.

10.2.2. PROGRAMACIÓN

La programación consiste en definir en primer lugar la trayectoria nominal, y luego en definir la diferencia correspondiente en tiempo real.
La trayectoria nominal se programa de la misma manera que los movimientos estándares, por medio de las instrucciones alterMovel(), alterMovej() y alterMovec(). Varios movimientos modificables pueden sucederse, o algunos movimientos modificables pueden alternarse con movimientos no alterables. Definiremos una trayectoria modificable como la sucesión de comandos de movimiento modificables situados entre dos comandos de movimiento no modificables.

La modificación propiamente dicha se programa por medio de la instrucción alter(). Distintos modos alter son posibles en función de la transformación geométrica por practicar; el modo se define por medio de la instrucción alterBegin(). La instrucción alterEnd() sirve, finalmente, para indicar cómo debe terminarse la modificación, ya sea antes del final del movimiento nominal, para poder secuenciar el próximo movimiento no alterable sin causar un paro; o bien después, para poder desplazar el brazo por medio de alter mientras que el movimiento nominal está parado.

Las otras instrucciones de mando de movimiento siguen siendo efectivas en modo alter.

> **ATENCIÓN:**
> **Las instrucciones waitEndMove, open y close esperan el final del movimiento nominal, y no el final del movimiento modificado. Por consiguiente, se reanuda la ejecución de VAL3, tras un waitEndMove, incluso si brazo continúa desplazándose, debido a una alteración cambiante.**

10.2.3. EXIGENCIAS

Sincronización, desincronización: Dado que el mando alter se aplica inmediatamente, el cambio de la modificación se debe controlar de tal modo que la trayectoria del brazo resultante no presente ninguna discontinuidad y ningún ruido:

- Un cambio importante de la modificación sólo puede aplicarse progresivamente con un mando de acercamiento específico.
- El final de la modificación exige una velocidad de modificación nula, obtenida progresivamente gracias a un mando de parada específico.

Mando síncrono: El controlador envía mandos de posición y velocidad cada 4 ms a los amplificadores. En consecuencia, el mando alter debe sincronizarse con este período de comunicación de tal modo que la velocidad de modificación permanece bajo control. Para ello, se utilizará una tarea **VAL3** síncrona (véase capítulo Tareas). De la misma manera, será necesario a veces filtrar previamente la entrada del sensor si los datos son ruidosos o si su período de muestreo no se sincroniza con el período del controlador.

Secuenciamiento progresivo: El primer movimiento no modificable según una trayectoria modificable sólo puede calcularse después de la ejecución alterEnd. En consecuencia, si alterEnd se ejecuta muy poco tiempo después del final del movimiento modificable, el brazo corre el riesgo de relantizarse o incluso de detenerse cerca de este punto mientras no se calcule el siguente movimiento.

Además la capacidad de calcular de antemano el movimiento siguiente impone restricciones a la trayectoria modificada después de la ejecución de alterEnd: Debe conservar la misma configuración, y es necesario cerciorarse que todos los campos permanezcan en el mismo torno de eje. Es posible entonces que se genere un error durante el movimiento, lo que no habría podido ocurrir si alterEnd no se hubiese realizado de antemano.

10.2.4. SEGURIDAD

En cualquier momento, la modificación del usuario puede ser inválida: objetivo fuera de alcance, velocidad o aceleración demasiado elevadas. Cuando el sistema detecta situaciones de este tipo, se genera un error y se detiene bruscamente el brazo en la última posición válida. Es necesario restablecer en primer lugar el movimiento antes de volver a iniciar el funcionamiento.

Cuando el movimiento del brazo se desactiva en el transcurso de un movimiento (modo mantenimiento, solicitud de parada o parada de emergencia), el mando de parada se ejerce tanto sobre los movimientos nominales como sobre los movimientos estándares. Al cabo de un determinado tiempo de espera, el modo alter también se desactiva automáticamente con el fin de garantizar una parada completa del brazo. Cuando desaparece el estado de parada, el movimiento se reanuda y se reactiva automáticamente el modo alter.

10.2.5. LIMITACIONES

Los movimientos nulos (cuando el objetivo de movimiento se encuentra en posición de inicio) no son considerados por el sistema. En consecuencia, debe recurrir a un movimiento no nulo para pasar a modo alter.

No se puede especificar la configuración que debe aplicarse a la trayectoria modificada; en efecto, el sistema siempre utiliza la misma configuración. Por lo tanto, es imposible modificar la configuración del brazo dentro de una trayectoria modificada (incluso utilizando la instrucción alterMovej).

10.2.6. INSTRUCCIONES

void **alterMovej**(joint jPosición, tool tHerramienta, mdesc mDesc)

void **alterMovej**(point pPosición, tool tHerramienta, mdesc mDesc)

Sintaxis

void alterMovej(joint jPosición, tool tHerramienta, mdesc mDesc)
void alterMovej(point pPosición, tool tHerramienta, mdesc mDesc)

Función

Registra un mando de movimiento de campo modificable (una línea en el espacio del campo)

Parámetros

jPosición/pPosición	Expresión de tipo punto o campo que define la posición de fin del movimiento.
tHerramienta	Expresión de tipo herramienta que define el punto del centro de herramienta utilizado durante el movimiento para el mando de velocidad cartesiana.
mDesc	Expresión **mDesc** que define el control de velocidad y el parámetro blending del movimiento.

Detalles

Esta instrucción se comporta exactamente como la instrucción movej, salvo que no autoriza el modo alter para el movimiento. Ver movej para más información.

void **alterMovel**(point pPosición, tool tHerramienta, mdesc mDesc)

Sintaxis
void alterMovel(point pPosición, tool tHerramienta, mdesc mDesc)

Función
Registra un mando de movimiento lineal modificable (una línea en el espacio cartesiano)

Parámetros

pPosición	Expresión de tipo punto que define la posición de fin del movimiento.
tHerramienta	Expresión de tipo herramienta que define el punto del centro de herramienta utilizado durante el movimiento para el mando de velocidad cartesiana. Al final del movimiento, el punto del centro de herramienta se encuentra en la posición objetivo especificada.
mDesc	Expresión mdesc que define el control de velocidad y el parámetro blending del movimiento.

Detalles
Esta instrucción se comporta exactamente como la instrucción movel, salvo que no autoriza el modo alter para el movimiento. Ver movel para más información.

void **alterMovec** (point pIntermediario, point pObjetivo, tool tHerramienta, mdesc m Desc)

Sintaxis
void alterMovec(point pIntermediario, point pObjetivo, tool tHerramienta, mdesc mDesc)

Función
Registra un mando de movimiento circular modificable.

Parámetros

pIntermediario	Expresión de tipo punto que define un punto intermedio del círculo
pObjetivo	Expresión de tipo punto que define la posición de fin del movimiento.
tHerramienta	Expresión de tipo herramienta que define el punto del centro de herramienta utilizado durante el movimiento para el mando de velocidad cartesiana. Al final del movimiento, el punto del centro de herramienta se encuentra en la posición objetivo especificada.
mDesc	Expresión mdesc que define el control de velocidad y el parámetro blending del movimiento.

Detalles
Esta instrucción se comporta exactamente como la instrucción movec, salvo que no autoriza el modo alter para el movimiento. Ver movec para más información.

num **alterBegin**(frame fReferenciaAlter, mdesc mVelocidadMáxima)

num **alterBegin**(tool tReferenciaAlter, mdesc mVelocidadMáxima)

Sintaxis
num alterBegin(frame fReferenciaAlter, mdesc mVelocidadMáxima)
num alterBegin(tool tReferenciaAlter, mdesc mVelocidadMáxima)

Función
Inicializa el modo alter para la trayectoria modificable en curso de ejecución.

Parámetros

fReferenciaAlter/ tReferenciaAlter	Expresión de tipo cuadro o herramienta que define la referencia de la diferencia alter.
mVelocidadMáxima	Expresión mdesc que define los parámetros de verificación de seguridad de la diferencia alter.

Detalles
El modo alter iniciado con alterBegin sólo se termina mediante un mando alterEnd, o mediante un resetMotion. Cuando se alcanza el fin de una trayectoria modificable, el modo alter permanece activo hasta la ejecución de alterEnd.

La expresión trsf del mando alter define una transformación de la trayectoria entera en torno a alterReference:

- La trayectoria gira alrededor del centro del marco o de la herramienta utilizando la parte de rotación del trsf.
- La trayectoria luego se somete a una translación por la parte de translación del trsf.

Las coordenadas trsf del mando alter se definen en la base alterReference.

Cuando se utiliza un marco como referencia, el alterReference es fijo en el espacio (World). Este modo debe utilizarse cuando la divergencia de la trayectoria se conoce o mide en el espacio cartesiano (parte móvil como el rastreo del transportador).

Cuando se utiliza una herramienta como referencia, el alterReference es fijo con relación al punto del centro de la herramienta. Este modo debe utilizarse cuando la diferencia de la trayectoria se conoce o se mide con relación al punto del centro de la herramienta (por ejemplo el sensor de forma de pieza montado sobre la herramienta).

Se utiliza el descriptor de movimiento para definir la velocidad de campo y la velocidad cartesiana máxima sobre la trayectoria modificada (con la ayuda de los campos vel, tvel y rvel del descriptor de movimiento). Se genera un error y se detiene el brazo en la trayectoria si la velocidad modificada supera los límites especificados.

Cuando se produce una parada (eStop, modo espera, VAL3 stopMove()), los campos de aceleración y deceleración del descriptor de movimiento controlan el tiempo de parada: La modificación de la trayectoria se debe parar utilizando estos parámetros de desaceleración (véase alterStopTime).

alterBegin reenvía un valor numérico que indica el resultado de la instrucción:

1	alterBegin se ha ejecutado con éxito
0	alterBegin espera el inicio del movimiento modificable
-1	alterBegin no se ha tenido en cuenta porque el modo alter se inició
-2	alterBegin ha sido rechazado (la opción alter no está activada)
-3	alterBegin se ha rechazado porque el movimiento está en error. Se requiere de un resetMotion.

Véase también
num alterEnd()
num alter(trsf trAlteración)
num alterStopTime()

num **alterEnd()**

Sintaxis
num alterEnd()

Función
Sale del modo alter y procura que el movimiento actual ya no sea modificable.

Detalles
Si se ejecuta alterEnd mientras que se alcanza el final de la trayectoria modificable, el movimiento no modificable siguiente (si existe) se inicia inmediatamente.
Si se ejecuta alterEnd antes del final del movimiento modificable, el valor actual de la diferencia alter se aplica al resto de la trayectoria modificable, hasta el siguiente movimiento no modificable. No es posible pasar de nuevo al modo alter sobre la misma trayectoria modificable.
El siguiente movimiento no modificable, si existe alguno, se calcula a partir de la ejecución de alterEnd para que la transición entre la trayectoria modificable y el movimiento no modificable siguiente se realice sin ruptura.

alterEnd reenvía un valor numérico que indica el resultado de la instrucción:

1	alterEnd se ha ejecutado con éxito
-1	alterEnd no ha sido considerado porque el modo alter aún no se ha iniciado
-3	alterEnd se ha rechazado porque el movimiento está en error. Se requiere de un resetMotion.

Véase también
num alterBegin(frame fReferenciaAlter, mdesc mVelocidadMáxima)
num alterBegin(tool tReferenciaAlter, mdesc mVelocidadMáxima)

num **alter**(trsf trAlteración**)**

Sintaxis
num alter(trsf trAlteración)

Función
Especifica una nueva modificación de la trayectoria modificable.

Parámetros

trAlteración	Expresión Trsf que define la modificación que debe aplicarse hasta la instrucción alter siguiente.

Detalles
La transformación inducida por el trsf de modificación depende del modo alter seleccionado por la instrucción alterBegin. Los datos de la modificación se definen en el marco o en la herramienta especificados por la instrucción alterBegin.

La modificación es aplicada por el sistema cada 4 ms: Cuando se realizan varias instrucciones alter en menos de 4 ms, la última es la que se aplica. Generalmente, la instrucción alter debe realizarse en una tarea síncrona para forzar una reactualización de la modificación cada 4 ms.

Se debe calcular minuciosamente la modificación de tal modo que los mandos de posición y de velocidad del brazo que resulten se realicen sin ruptura y sin ruido. A veces será necesario filtrar previamente la entrada del sensor para alcanzar la calidad buscada sobre la trayectoria y el comportamiento del brazo.

Cuando se para el movimiento (modo mantenimiento, parada de emergencia, instrucción stopMove()), la modificación de la trayectoria se bloquea hasta que todas las condiciones de parada se hayan borrado.

Cuando la modificación de la trayectoria no es válida (posición inalcanzable, límites de velocidad superados), el brazo se detiene de golpe en la última posición válida y se bloquea el modo alter en posición de error. Se requiere de un resetMotion antes de poder reiniciar el funcionamiento. Los límites de velocidad del movimiento alter son definidos por la instrucción alterBegin.

Alter reenvía un valor numérico que indica el resultado de la instrucción:

1	alter se ha ejecutado con éxito.
0	alter espera el reinicio del movimiento (alterStopTime es nulo).
-1	alter no ha sido considerado porque el modo alter no se ha iniciado o ya ha terminado.
-2	alter ha sido rechazado (la opción alter no está activada).
-3	alter se ha rechazado porque el movimiento está en error. Se requiere de un resetMotion.

Véase también

num alterBegin(frame fReferenciaAlter, mdesc mVelocidadMáxima)
num alterBegin(tool tReferenciaAlter, mdesc mVelocidadMáxima)
void taskCreateSync string sNombre, num nPeriodo, bool& bSobreejecución, programa(...)

num **alterStopTime**()

Sintaxis

num alterStopTime()

Función

Reenvía el tiempo restante antes del bloqueo de la diferencia alter, cuando ocurre una parada.

Detalles

Cuando ocurre una parada, el sistema evalúa el tiempo necesario para la parada del brazo si se utilizan los parámetros accel y decel del descriptor de movimiento especificados por alterBegin. La duración mínima necesaria para esta parada y el tiempo impuesto por el sistema (generalmente 0.5s cuando ocurre un eStop) son reenviados por alterStopTime.

Cuando alterStopTime reenvía un valor negativo, no hay condición de parada pendiente. Cuando alterStopTime reenvía un valor nulo, el mando alter se bloquea hasta que todas las condiciones de parada se hayan reinicializado.

alterStopTime reenvía un valor nulo cuando no se activa el modo alter.

Véase también

num alterBegin(frame fReferenciaAlter, mdesc mVelocidadMáxima)
num alterBegin(tool tReferenciaAlter, mdesc mVelocidadMáxima)
num alter(trsf trAlteración)

S6.1

10.3. CONTROL DE LICENCIA OEM

10.3.1. PRINCIPIOS

Una licencia OEM es una llave específica del controlador que permite restringir el uso de una proyecto **VAL3** a algunos controladores de robot seleccionados.

Con Stäubli Robotics Studio(*)se suministra una herramienta para codificar una contraseña OEM secreta en una licencia OEM pública, específica del controlador, que puede luego instalarse como una opción de software sobre el controlador. Mediante la utilización de la instrucción **getLicence()**, un proyecto o biblioteca puede comprobar si la licencia OEM está instalada y, por consiguiente, asegurarse de que esta es utilizada únicamente por los controladores de robot seleccionados.

Para mantener secreto la contraseña OEM y proteger el código durante la prueba de la licencia, se debe utilizar la instrucción **getLicence()** en una biblioteca codificada.

Está soportado el modo de demostración de licencias OEM; en este caso, el controlador es simplemente configurado con la llave "demo", lo cual es notificado al llamante mediante la instrucción **getLicence()**. Con el emulador **VAL3**, basta con pulsar sobre la tecla "demo" para activar completamente la licencia OEM.

La instrucción **getLicence()** es una opción **VAL3** y requiere la instalación de una licencia de tiempo de ejecución sobre el controlador. Si esta licencia de ejecución no está definida, **getLicence()** reenvía un código de error.

(*) Esta herramienta, un licence.exe ejecutable, suministrada con el emulador **VAL3**, requiere la utilización de una licencia SRS específica.

10.3.2. INSTRUCCIONES

string **getLicence**(string sOemNombreLicencia, string sOemContraseña)

Sintaxis

string getLicence(string sOemNombreLicencia, string sOemContraseña)

Función

Retorna el estado de la licencia OEM especificada:

"oemLicenceDisabled"	La licencia de tiempo de ejecución **VAL3** "oemLicencia" no está activada en el controlador: la licencia OEM no puede comprobarse.
""	La licencia OEM sOemNombreLicencia no está activada (no definida o contraseña inválida).
"demo"	La licencia OEM sOemNombreLicencia está activada en modo demostración.
"enabled"	La licencia OEM sOemNombreLicencia está activada.

Véase también

Codificación

CAPÍTULO 11

ADJUNTO

11.1. CÓDIGOS DE ERROR DE EJECUCIÓN

Código	Descripción
-1	No se ha creado ninguna tarea que lleve el nombre especificado por esta librería o aplicación
0	Ningún error de ejecución
1	La tarea especificada está en curso de ejecución
10	Cálculo numérico no válido (división por cero).
11	Cálculo numérico no válido (por ejemplo ln(-1))
20	Acceso a un conjunto con un índice mayor que el tamaño del conjunto.
21	Acceso a un conjunto con un índice negativo.
29	Nombre de tarea no válido. Véase instrucción taskCreate().
30	El nombre especificado no corresponde a ninguna tarea **VAL3**.
31	Existe ya una tarea del mismo nombre. Véase instrucción taskCreate.
32	Sólo se soportan 2 períodos diferentes para las tareas síncronas. Modificar el período de programación.
40	No hay suficiente espacio de memoria sistema para los datos.
41	No hay suficiente espacio de memoria de ejecución para la tarea. Véase Tamaño de memoria de ejecución.
60	Tiempo máximo de ejecución de la instrucción excedido.
61	Error interno al interpretador **VAL3**
70	Valor de parámetro de instrucción no válido. Véase instrucción correspondiente.
80	Utilización de un dato o un programa de una biblioteca no cargada en la memoria.
81	Cinemática incompatible: Utilización de un punto/campo/config incompatible con el brazo cinemático.
82	El plano o herramienta de referencia de una variable pertenece a una biblioteca y no es accesible a partir del campo de la variable (biblioteca no declarada en el proyecto de la variable o la variable de referencia es privada).
90	La tarea no puede reanudarse en el lugar especificado. Véase instrucción **taskResume()**.
100	La velocidad especificada en el descriptor de movimiento no es válida (negativa o demasiado alta).
101	La aceleración especificada en el descriptor de movimiento no es válida (negativa o demasiado alta).
102	La deceleración especificada en el descriptor de movimiento no es válida (negativa, demasiado alta o inferior a la velocidad).
103	La velocidad de traslación especificada en el descriptor de movimiento es inválida (negativa o demasiado grande).
104	La velocidad de rotación especificada en el descriptor de movimiento es inválida (negativa o demasiado grande).
105	El parámetro reach especificado en el descriptor de movimiento no es válido (negativo).
106	El parámetro leave especificado en el descriptor de movimiento no es válido (negativo).
122	Tentativa de escritura sobre una entrada del sistema.
123	Utilización de una entrada-salida dio, aio o sio no enlazada a una entrada-salida del sistema.
124	Tentativa de acceso a una entrada-salida protegida del sistema
125	Error de lectura o de escritura en una dio, aio o sio (error en un bus de campo)
150	Imposible ejecutar esta instrucción de movimiento: un movimiento solicitado anteriormente no ha podido ser terminado (punto fuera de alcance, singularidad, problema de configuración...)
153	Comando de movimiento no soportado
154	Instrucción de movimiento no válido: compruebe el descriptor de movimiento.
160	Coordenadas de la herramienta flange no válidas
161	Coordenadas del sistema de referencia world no válidas
162	Utilización de un point sin plano. Véase Definición.
163	Utilización de un plano sin sistema de referencia. Véase Definición.
164	Utilización de una herramienta sin herramienta de referencia. Véase Definición.
165	Sistema de referencia o herramienta de referencia no válido (variable global enlazada a una variable local)
250	No hay licencia de duración de ejecución (runtime) para esta instrucción, o se ha terminado la validez de la licencia de demo.

11.2. CÓDIGOS DE LAS TECLAS DEL TECLADO DE LA CONSOLA

Sin **Shift**

3	Caps	Space			Move
283	-	32			-
2	Shift	Esc	Help	Ret.	Run
282	-	255	-	270	-
	Menu	Tab	Up	Bksp	Stop
	-	259	261	263	-
1	User	Left	Down	Right	
281	-	264	266	268	

Con **Shift**

3	Caps	Space			Move
283	-	32			-
2	Shift	Esc	Help	Ret.	Run
282	-	255	-	270	-
	Menu	UnTab	PgUp	Bksp	Stop
	-	260	262	263	-
1	User	Home	PgDn	End	
281	-	265	267	269	

Menús (con o sin **Shift**):

F1	F2	F3	F4	F5	F6	F7	F8
271	272	273	274	275	276	277	278

Para las teclas estándares, el código retornado es el código **ASCII** del carácter correspondiente:

Sin **Shift**									
q	w	e	r	t	y	u	i	o	p
113	119	101	114	116	121	117	105	111	112
a	s	d	f	g	h	j	k	l	<
97	115	100	102	103	104	106	107	108	60
z	x	c	v	b	n	m	.	,	=
122	120	99	118	98	110	109	46	44	61

Con **Shift**									
7	8	9	+	*	;	(	)	[	]
55	56	57	43	42	59	40	41	91	93
4	5	6	-	/	?	:	!	{	}
52	53	54	45	47	63	58	33	123	125
1	2	3	0	"	%	-	.	,	>
49	50	51	48	34	37	95	46	44	62

Con doble **Shift**									
Q	W	E	R	T	Y	U	I	O	P
81	87	69	82	84	89	85	73	79	80
A	S	D	F	G	H	J	K	L	}
65	83	68	70	71	72	74	75	76	125
Z	X	C	V	B	N	M	$	\	=
90	88	67	86	66	78	77	36	92	61

ILUSTRACIÓN

INDEX

J

L

M

N

O

P

R

S

T

U

V

W