KUKA

Training

KUKA Roboter GmbH

Programación de robots 3

KUKA System Software 8

Edición: 20.12.2011

Versión: P3KSS8 Roboterprogrammierung 3 V1 es

KUKA Roboter GmbH
Zugspitzstraße 140
D-86165 Augsburg
Alemania

Traducción de la documentación original

KIM-PS5-DOC

Publicación:	Pub COLLEGE P3KSS8 Roboterprogrammierung 3 (PDF-COL) es
Estructura de libro:	P3KSS8 Roboterprogrammierung 3 V1.1
Versión:	P3KSS8 Roboterprogrammierung 3 V1 es

Indice

1 Programación estructurada

1.1 Objetivo de la metodología de programación uniforme

Objetivo de la metodología de programación uniforme

Una metodología de programación sirve para:

- hacer frente a problemas complejos de un modo más sencillo mediante una estructura dividida de forma rigurosa
- representar de forma comprensible el proceso en cuestión (sin tener conocimientos de programación más avanzados)
- aumentar la efectividad en el mantenimiento, la modificación y la ampliación de programas

La planificación del programa con previsión tiene como consecuencia:

- poder dividir tareas complejas en tareas parciales simples
- reducir el tiempo total necesario para la programación
- permitir la posibilidad de sustitución de componentes con el mismo rendimiento
- poder desarrollar componentes de forma separada

Los 6 requisitos para un programa del robot:

1. Eficacia
2. Ausencia de errores
3. Inteligibilidad
4. Mantenimiento sencillo
5. Control visual completo
6. Rentabilidad

1.2 Elementos auxiliares para la creación de programas de robot estructurados

¿Qué utilidad tiene un comentario?

Los comentarios son complementos/observaciones dentro del lenguaje de programación. Todos los lenguajes de programación se componen de instrucciones para el ordenador (código) e indicaciones para el procesador de textos (comentarios). Si se continúa el procesamiento (se compila, interpreta, etc.) de un texto fuente, se ignorarán los comentarios del software de procesamiento y, por ello, no tienen ninguna influencia en el resultado.

En el controlador de KUKA se utilizan comentarios de líneas, es decir, los comentarios terminan automáticamente al final de la línea.

Los comentarios por sí solos no pueden hacer que un programa sea legible, pero pueden aumentar considerablemente la legibilidad de un programa bien estructurado. El programador tiene la posibilidad mediante los comentarios de añadir observaciones y explicaciones en el programa sin que sean registradas como sintaxis por el controlador.

El programador es responsable de que el contenido de los comentarios coincida con el estado actual de las indicaciones de programación. En caso de modificaciones del programa, también se deberá comprobar los comentarios y adaptarse si es necesario.

El contenido de un comentario y, de este modo, también su utilización se puede seleccionar libremente por el autor y no está sometido a ninguna sintaxis vinculante. Por regla general, los comentarios se realizan en lenguaje "humano", bien en la lengua materna del autor o en una lengua universal.

- Observaciones sobre el contenido o función de un programa
- El contenido y la utilización se pueden seleccionar libremente
- Se mejora la legibilidad de un programa

- Contribución a la estructuración de un programa
- La responsabilidad de la actualización corresponde al programador
- KUKA utiliza comentarios de líneas
- Los comentarios no son registrados como sintaxis por el controlador

¿Dónde y cuándo se utilizan los comentarios?

Informaciones sobre el texto fuente completo:

Al principio de un texto fuente el autor puede aplicar comentarios previos, incluyendo aquí el autor, la licencia, la fecha de creación, la dirección de contacto en caso de preguntas, la liste de otros ficheros necesarios, etc.

```
DEF PICK_CUBE()
;Este programa recoge el cubo del depósito
;Autor: Max Mustermann
;Fecha de creación: 09.08.2011
INI
...
END
```

Subdivisión del texto fuente:

Los títulos y los apartados se pueden identificar como tales. Para ello, no solo se utilizan frecuentemente medios lingüísticos, sino también medios gráficos que se pueden aplicar a lo largo del texto.

```
DEF PALLETIZE()
;*****************************************************
;*Este programa paletiza 16 cubos sobre la mesa*
;*Autor: Max Mustermann------------------------------*
;*Fecha de creación: 09.08.2011----------------------*
;*****************************************************
INI
...
;------------Cálculo de las posiciones----------------
...
;------------Paletizado de los 16 cubos---------------
...
;------------Despaletizado de los 16 cubos---------------
...
END
```

Explicación de una línea individual:

De este modo se puede explicar el procedimiento o el significado de una parte del texto (p. ej. línea del programa) para que otros autores o el propio autor puedan entenderlo mejor posteriormente.

```
DEF PICK_CUBE()

INI

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; Desplazamiento a posición previa para agarrar

LIN Grip_Pos ; Cubo desplazamiento a posición de agarre
...

END
```

Indicación sobre el trabajo a realizar:

Los comentarios pueden identificar fragmentos de códigos insuficientes o ser un comodín para fragmentos de códigos completamente ausentes.

```
DEF PICK_CUBE()

INI

;Aquí se debe insertar el cálculo de las posiciones de palets!

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; Desplazamiento a posición previa para agarrar

LIN Grip_Pos ; Cubo desplazamiento a posición de agarre

;Aquí falta el cierre de la garra

END
```

Insertar punto y coma:

Si se borra provisionalmente un componente del código, aunque posiblemente se vuelve a introducir de forma posterior, se insertará un punto y coma. En cuanto está incluido en el comentario, el fragmento del código ya no es un código desde el punto de vista del compilador, es decir, prácticamente ya no está disponible.

```
DEF Palletize()

INI

PICK_CUBE()

;CUBE_TO_TABLE()

CUBE_TO_MAGAZINE()

END
```

¿Qué provoca la utilización de Folds en un programa del robot?

- En FOLDS se pueden ocultar partes del programa
- Los contenidos de FOLDS no son visibles para el usuario
- Los contenidos de FOLDS se procesan de forma totalmente normal en la ejecución del programa
- Mediante la utilización de Folds se puede mejorar la legibilidad de un programa

¿Qué ejemplos existen para la utilización de Folds?

En el controlador de KUKA ya se utilizan Folds por el sistema de forma estándar, p. ej. en la indicación de formularios en línea. Estos Folds facilitan el control visual completo de los valores introducidos en el formulario en línea y ocultan las partes del programa que no son relevantes para el operario.

Además, para el usuario existe la posibilidad de crear Folds propios (a partir de grupo de usuario experto). Estos Folds se pueden utilizar por el programador, p. ej. para comunicarle al operario lo que está ocurriendo en un lugar determinado del programa, pero manteniendo la sintaxis KRL real en segundo plano.

En general, los Folds se representan primero cerrados tras su creación.

```
DEF Main()
...
INI                          ; KUKA FOLD cerrado

SET_EA                       ; FOLD cerrado por utilizador

PTP HOME Vel=100% DEFAULT  ; KUKA FOLD cerrado

PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table
...
PTP HOME Vel=100% Default

END
```

```
DEF Main()
...
INI                          ; KUKA FOLD cerrado

SET_EA                       ; FOLD creado por utilizador
$OUT[12]=TRUE
$OUT[102]=FALSE
PART=0
Position=0

PTP HOME Vel=100% DEFAULT  ; KUKA FOLD cerrado
...
PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table

PTP HOME Vel=100% Default

END
```

```
DEF Main()
...
INI                          ; KUKA FOLD cerrado

SET_EA                       ; FOLD cerrado por utilizador

PTP HOME Vel=100% DEFAULT  ; KUKA FOLD abierto
$BWDSSTART=FALSE
PDAT_ACT=PDEFAULT
FDAT_ACT=FHOME
BAS(#PTP_PARAMS,100)
$H_POS=XHOME
PTP XHOME
...

PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table

PTP HOME Vel=100% Default

END
```

¿Por qué se trabaja con tecnología de subprogramas?

En la programación los subprogramas se utilizan principalmente para permitir una utilización múltiple de partes de tareas iguales y, de este modo, conseguir la prevención de repeticiones de códigos. Esto también permite ahorrar en espacio de almacenamiento, entre otros aspectos.
Otro motivo importante para la utilización de subprogramas es las estructuración resultante del programa.
Un subprograma debe realizar una tarea parcial acotada a sí misma y fácilmente descriptible.
Los subprogramas son actualmente más cortos y con una presentación clara en beneficio de la mantenibilidad y la eliminación de errores de programación, ya que el tiempo y la administración invertidos desde el punto de vista interno del ordenador para la llamada de subprogramas, ya no tiene prácticamente ninguna importancia en los ordenadores modernos.

- Posibilidad de utilización múltiple
- Prevención de repeticiones de códigos

- Ahorro de espacio de almacenamiento
- Los componentes se pueden desarrollar separados unos de otros
- Posibilidad de intercambio de componentes del mismo nivel de rendimiento en cualquier momento
- Estructuración del programa
- Tarea completa dividida en tareas parciales
- Mantenibilidad y eliminación de errores de programación mejoradas

Aplicación de subprogramas

```
DEF MAIN()

INI

LOOP

    GET_PEN()
    PAINT_PATH()
    PEN_BACK()
    GET_PLATE()
    GLUE_PLATE()
    PLATE_BACK()

    IF $IN[1] THEN
        EXIT
    ENDIF

ENDLOOP

END
```

¿Qué provoca la inserción de líneas de instrucciones?

Para aclarar la relación de los módulos del programa, se recomienda insertar secuencias de instrucciones encajadas en el texto de programación y escribir indicaciones de la misma profundidad de apilamiento directamente unas debajo de otras.
El efecto logrado es puramente visual, solo afecta al valor de un programa como medio de información de persona a persona.

```
DEF INSERT()
INT PART, COUNTER
INI
PTP HOME Vel=100% DEFAULT
LOOP
     FOR COUNTER = 1 TO 20
         PART = PART+1
         ;Imposible sangrar formularios inline!!!
PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table
         PTP XP5
     ENDFOR
...
ENDLOOP
```

¿Qué se consigue mediante una identificación adecuada de nombres de datos?

Para poder indicar correctamente la función de datos y señales en un programa del robot, se recomienda utilizar términos significativos en la asignación de los nombres. Aquí se incluyen p. ej.:

- Nombre de texto largo para señales de entrada y de salida
- Nombres de Tool y Base
- Declaraciones de señales para señales de entrada y de salida
- Nombres de puntos

1.3 Cómo crear un plan de ejecución del programa

¿Qué es un PEP?

Un plan de ejecución del programa (PEP) es un diagrama de flujo para un programa que también se denomina como plan de estructura del programa. Es una representación gráfica para la transformación de un algoritmo en un programa y describe la secuencia de operaciones para la solución de una tarea. Los símbolos para los planes de ejecución del programa están regulados en la norma DIN 66001. Los planes de ejecución del programa también se utilizan a menudo por programas de ordenador para la representación de procesos y tareas.
En comparación con una descripción basada en un código, el algoritmo del programa presenta una legibilidad considerablemente más sencilla, ya que gracias a la representación gráfica, la estructura se puede reconocer notablemente mejor.
Los errores en la estructura y de programación se evitan de forma más sencilla en la transformación posterior en código de programación, ya que en caso de aplicación correcta de un PEP, es posible la transformación directa en un código de programación. Al mismo tiempo, con la creación de un PEP se obtiene una documentación del programa que se va a crear.

- Herramienta para la estructuración del proceso de un programa
- La ejecución del programa es fácilmente legible
- Los errores en la estructura se pueden reconocer de forma más sencilla
- Documentación simultánea del programa

Símbolos PEP

Comienzo o fin de un proceso o de un programa

Fig. 1-1

Combinación de indicaciones y operaciones

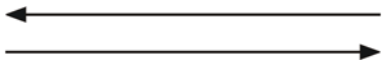

Fig. 1-2

Ramificación

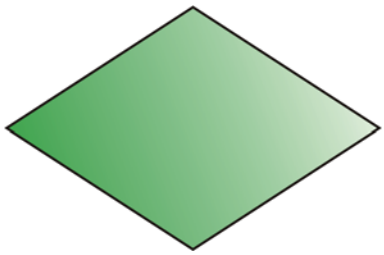

Fig. 1-3

Indicaciones generales en el código de programa

Fig. 1-4

Llamada de subprograma

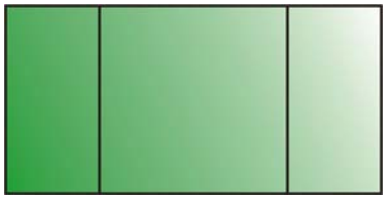

Fig. 1-5

Indicación de entrada/salida

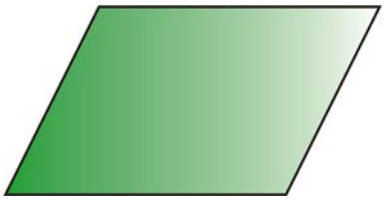

Fig. 1-6

Ejemplo PEP

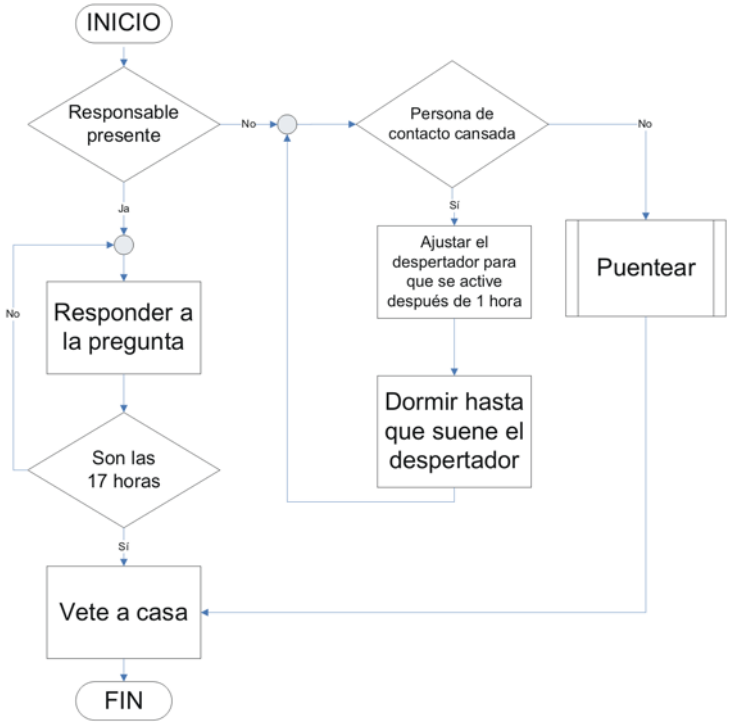

Fig. 1-7

Cómo crear un PEP

Partiendo de la presentación de usuario, el problema se perfecciona paso por paso, hasta que se tenga un control visual completo de los componentes elaborados de este modo para poder adaptarlos en KRL.
Los proyectos que se creen en los pasos de desarrollo seguidos se diferencian por la precisión de detalles creciente.

1. Subdivisión aproximada del proceso completo en aprox. 1-2 páginas
2. División de la tarea completa en tareas parciales pequeñas
3. Subdivisión aproximada de las tareas parciales
4. Perfeccionamiento de la subdivisión de las tareas parciales
5. Aplicación en código KRL

2 Interpretador Submit

2.1 Utilización del interpretador Submit

Descripción del interpretador Submit

En el KSS 8.x se ejecutan dos tareas:

- Interpretador de robots (secuencia de los programas de movimiento de los robots y su lógica)
- Interpretador de la unidad de control (secuencia de un programa de la unidad de control ejecutado en paralelo)

  Estructura del programa SPS.SUB:

```
 1  DEF  SPS ( )
 2    DECLARATIONS
 3    INI
 4
 5    LOOP
 6       WAIT FOR NOT($POWER_FAIL)
 7       TORQUE_MONITORING()
 8
 9      USER PLC
10    ENDLOOP
```

El estado del interpretador Submit

		
El interpretador Submit está en funcionamiento	El interpretador Submit está detenido	El interpretador Submit no está seleccionado

El interpretador de la unidad de control:

- Se puede iniciar automática o manualmente.
- También se puede detener o cancelar manualmente.
- Puede incluir tareas de manejo y control en el entorno del robot.
- Se crea de manera estándar con el nombre **SPS.sub** en el directorio R1/SYSTEM.
- Se puede programar con el conjunto de instrucciones KRL.
- **No** se pueden ejecutar las instrucciones KRL relacionadas con los movimientos del robot.
- Se permiten movimientos asincrónicos de los ejes adicionales.
- Se puede acceder a las variables del sistema a través de la lectura y de la escritura.
- Se puede acceder a las entradas/salidas a través de la lectura y de la escritura.

Relaciones en la programación del interpretador Submit

¡Atención!
¡El interpretador Submit no puede utilizarse para aplicaciones críticas con respecto al tiempo! En estos casos deberá utilizarse un PLC. Causas:

- El interpretador Submit comparte la capacidad del sistema con el interpretador de robots y la gestión de E/S con la mayor prioridad. Por esta razón, el interpretador Submit no se ejecuta regularmente en el ciclo de interpolación de 12 ms de la unidad de control del robot.
- Además, el tiempo de ejecución del interpretador Submit es irregular. El tiempo de ejecución del interpretador Submit dependerá del número de líneas en el programa SUB. Las líneas de comentarios y las líneas vacías también influyen.

- Inicio automático del interpretador Submit
  - El interpretador Submit se activa de forma automática al conectar la unidad de control del robot.
  - Se inicia el programa definido en el archivo KRC/STEU/MADA/$custom.dat.

```
$PRO_I_O[]="/R1/SPS()"
```

- Servicio automático del interpretador Submit
  - Seleccionar la secuencia de menú **Configuración** > **Interpretador SUBMIT** > **Arrancar/Seleccionar** para el servicio.
  - Servicio directo a través de la barra de estado del indicador de estado **Interpretador Submit**. En caso de contacto, se abre una ventana con las opciones ejecutables.

Si se modifica un archivo del sistema, por ejemplo $config.dat o $custom.dat, y éste resulta dañado a raíz de esta modificación, el interpretador Submit se desactiva automáticamente. Si se subsana el error en el archivo del sistema se deberá seleccionar de nuevo manualmente el interpretador Submit.

Particularidades de la programación del interpretador Submit

- **No** se pueden ejecutar instrucciones para movimientos del robot, por ejemplo:
  - `PTP`, `LIN`, `CIRC`, etc.
  - Activaciones de subprogramas que contienen movimientos de robot
  - Instrucciones que hacen referencia a movimientos del robot, `TRIGGER` o `BRAKE`
- Se pueden controlar ejes asincrónicos, como E1.

```
IF (($IN[12] == TRUE) AND ( NOT $IN[13] == TRUE)) THEN
ASYPTP {E1 45}
...
IF ((NOT $IN[12] == TRUE) AND  ($IN[13] == TRUE)) THEN
ASYPTP {E1 0}
```

- Las instrucciones entre las líneas `LOOP` y `ENDLOOP` se ejecutan constantemente en "segundo plano".
- Se debe evitar cualquier parada provocada por instrucciones o bucles de espera que puedan retrasar más la ejecución del interpretador Submit.
- Es posible activar salidas.

Advertencia
No se comprobará si los interpretadores Submit y de robots acceden simultáneamente a la misma salida, ya que esto puede ser lo que se desee que ocurra en determinados casos.
Por esta razón, el usuario deberá verificar con atención la asignación de las salidas. De lo contrario pueden producirse señales de salida inesperadas, como por ejemplo en los dispositivos de seguridad. Como consecuencia podrían ocasionarse importantes daños materiales, lesiones graves e incluso la muerte.

⚠ ADVERTENCIA En los modos de servicio del test, $OV_PRO no se puede describir desde el interpretador Submit, porque la modificación puede tener resultados inesperados para los usuarios que trabajan en el robot industrial. Las consecuencias puden ser daños materiales importantes, lesiones graves e incluso la muerte.

Advertencia
En la medida de lo posible, no modificar las señales y variables relevantes en materia de seguridad (p. ej. modo de servicio, PARADA DE EMERGENCIA, contacto de puerta de seguridad) con el interpretador Submit.
Si a pesar de todo es necesario efectuar cambios, todas las señales y variables relevantes para la seguridad deben estar enlazadas de forma que el interpretador Submit o el PLC no pueda colocarlas en un estado potencialmente peligroso.

Procedimiento para programar el interpretador Submit

1. La programación se realiza en estado detenido o cancelado.
2. El programa estándar SPS.sub se carga en el editor.
3. Realizar las declaraciones e inicializaciones necesarias. Para ello se deberían utilizar las folds preparadas.
4. Realizar ampliaciones del programa en la fold `USER PLC`.
5. Cerrar y guardar el interpretador Submit.
6. Si el Submit no se inicia automáticamente, iniciarlo manualmente.

Ejemplo de programa basado en una programación de los intermitentes en el interpretador Submit

```
DEF SPS( )
DECLARATIONS
DECL BOOL flash ;Declaración en $CONFIG.dat
INI
flash = FALSE
$TIMER[32]=0 ; Reiniciar TIMER[32]
$TIMER_STOP[32]=false ; Iniciar TIMER[32]
...
LOOP
...
USER PLC
IF ($TIMER[32]>500) AND (flash==FALSE) THEN
   flash=TRUE
ENDIF
IF $TIMER[32]>1000 THEN
   flash=FALSE
   $TIMER[32]=0
ENDIF
; Asignación a lámpara (salida 99)
$OUT[99] = flash
...
ENDLOOP
```

3 Campos de trabajo con KRL

3.1 Utilizar campos de trabajo

Descripción

Campos de trabajo seguros y no seguros

- Los campos de trabajo seguros sirven para la protección personal y solo se pueden configurar mediante la opción adicional SafeOperation.
- Mediante el software del sistema KUKA 8.x se pueden configurar campos de trabajo para el robot. Estos campos **sólo sirven para proteger las instalaciones**.

Campos de trabajo no seguros

- Estos campos de trabajo no seguros se configuran directamente en el software de sistema KUKA.
- Se pueden crear 8 campos de trabajo específicos del eje.

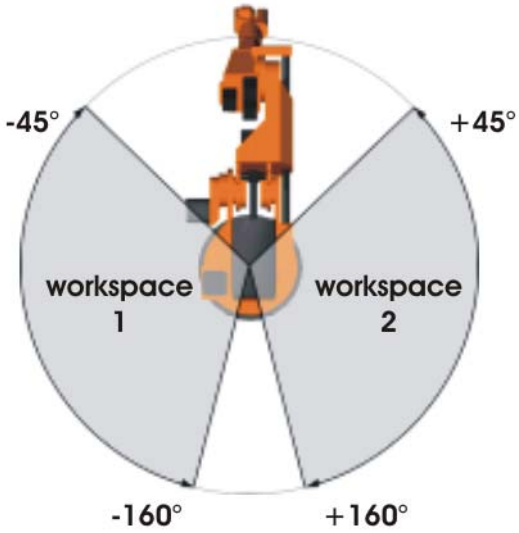

Fig. 3-1: Ejemplo de campos de trabajo específicos del eje para A1

- Con los campos de trabajo específicos del eje se pueden restringir aún más las zonas definidas por los interruptores de final de carrera del software para proteger el robot, la herramienta o la pieza.
- Se pueden crear 8 campos de trabajo cartesianos.

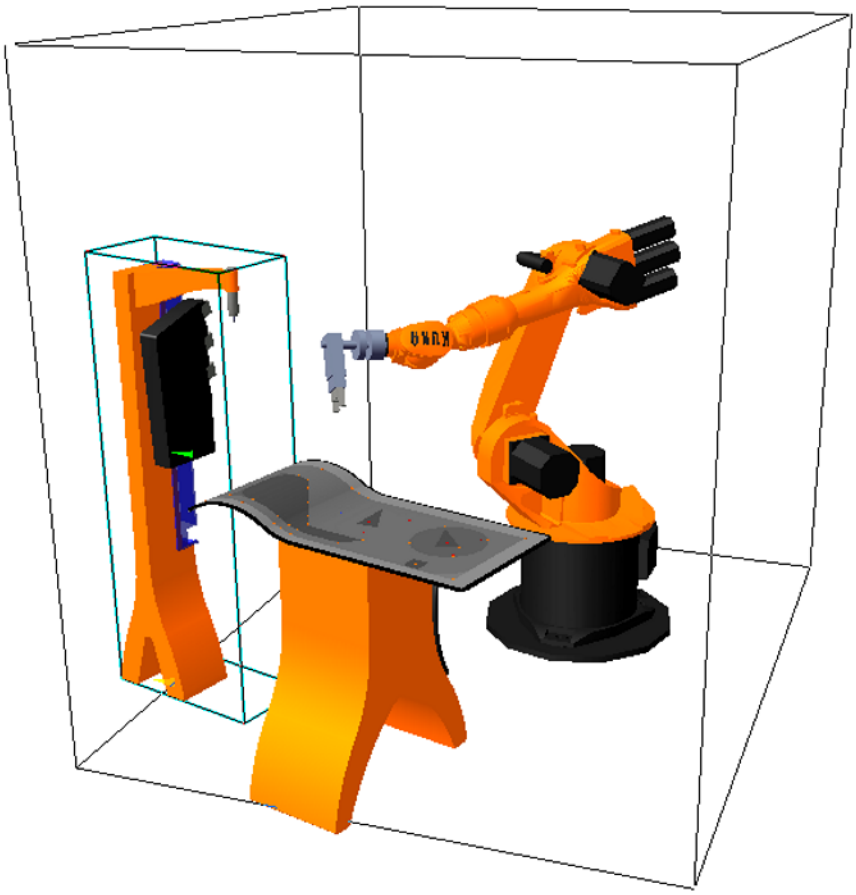

Fig. 3-2: Ejemplo de campo de trabajo cartesiano

- En el caso de los campos de trabajo cartesianos sólo se controlará la posición del TCP. No es posible controlar si otras piezas del robot dañan el campo de trabajo.
- Para crear formas complejas, se pueden activar varios campos de trabajo que se pueden solapar.
- **Campos no permitidos**: el robot sólo se puede mover fuera de este tipo de campos.

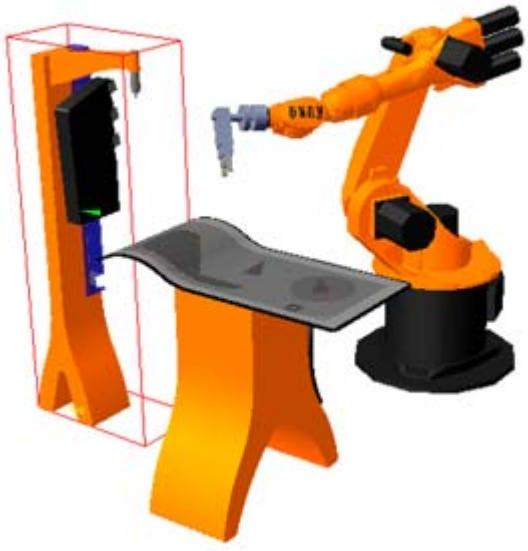

Fig. 3-3: Campos no permitidos

- **Campos permitidos**: el robot no se puede mover fuera de este tipo de campos.

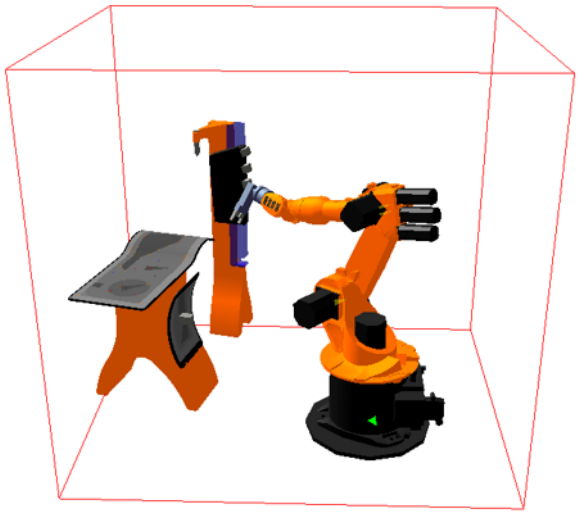

Fig. 3-4: Campos permitidos

- Las reacciones que se pueden producir en caso de que el robot dañe un campo de trabajo dependen de la configuración.
- Por cada campo de trabajo se puede emitir una salida (señal).

El principio de campos de trabajo y de bloqueo de campo de trabajo

Bloqueo de campo de trabajo

- Secuencia en caso de acoplamiento directo (sin PLC).

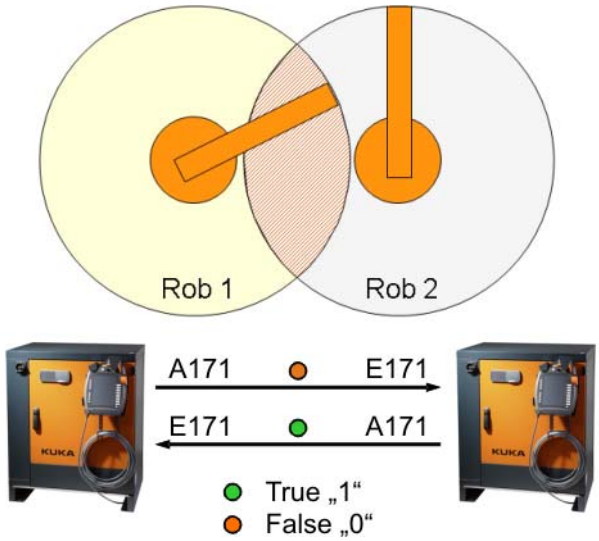

Fig. 3-6

- Secuencia con un PLC que sólo puede transmitir las señales o que además aplica una lógica.

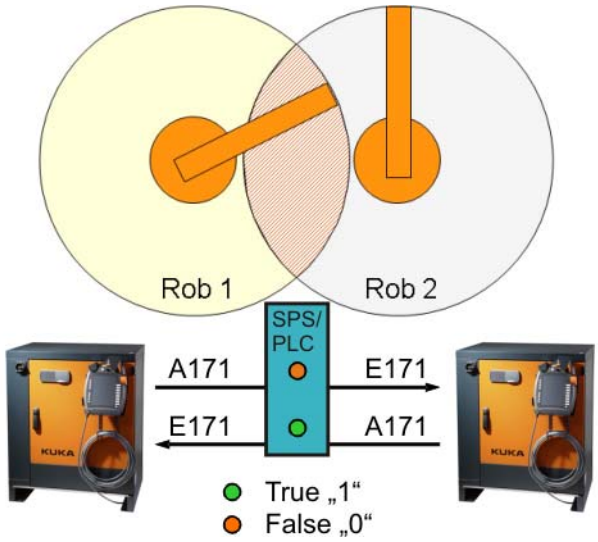

Fig. 3-10

- Transmisión directa de la señal (si se utiliza un PLC: sin lógica).
  - **Sin tiempo de espera**: se crea una solicitud de entrada y, si el campo no está bloqueado, el robot puede entrar en él de inmediato.

AVISO Sin embargo, si la solicitud de entrada es simultánea, los dos robots obtendrán permiso de entrada, lo que por norma general dará lugar a un choque.

  - **Con tiempo de control**: se crea una solicitud de entrada y se bloquea el campo propio. El nuevo campo se comprueba una vez transcurrido un tiempo de control. Si el campo no está bloqueado, el robot podrá entrar de inmediato en él. Si las dos solicitudes se reciben casi simultáneamente, los campos se bloquearán.
- Transmisión de la señal con control lógico (prioridad)
  - Las solicitudes de entrada y la liberación de la entrada se combinan entre sí mediante la lógica. En caso de solicitud de entrada simultánea, el **control de prioridades** regula también qué robot puede entrar en la zona de trabajo común.
  - Además de controlar la prioridad, se puede comprobar si el robot (TCP de robot) se encuentra en la zona de trabajo para autorizar la entrada. Para ello se deben **definir campos de trabajo**.

Principio de la configuración de campo de trabajo

Modo para campos de trabajo

- **#OFF**

  El control del campo de trabajo está desactivado.
- **#INSIDE**
  - Campo de trabajo cartesiano: la salida definida se establece cuando el TCP o la brida se encuentran dentro del campo de trabajo.
  - Campo de trabajo específico del eje: la salida definida se fija cuando el eje se encuentra dentro del campo de trabajo.
- **#OUTSIDE**
  - Campo de trabajo cartesiano: la salida definida se establece cuando el TCP o la brida se encuentran fuera del campo de trabajo.
  - Campo de trabajo específico del eje: la salida definida se fija cuando el eje se encuentra fuera del campo de trabajo.
- **#INSIDE_STOP**
  - Campo de trabajo cartesiano: la salida definida se establece cuando el TCP, la brida o el punto de la raíz de la muñeca se encuentran den-

tro del campo de trabajo (punto de la raíz de la muñeca = punto central del eje A5)

- Campo de trabajo específico del eje: la salida definida se fija cuando el eje se encuentra dentro del campo de trabajo.

Además, el robot se detiene y se muestran mensajes. El robot sólo se puede volver a mover si se ha desactivado o puenteado el control del campo de trabajo.

- **#OUTSIDE_STOP**
  - Campo de trabajo cartesiano: la salida definida se establece cuando el TCP o la brida se encuentran fuera del campo de trabajo.
  - Campo de trabajo específico del eje: la salida definida se fija cuando el eje se encuentra fuera del campo de trabajo.

  Además, el robot se detiene y se muestran mensajes. El robot sólo se puede volver a mover si se ha desactivado o puenteado el control del campo de trabajo.

Los siguientes parámetros definen la posición y el tamaño de un campo de trabajo cartesiano:

- Origen del campo de trabajo respecto del sistema de coordenadas WORLD.

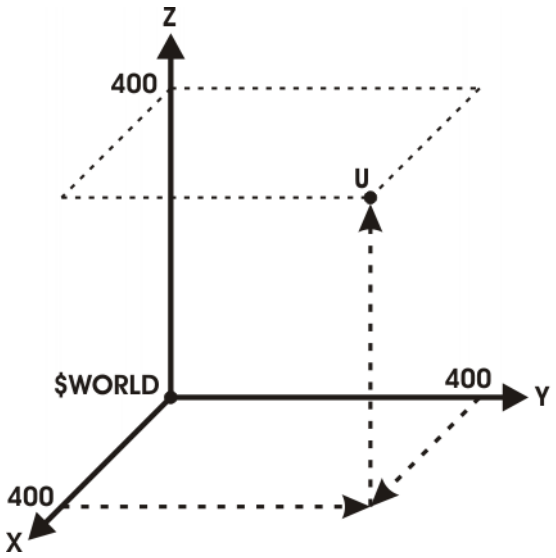

Fig. 3-13: Campo de trabajo cartesiano, origen U

- Dimensiones del campo de trabajo desde el origen

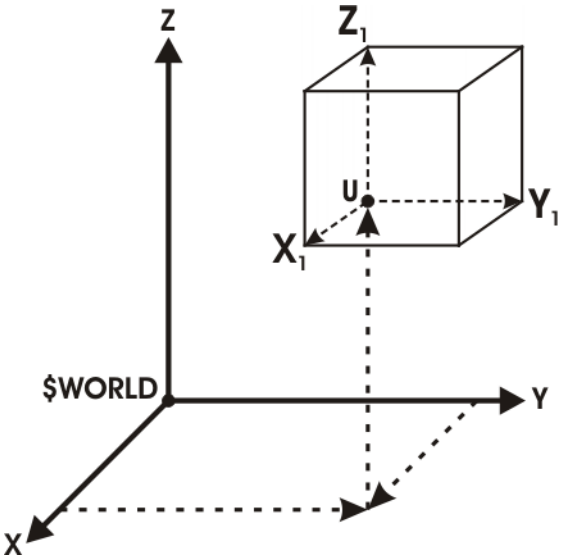

Fig. 3-14: Campo de trabajo cartesiano, medidas

Procedimiento para configurar y utilizar campos de trabajo

Configuración de campos de trabajo específicos del eje

1. Seleccionar en el menú principal la secuencia **Configuración** > **Extras** > **Monitorización del campo de trabajo** > **Configuración**.
   Se abre la ventana **Campos de trabajo cartesianos**.
2. Pulsar **Espec. ejes** para cambiar a la ventana **Campos de trabajo específicos del eje**.

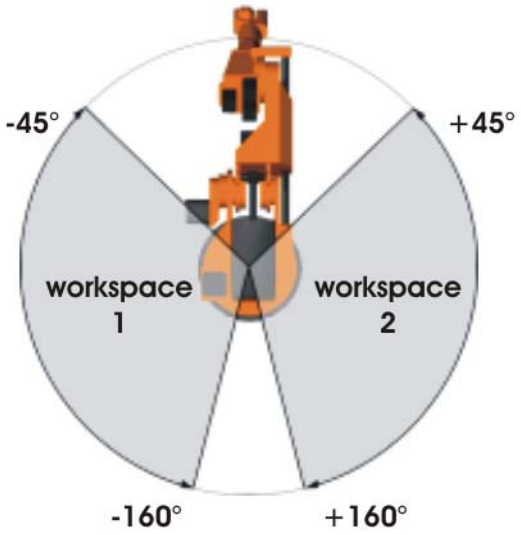

Fig. 3-15: Ejemplo de campos de trabajo específicos del eje para A1

3. Introducir los valores y pulsar **Guardar**.

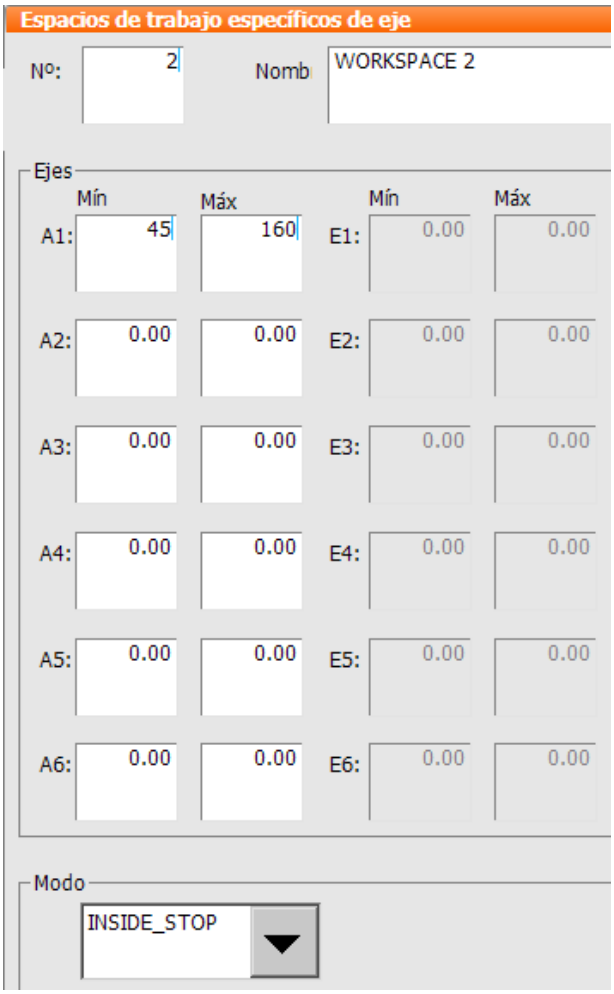

Fig. 3-16: Ejemplo de campo de trabajo específico del eje

4. Pulsar **Señal**. Se abre la ventana **Señales**.

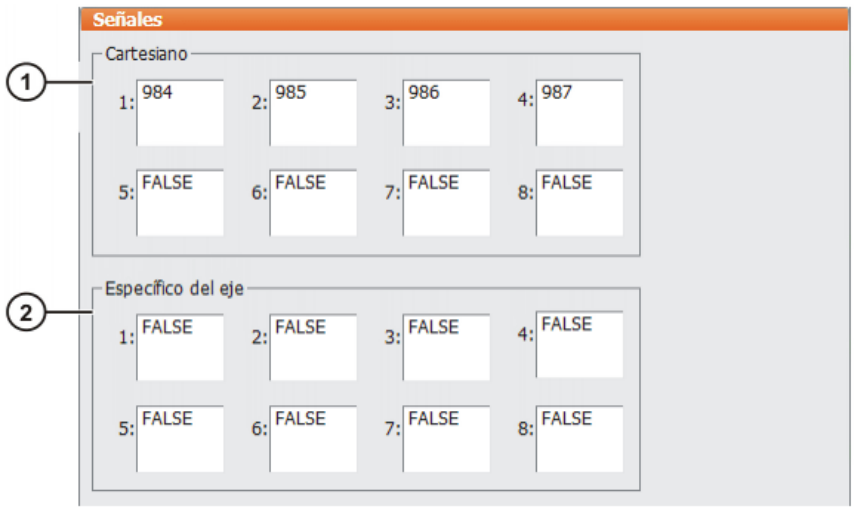

Fig. 3-17: Señales del campo de trabajo

Pos.	Descripción
1	Salidas para el control de los campos de trabajo cartesianos
2	Salidas para el control de los campos de trabajo específicos de los ejes

Si al lesionarse un campo de trabajo no se fija ninguna salida, se debe introducir FALSE.

5. En el grupo **Espec. ejes**: en el número del campo de trabajo introducir la entrada que se debe fijar, si el campo de trabajo está afectado.
6. Pulsar **Guardar**.
7. Cerrar la ventana.

Configuración de los campos de trabajo cartesianos

1. Seleccionar en el menú principal la secuencia **Configuración** > **Extras** > **Monitorización del campo de trabajo** > **Configuración**.
   Se abre la ventana **Campos de trabajo cartesianos**.
2. Introducir los valores y pulsar **Guardar**.
3. Pulsar **Señal**. Se abre la ventana **Señales**.
4. En el grupo **Cartesiano**: en el número del campo de trabajo introducir la entrada que se debe fijar, si el campo de trabajo está afectado.
5. Pulsar **Guardar**.
6. Cerrar la ventana.

Ejemplos de campos de trabajo cartesianos

- Si el punto "P2" se encuentra sobre el origen del campo de trabajo, sólo es necesario averiguar las coordenadas de "P1".

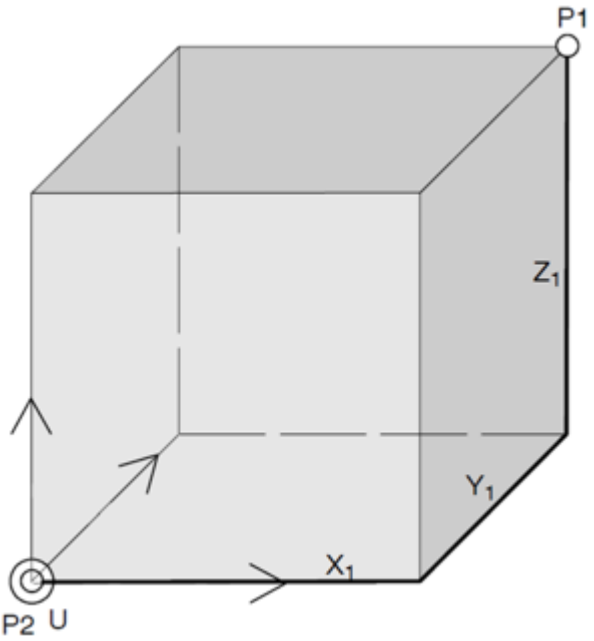

Fig. 3-18: Ejemplo de campo de trabajo cartesiano (P2 se encuentra en el origen)

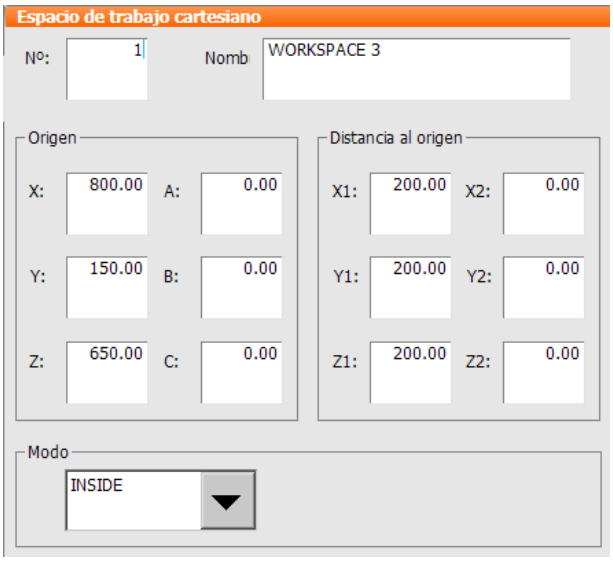

Fig. 3-19: Ejemplo de configuración de campo de trabajo cartesiano (P2 se encuentra en el origen)

- En este ejemplo, el campo de trabajo tiene las medidas x=300 mm, y=250 mm y z=450 mm. Con respecto al sistema de coordenadas universales, dicho campo se encuentra girado 30 grados sobre el eje Y. El origen "U" no se encuentra en el centro del paralelepípedo.

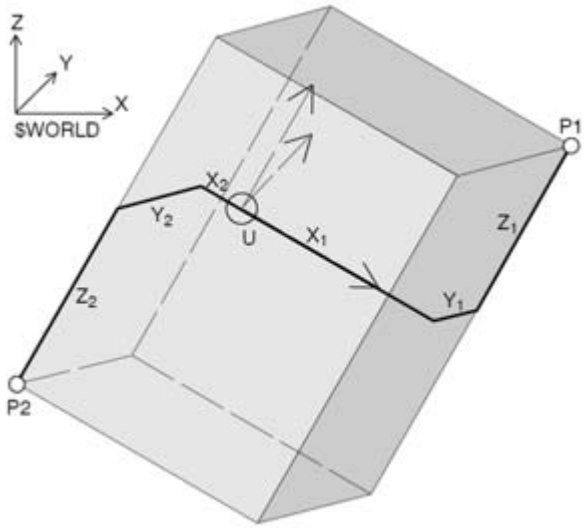

Fig. 3-20: Ejemplo de campo de trabajo cartesiano (girado)

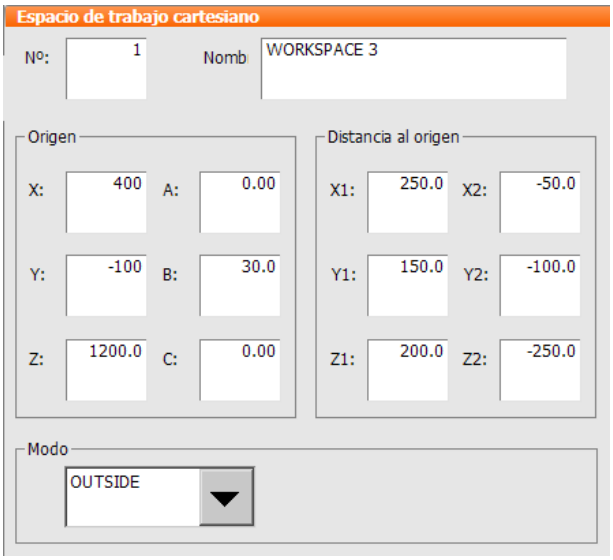

Fig. 3-21: Ejemplo de configuración del campo de trabajo cartesiano (girado)

Trabajos con campos de trabajo

- Campos de trabajo específicos de eje (R1\Mada\$machine.dat)

```
DEFDAT  $MACHINE PUBLIC
...
$AXWORKSPACE[1]={A1_N 0.0,A1_P 0.0,A2_N 0.0,A2_P 0.0,A3_N 0.0,A3_P
0.0,A4_N 0.0,A4_P 0.0,A5_N 0.0,A5_P 0.0,A6_N 0.0,A6_P 0.0,E1_N
0.0,E1_P 0.0,E2_N 0.0,E2_P 0.0,E3_N 0.0,E3_P 0.0,E4_N 0.0,E4_P
0.0,E5_N 0.0,E5_P 0.0,E6_N 0.0,E6_P 0.0,MODE #OFF}
$AXWORKSPACE[2]={A1_N 45.0,A1_P 160.0,A2_N 0.0,A2_P 0.0,A3_N
0.0,A3_P 0.0,A4_N 0.0,A4_P 0.0,A5_N 0.0,A5_P 0.0,A6_N 0.0,A6_P
0.0,E1_N 0.0,E1_P 0.0,E2_N 0.0,E2_P 0.0,E3_N 0.0,E3_P 0.0,E4_N
0.0,E4_P 0.0,E5_N 0.0,E5_P 0.0,E6_N 0.0,E6_P 0.0,MODE #INSIDE_STOP}
```

- Campos de trabajo cartesianos (STEU\Mada\$custom.dat)

```
DEFDAT  $CUSTOM PUBLIC
...
$WORKSPACE[1]={X 400.0,Y -100.0,Z 1200.0,A 0.0,B 30.0,C 0.0,X1
250.0,Y1 150.0,Z1 200.0,X2 -50.0,Y2 -100.0,Z2 -250.0,MODE #OUTSIDE}
$WORKSPACE[2]={X 0.0,Y 0.0,Z 0.0,A 0.0,B 0.0,C 0.0,X1 0.0,Y1 0.0,Z1
0.0,X2 0.0,Y2 0.0,Z2 0.0,MODE #OFF}
```

- Señales de los campos de trabajo (STEU\Mada\$machine.dat)

```
DEFDAT  $MACHINE PUBLIC
...
SIGNAL $WORKSTATE1 $OUT[912]
SIGNAL $WORKSTATE2 $OUT[915]
SIGNAL $WORKSTATE3 $OUT[921]
SIGNAL $WORKSTATE4 FALSE
...
SIGNAL $AXWORKSTATE1 $OUT[712]
SIGNAL $AXWORKSTATE2 $OUT[713]
SIGNAL $AXWORKSTATE3 FALSE
```

- Conectar/desconectar un campo de trabajo mediante KRL

```
DEF myprog( )
...
$WORKSPACE[3].MODE = #INSIDE
...
$WORKSPACE[3].MODE = #OFF
...
$AXWORKSPACE[1].MODE = #OUTSIDE_STOP
...
$AXWORKSPACE[1].MODE = #OFF
```

3.2 Ejercicio: Control del campo de trabajo

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Configuración de campos de trabajo
- Utilización de los distintos modos para campos de trabajo
- Puenteado del control de campo de trabajo

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos teóricos sobre el control de campo de trabajo

Formulación

Tarea parcial 1

1. Configure el campo de trabajo 1 como cubo con una longitud de canto de 200 mm.
2. Transmita una señal cuando se realice la entrada en el campo. Para ello utilice la salida 14.
3. Configure el campo de trabajo 2 como cubo con una longitud de canto de 200 mm.
4. Transmita una señal cuando se realice la entrada en el campo. Para ello utilice la salida 15.
5. Compruebe los dos campos de trabajo y compare la información con la visualización en el panel de mando.

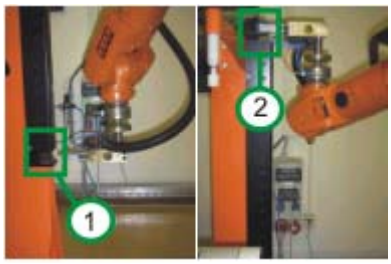

Fig. 3-22

Tarea parcial 2

1. Configure el campo de trabajo 3 como paralelepípedo con una longitud de canto de 400 mm y 200 mm.
2. Bloquee la entrada en este campo de trabajo y transmita una señal. Para ello utilice la salida 16.
3. Compruebe este campo de trabajo y compare la información con la visualización en el panel de mando.
4. Para salir del campo de trabajo, puentéelo con la opción de menú prevista para ello.

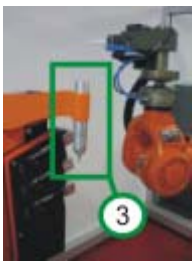

Fig. 3-23

Lo que se debe saber tras el ejercicio:

1. ¿Cuántos campos de trabajo se pueden configurar como máximo?

. . . . . .

. . . . . .

2. ¿Con qué opciones de ajuste para MODE cuenta para configurar un campo de trabajo?

. . . . . .

. . . . . .

3. ¿A qué sistema de coordenadas hace referencia el ORIGEN en la configuración cartesiana de campo de trabajo?

. . . . . .

. . . . . .

4. ¿Qué ventajas ofrece un bloqueo del robot realizado mediante un acoplamiento directo de E/S y el tiempo de control?

. . . . . .

. . . . . .

5. En el caso mencionado anteriormente (pregunta 4), ¿qué desventajas existen cuando se trabaja sin tiempo de control?

. . . . . .

. . . . . .

4 Programación de un mensaje con KRL

4.1 Información general sobre los mensajes definidos por el usuario

Descripción de mensajes definidos por el usuario

Características de la programación de mensajes

- El programador puede programar mensajes propios con KRL.
- Se pueden emitir varios mensajes simultáneamente.
- Los mensajes emitidos se encuentran en una memoria intermedia de mensajes hasta que se vuelven a borrar.
- Los mensajes de observación no se administran en la memoria intermedia de mensajes (principio "fire and forget" [dispare y olvídese]).
- Los mensajes se pueden comprobar o borrar fácilmente, salvo los mensajes de observación.
- En cada mensaje se pueden integrar hasta 3 parámetros.

En la ventana del mensaje de KUKA.HMI se muestra un símbolo para cada mensaje. Los símbolos están asignados a los tipos de mensaje de forma estricta y no pueden ser modificados por el programador.

Se pueden programar los siguientes tipos de mensajes:

Símbolo	Tipo
	**Mensaje de acuse de recibo**
	**Mensaje de estado**
	**Mensaje de observación**
	**Mensaje de espera**

	**Mensaje de diálogo** (se muestra en una ventana emergente propia)

No existen reacciones predefinidas del sistema de robot asociadas a los distintos tipos de mensaje (p. ej. el robot frena o se detiene el programa). Es necesario programar las reacciones deseadas.

Los mensajes se emiten, borran o comprueban mediante funciones estándar predefinidas de KUKA. Para ello se necesitan distintas variables.

Funciones para programar mensajes

- Emitir un mensaje
- Comprobar un mensaje
- Borrar un mensaje
- Emitir un diálogo
- Comprobar un diálogo

Variables complejas para programar mensajes

- Estructura para remitente, número de mensaje, texto del mensaje
- Estructura como marcador de posición para 3 parámetros posibles
- Estructura para el comportamiento general de un mensaje
- Estructura para la rotulación de los botones en mensajes de diálogo

Principio de la programación de mensaje definida por el usuario: variables/estructuras

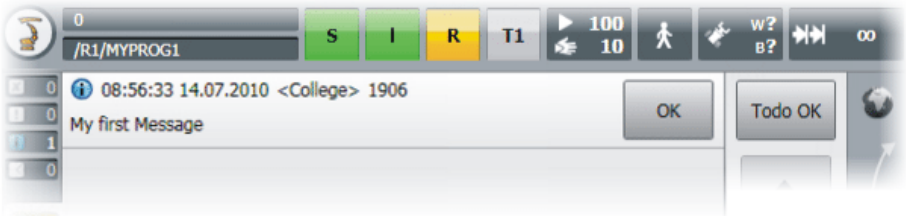

Fig. 4-1: Mensaje de observación

Estructura para remitente, número de mensaje, texto del mensaje

- Estructura KUKA predefinida: `KrlMsg_T`

```
STRUC KrlMsg_T CHAR Modul[24], INT Nr, CHAR Msg_txt[80]
```

-

```
DECL KrlMsg_T mymessage
mymessage = {Modul[ ] "College", Nr 1906, Msg_txt[ ] "My first
Message"}
```

- Remitente: `Modul[ ]"College"`
  - 24 caracteres como máximo
  - El sistema integra el texto del remitente en la visualización entre "< >".
- Número de mensaje: `N.° 1906`
  - Número entero de libre elección.
  - Los números duplicados seleccionados no se detectan.

  Texto del mensaje: `Msg_txt[ ] "My first Message"`
  - 80 caracteres como máximo
  - El texto se muestra en la segunda línea del mensaje.

Cuando se envía un mensaje, se debe elegir el tipo de mensaje:

- Tipo de dato de enumeración `EKrlMsgType`

```
ENUM EKrlMsgType Notify, State, Quit, Waiting
```

-  `#Quit`: da salida a este mensaje como mensaje de confirmación.
-  `#STATE`: da salida a este mensaje como mensaje de estado.
-  `#NOTIFY`: da salida a este mensaje como mensaje de observación.
-  `#WAITING`: da salida a este mensaje como mensaje de espera.

En el texto de un mensaje se debe mostrar el valor de una variable. Por ejemplo, se debe mostrar la cantidad de piezas actual. Para ello, en el texto del mensaje se necesitan los denominados marcadores de posición. La cantidad máxima de marcadores de posición es 3. La visualización es `%1`, `%2` y `%3`.

Por este motivo se necesitan 3 juegos de parámetros. Cada juego de parámetros está compuesto por la estructura KUKA `KrlMsgPar_T`:

```
Enum KrlMsgParType_T Value, Key, Empty
STRUC KrlMsgPar_T KrlMsgParType_T Par_Type, CHAR Par_txt[26], INT
Par_Int, REAL Par_Real, BOOL Par_Bool
```

Utilización de las unidades individuales

- `Par_Type`: clase de parámetro/marcador de posición.
  - `#VALUE`: el parámetro se procesa directamente en la forma transferida en el texto del mensaje (es decir, como String, valor INT, REAL o BOOL).
  - `#KEY`: el parámetro es una palabra clave que se debe buscar en la base de datos de mensajes para cargar el texto correspondiente.
  - `#EMPTY`: el parámetro está en blanco.
- `Par_txt[26]`: el texto o la palabra clave para el parámetro.
- `Par_Int`: se transmite un valor entero como parámetro.
- `Par_Real`: se transmite un valor real como parámetro.
- `Par_Bool`: se transmite un valor booleano como parámetro, se muestra el texto `TRUE` o `FALSE`.

Ejemplos de programa para la transmisión directa de parámetros a los marcadores de posición

El texto del mensaje es `Msg_txt[ ] "Avería en %1"`.

```
DECL KrlMsgPar_T Parameter[3] ; 3 Crear juegos de parámetros
...
Parameter[1] = {Par_Type #VALUE, Par_txt[ ] "Finisher"}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
...
```

Salida de mensaje: **avería en la garra**

Puesto que los parámetros no se suelen describir mediante constantes, las unidades individuales se transmiten con el punto separador.

Ejemplos de programa para la transmisión de parámetros con punto separador a los marcadores de posición

El texto del mensaje es `Msg_txt[ ] "Faltan %1 componentes"`

```
DECL KrlMsgPar_T Parameter[3] ; 3 Crear juegos de parámetros
DECL INT missing_part
...
missing_part = 13
...
Parameter[1] = {Par_Type #VALUE}
Parameter[1].Par_Int = missing_part
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
...
```

Salida de mensaje: **faltan 13 componentes**

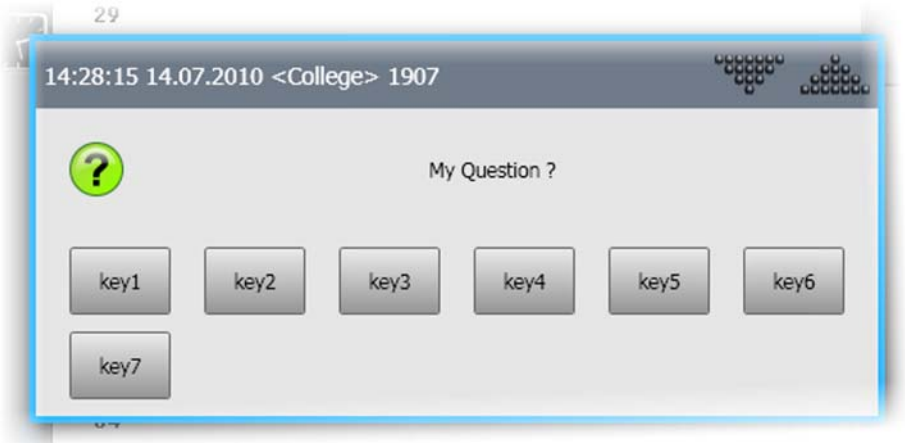

Fig. 4-2: Mensaje de diálogo

Estructura para la asignación de botones en diálogos:

- Estructura KUKA predefinida: `KrlMsgDlgSK_T`

```
Enum KrlMsgParType_T Value, Key, Empty
Struc KrlMsgDlgSK_T KrlMsgParType_T Sk_Type, Char SK_txt[10]
```

- `Sk_Type`: clase de rotulación de botón
  - `#VALUE`: el parámetro se procesa directamente en la forma transferida en el texto del mensaje.
  - `#KEY`: el parámetro es una palabra clave que se debe buscar en la base de datos de mensajes para cargar el texto correspondiente.
  - `#EMPTY`: el botón no tiene asignación.
- `Sk_txt[ ]`: el texto o la palabra clave para el botón.

Ejemplo de programa para la rotulación de 7 botones de un diálogo

```
DECL KRLMSGDLGSK_T Softkey[7] ; 7 Preparar teclas de función
programable posibles
...
softkey[1]={sk_type #value, sk_txt[] "key1"}
softkey[2]={sk_type #value, sk_txt[] "key2"}
softkey[3]={sk_type #value, sk_txt[] "key3"}
softkey[4]={sk_type #value, sk_txt[] "key4"}
softkey[5]={sk_type #value, sk_txt[] "key5"}
softkey[6]={sk_type #value, sk_txt[] "key6"}
softkey[7]={sk_type #value, sk_txt[] "key7"}
...
```

Se pueden asignar como máximo 10 caracteres por botón. En función de los caracteres utilizados, los botones pueden tener distinta anchura.

Cuando se emite un mensaje o un diálogo, se transfieren otras 4 opciones de mensaje. Con estas opciones se puede influir en el avance, el borrado de mensajes y la base de datos Log.

Estructura para opciones de mensaje generales:

- Estructura KUKA predefinida: `KrlMsgOpt_T`

```
STRUC KrlMsgOpt_T   BOOL VL_Stop, BOOL Clear_P_Reset, BOOL
Clear_SAW, BOOL Log_To_DB
```

- `VL_Stop`: `TRUE` desencadena una parada del avance.
  - Por defecto: `TRUE`

- `Clear_P_Reset` : `TRUE` borra todos los mensajes de estado, confirmación y espera cuando se cancela el programa o se restablece.
  - Por defecto: `TRUE`

> Los mensajes de observación sólo se pueden borrar con los menús OK o Todo OK. En el caso de los mensajes de diálogo, siempre es válido lo siguiente: `Clear_P_Reset=TRUE`.

- `Clear_P_SAW`: `TRUE` borra todos los mensajes de estado, confirmación y espera cuando se selecciona un paso mediante el menú correspondiente.
  - Por defecto: `FALSE`
- `Log_To_DB`: `TRUE` hace que este mensaje se registre en la base de datos Log.
  - Por defecto: `FALSE`

Principio de la programación de mensaje definida por el usuario: Funciones

Definición, comprobación y borrado de un mensaje

- **Definición o emisión de un mensaje**

  Con esta función se define un mensaje en el programa KRL. Esto significa que el mensaje correspondiente se procesa en la memoria intermedia de mensajes interna. **Excepción:** los mensajes de observación, que no se administran en la memoria intermedia de mensajes.

  - Funciones integradas para emitir un mensaje

```
DEFFCT INT Set_KrlMsg(Type:IN, MyMessage:OUT, Parameter[ ]:OUT,
Option:OUT)
DECL EKrlMsgType Type
DECL KrlMsg_T MyMessage
DECL KrlMsgPar_T Parameter[ ]
DECL KrlMsgOpt_T Option
```

  - `Tipo`: clase de mensaje (#Notify, #State, #Quit, #Waiting).
  - `MyMessage`: información general de mensaje (remitente, número de mensaje, texto del mensaje)
  - `Parameter[ ]`: los **3** parámetros posibles para los marcadores de posición %1, %2 y %3 (se deben transferir también aunque no se utilicen)
  - `Opción`: Opciones de mensaje generales (parada del procesamiento, inicio de sesión en la base de datos de mensajes, mensaje en caso de borrado implícito de selección de paso o de reinicio del programa)
  - **Valor de retorno de la función:** se denomina "handle" (número de billete). Con este handle se puede controlar si el mensaje se ha podido emitir correctamente; además, el handle es también el número de identificación en la memoria intermedia de mensajes. De este modo se puede comprobar o borrar un mensaje determinado.

```
DEF MyProg( )
DECL INT handle
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
```

  - `handle == -1`: no se ha podido emitir el mensaje (p. ej., porque la memoria intermedia de mensajes está saturada).
  - `handle > 0`: El mensaje se ha emitido correctamente y se gestionará en la memoria intermedia de mensajes con el número de identificación correspondiente.

Los mensajes de observación se procesan de acuerdo con el principio "fire and forget" (dispare y olvídese). En el caso de los mensajes de observación, el retorno siempre es `handle = 0` si el mensaje se ha emitido correctamente.

- **Comprobación de un mensaje**

  Con esta función se puede comprobar si un mensaje determinado con un handle definido existe todavía. También se comprueba si dicho mensaje se encuentra aún en la memoria intermedia de mensajes interna.

  - Funciones integradas para comprobar un mensaje

```
DEFFCT BOOL Exists_KrlMsg(nHandle:IN)
DECL INT nHandle
```

  - `nHandle`: el handle facilitado por la función "`Set_KrlMsg(…)`" para el mensaje.
  - **Valor de retorno de la función:**

```
DEF MyProg( )
DECL INT handle
DECL BOOL present
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
...
present= Exists_KrlMsg(handle)
```

    - `present == TRUE`: este mensaje existe aún en la memoria intermedia de mensajes.
    - `present == FALSE`: este mensaje ya no se encuentra en la memoria intermedia de mensajes (por lo tanto, se ha confirmado o se ha borrado).

- **Borrado de un mensaje**

  Con esta función se puede borrar un mensaje. Esto significa que el mensaje correspondiente se borra de la memoria intermedia de mensajes interna.

  - Funciones integradas para comprobar un mensaje

```
DEFFCT BOOL Clear_KrlMsg(nHandle:IN)
DECL INT nHandle
```

  - `nHandle`: el handle facilitado por la función "`Set_KrlMsg(…)`" para el mensaje.
- **Valor de retorno de la función:**

```
DEF MyProg( )
DECL INT handle
DECL BOOL erase
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
...
erase = Clear_KrlMsg(handle)
```

  - `erase == TRUE`: se ha podido borrar este mensaje.
  - `erase == FALSE`: no se ha podido borrar este mensaje.

Funciones especiales para borrar con la función `Clear_KrlMsg(handle)`:
`Clear_KrlMsg(-1)`: se borran todos los mensajes iniciados por este proceso.
`Clear_KrlMsg(-99)`: se borran todos los mensajes iniciados por el usuario KRL.

Principio de la programación de diálogo definida por el usuario: Funciones

Definición y comprobación de un diálogo

Definición o emisión de un diálogo

Con la función `Set_KrlDlg( )` se emite un mensaje de diálogo. Esto significa que el mensaje se transfiere a la memoria intermedia de mensajes y se muestra en una ventana de mensajes propia con botones.

- Funciones integradas para emitir un diálogo

```
DEFFCT Extfctp Int Set_KrlDlg (MyQuestion:OUT, Parameter[ ]:OUT,
Button[ ]:OUT, Option:OUT)
DECL KrlMsg_T MyQuestion
DECL KrlMsgPar_T Parameter[ ]
DECL KrlMsgDlgSK_T Button[ ]
DECL KrlMsgOpt_T Option
```

- `MyQuestion`: información general de mensaje (remitente, número de mensaje, texto del mensaje)
- `Parameter[ ]`: los **3** parámetros posibles para los marcadores de posición %1, %2 y %3 (se deben transferir también aunque no se utilicen)
- `Button[ ]`: la rotulación para los **7** botones posibles (se deben transferir también aunque no se utilicen)
- `Opción`: opciones de mensaje generales (parada del procesamiento, inicio de sesión en la base de datos de mensajes, mensaje en caso de borrado implícito de selección de paso o de reinicio del programa)
- **Valor de retorno de la función:** handle para el diálogo. Con este handle se puede controlar si el diálogo se ha podido emitir correctamente; además, el handle es también el número de identificación en la memoria intermedia de mensajes.

```
DEF MyProg( )
DECL INT handle
...
handle = Set_KrlDlg(MyQuestion, Parameter[ ], Button[ ], Option)
```

- `handle == -1`: no se ha podido emitir el diálogo (p. ej., porque todavía hay otro diálogo activo al que no se ha respondido aún o porque la memoria intermedia de mensajes está saturada)
- `handle > 0`: el diálogo se ha emitido correctamente y se gestionará en la memoria intermedia de mensajes con el número de identificación correspondiente.

En todos los casos, un diálogo se puede emitir sólo si no hay ningún otro diálogo activo.
La función sólo emite el diálogo. No espera a la respuesta del diálogo.

Comprobación de un diálogo

Con la función `Exists_KrlDlg( )` se puede comprobar si aún existe un diálogo concreto. También se comprueba si dicho diálogo se encuentra aún en la memoria intermedia de mensajes.

- Funciones integradas para comprobar un mensaje

```
DEFFCT BOOL Exists_KrlDlg(INT nHandle:IN, INT Answer:OUT)
DECL INT nHandle, answer
```

- `nHandle`: el handle facilitado por la función "`Set_KrlDlg(…)`" para el diálogo.
- `answer`: confirmación del botón pulsado. De este modo, el botón 1, definido como "`Button[1]`", devuelve el valor 1.

La función no espera a la respuesta del diálogo, sino que sólo explora la memoria intermedia en busca del diálogo que tenga este handle. La consulta en el programa KRL se debe realizar cíclicamente mientras el diálogo responda o se haya borrado de otro modo.

- **Valor de retorno de la función:**

```
DEF MyProg( )
DECL INT handle, answer
DECL BOOL noch_da
...
handle = Set_KrlDlg(MyQuestion, Parameter[ ], Button[ ], Option)
...
noch_da = Exists_KrlDlg(handle, answer)
```

- `present == TRUE`: este diálogo existe aún en la memoria intermedia de mensajes.
- `present == FALSE`: este diálogo ya no se encuentra en la memoria intermedia de mensajes (por lo tanto, se ha respondido).

`answer` se restaura ahora con el valor del botón pulsado. Los valores válidos son del 1 al 7, en función de los números de los botones programados.

4.2 Trabajos con un mensaje de observación

Descripción de un mensaje de observación definido por el usuario

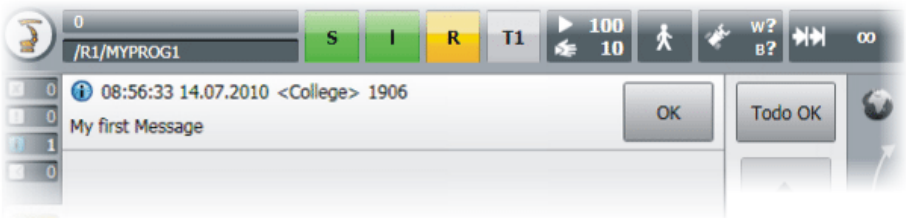

Fig. 4-3: Mensaje de observación

- Los mensajes de observación no se administran en la memoria intermedia de mensajes.
- Los mensajes de observación sólo se pueden borrar con los menús OK o Todo OK.

Función de un mensaje de observación definido por el usuario

- Los mensajes de observación son idóneos para mostrar información general.
- Sólo se puede emitir un mensaje de observación. Se puede comprobar si el mensaje se ha recibido correctamente.
- Puesto que los mensajes de observación no se administran, se pueden emitir **3 millones aproximadamente**

Programación de mensajes de observación definidos por el usuario

1. Cargar un programa principal en el editor.
2. Declarar variables de trabajo para:
   - Remitente, número de mensaje, texto del mensaje (de `KrlMsg_T`)
   - Campos con 3 elementos para el parámetro (de `KrlMsgPar_T`)
   - Opciones de mensaje generales (de `KrlMsgOpt_T`)
   - "Handle" (como `INT`)
3. Inicializar variables de trabajo con los valores deseados.
4. Programar la activación de la función `Set_KrlMsg(…)`.
5. En caso necesario, evaluar el "handle" para saber si la transmisión ha sido satisfactoria.
6. Cerrar y guardar el programa principal.

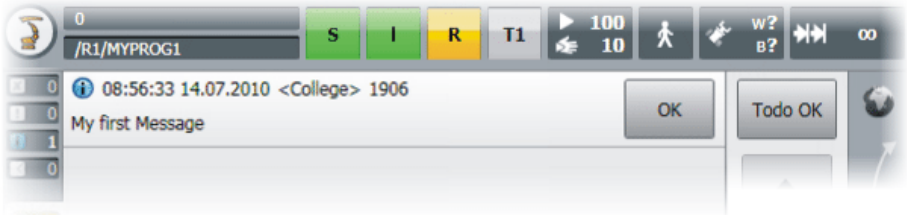

Fig. 4-4: Mensaje de observación

Ejemplo de programación para visualización mencionada anteriormente:

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
...
mymessage={modul[] "College", Nr 1906, msg_txt[] "My first Message"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Los marcadores de posición están en blanco Marcadores de
posición[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#NOTIFY, mymessage, Parameter[ ], Option)
```

4.3 Ejercicio: Programación de un mensaje de observación

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar mensajes de observación propios. •
- Dar salida en mensajes a los parámetros que se desee

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos de la programación de mensajes

Formulación

Tarea parcial 1: Mensaje de observación

1. Cree un mensaje de observación con el texto "Cargador casi vacío, rellenar".
2. Mediante la entrada 13 en el panel de mando se debe visualizar este mensaje.
3. Compruebe su programa según prescripción.

Tarea parcial 2: Mensaje de observación con parámetros

1. Cree un mensaje de observación con el texto "Componente número xxx listo".
2. Mediante la entrada 16 en el panel de mando se debe mostrar este mensaje; el contador de piezas del componente se debe incrementar y se debe mostrar en lugar de xxx.
3. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Cómo se vuelve a borrar un mensaje de observación?

. . . . . .

. . . . . .

2. ¿Qué componente de la estructura del mensaje es responsable de "lanzar" el texto del mensaje?

4.4 Trabajos con un mensaje de estado

Descripción de un mensaje de estado definido por el usuario

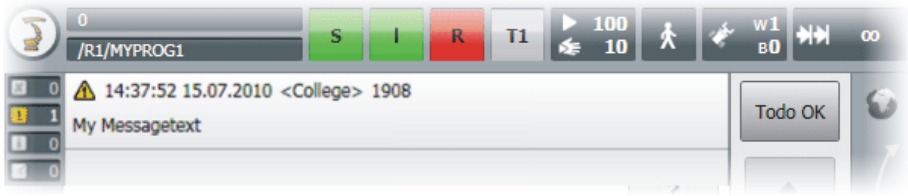

Fig. 4-5: Mensaje de estado

- Los mensajes de estado se administran en la memoria intermedia de mensajes.
- Los mensajes de estado **no** se pueden volver a borrar con el menú "Todo OK".
- Los mensajes de estado se deben volver a borrar en el programa con una función.
- Los mensajes adicionales también se pueden borrar mediante los ajustes de las opciones de mensaje durante el reset del programa, al salir del programa o al seleccionar un paso.

Función de un mensaje de estado definido por el usuario

- Los mensajes de estado son adecuados para indicar un cambio de estado (p. ej. la supresión de una entrada).
- En la memoria intermedia de mensajes se administran 100 mensajes como máximo.
- El programa persiste, por ejemplo, hasta que se vuelva a dar el estado desencadenante.
- El mensaje de estado se vuelve a borrar mediante la función `Clear_KrlMsg( )`.

> No existen reacciones predefinidas del sistema de robot asociadas a los distintos tipos de mensaje (p. ej., el robot frena o se detiene el programa). Es necesario programar las reacciones deseadas.

Programación de mensajes de estado definidos por el usuario

1. Cargar un programa principal en el editor.
2. Declarar variables de trabajo para:
   - Remitente, número de mensaje, texto del mensaje (de `KrlMsg_T`)
   - Campos con 3 elementos para el parámetro (de `KrlMsgPar_T`)
   - Opciones de mensaje generales (de `KrlMsgOpt_T`)
   - "Handle" (como `INT`)
   - Variable para el resultado de la comprobación (como `BOOL`)
   - Variable para el resultado del borrado (como `BOOL`)
3. Inicializar variables de trabajo con los valores deseados.
4. Programar la activación de la función `Set_KrlMsg(…)`.
5. Detener el programa con un bucle hasta que se desaparezca el estado desencadenante.
6. Borrar el mensaje de estado con la activación de la función `Clear_KrlMsg( )`.
7. Cerrar y guardar el programa principal.

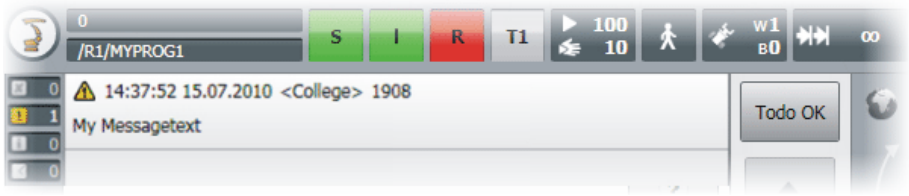

Fig. 4-6: Mensaje de estado

Ejemplo de programación para visualización/mensaje mencionados anteriormente:

El mensaje de estado se desencadena con el estado de la entrada 17 (FALSE). el programa se detiene tras la transmisión del mensaje. El mensaje se borra con el estado de la entrada 17 (TRUE). Después, el programa prosigue.
El mensaje también desaparece cuando se reinicializa el programa o se sale del mismo. Para ello se utiliza el ajuste en las opciones de mensaje Clear_P_Reset TRUE.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present, erase
...
IF $IN[17]==FALSE THEN
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Los marcadores de posición están en blanco Marcadores de
posición[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#STATE, mymessage, Parameter[ ], Option)
ENDIF
erase=FALSE
;Bucle de parada hasta que se haya borrado este mensaje
REPAEAT
IF $IN[17]==TRUE THEN
erase=Clear_KrlMsg(handle) ;Borrar un mensaje
ENDIF
present=Exists_KrlMsg(handle) ;Comprobación adicional
UNTIL NOT(present) or erase
```

4.5 Ejercicio: Programación de un mensaje de estado

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar mensajes de estado propios
- Dar salida en mensajes a los parámetros que se desee

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos de la programación de mensajes

Formulación

Tarea parcial 1: Mensaje de estado

1. Cree un mensaje de estado con el texto "Cargador casi vacío".

2. Mediante la entrada 14 en el panel de mando se debe visualizar este mensaje.
3. Mediante la retirada de la entrada 14 del panel de mando se debe volver a borrar el mensaje.
4. Compruebe su programa según prescripción.

Tarea parcial 2: Mensaje de estado con parámetros

1. Cree un mensaje de estado con el texto "Todavía quedan xxx cubos de yyy en el cargador".
2. Mediante la entrada 15 en el panel de mando se debe visualizar este mensaje.
3. Mediante la retirada de la entrada 15 del panel de mando se debe volver a borrar el mensaje.
4. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Qué significa %2 en el texto del mensaje?

. . . . . .

. . . . . .

4.6 Trabajos con un mensaje de acuse de recibo

Descripción de un mensaje de acuse de recibo definido por el usuario

Fig. 4-7: Mensaje de acuse de recibo

- Los mensajes de acuse de recibo se administran en la memoria intermedia de mensajes.
- Los mensajes de acuse de recibo sólo se pueden borrar con los menús OK o Todo OK.
- Los mensajes de acuse de recibo se pueden volver a borrar también mediante una función en el programa.
- Asimismo, los mensajes de acuse de recibo se pueden borrar mediante los ajustes de las opciones de mensaje durante el reset del programa, al salir del programa o al seleccionar un paso.

Función de un mensaje de acuse de recibo definido por el usuario

- Los mensajes de acuse de recibo son idóneos para mostrar información que el usuario debe conocer.
- En la memoria intermedia de mensajes se administran 100 mensajes como máximo.
- En el caso del mensaje de acuse de recibo (a diferencia del mensaje de observación), se puede comprobar si el usuario lo ha confirmado o no.
- El programa persiste, por ejemplo, hasta que se confirme el mensaje.

> No existen reacciones predefinidas del sistema de robot asociadas a los distintos tipos de mensaje (p. ej. el robot frena o se detiene el programa). Es necesario programar las reacciones deseadas.

Programación de mensajes de acuse de recibo definidos por el usuario

1. Cargar un programa principal en el editor.
2. Declarar variables de trabajo para:
   - Remitente, número de mensaje, texto del mensaje (de `KrlMsg_T`)
   - Campos con 3 elementos para el parámetro (de `KrlMsgPar_T`)
   - Opciones de mensaje generales (de `KrlMsgOpt_T`)
   - "Handle" (como `INT`)
   - Variable para el resultado de la comprobación (como `BOOL`)
3. Inicializar variables de trabajo con los valores deseados.
4. Programar la activación de la función `Set_KrlMsg(…)`.
5. Detener el programa con un bucle.
6. Activar la función `Exists_KrlMsg(...)` para comprobar si el usuario ya ha confirmado el mensaje; en caso afirmativo, se habrá abandonado el bucle mencionado anteriormente.
7. Cerrar y guardar el programa principal.

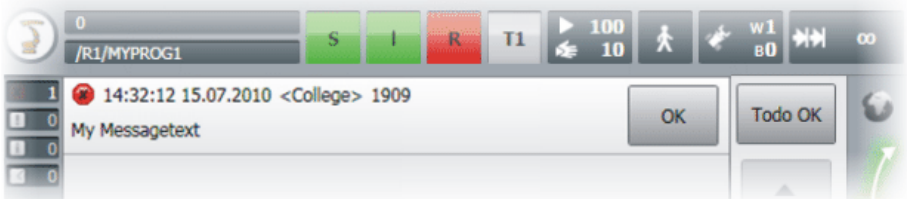

Fig. 4-8: Mensaje de acuse de recibo

Ejemplo de programación para visualización/mensaje mencionados anteriormente:

el programa se detiene tras la transmisión del mensaje. El mensaje se borra accionando el botón OK o Todo OK. Después, el programa prosigue.
El mensaje también desaparece cuando se reinicializa el programa o se sale del mismo. Para ello se utiliza el ajuste en las opciones de mensaje `Clear_P_Reset TRUE`.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present
...
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Los marcadores de posición están en blanco Marcadores de
posición[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#QUIT, mymessage, Parameter[ ], Option)

;Bucle de parada hasta que se haya borrado este mensaje
REPAEAT
present=Exists_KrlMsg(handle)
UNTIL NOT(present)
```

4.7 Ejercicio: Programación de mensajes de acuse de recibo

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar mensajes de confirmación propios.
- Dar salida en mensajes a los parámetros que se desee

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos de la programación de mensajes

Formulación

Tarea parcial 1: Mensaje de confirmación

1. Cree un mensaje de confirmación con el texto "Confirmar avería Vacío no alcanzado".
2. Mediante la entrada 15 en el panel de mando se debe visualizar este mensaje.
3. Compruebe su programa según prescripción.

Tarea parcial 2: mensaje de estado con mensaje de confirmación

1. Cree un mensaje de estado con el texto "Avería Vacío no alcanzado".
2. Mediante la entrada 18 en el panel de mando se debe visualizar este mensaje.
3. Tras resetear la entrada, se debe anular el mensaje de estado y se debe mostrar el mensaje de confirmación programado en la tarea parcial 1.
4. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿xxx?

. . . . . .

. . . . . .

4.8 Trabajos con un mensaje de espera

Descripción de un mensaje de espera definido por el usuario

Fig. 4-9: Mensaje de espera

- Los mensajes de espera se administran en la memoria intermedia de mensajes.
- Los mensajes de espera se pueden volver a borrar con el menú "Simular".
- Los mensajes de espera **no** se pueden volver a borrar con el menú "Todo OK".
- Los mensajes de espera también se pueden borrar mediante los ajustes de las opciones de mensaje durante el reset del programa, al salir del programa o al seleccionar un paso.

Función de un mensaje de espera definido por el usuario

- Los mensajes de espera son adecuados para esperar a un estado y mostrar el símbolo de espera mientras tanto.
- En la memoria intermedia de mensajes se administran 100 mensajes como máximo.
- El programa persiste, por ejemplo, hasta que se vuelva a dar el estado esperado.
- El mensaje de espera se vuelve a borrar mediante la función `Clear_KrlMsg( )`.

Programación de mensajes de espera definidos por el usuario

1. Cargar un programa principal en el editor.
2. Declarar variables de trabajo para:
   - Remitente, número de mensaje, texto del mensaje (de `KrlMsg_T`)
   - Campos con 3 elementos para el parámetro (de `KrlMsgPar_T`)
   - Opciones de mensaje generales (de `KrlMsgOpt_T`)
   - "Handle" (como `INT`)
   - Variable para el resultado de la comprobación (como `BOOL`)
   - Variable para el resultado del borrado (como `BOOL`)
3. Inicializar variables de trabajo con los valores deseados.
4. Programar la activación de la función `Set_KrlMsg(…)`.
5. Detener el programa con un bucle hasta que se dé el estado esperado o hasta que se haya borrado el mensaje mediante el botón Simular.
6. Borrar el mensaje de espera con la activación de la función `Clear_KrlMsg( )`.
7. Cerrar y guardar el programa principal.

Fig. 4-10: Mensaje de espera

Ejemplo de programación para visualización/mensaje mencionados anteriormente:

el programa se detiene tras la transmisión del mensaje. El mensaje se borra con el estado de la entrada 17 (TRUE). Después, el programa prosigue.
El mensaje también desaparece cuando se reinicializa el programa o se sale del mismo. Para ello se utiliza el ajuste en las opciones de mensaje Clear_P_Reset TRUE.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present, erase
...
IF $IN[17]==FALSE THEN
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Los marcadores de posición están en blanco Marcadores de
posición[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#WAITING, mymessage, Parameter[ ], Option)
ENDIF
erase=FALSE
;Bucle de parada hasta que se haya borrado este mensaje
REPAEAT
IF $IN[17]==TRUE THEN
erase=Clear_KrlMsg(handle) ;Borrar un mensaje
ENDIF
present=Exists_KrlMsg(handle) ;Se puede haber borrado mediante
Simular
UNTIL NOT(present) or erase
```

4.9 Ejercicio: Programación de mensajes de espera

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar mensajes de espera propios
- Dar salida en mensajes a los parámetros que se desee

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos de la programación de mensajes

Formulación

Tarea parcial 1: Mensaje de espera

1. Cree un mensaje de espera con el texto "Esperar a entrada de usuario".
2. Ponga 4 componentes distintos a disposición y asigne "FIN" a la tecla de función programable 5.
3. Tras la selección del componente, emita un mensaje de observación con el texto "Componente xxx seleccionado". Para ello puede utilizar módulos básicos ya existentes.
4. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Cuál es la diferencia entre un mensaje "STATE" y otro "WAITING"?

. . . . . .

4.10 Trabajos con un mensaje de diálogo

Descripción de un mensaje de diálogo definido por el usuario

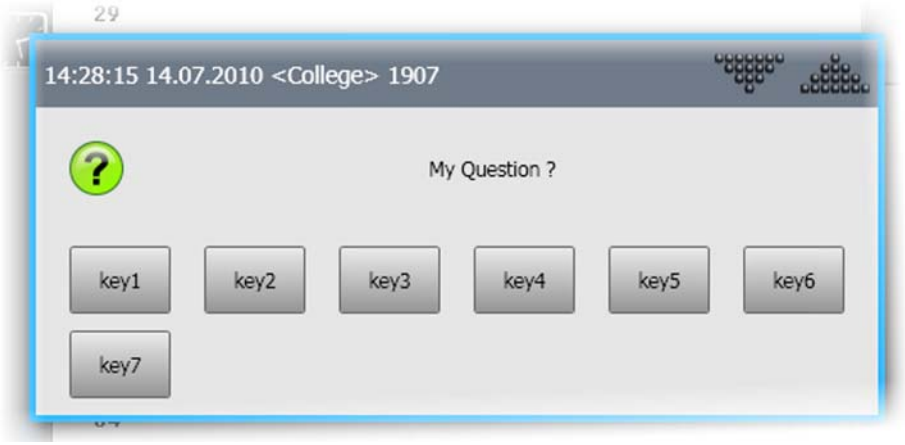

Fig. 4-11: Mensaje de diálogo

- En todos los casos, un diálogo se puede emitir sólo si no hay ningún otro diálogo activo.
- Los mensajes de diálogo se pueden borrar con una tecla de función programable cuya rotulación haya sido definida por el programador.
- Es posible definir hasta 7 teclas de función programables.

Función de un mensaje de diálogo definido por el usuario

- Los mensajes de diálogo son idóneos para mostrar preguntas a las que el usuario debe contestar.
- Con la función `Set_KrlDlg( )` se emite un mensaje de diálogo.
- La función sólo emite el diálogo.
- No espera a la respuesta del diálogo.
- Con la función `Exists_KrlDlg( )` se puede comprobar si aún existe un diálogo concreto.
- Esta función tampoco espera a la respuesta del diálogo, sino que sólo explora la memoria intermedia en busca del diálogo que tenga este handle.
- La consulta en el programa KRL se debe realizar cíclicamente mientras el diálogo responda o se haya borrado de otro modo.
- Se puede conseguir que el resto del flujo de programa dependa de la tecla de función programable seleccionada por el usuario.

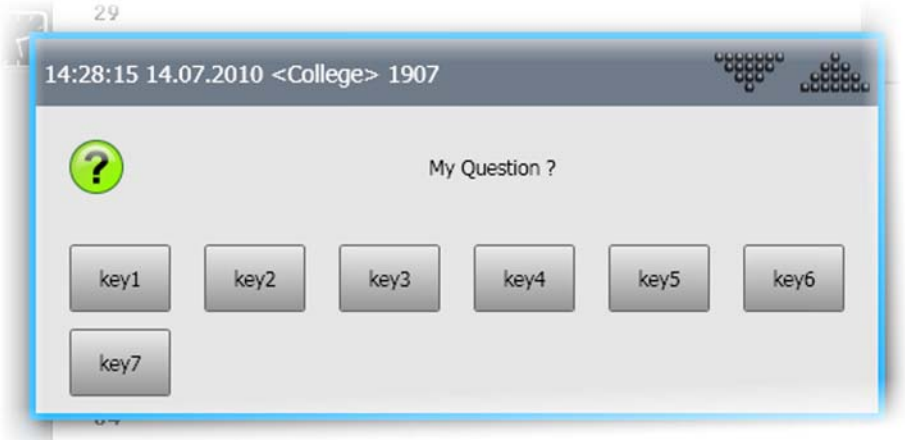

Fig. 4-12: Mensaje de diálogo

Evaluación de botones

- Declaración e inicialización de botones

```
DECL KRLMSGDLGSK_T Softkey[7] ; 7 Preparar teclas de función
programable posibles
softkey[1]={sk_type #value, sk_txt[] "key1"}
softkey[2]={sk_type #value, sk_txt[] "key2"}
softkey[3]={sk_type #value, sk_txt[] "key3"}
softkey[4]={sk_type #value, sk_txt[] "key4"}
softkey[5]={sk_type #value, sk_txt[] "key5"}
softkey[6]={sk_type #value, sk_txt[] "key6"}
softkey[7]={sk_type #value, sk_txt[] "key7"}
```

- Evaluación mediante `Exists_KrlDlg()`: el botón creado en el índice 4 retorna también 4.

```
; La tecla de función programable 4 retorna 4
softkey[4]={sk_type #value, sk_txt[] "key4"}
```

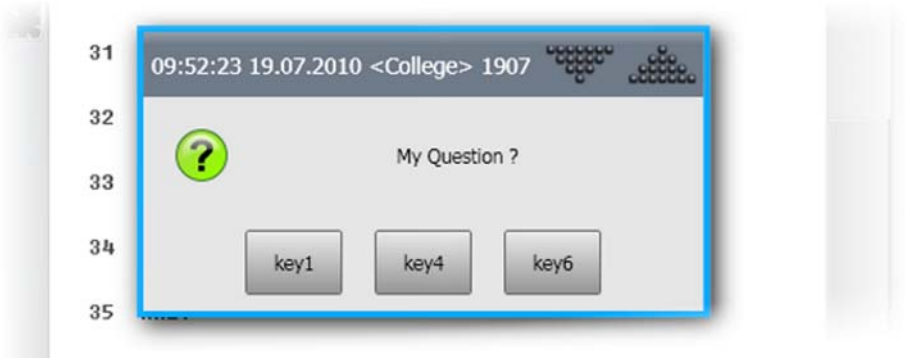

Fig. 4-13: Mensaje de diálogo con 3 botones

Si no se programan todos los botones o si existen espacios en blanco (n.º 1, 4, 6), los botones se agruparán. Si sólo se utilizan los botones 1, 4 y 6, únicamente son posibles las respuestas 1, 4 y 6.

Programación de mensajes de diálogo definidos por el usuario

1. Cargar un programa principal en el editor.
2. Declarar variables de trabajo para:
   - Remitente, número de mensaje, texto del mensaje (de `KrlMsg_T`)
   - Campos con 3 elementos para el parámetro (de `KrlMsgPar_T`)
   - 7 botones posibles (de `KrlMsgDlgSK_T`)
   - Opciones de mensaje generales (de `KrlMsgOpt_T`)
   - "Handle" (como `INT`)
   - Variable para el resultado de la comprobación (como `BOOL`)
   - Variable para el resultado de la respuesta cuyo botón se haya pulsado (como `INT`)
3. Inicializar variables de trabajo con los valores deseados.
4. Programar la activación de la función `Set_KrlDlg(…)`.
5. Detener el programa con un bucle hasta que se haya respondido al diálogo.
6. Evaluar el mensaje de diálogo con la activación de la función `Exists_KrlDlg( )`.
7. Planificar y programar más ramificaciones en el programa.
8. Cerrar y guardar el programa principal.

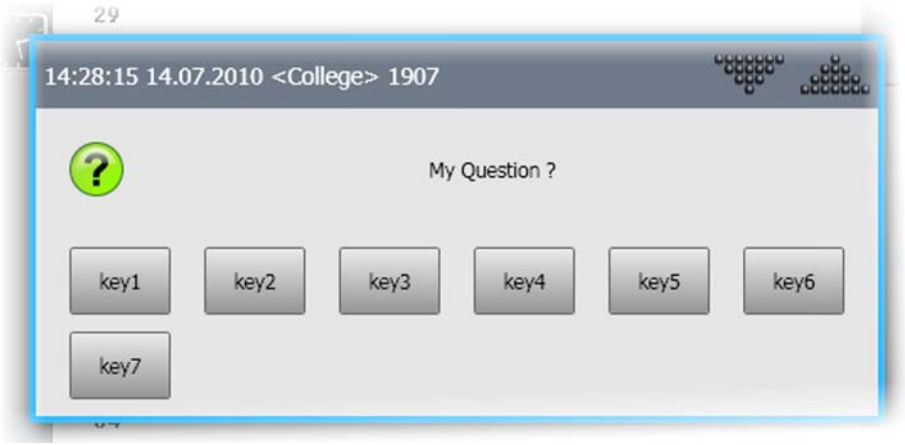

Fig. 4-14: Mensaje de diálogo

Ejemplo de programación para visualización/mensaje mencionados anteriormente:

el programa se detiene tras la transmisión del diálogo. El mensaje se borra tras la respuesta. Después, el programa prosigue. A continuación, se programa un distribuidor.
El mensaje también desaparece cuando se reinicializa el programa o se sale del mismo. Para ello se utiliza el ajuste en las opciones de mensaje `Clear_P_Reset TRUE`.

```
DECL KRLMSG_T myQuestion
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGDLGSK_T Softkey[7] ;7 Preparar teclas de función
programable posibles
DECL KRLMSGOPT_T Option
DECL INT handle, answer
DECL BOOL present
...

myQuestion={modul[] "College", Nr 1909, msg_txt[] "My Questiontext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Los marcadores de posición están en blanco Marcadores de
posición[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
softkey[1]={sk_type #value, sk_txt[] "key1"} ; Botón 1
softkey[2]={sk_type #value, sk_txt[] "key2"} ; Botón 2
softkey[3]={sk_type #value, sk_txt[] "key3"} ; Botón 3
softkey[4]={sk_type #value, sk_txt[] "key4"} ; Botón 4
softkey[5]={sk_type #value, sk_txt[] "key5"} ; Botón 5
softkey[6]={sk_type #value, sk_txt[] "key6"} ; Botón 6
softkey[7]={sk_type #value, sk_txt[] "key7"} ; Botón 7
...
handle = Set_KrlMsg(#STATE, mymessage, Parameter[ ], Option)
ENDIF
erase=FALSE
;Bucle de parada hasta que se haya borrado este mensaje
REPAEAT
IF $IN[17]==TRUE THEN
erase=Clear_KrlMsg(handle) ;Borrar un mensaje
ENDIF
present=Exists_KrlMsg(handle) ;Comprobación adicional
UNTIL NOT(present) or erase
```

```
...; Emitir un diálogo
handle = Set_KrlDlg(myQuestion, Parameter[ ],Softkey[ ], Option)
answer=0
REPEAT ; Bucle de parada hasta que se haya contestado a este diálogo
present = exists_KrlDlg(handle ,answer) ; El sistema describe la
respuesta
UNTIL NOT(present)
...
SWITCH answer
CASE 1 ; Botón 1
; Acción con botón 1
...
CASE 2 ; Botón 2
; Acción con botón 2
...
...
CASE 7 ; Botón 7
; Acción con botón 7
ENDSWITCH
...
```

4.11 Ejercicio: Programación de un diálogo

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar mensajes de observación, estado y confirmación propios. •
- Programar preguntas de diálogo propias.
- Dar salida en mensajes a los parámetros que se desee

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos de la programación de mensajes

Formulación

Tarea parcial 1: Mensaje de diálogo

1. Cree un mensaje de diálogo con el texto "Seleccione un nuevo componente".
2. Ponga 4 componentes distintos a disposición y asigne "FIN" a la tecla de función programable 5.
3. Tras la selección del componente, emita un mensaje de observación con el texto "Componente xxx seleccionado". Para ello puede utilizar módulos básicos ya existentes.
4. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Cómo se rotulan las teclas de función programables en el diálogo?

. . . . . .

. . . . . .

5 Programación de interrupción

5.1 Programación de rutinas de interrupción

Descripción de rutinas de interrupción

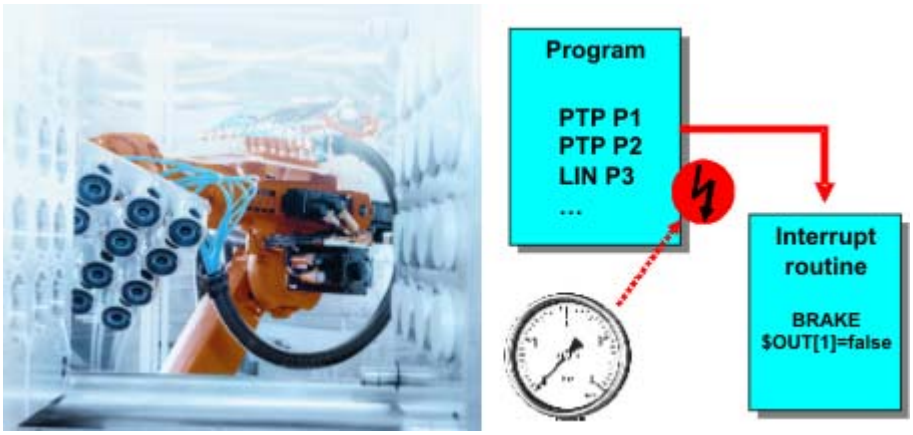

Fig. 5-1: Trabajos con rutinas de interrupción

- Si se da un evento determinado, por ejemplo una entrada, la unidad de control interrumpe el programa actual y ejecuta un subprograma definido.
- Un subprograma activado por una interrupción se denomina programa de interrupción.
- Puede haber como máximo 32 interrupciones declaradas al mismo tiempo.
- Simultáneamente pueden estar activas un máximo de 16 interrupciones.

Pasos importantes cuando se utiliza la interrupción

- Declaración de interrupción
- Conectar/desconectar o bloquear/liberar la interrupción
- Detener el robot dado el caso
- Rechazar la planificación de trayectoria actual y recorrer una nueva trayectoria dado el caso

Principio de la declaración de interrupción

Aspectos generales de la declaración de interrupciones

- Si se da un suceso determinado, por ejemplo una entrada, la unidad de control interrumpe el programa actual y ejecuta un subprograma predefinido.
- El evento y el subprograma se definen con `INTERRUPT ... DECL ... WHEN ... DO ...`.
-

La declaración de interrupción es una instrucción. Debe encontrarse en la sección de instrucciones del programa y no puede estar en la sección de declaraciones.

-

Después de la declaración está desactivada una interrupción. La interrupción se debe activar antes de que se pueda reaccionar al evento definido.

Sintaxis de la declaración de interrupción

-

```
<GLOBAL> INTERRUPT DECL Prio WHEN Evento DO Programa de interrupción
```

- Global
  - Una interrupción se reconoce a partir del nivel en el cual está declarada.

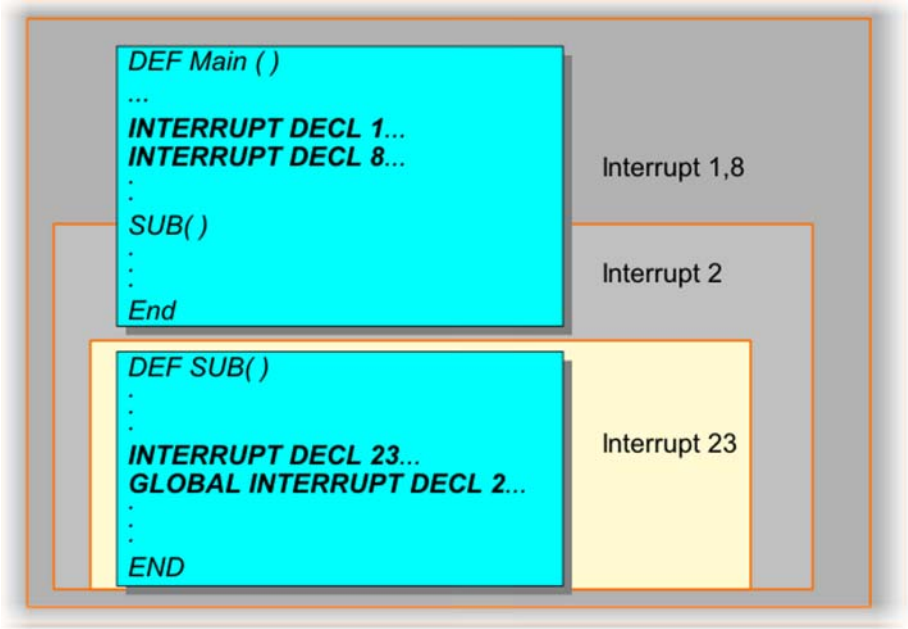

Fig. 5-2: Validez de la interrupción

  - Una interrupción declarada en un subprograma no es conocida en el programa principal (aquí, interrupción 23).
  - Una interrupción con la palabra clave GLOBAL antepuesta en la declaración es conocida también en los niveles superiores (aquí, interrupción 2).
-

```
<GLOBAL> INTERRUPT DECL Prio WHEN Evento DO Subprograma
```

- *Prio*: prioridad
  - Están disponibles las prioridades 1, 2, 4 - 39 y 81 -128.
  - Las prioridades 3 y 40 - 80 están reservadas para el uso por parte del sistema.
  - Puede que la interrupción 19 esté preasignada para el test de frenos.
  - Si se producen simultáneamente varias interrupciones, se procesará en primer lugar la que tenga la prioridad más alta y a continuación las de prioridades más bajas. **(1 = prioridad más alta)**

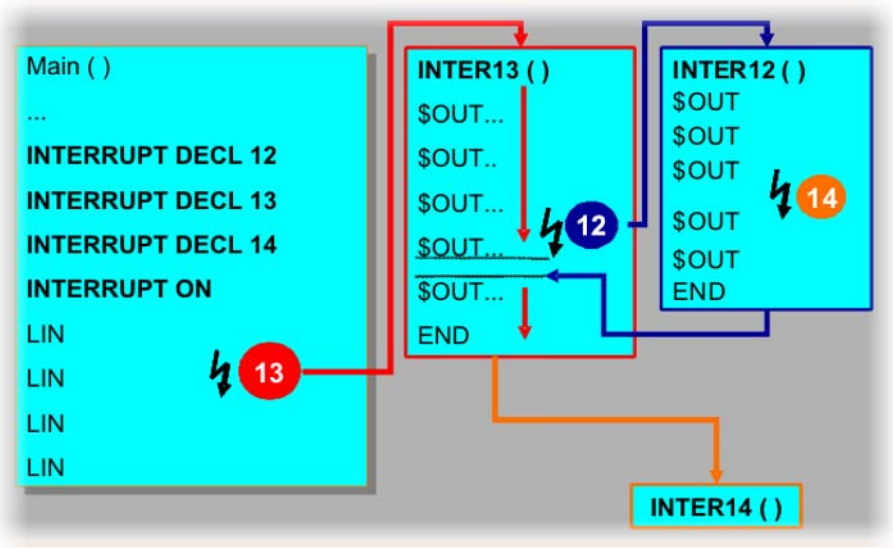

Fig. 5-3: Prioridades de las interrupciones

-

```
<GLOBAL> INTERRUPT DECL Prio WHEN Evento DO Subprograma
```

- *Evento*: evento por el que se debe producir la interrupción.

Este evento es reconocido por un flanco cuando se produce (disparo por flancos).

- *Programa de interrupción*
  - Nombre del programa de interrupción que debe ejecutarse.
  - Este subprograma se denomina programa de interrupción.
  - Las variables de duración temporal no se pueden transmitir como parámetros al programa de interrupción.
  - Se permiten variables declaradas en una lista de datos.
- Ejemplo: Declaración de una interrupción

```
INTERRUPT DECL 23 WHEN $IN[12]==TRUE DO INTERRUPT_PROG(20,VALUE)
```

  - Ninguna interrupción global
  - Prioridad: 23
  - Suceso: entrada 12 flanco positivo
  - Programa de interrupción: INTERRUPT_PROG(20,VALUE)
  -

Después de la declaración está desactivada una interrupción. La interrupción se debe activar antes de que se pueda reaccionar al evento definido.

Descripción de la conexión/desconexión/bloqueo/liberación de la interrupción

Tras declarar una interrupción, se debe activar.

Opciones con la instrucción `INTERRUPT ...`

- Activa una interrupción.
- Desactiva una interrupción.
- Bloquea una interrupción.
- Desbloquea una interrupción.

Sintaxis

- `INTERRUPT` *Acción* <*Número*>

  Acción

  - `ON`: Activa una interrupción.
  - `OFF`: Desactiva una interrupción.
  - `DISABLE`: Bloquea una interrupción activada.
  - `ENABLE`: Desbloquea una interrupción bloqueada.

  Número

  - Número (=prioridad) de la interrupción a la que hace referencia la *acción*.
  - El *número* se puede omitir.
    En este caso `ON` o `OFF` hacen referencia a todas las interrupciones declaradas, `DISABLE` o `ENABLE` a todas las interrupciones activas.

Activar y desactivar una interrupción.

```
INTERRUPT DECL 20 WHEN $IN[22]==TRUE DO SAVE_POS( )
...
INTERRUPT ON 20
;Se detecta la interrupción y se ejecuta (flanco positivo)
...
INTERRUPT OFF 20 ; La interrupción está desactivada
```

- En este caso, la interrupción se desencadena con el cambio de estado, p. ej., para $IN[x]==TRUE, con el cambio de FALSE a TRUE. El estado no debe existir ya cuando se dé INTERRUPT ON; en caso contrario, no se podrá desencadenar la interrupción.
- En este caso, además se debe tener en cuenta lo siguiente: El cambio de estado se debe producir como muy pronto un ciclo de interpolación después de INTERRUPT ON.

  (Esto se puede conseguir programando WAIT SEC 0.012 tras INTERRUPT ON. Si no se desea que se produzca una parada del procesamiento en avance, además se puede programar CONTINUE antes de WAIT SEC.)

  El motivo es que INTERRUPT ON necesita un ciclo de interpolación (= 12 ms) para que la interrupción se active realmente. Si se cambia el estado antes, la interrupción no podrá reconocer el cambio.

Activar y desactivar una interrupción: rebotes de teclas

Si existe el riesgo de que una interrupción se desencadene, por error, dos veces debido a la sensibilidad de un sensor ("rebotes de teclas"), puede evitarlo desactivando la interrupción en la primera línea del programa de interrupción.
Sin embargo, se ha de tener presente que aquí tampoco se detectará, en ningún momento, una interrupción real durante el procesamiento de la interrupción. Antes de retroceder, si tiene que continuar estando activa la interrupción, se debe volver a conectar.

Interrupción: bloqueo y liberación

```
INTERRUPT DECL 21 WHEN $IN[25]==TRUE DO INTERRUPT_PROG( )
...
INTERRUPT ON 21
;Se detecta la interrupción y se ejecuta de inmediato (flanco
positivo)
...
INTERRUPT DISABLE 21
;Se detecta la interrupción y se guarda, pero no se ejecuta (flanco
positivo)
...
INTERRUPT ENABLE 21
; Ahora se ejecutan las interrupciones guardadas
...
INTERRUPT OFF 21 ; La interrupción está desactivada
...
```

Una interrupción bloqueada es detectada y guardada. La ejecución tiene lugar directamente tras la liberación de la interrupción. En este caso se debe prestar atención a la ausencia de colisiones cuando se realicen movimientos.

Descripción del frenado del robot o de la cancelación del movimiento actual mediante una rutina de interrupción

Frenado del robot

- Un robot se debe detener inmediatamente después de que se produzca un evento.
- Para detenerlo se cuenta con dos rampas de frenado (STOP 1 y STOP 2)
- El programa de interrupción no se reanuda hasta que el robot se haya parado.
- En cuanto finaliza el programa de interrupción, se reanuda el movimiento iniciado por el robot.
- Sintaxis:
  - `BRAKE`: STOP 2
  - `BRAKE F`: STOP 1

BRAKE sólo se puede utilizar en un programa de interrupción.

Movimientos y rutinas de interrupción

- El robot se desplaza mientras en paralelo se ejecuta la rutina de interrupción.

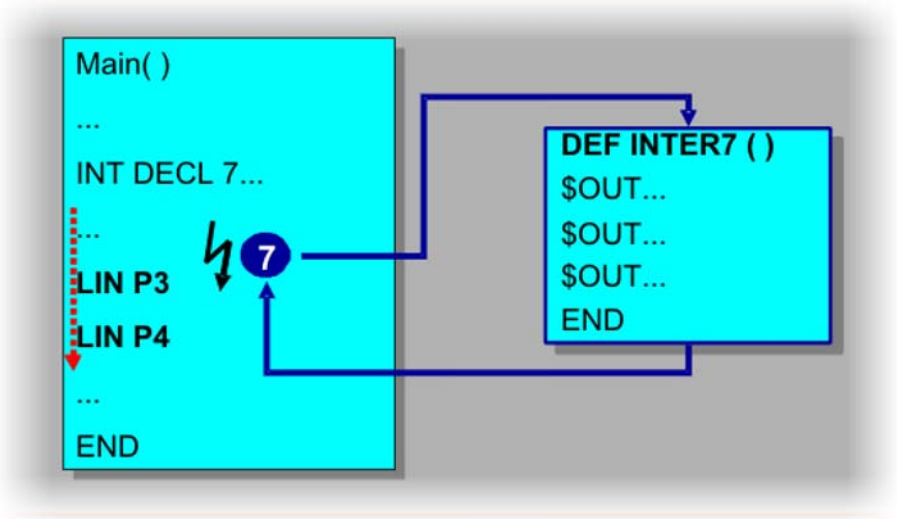

Fig. 5-4: Tramitación de rutinas de interrupción

Si la ejecución de la rutina de interrupción es más breve que la trayectoria planificada en el programa principal, el robot puede avanzar sin interrupción. Si para la rutina de interrupción se necesita más tiempo que para la trayectoria planificada, el robot se detiene al final de su planificación de trayectoria y continúa en cuanto se haya ejecutado toda la rutina.

- No se permiten formularios en línea para la inicialización (`INI`) ni para los movimientos (p. ej. `PTP` o `LIN`...). Durante la ejecución se producen mensajes de error.
- El robot se detiene con `BRAKE` y reanuda la marcha al final de la rutina de interrupción con la trayectoria realizada en el programa principal.
- El robot se detiene con `BRAKE` y pasa a la rutina de interrupción. Una vez finalizada la rutina de interrupción, la trayectoria se reanuda desde el programa principal.

Se debe evitar que se produzcan colisiones.
Si no se respeta esta medida, pueden ocasionarse importantes daños materiales, lesiones graves e incluso la muerte.

- El robot se detiene con `BRAKE` y debe realizar una nueva trayectoria una vez finalizada la rutina de interrupción. Para ello se puede utilizar la instrucción `RESUME`.
- El robot se detiene con `BRAKE` y pasa a la rutina de interrupción. Una vez finalizada la rutina de interrupción, no se debe continuar con la primera, sino que se debe realizar una nueva planificación de trayectoria. Para ello se puede utilizar también la instrucción `RESUME`.

Puesto que no se puede estimar con exactitud cuándo se desencadenará la interrupción, debe ser posible desplazarse sin colisionar dentro de la rutina de interrupción y de la marcha siguiente en todas las posiciones posibles de la marcha actual del robot.
Si no se respeta esta medida, pueden ocasionarse importantes daños materiales, lesiones graves e incluso la muerte.

Cancelación del movimiento actual mediante `RESUME`.

- `RESUME` cancela todos los programas de interrupción y subprogramas en curso hasta el nivel en el cual se encuentra declarada la interrupción actual.

- En el momento de la instrucción RESUME, el puntero de movimiento de avance no debe estar en el mismo nivel en el que se declaró la interrupción, sino en un nivel inferior como mínimo.
- RESUME sólo puede estar presente en programas de interrupción.
- En cuanto se declare una interrupción como GLOBAL, ya **no** se podrá utilizar RESUME en la rutina de interrupción.
- Las modificaciones de las variables $BASE en el programa de interrupción sólo son vigentes allí.
- El procesamiento en avance, es decir, la variable $ADVANCE, no se puede modificar en el programa de interrupción.
- Los movimientos que deben cancelarse con BRAKE y RESUME en principio se deben programar en un subprograma.
- El comportamiento de la unidad de control del robot tras RESUME depende de la instrucción de movimiento siguiente:
  - Instrucción PTP: la marcha se realiza como movimiento PTP.
  - Instrucción LIN: la marcha se realiza como movimiento LIN.
  - Instrucción CIRC: la marcha se realiza siempre como movimiento LIN.

    Tras RESUME, el robot no se encuentra en el punto de inicio original del movimiento CIRC. Esto haría que el movimiento se realizará de forma distinta a la planificada originalmente, con el consiguiente potencial de peligro, que es considerable, en especial en el caso de los movimientos CIRC.

⚠ ADVERTENCIA Si la primera instrucción de movimiento después de RESUME es un movimiento CIRC, se realizará siempre como LIN. Este comportamiento se debe tener en cuenta cuando se programen instrucciones RESUME. El robot debe poder avanzar al punto de destino del movimiento CIRC como LIN sin peligro desde cualquier posición posible en la que se pueda encontrar RESUME.
Si no se respeta esta medida, pueden ocasionarse importantes daños materiales, lesiones graves e incluso la muerte.

- Variables de sistema útiles en caso de parada exacta:

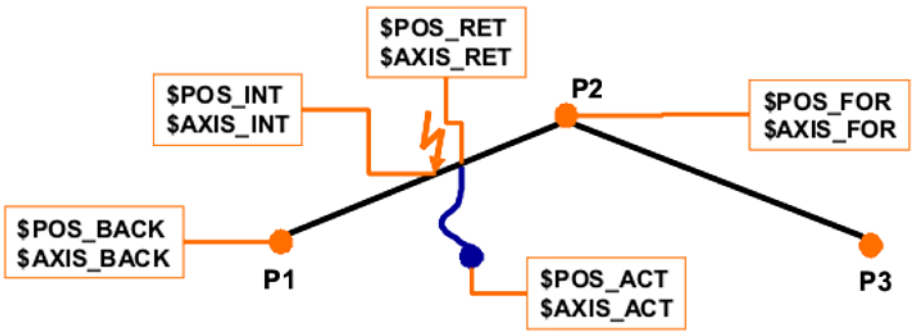

Fig. 5-5: Variables de sistema para parada exacta

- Variables de sistema útiles en caso de posicionamiento aproximado:

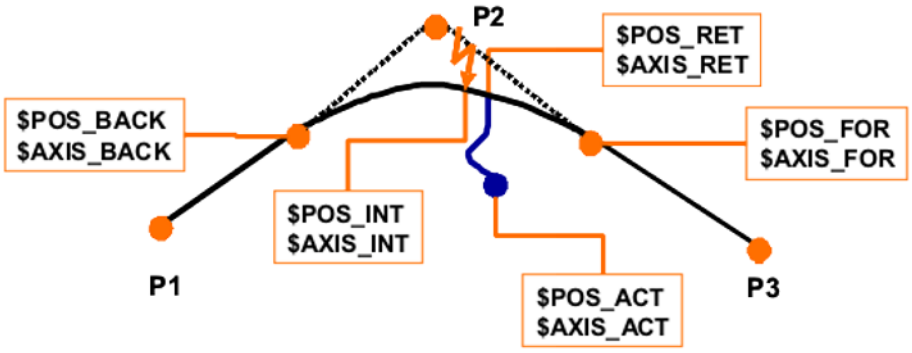

Fig. 5-6: Variables de sistema en caso de posicionamiento aproximado

Programación de rutinas de interrupción

Ejecución de la lógica en paralelo al movimiento del robot

1. Declaración de interrupción
   - Establecer la prioridad.
   - Determinar el evento desencadenante.
   - Definir y crear una rutina de interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )

END

_____________________________________________
DEF ERROR()

END
```

2. Activar y desactivar una interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
...
...
INTERRUPT OFF 25

END

_____________________________________________
DEF ERROR()

END
```

3. Ampliar un programa con movimientos y definir acciones en la rutina de interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
INTERRUPT OFF 25
END
_____________________________________________________
DEF ERROR()
$OUT[20]=FALSE
$OUT[21]=TRUE
END
```

Ejecución de la lógica tras la detención del robot y, a continuación, reanudación del movimiento del robot

1. Declaración de interrupción
   - Establecer la prioridad.
   - Determinar el evento desencadenante.
   - Definir y crear una rutina de interrupción.
   - Activar y desactivar una interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
...
...
INTERRUPT OFF 25

END
_____________________________________________________
DEF ERROR()

END
```

2. Ampliar un programa con movimientos, frenar el robot en la rutina de interrupción y definir la lógica.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
INTERRUPT OFF 25
END
_____________________________________________________
DEF ERROR()
BRAKE
$OUT[20]=FALSE
$OUT[21]=TRUE
END
```

Detener el movimiento actual del robot, reposicionarlo, desechar la planificación de trayectoria actual y desplazarse por una nueva trayectoria

1. Declaración de interrupción
   - Establecer la prioridad.

- Determinar el evento desencadenante.
- Definir y crear una rutina de interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
...
END

_____________________________________________________
DEF ERROR()
...
END
```

2. Ampliar un programa con movimientos:
   - Para poder cancelarlo, el movimiento se debe realizar en un subprograma.
   - El puntero de movimiento de avance debe permanecer en el subprograma.
   - Activar y desactivar una interrupción.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
SEARCH()
END

_____________________________________________________
DEF SEARCH()
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
WAIT SEC 0 ; Detener puntero de movimiento de avance
INTERRUPT OFF 25
END

_____________________________________________________
DEF ERROR()
...
END
```

3. Editar una rutina de interrupción.
   - Detener robot
   - Reposicionar el robot a $POS_INT.
   - Rechazar el movimiento actual.
   - Nuevo movimiento en el programa principal.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
SEARCH()
END

_____________________________________________________
DEF SEARCH()
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
WAIT SEC 0 ; Detener puntero de movimiento de avance
INTERRUPT OFF 25
END

_____________________________________________________
DEF ERROR()
BRAKE
PTP $POS_INT
RESUME
END
```

5.2 Ejercicio: Trabajos con interrupciones

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Declarar una interrupción.
- Crear un subprograma de interrupción.
- Evaluar y procesar interrupciones durante la ejecución del programa.

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos sobre la programación de interrupciones

Formulación

El objetivo de esta tarea es detectar la posición de 3 cubos mediante un desplazamiento de medición definido y guardar dichas posiciones.

1. Cree un nuevo programa con el nombre "BUSCAR".
2. Tome tres cubos del cargador (sin utilizar el robot) y colóquelos en línea sobre la mesa.
3. Programe un movimiento LIN como desplazamiento de búsqueda que pase por encima de los tres cubos. La velocidad se debe ajustar a 0,2 m/s.
4. Es necesario activar o desactivar el sensor con la salida 27. Como confirmación de la determinación de posición recibirá una señal en la entrada 27.
5. Cuando se detecte un cubo, la salida 10 se debería activar durante 1 s. La posición se debe guardar al mismo tiempo que se detecta. Para ello, utilice un campo creado en el fichero DAT local o $config.dat.
6. Al final del desplazamiento de búsqueda se deben mostrar las tres posiciones guardadas avanzando hasta ellas, es decir, se debe avanzar a la posición 1, esperar 1 segundo y luego avanzar a la posición siguiente.
7. Compruebe su programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿En qué parte del programa se declara la interrupción?

. . . . . .

. . . . . .

2. ¿Cuál es la diferencia entre INTERRUPT OFF 99 e INTERRUPT DISABLE 99?

. . . . . .

. . . . . .

. . . . . .

. . . . . .

3. ¿Cuándo se activa el subprograma de interrupción?

. . . . . .

. . . . . .

4. ¿Qué provoca la instrucción INTERRUPT OFF al comienzo de un subprograma de interrupción?

. . . . . .

. . . . . .

5. ¿Qué rango de prioridad no se ha liberado para la interrupción?

. . . . . .

. . . . . .

5.3 Ejercicio: Cancelación de movimientos con interrupciones

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Declarar una interrupción.
- Crear un subprograma de interrupción.
- Evaluar y procesar interrupciones en el flujo de programa.
- Frenar el movimiento del robot mediante una instrucción KRL.
- Frenar y cancelar el movimiento del robot mediante instrucciones KRL.

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL
- Conocimientos teóricos sobre la programación de interrupciones
- Conocimientos teóricos sobre las instrucciones KRL para frenar y cancelar el movimiento del robot y su correcta aplicación

Formulación

Debe poder detectar la posición de 3 cubos mediante un desplazamiento de medición definido y guardar dichas posiciones. Además, el desplazamiento de medición se debe cancelar directamente después de la detección del tercer cubo.

1. Duplique su programa BUSCAR y asígnele el nombre "BUSCAR_CANCELAR".
2. Tome tres cubos del cargador (sin utilizar el robot) y colóquelos en línea sobre la mesa.
3. Programe un movimiento LIN como desplazamiento de búsqueda que pase por encima de los tres cubos. La velocidad se debe ajustar a 0,2 m/s. Es necesario activar o desactivar el sensor con la salida 27. Como confirmación de la determinación de posición recibirá una señal en la entrada 27.
4. Cuando se detecte un cubo, la salida 10 se debería activar durante 1 s. La posición se debe guardar al mismo tiempo que se detecta. Para ello, utilice un campo creado en el fichero DAT local.
5. Inmediatamente después de haber encontrado el tercer cubo, se debe detener el robot y se debe cancelar el desplazamiento de búsqueda.
6. Al final del desplazamiento de búsqueda se deben mostrar las tres posiciones guardadas avanzando hasta ellas, es decir, se debe avanzar a la posición 1, esperar 1 segundo y luego avanzar a la posición siguiente.
7. Comprobar el programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Cuál es la diferencia entre BRAKE y BRAKE F?

. . . . . .

. . . . . .

2. ¿Por qué la instrucción RESUME no funciona correctamente en este caso?

```
INTERRUPT DECL 21 WHEN $IN[1] DO encontrado( )
INTERRUPT ON 21
LIN ptoarr
LIN ptofn
$ADVANCE = 0
INTERRUPT OFF 21 ...
END

DEF encontrado( )
INTERRUPT OFF 21
```

```
BRAKE
;tomar pieza
RESUME
END
```

3. ¿Cuando se desencadena una interrupción?

6 Programación de estrategias de retorno

6.1 Programación de estrategias de retorno

¿Qué es una estrategia de retorno?

Tras la creación de un programa de ejecución y comprobarlo en el terreno práctico, se plantea siempre la pregunta de cómo reaccionará el programa en caso de avería.

En caso de avería, evidentemente lo deseable sería que el sistema reaccionara automáticamente.

Para ello se utilizan estrategias de retorno.

Por estrategia de retorno se entienden los movimientos de retorno que el robot ejecuta en caso de avería, por ejemplo, para poder desplazarse automáticamente a la posición base con independencia del lugar en el que se encuentre en ese momento.

Estos movimientos de retroceso debe programarlos libremente el programador.

¿Dónde se aplican estrategias de retorno?

Las estrategias de retorno se utilizan allí donde se desee conseguir una automatización total de una célula de producción, también en caso de avería.

Una estrategia de retorno correctamente programada puede brindar al usuario la oportunidad de decidir qué debe ocurrir en adelante.

Incluso se puede omitir un desplazamiento manual por una situación de peligro.

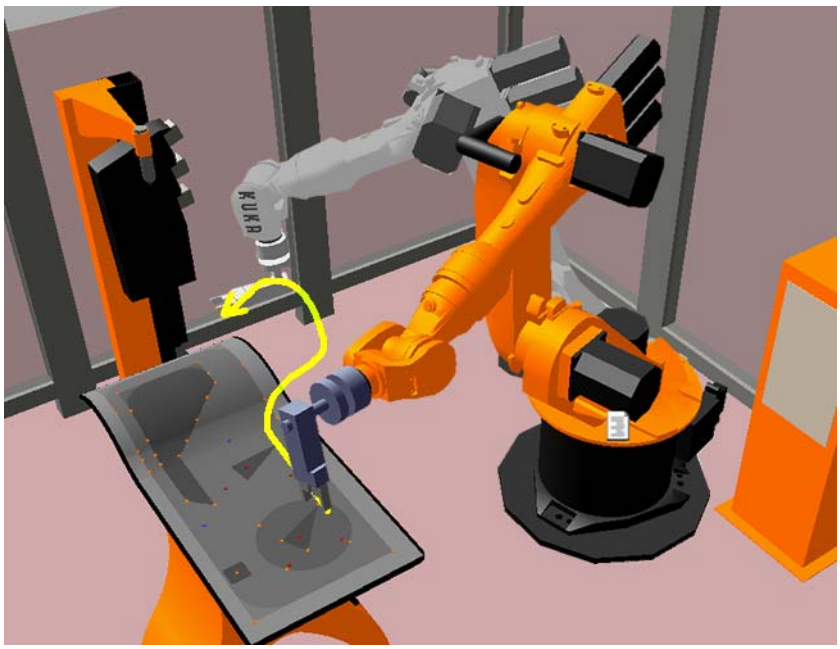

Fig. 6-1

¿Cómo se programa una estrategia de retorno?

- Crear un rango de desplazamiento en las zonas de trabajo
- Configuración de IO
- Declarar interrupciones
- Guardar posiciones
- Programar mensajes de usuario
- Definir distintas posiciones iniciales dado el caso
- Utilizar puntos globales dado el caso

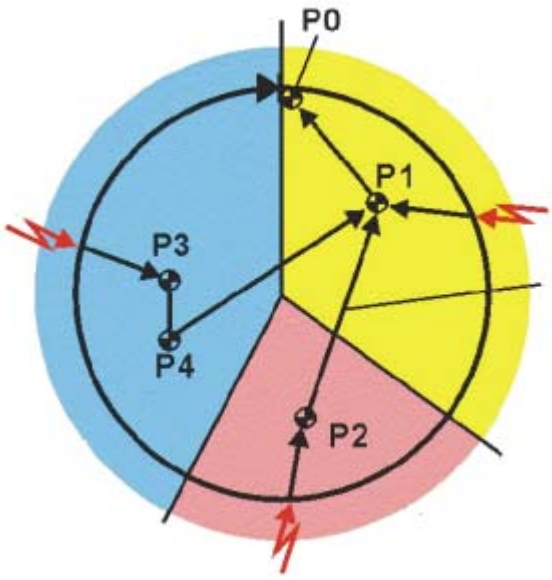

Fig. 6-2

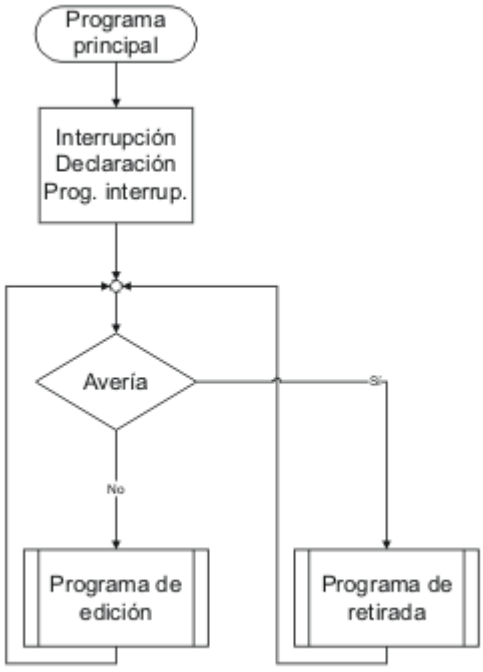

Fig. 6-3

6.2 Ejercicio: Programación de estrategia de retorno

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Programar movimientos de retroceso automáticos.
- Integrar mensajes en un proceso de trabajo.
- Detección de averías mediante la utilización de interrupciones.
- Finalizar el movimiento del robot en función del proceso.

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos del lenguaje de programación KRL

- Conocimientos sobre la programación de mensajes
- Conocimientos sobre la programación de interrupciones
- Conocimientos sobre las instrucciones KRL para frenar y cancelar el movimiento del robot y su correcta aplicación
- Conocimientos teóricos sobre la instrucción Trigger

Formulación

El programa base consiste en tomar el cubo del cargador y volver a depositarlo en él. La liberación del PLC se deshabilita mediante una entrada (n.º 11). El robot se debe detener de inmediato. Mediante una consulta, el operario debe decidir si el robot se desplaza a la posición de salida o si se continúa con el procedimiento. En cualquier caso, el robot sólo se podrá desplazar después de decidir cuándo volverá a estar disponible la liberación y confirmar esta avería. Si se selecciona la posición de salida, el desplazamiento se realizará a velocidad reducida (POV=10%). En la posición base se consulta si la instalación está lista. En caso afirmativo, se puede continuar con el override del programa ajustado antes de la avería. En caso negativo, se finaliza el programa.

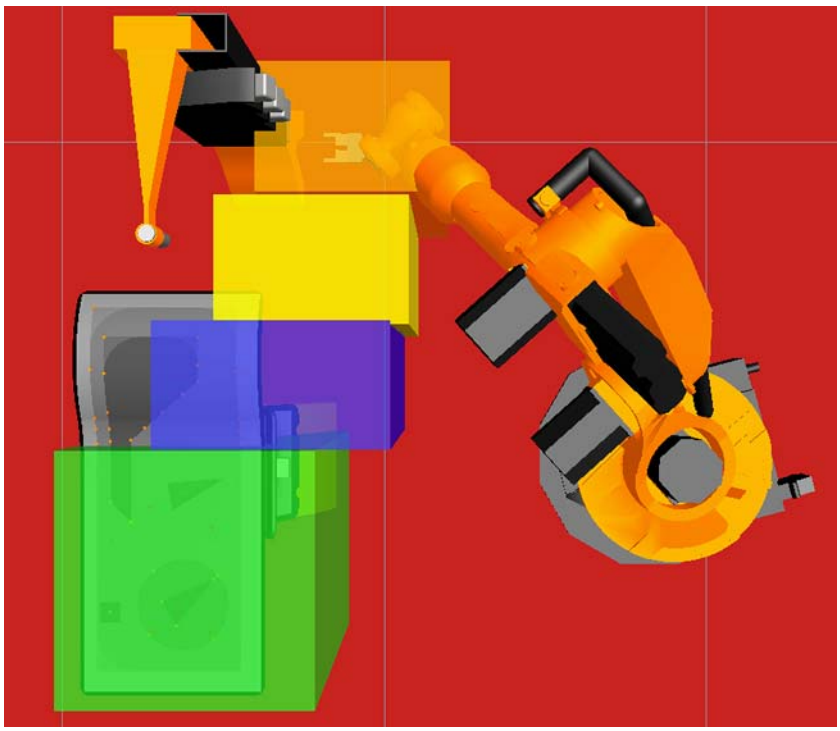

Fig. 6-4

1. Empiece por crear el plan de ejecución del programa.
2. Preste atención a la estructuración de su concepto global cuando lo aplique a la estructura del programa.
3. El objetivo de este proyecto es conseguir una programación clara y depurada, y que los programas o módulos funcionen.
4. Cuando asigne nombres de archivo y de variables, asegúrese de que sean fácilmente comprensibles.
5. Compruebe que pueda desplazarse a la posición de salida sin colisiones en cualquier momento.
6. Durante el rearranque desde la posición de salida, asegúrese de realizar el procedimiento correcto (tomar o depositar) en función de la posición de la garra. Indicación: La entrada 26 significa que la garra está abierta.
7. Comprobar el programa según prescripción.

Lo que se debe saber tras el ejercicio:

1. ¿Con qué instrucción de experto se pueden activar variables definidas por el usuario sobre la trayectoria?

. . . . . .

. . . . . .

2. ¿Cuál es la instrucción KRL para finalizar de inmediato un subprograma y un subprograma de interrupción?

. . . . . .

. . . . . .

3. ¿Para qué se utiliza un desplazamiento COI?

. . . . . .

. . . . . .

4. ¿Con qué variables se puede influir en el override del programa?

. . . . . .

. . . . . .

5. En comparación con el formulario inline SYNOUT, ¿qué se puede activar además con la instrucción Trigger?

. . . . . .

. . . . . .

7 Trabajos con señales analógicas

7.1 Programación de entradas analógicas

Descripción

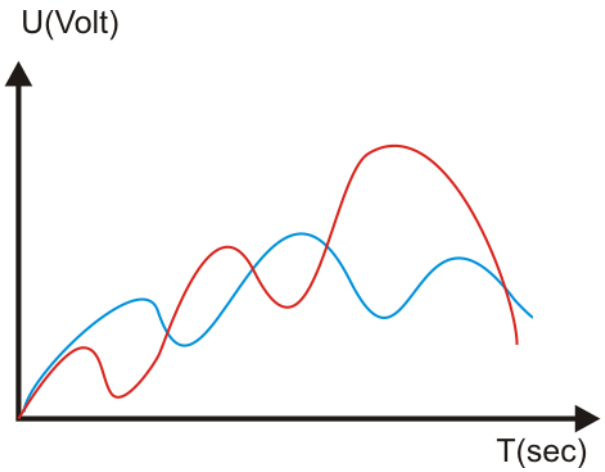

Fig. 7-1: Señales analógicas

- KR C4 cuenta con más de 32 entradas analógicas.
- Para las señales analógicas se necesita un sistema de bus opcional y la proyección se debe realizar con WorkVisual.
- Las entradas analógicas se leen mediante las variables de sistema `$ANIN[1]` ... `$ANIN[32]`.
- Lectura cíclica (cada 12 ms) de una entrada analógica.
- Los valores `$ANIN[nr]` oscilan entre 1,0 y -1,0, y representan una tensión de entrada de entre +10 V y -10 V

Función

Asignación estática de valor

- Asignación directa de valores

```
...
REAL value

value = $ANIN[2]
...
```

- Asignación de valores de una declaración de señal

```
...
SIGNAL sensor $ANIN[6]
REAL value

value = sensor
...
```

Asignación dinámica de valor

- Todas las variables empleadas en una instrucción ANIN deben estar declaradas en **listas de datos** (localmente o en el archivo `$CONFIG.DAT`).
- Como máximo están permitidas **tres** instrucciones ANIN ON a la vez.
- Cómo máximo dos instrucciones ANIN ON pueden utilizar la misma variable *Valor* o acceder a la misma entrada analógica.
- Sintaxis
  - Iniciar lectura cíclica:

    `ANIN ON` *Valor* `=` *Factor* `*` *Nombre de señal* <*±Offset*>

Elemento	Descripción
Valor	Tipo: REAL En *Valor* se registra el resultado de la lectura cíclica. Este *Valor* puede ser una variable o un nombre de señal para una salida.
Factor	Tipo: REAL Un factor cualquiera. Puede ser una constante, una variable o un nombre de señal.
Nombre de la señal	Tipo: REAL Indica la entrada analógica. El *nombre de señal* se debe haber declarado antes con `SIGNAL` . No está permitido indicar directamente la entrada analógica $ANIN[x] en lugar del nombre de señal. Los valores de una entrada analógica $ANIN[x] oscilan entre +1,0 y -1,0 y representan una tensión de +10 V a -10 V dar.
Offset	Tipo: REAL Puede ser una constante, una variable o un nombre de señal.

- Finalizar lectura cíclica:

  `ANIN OFF` *nombre de señal*

- Ejemplo 1

```
DEFDAT myprog
DECL REAL value = 0
ENDDAT
```

```
DEF myprog( )
SIGNAL sensor $ANIN[3]
...
ANIN ON value = 1.99*sensor-0.75
...
ANIN OFF sensor
```

- Ejemplo 2

```
DEFDAT myprog
DECL REAL value = 0
DECL REAL corr = 0.25
DECL REAL offset = 0.45
ENDDAT
```

```
DEF myprog( )
SIGNAL sensor $ANIN[7]
...
ANIN ON value = corr*sensor-offset
...
ANIN OFF sensor
```

Procedimiento para programar con entradas analógicas

AVISO El requisito para utilizar las señales analógicas es proyectar correctamente el sistema de bus con las señales analógicas conectadas.

Programación de ANIN ON/OFF

1. Selección de la entrada analógica correcta.
2. Ejecución de la declaración de señal.
3. Declaración de las variables necesarias en una lista de datos.
4. **Conectar:** programación de la instrucción `ANIN ON`.

5. Comprobación de que haya 3 entradas dinámicas activas como máximo.
6. **Desconectar:** programación de la instrucción `ANIN OFF`.

7.2 Programación de salidas analógicas

Descripción

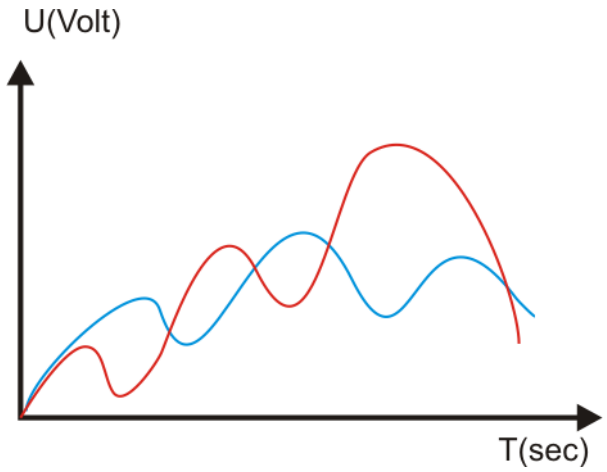

Fig. 7-2: Señales analógicas

- KR C4 cuenta con más de 32 salidas analógicas.
- Para las señales analógicas se necesita un sistema de bus opcional y la proyección se debe realizar con WorkVisual.
- Las entradas analógicas se leen mediante las variables de sistema `$ANOUT[1]` ... `$ANOUT[32]`.
- Escritura cíclica (cada 12 ms) en una salida analógica.
- Los valores `$ANOUT[nr]` oscilan entre 1,0 y -1,0, y representan una tensión de salida de entre +10 V y -10 V

Función

AVISO Se pueden utilizar, como máximo, 8 salidas analógicas al mismo tiempo (estáticas y dinámicas juntas). ANOUT genera una parada del procesamiento en avance.

Asignación estática de valor

- Asignación directa de valores

```
...
ANOUT[2] = 0.7 ; 7 V en salida analógica 2
...
```

- Asignación de valores mediante variables

```
...
REAL value
value = -0.8
ANOUT[4] = value ; -8 V en salida analógica 4
...
```

- Programación mediante formulario inline

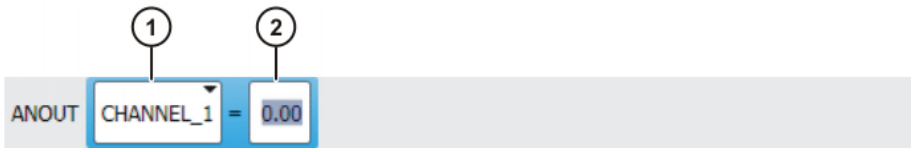

Fig. 7-3: Formulario inline ANOUT estática

Pos.	Descripción
1	Número de la salida analógica ▪ **CHANNEL_1 … CHANNEL_32**
2	Factor para la tensión ▪ **0 … 1** (graduación: 0.01)

Asignación dinámica de valor

- Todas las variables utilizadas en una instrucción ANOUT deben estar declaradas en **listas de datos** (localmente o en $CONFIG.DAT).
- Se permite un máximo de **cuatro** instrucciones ANOUT ON al mismo tiempo.
- ANOUT genera una parada del procesamiento en avance.
- Sintaxis
  - Iniciar lectura cíclica:

    `ANOUT ON` *Nombre de señal* `=` *Factor* `*` *Elemento de regulación* `<`*±Offset*`> <DELAY = ±`*Tiempo*`> <MINIMUM =` *Valor mínimo*`> <MAXIMUM =` *Valor máximo*`>`

Elemento	Descripción
Nombre de la señal	Tipo: REAL Indica la salida analógica. El *Nombre de señal* debe haberse declarado con antelación con SIGNAL. No está permitido indicar directamente la salida analógica $ANOUT[x] en lugar del nombre de señal. Los valores de una salida analógica $ANOUT[x] oscilan entre +1,0 y -1,0 y representan una tensión de +10 V a -10 V dar.
Factor	Tipo: REAL Un factor cualquiera. Puede ser una constante, una variable o un nombre de señal.
Elemento de regulación	Tipo: REAL Puede ser una constante, una variable o un nombre de señal.
Offset	Tipo: REAL Puede ser una constante, una variable o un nombre de señal.
Horas	Tipo: REAL Unidad: Segundos. Con la palabra clave DELAY y una indicación de tiempo positiva o negativa, se puede emitir una señal de salida con retardo (+) o adelanto (-).
Valor mínimo, valor máximo	Tipo: REAL Tensión mínima y/o máxima que debe existir en la salida. No se sobrepasa ni por encima ni por debajo, ni siquiera si los valores calculados están por encima o por debajo. Valores permitidos: -1,0 a +1,0 (corresponde de -10 V a +10 V). Puede ser una constante, una variable, un componente estructural o un elemento de campo. El valor mínimo debe ser en cualquier caso menor al valor máximo. Debe respetarse la secuencia de las palabras clave MINIMUM y MAXIMUM.

- Finalizar lectura cíclica:

  ANOUT OFF *Nombre de señal*

- Ejemplo 1

```
DEF myprog( )
SIGNAL motor $ANOUT[3]
...
ANOUT ON motor = 3.5*$VEL_ACT-0.75 DELAY=0.5
...
ANOUT OFF motor
```

- Ejemplo 2

```
DEFDAT myprog
DECL REAL corr = 1.45
DECL REAL offset = 0.25
ENDDAT
```

```
DEF myprog( )
SIGNAL motor $ANOUT[7]
...
ANOUT ON motor = corr*$VEL_ACT-offset
...
ANOUT OFF motor
```

Procedimiento para programar con entradas analógicas

AVISO El requisito para utilizar las señales analógicas es proyectar correctamente el sistema de bus con las señales analógicas conectadas.

Programación de ANOUT ON/OFF

1. Selección de la salida analógica correcta.
2. Ejecución de la declaración de señal.
3. Declaración de las variables necesarias en una lista de datos.
4. **Conectar:** programación de la instrucción ANOUT ON.
5. Comprobación de que haya 4 salidas dinámicas activas como máximo.
6. **Desconectar:** programación de la instrucción ANOUT OFF.

Ejemplo:

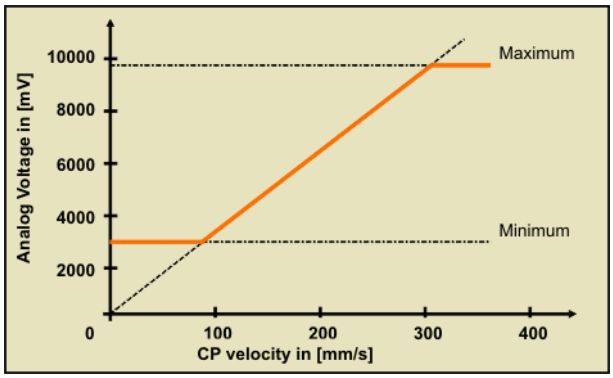

Fig. 7-4: Ejemplo de señal de salida analógica

```
DEF myprog( )
SIGNAL motor $ANOUT[3]
...
ANOUT ON motor = 3.375*$VEL_ACT MINIMUM=0.30 MAXIMUM=0.97
...
ANOUT OFF motor
```

7.3 Ejercicio: Trabajos con E/S analógicas

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Utilización de declaraciones de señales en entradas/salidas
- Integración estática o dinámica de entradas analógicas en los procesos de trabajo
- Integración estática o dinámica de salidas analógicas en los procesos de trabajo

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos teóricos sobre las declaraciones de señales
- Conocimientos teóricos sobre la integración de entradas/salidas analógicas

Formulación

Configure su sistema de modo que pueda modificar el override del programa mediante una entrada analógica. Además, la velocidad real de desplazamiento del robot debe controlar una salida analógica.

Tarea parcial 1

1. Cree un programa con el nombre Velocidad.
2. Utilice la entrada analógica 1 controlada mediante el potenciómetro.
3. Modifique el override del programa en el interpretador Submit.
4. Compruebe su programa según prescripción.

Tarea parcial 2

1. Amplíe su programa con movimientos de trayectoria (velocidad: hasta 2 m/s) que formen parte de un bucle infinito.
2. Utilice la salida analógica 1, visualización en panel.
3. Utilice la variable de sistema $VEL_ACT para la velocidad de desplazamiento actual.
4. Compruebe su programa según prescripción.
5. Adicionalmente: Si la velocidad es inferior a los 0,2 m/s, se debe excitar la salida de todos modos con 1,0 V; si la velocidad es superior a los 1,8 m/s, la salida no debe emitir más de 9,0 V.

Asegúrese de activar la E/S analógica una única vez.

Lo que se debe saber tras el ejercicio:

1. ¿Cuántas E/S analógicas se pueden utilizar en la unidad de control KRC?

. . . . . .

. . . . . .

2. ¿Cuántas entradas digitales, entradas analógicas y salidas analógicas predefinidas puede utilizar simultáneamente la unidad de control de KUKA?

. . . . . .

. . . . . .

3. ¿Cuáles son las instrucciones KRL para iniciar y finalizar cíclicamente la salida analógica?

. . . . . .

. . . . . .

4. ¿Cómo se interroga una entrada analógica estáticamente?

. . . . . .

. . . . . .

8 Secuencia y configuración del modo automático externo

8.1 Configuración y aplicación del modo automático externo

Descripción

Fig. 8-1: Enlace PLC

- Con la interfaz del modo automático externo, los procesos del robot se controlan mediante una unidad de control superior (p. ej., un PLC).
- Con la interfaz Automático externo la unidad de control superior transmite a la unidad de control del robot las señales para los procesos del robot (p. ej. validación del movimiento, confirmación de fallos, arranque del programa, etc.). La unidad de control del robot transmite a la unidad de control superior información sobre los estados de funcionamiento y averías.

Para poder utilizar la interfaz Automático externo se debe efectuar la configuración siguiente:

1. Configurar programa CELL.SRC.
2. Configurar entradas/salidas de la interfaz de Automático Externo.

Utilización de entradas/salidas de la interfaz en modo automático externo

Cuadro resumen de las señales más importantes de la interfaz

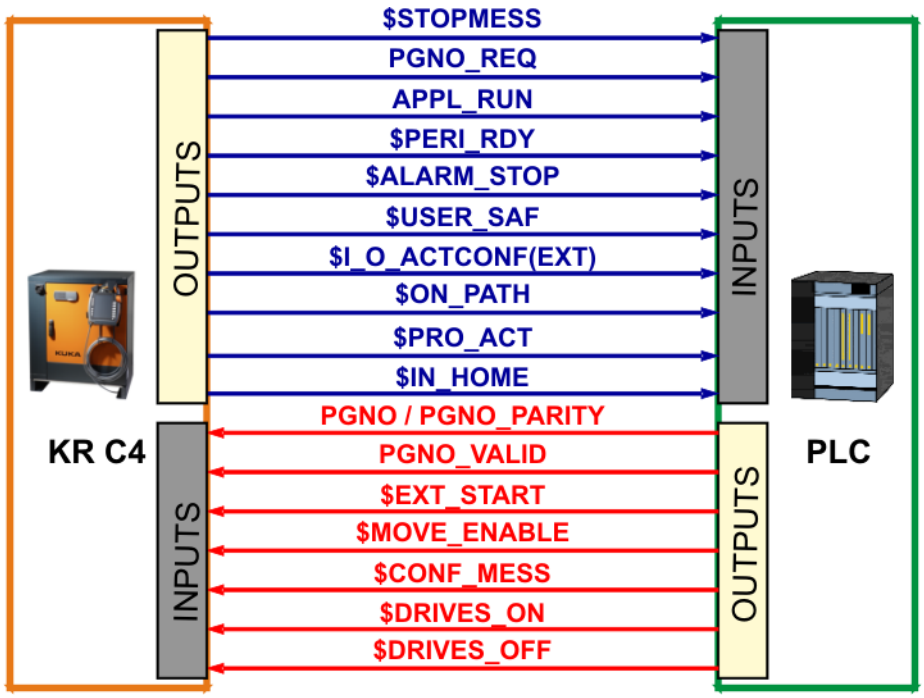

Fig. 8-2: Resumen de las señales más importantes en modo automático externo

Entradas (desde la perspectiva de la unidad de control del robot)

- **PGNO_TYPE**: tipo de números del programa

  Esta variable define en qué formato se leerán los números de programa transmitidos por la unidad de control superior.

Valor	Descripción	Ejemplo
1	Leer como número binario. El número de programa se transmite por la unidad de control superior en forma de valor entero con codificación binaria.	0 0 1 0 0 1 1 1 => PGNO = 39
2	Leer como valor BCD. El número de programa se transmite por la unidad de control superior en forma de valor decimal con codificación binaria.	0 0 1 0 0 1 1 1 => PGNO = 27
3	Leer como "1 de N"*. El número de programa es transmitido por la unidad de control superior o por los dispositivos periféricos en forma de valor codificado "1 de n".	0 0 0 0 0 0 0 1 => PGNO = 1 0 0 0 0 1 0 0 0 => PGNO = 4

  * En este formato de transferencia, los valores de PGNO_REQ, PGNO_PARITY y PGNO_VALID no son evaluados, y carecen por tanto de importancia.

- **PGNO_LENGTH**: longitud de números del programa

  Esta variable establece el ancho de bit del número de programa transmitido por la unidad de control superior. Gama de valores: 1 … 16.

  Si PGNO_TYPE posee el valor 2, sólo se permiten anchos de 4, 8, 12 y 16 bits.

- **PGNO_PARITY**: bit de paridad de número del programa

  Entrada por la que la unidad de control superior transmite el bit de paridad.

Entrada	Función
Valor negativo	Paridad impar
0	Sin evaluación!
Valor positivo	Paridad par

Si PGNO_TYPE tiene valor 3, PGNO_PARITY no se evaluará..

- **PGNO_VALID**: número de programa válido

  Entrada por la que la unidad de control superior transmite el comando de lectura del número de programa.

Entrada	Función
Valor negativo	Se acepta el número con el flanco decreciente de la señal.
0	Se acepta el número con el flanco creciente de la señal en el conductor EXT_START.
Valor positivo	Se acepta el número con el flanco creciente de la señal.

- **$EXT_START**: arranque externo

  Al establecer esta entrada, se puede iniciar o continuar un programa (normalmente CELL.SRC) si la interfaz de E/S se encuentra activa.

Sólo se evalúa el flanco creciente de la señal.

⚠ ADVERTENCIA En el modo de servicio automático externo no se efectúa ningún desplazamiento COI. Esto significa que, una vez arrancado, el robot se desplaza a la primera posición programada a la velocidad programada (no reducida). El robot no para allí.

- **$MOVE_ENABLE**: marcha habilitada

  Esta entrada se utiliza para controlar los accionamientos del robot a través de la unidad de control superior.

Señal	Función
TRUE	Se puede efectuar un desplazamiento manual y se puede ejecutar el programa
FALSE	Detención de todos los accionamientos y bloqueo de todos los comandos activos

Si los accionamientos han sido detenidos por la unidad de control superior, aparecerá el mensaje "LIBERACIÓN DE MOVIMIENTO GENERAL". El robot sólo se podrá mover una vez que se haya borrado este mensaje y se haya recibido una nueva señal de arranque externa.

Durante la puesta en servicio, a la variable "$MOVE_ENABLE" se le asigna con frecuencia el valor "$IN[1025]". Si después se olvida configurar otra entrada, no será posible efectuar un arranque externo.

- **$CONF_MESS**: confirmación de mensaje

  Al activar esta entrada, la unidad de control superior confirmará los mensajes de error en cuanto se haya subsanado la causa del fallo.

Sólo se evalúa el flanco creciente de la señal.

- **$DRIVES_ON**: accionamientos conectados

  Si en esta entrada se crea un impulso de nivel high de 20 ms de duración como mínimo, la unidad de control principal activa los accionamientos del robot.
- **$DRIVES_OFF**: accionamientos desconectados

  Si en esta entrada se crea un impulso de nivel low de 20 ms de duración como mínimo, la unidad de control principal desactiva los accionamientos del robot.

Salidas (desde la perspectiva de la unidad de control del robot)

- **$ALARM_STOP**: parada de emergencia

  Esta salida se reinicia en las siguientes situaciones de PARADA DE EMERGENCIA:
  - Se acciona el pulsador de PARADA DE EMERGENCIA en el KCP (Int. NotAus).
  - PARADA DE EMERGENCIA externa

En caso de una PARADA DE EMERGENCIA, en los estados de las salidas **$ALARM_STOP** e **Int. NotAus** se detecta de qué PARADA DE EMERGENCIA se trata:

- Ambas salidas son FALSE: la PARADA DE EMERGENCIA se ha desencadenado en el KCP.
- **$ALARM_STOP** es FALSE y **Int. NotAus** es TRUE: PARADA DE EMERGENCIA externa

- **$USER_SAF**: protección del operario/puertas de protección

  Esta salida se reinicializa al abrir un conmutador de muestreo de la valla protectora (en el modo de servicio AUT) o bien al soltar un pulsador de validación (en el modo de servicio T1 o T2).
- **$PERI_RDY**: los accionamientos están listos

  Al establecer esta salida, la unidad de control del robot comunica a la unidad de control superior que los accionamientos del robot están activados.
- **$STOPMESS**: mensajes de parada

  Esta salida es establecida por la unidad de control del robot para indicar a la unidad de control superior que se ha producido un mensaje que requiere la parada del robot. (Ejemplos: PARADA-EMERGENCIA, liberación de la marcha o protección del operador).
- **$I_O_ACTCONF**: el modo automático externo está activado

  Esta entrada es TRUE si se ha seleccionado el modo automático externo y si la entrada $I_O_ACT es TRUE (normalmente, siempre en $IN[1025]).
- **$PRO_ACT**: el programa está activo/en ejecución

  Esta salida estará fijada cuando esté activo un proceso en el nivel de robot. El proceso continúa activo mientras se esté procesando el programa o una interrupción. Al final del programa la ejecución del mismo pasará a estado inactivo justo cuando se hayan procesado todas las salidas de impulsos y de activación (trigger).
- **PGNO_REQ**: solicitud de números de programa

  Con un cambio de señal en esta salida se requiere a la unidad de control superior que transmita un número de programa.

  Si PGNO_TYPE tiene valor 3, PGNO_REQ no se evaluará.
- **APPL_RUN**: el programa de la aplicación está en ejecución

  Al establecer esta salida, la unidad de control del robot comunica a la unidad de control superior que se está procesando un programa en ese preciso instante.

- **$IN_HOME**: el robot se encuentra en la posición inicial

  Esta salida comunica a la unidad de control superior si el robot se encuentra en su posición inicial.
- **$ON_PATH**: el robot se encuentra sobre la trayectoria

  Esta salida estará activa mientras el robot se encuentra dentro de su trayectoria programada. Después de un desplazamiento COI, se activa la salida ON_PATH. Esta salida permanecerá activa hasta que el robot se salga de su trayectoria, se resetee el programa o se ejecute una selección de paso. La señal ON_PATH no dispone de una ventana de tolerancia; en el momento en que el robot abandona la trayectoria, se reinicia la señal.

Principio de la comunicación en el modo automático externo

Resumen de toda la secuencia

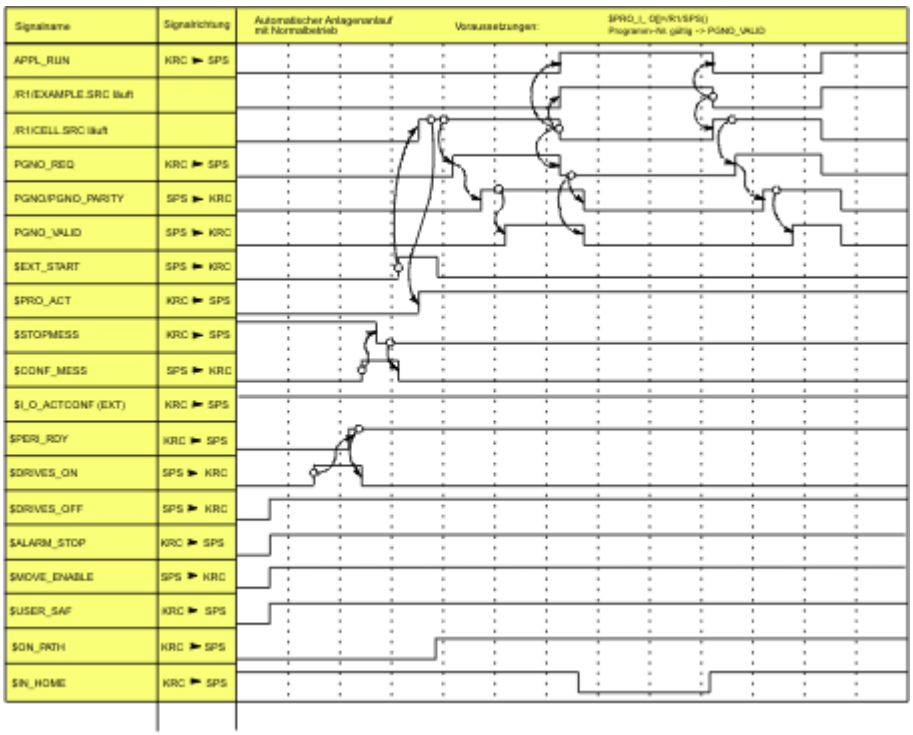

Fig. 8-3: Arranque automático de la instalación y servicio normal con confirmación del número de programa por medio de PGNO_VALID

División en subcampos

1. Conectar los accionamientos
2. Confirmar mensajes
3. Iniciar el programa Cell
4. Transmitir el número de programa y ejecutar la aplicación

Para cada uno de estos campos se deben cumplir condiciones, además, cabe la posibilidad de informar al PLC de los estados del robot.

Fig. 8-4: Protocolo de enlace

Es conveniente utilizar este protocolo de enlace predefinido.

Conectar los accionamientos

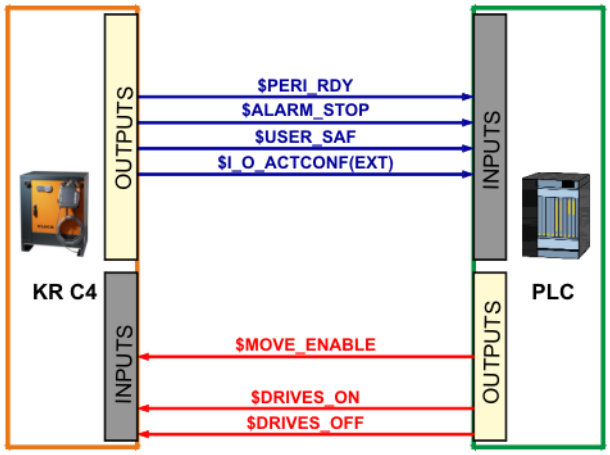

Fig. 8-7

- Condiciones previas
  - $USER_SAF: puertas de protección cerradas.
  - $ALARM_STOP: no hay activa ninguna parada de emergencia.
  - $I_O_ACTCONF: está activo el modo automático externo.
  - $MOVE_ENABLE: marcha habilitada disponible.
  - $DRIVER_OFF: no está activada la desconexión de los accionamientos.
- Conectar los accionamientos

  $DRIVES_ON: los accionamientos están activados como mínimo durante 20 ms.
- Accionamientos preparados

  $PERI_RDY: en cuanto se recibe la confirmación para los accionamientos, se anula la señal $DRIVES_ON.

Confirmar mensajes

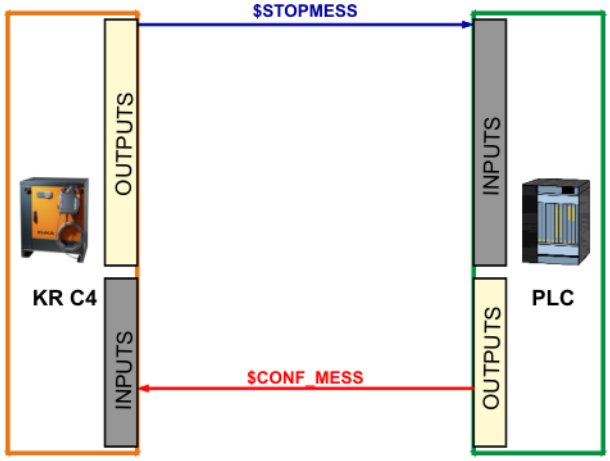

Fig. 8-11

- Condiciones previas

  $STOPMESS: hay activo un mensaje de Stop.
- Confirmar el mensaje

  $CONF_MESS: confirmar el mensaje.
- Se han borrado los mensajes que se pueden confirmar.

  $STOPMESS: el mensaje de Stop ya no está en $CONF_MESS, ahora se puede anular.

Iniciar el programa (CELL.SRC) externamente.

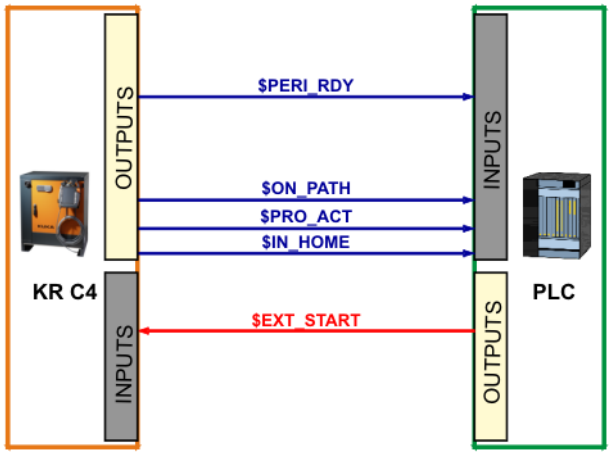

Fig. 8-16

- Condiciones previas
  - $PERI_RDY: los accionamientos están listos.
  - $IN_HOME: el robot se encuentra en la posición HOME.
  - Sin $STOPMESS: no hay activo ningún mensaje de Stop.
- Arranque externo

  $EXT_START: activar el arranque externo (flanco positivo).
- El programa CELL está en ejecución.
  - $PRO_ACT: indica que el programa CELL está en ejecución.
  - $ON_PATH: en cuanto se recibe la notificación de que el robot se encuentra sobre la trayectoria, se anula la señal $EXT_START.

Ejecutar la transferencia de programa y el programa de aplicación.

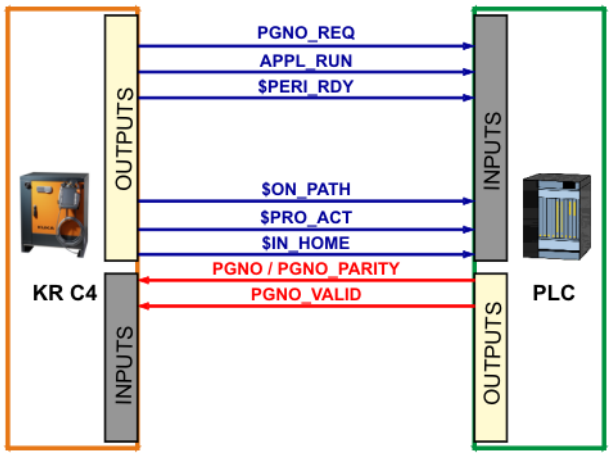

Fig. 8-23

- Condiciones previas
  - $PERI_RDY: los accionamientos están listos.
  - $PRO_ACT: el programa CELL está en ejecución.
  - $ON_PATH: el robot se encuentra sobre la trayectoria.
  - $IN_HOME: el robot se encuentra en la posición inicial; esto no es necesario en caso de reanudación.
  - PGNO_REQ: la solicitud de números de programa está activada.
- Transmisión de números de programa y confirmación
  - Transmisión del número de programa

(se han ajustado el tipo de datos correcto (PGNO_TYPE), la longitud de número de programa (PGNO_LENGTH) y el primer bit del número de programa (PGNO_FBIT)).

- PGNO_VALID: activar el número de programa válido (confirmación, flanco positivo)

- El programa de aplicación está en ejecución.
  - APPL_RUN: indica que el programa de aplicación está en ejecución.
  - El robot abandona la posición inicial; una vez finalizado el programa de aplicación, el robot regresa a la posición inicial.

Procedimiento

1. En el menú principal seleccionar **Configuración** > **Entradas/Salidas** > **Automático externo**.
2. En la columna **Valor** marcar la celda que se debe editar y pulsar **Editar**.
3. Introducir el valor que se desee y guardarlo con **OK**.
4. Repetir los pasos 2 y 3 para todos los valores que se desean editar.
5. Cerrar la ventana. Las modificaciones son aceptadas.

Fig. 8-26: Configuración entradas de Automático externo

Pos.	Descripción
1	Número
2	Nombre largo de la entrada/salida
3	Tipo ■ **Verde**: Entrada/Salida ■ **Amarillo**: Variable o variable del sistema ($...)
4	Nombre de la señal o de la variable
5	Número de la entrada/salida o número del canal
6	Las salidas se encuentran clasificadas en pestañas por temas.

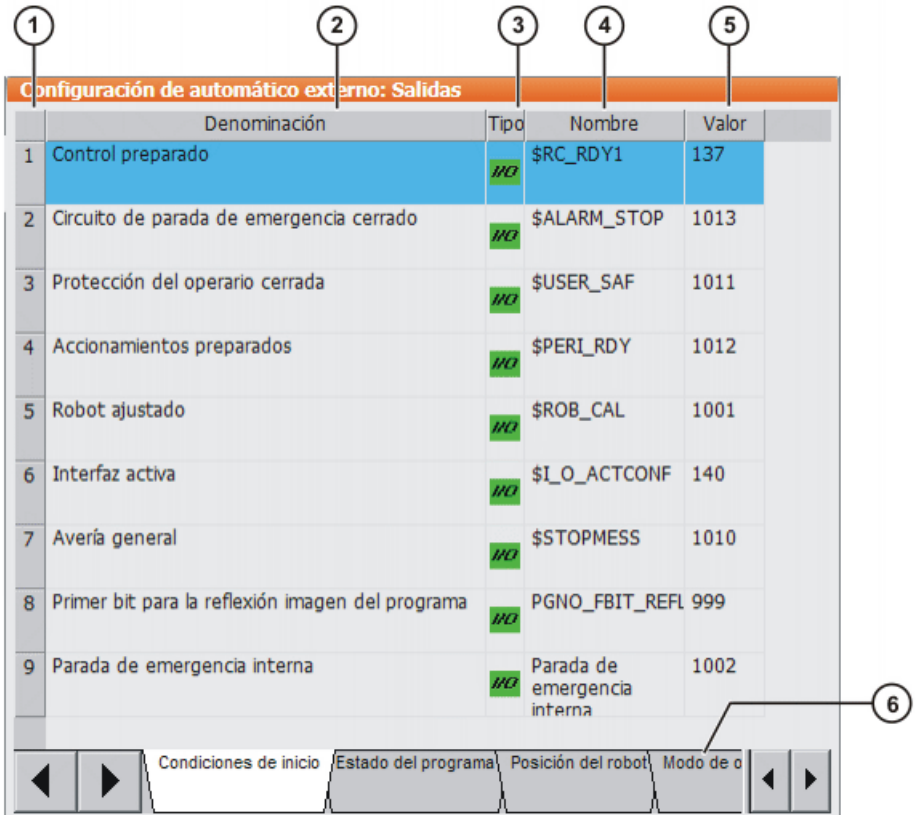

Fig. 8-27: Configuración de salidas en el modo automático externo

8.2 Ejercicio: El modo automático externo

Objetivo del ejercicio

Después de completar correctamente este ejercicio, se dispondrá de la competencia necesaria para efectuar las siguientes tareas:

- Integración selectiva de un programa de robot en el modo automático externo.
- Adaptación del programa "Cell".
- Configuración de la interfaz para el modo automático externo.
- Conocer la secuencia del modo Automático externo.

Requisitos

Los siguientes requisitos son necesarios para completar este ejercicio correctamente:

- Conocimientos sobre el procesamiento del programa "Cell".
- Conocimientos sobre la configuración de la interfaz para el modo automático externo.
- Conocimientos teóricos sobre la secuencia técnica de señales para el modo automático externo.

Formulación

1. Configure la interfaz para el modo automático externo según las especificaciones de su panel de mando.
2. Amplíe su programa Cell con los 3 módulos que desee después de haber comprobado su función.
3. Compruebe el programa en los modos de servicio T1, T2 y Automático. Se deben tener en cuenta las prescripciones de seguridad enseñadas.
4. Simule la funcionalidad de la unidad de control del PLC mediante el pulsador.

Lo que se debe saber tras el ejercicio:

1. ¿Qué requisito se necesita para que no se evalúe PGNO_REQ?

. . . . . .

. . . . . .

2. ¿Con qué señal se conectan los accionamientos y a qué se debe prestar atención en este caso?

. . . . . .

. . . . . .

3. ¿Qué variable de la interfaz para el modo automático externo repercute también en el desplazamiento manual?

. . . . . .

. . . . . .

4. ¿Qué pliegue comprueba la posición inicial en el programa CELL?

. . . . . .

. . . . . .

5. ¿Qué requisitos se necesitan para el modo Automático externo?

. . . . . .

. . . . . .

9 Programación de la detección de colisiones

9.1 Programación de movimientos con detección de colisiones

Descripción

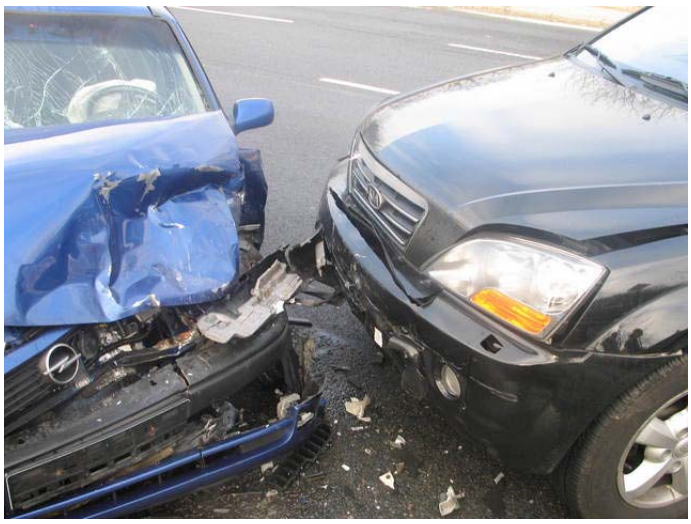

Fig. 9-1: Colisión

En la robótica se utiliza un control de los momentos axiales para detectar si el robot ha colisionado con un objeto. En la mayoría de los casos, esta colisión no es deseada y puede destruir el robot, la herramienta o componentes.

Control contra colisiones

- Si un robot colisiona con un objeto, la unidad de control del robot incrementa los momentos axiales para vencer la resistencia. Esto puede dañar el robot, la herramienta u otras piezas.
- La detección de colisiones reduce el riesgo y la gravedad de este tipo de daños. Controla los momentos axiales.
- El usuario puede determinar cómo se debe proceder tras una colisión después de que el algoritmo haya detectado una colisión y haya detenido el robot.
  - El robot se detiene con STOP 1.
  - La unidad de control del robot activa el programa tm_useraction. Se encuentra en la carpeta Programa y contiene la instrucción HALT. Como alternativa, el usuario puede programar otras reacciones en el programa tm_useraction.
- La unidad de control del robot calcula el rango de tolerancia automáticamente.
- Por norma general, un programa se debe ejecutar entre 2 y 3 veces hasta que la unidad de control del robot calcula un rango de tolerancia adecuado para la práctica.
- El usuario puede definir un offset mediante la interfaz de usuario para el rango de tolerancia calculado por la unidad de control del robot.
- Si el robot no se pone en marcha durante un período prolongado (p. ej., fin de semana), los motores, los engranajes, etc. se enfrían. Durante la primera marcha tras una pausa de este tipo, se necesitan momentos axiales distintos a los de un robot a temperatura de servicio. La unidad de control del robot adapta la detección de colisiones automáticamente a la temperatura modificada.

Restricciones

- En el **modo de servicio T1 no** es posible detectar las colisiones.

- En el caso de las posiciones iniciales y otras posiciones globales tampoco es posible detectar las colisiones.
- En el caso de los ejes adicionales tampoco se pueden detectar las colisiones.
- En caso de retroceso tampoco se pueden detectar las colisiones.
- Si el robot está parado, durante el arranque se dan momentos axiales muy altos. Por eso, en la fase de arranque (aprox. 700 ms) no se controlan los momentos axiales.
- Tras una modificación del override del programa, la detección de colisiones reacciona con una insensibilidad considerable durante 2 o 3 ejecuciones del programa. Después de esto la unidad de control del robot habrá adaptado ya el rango de tolerancia al nuevo override del programa.

Principio de la detección de colisiones

Programar por aprendizaje un programa con detección de colisiones

- La adaptación de la velocidad debe estar conectada con la variable de sistema `$ADAP_ACC`.
  - La variable del sistema se halla en el fichero C:\KRC\Roboter\KRC\R1\MaDa\$ROBCOR.DAT.
  - `$ADAP_ACC = #NONE`: la modificación de la aceleración no está activada.
  - `$ADAP_ACC = #STEP1`: modelo dinámico sin energía cinética.
  - `$ADAP_ACC = #STEP2`: modelo dinámico con energía cinética.
- Para conectar la detección de colisiones para un movimiento, durante la programación el parámetro **Detección de colisiones** se debe definir como TRUE. En el código del programa, esto se puede saber por el complemento `CD`:

```
PTP P2  Vel= 100 % PDAT1 Tool[1] Base[1] CD
```

> El parámetro **Detección de colisiones** sólo está disponible si el movimiento se programa mediante un formulario inline.

- El rango de tolerancia solo se calcula para conjuntos de movimiento realizados íntegramente.

Ajuste de los valores de offset

- Se puede definir un offset para el momento de fuerza y para el momento de impacto del rango de tolerancia.
- **Momento de fuerza:** el momento de fuerza tiene efecto cuando el robot se enfrenta a una resistencia prolongada. Ejemplos:
  - El robot colisiona con una pared y hace presión contra ella.
  - El robot colisiona con un contenedor. El robot hace presión contra el contenedor y lo mueve.
- **Momento de impacto:** el momento de impacto tiene efecto cuando el robot se enfrenta a una resistencia breve. Ejemplo:
  - El robot colisiona con una placa que sale despedida por el impacto.
- Cuanto menor sea el offset, más sensible será la reacción de la detección de colisiones.
- Cuanto mayor sea el offset, menos sensible será la reacción de la detección de colisiones.

> Si la detección de colisiones reacciona con demasiada sensibilidad, no incrementar el offset de forma inmediata. En lugar de ello, primero volver a calcular el rango de tolerancia y comprobar si la detección de colisiones reacciona ahora como se desea.

- Ventana de opciones: ventana de colisión

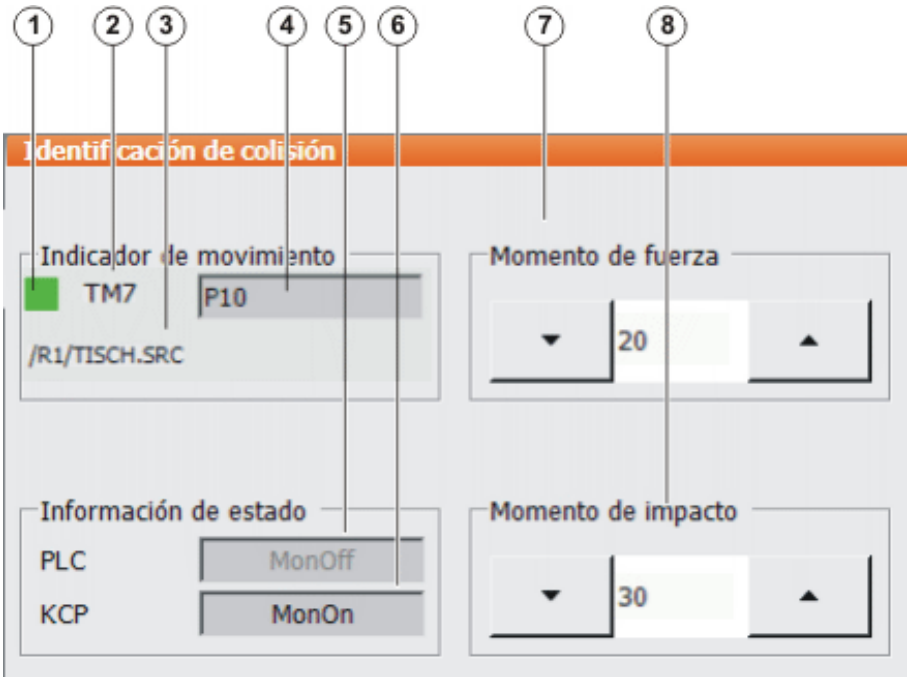

Fig. 9-2: Ventana de opciones Detección de colisiones

Los datos en la ventana de opción **Detección de colisiones** siempre hacen referencia al movimiento actual. Se pueden dar desviaciones, en especial en el caso de las distancias de punto cortas y de los movimientos aproximados.

Pos.	Descripción
1	El botón indica el estado de un movimiento. ▪ Rojo: el movimiento actual no se controla. ▪ Verde: el movimiento actual sí se controla. ▪ Naranja: se ha pulsado una tecla de ajuste de valores a izquierda y derecha, junto al valor numérico del momento de fuerza o del momento de impacto. La ventana permanece en el movimiento, y se puede modificar el offset. La modificación se puede aceptar con **Guardar**. ▪ Enclavado: por norma general, un programa se debe ejecutar entre 2 y 3 veces hasta que la unidad de control del robot calcula un rango de tolerancia adecuado para la práctica. Mientras la unidad de control del robot se encuentre en esta fase de aprendizaje, el botón se mostrará enclavado.
2	Número de las variables TMx Por cada conjunto de movimientos para los que el parámetro **Detección de colisiones** sea TRUE, la unidad de control del robot crea una variable TMx. TMx contiene todos los valores para el rango de tolerancia de este conjunto de movimientos. Si 2 conjunto de movimientos hacen referencia al mismo punto Px, la unidad de control del robot creará 2 variables TMx.
3	Ruta y nombre del programa seleccionado
4	Nombre del punto

Pos.	Descripción
5	Este campo sólo está activado en el modo de servicio "Automático externo". En los demás casos está en gris. **MonOn**: el PLC ha activado la detección de colisiones. Cuando se activa la detección de colisiones mediante el PLC, éste envía la señal de entrada **sTQM_SPSACTIVE** a la unidad de control del robot. La unidad de control del robot responde con la señal de salida **sTQM_SPSSTATUS**. Las señales están definidas en el fichero $config.dat. **Indicación:** En el modo Automático externo, la detección de colisiones sólo está activa si tanto en el campo **PLC** como en el campo **KCP** se muestra la entrada **MonOn**.
6	**MonOn**: el KCP ha activado la detección de colisiones. **Indicación:** En el modo Automático externo, la detección de colisiones sólo está activa si tanto en el campo **PLC** como en el campo **KCP** se muestra la entrada **MonOn**.
7	Offset para el momento de fuerza. Cuanto menor sea el offset, más sensible será la reacción de la detección de colisiones. Valor por defecto: 20. La ventana permanece en el movimiento, y se puede modificar el offset. La modificación se puede aceptar con **Guardar**. **Nota**: Para este movimiento, la opción **Detección de colisiones** en el formulario inline es FALSE.
8	Offset para el momento de fuerza. Cuanto menor sea el offset, más sensible será la reacción de la detección de colisiones. Valor por defecto: 30. La ventana permanece en el movimiento, y se puede modificar el offset. La modificación se puede aceptar con **Guardar**. **Nota**: Para este movimiento, la opción **Detección de colisiones** en el formulario inline es FALSE.

Botón	Descripción
Activar	Activa la detección de colisiones. Este botón no se muestra si se ha modificado el momento de fuerza o el momento de impacto pero las modificaciones no se han guardado todavía.
Desactivar	Desactiva la detección de colisiones. Este botón no se muestra si se ha modificado el momento de fuerza o el momento de impacto pero las modificaciones no se han guardado todavía.
Guardar	Acepta las modificaciones del momento de fuerza y/o el momento de impacto.
Cancelar	Desecha las modificaciones del momento de fuerza y/o el momento de impacto.

Procedimiento

Como alternativa, en este tipo de programas se pueden borrar las líneas con el control de momentos y utilizar la detección de colisiones en su lugar. La detección de colisiones no se debe utilizar en un mismo programa en combinación con el control de momentos.
La adaptación de la aceleración está activada cuando la variable del sistema $ADAP_ACC **no es igual** a #NONE. (Este es el ajuste por defecto). La variable del sistema se halla en el fichero C:\KRC\Roboter\KRC\R1\MaDa\$ROBCOR.DAT.

Programación de la detección de colisiones

1. Crear movimiento con el formulario en línea.
2. Abrir la ventana de opciones Frames y activar la detección de colisiones.

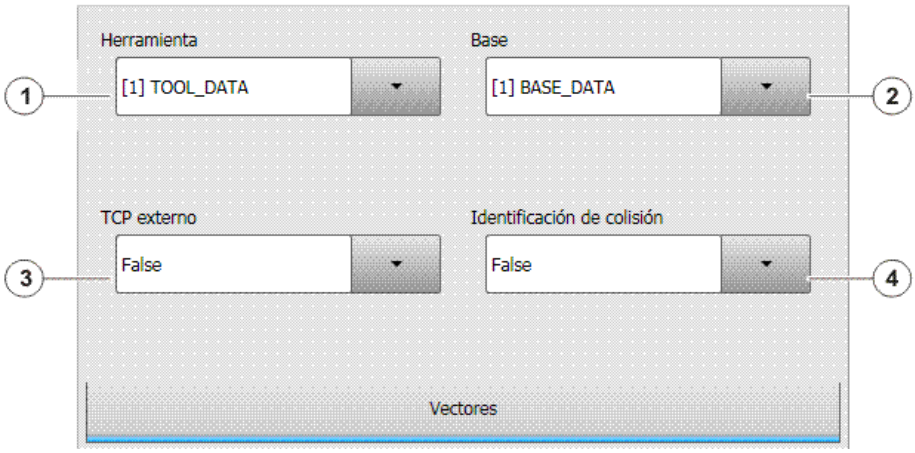

Fig. 9-3: Ventana de opciones Vectores

Pos.	Descripción
1	Seleccionar herramienta. Si consta **True** en el campo **TCP externo**: seleccionar pieza. Gama de valores: [1] … [16]
2	Seleccionar base. Si consta **True** en el campo **TCP externo**: seleccionar herramienta fija. Gama de valores: [1] … [32]
3	Modo de interpolación ▪ **False**: la herramienta se encuentra montada sobre la brida de acople. ▪ **True**: la herramienta es fija.
4	▪ **True**: para este movimiento la unidad de control del robot determina los momentos axiales, que son necesarios para la detección de colisiones. ▪ **False**: para este movimiento la unidad de control del robot no determina ningún momento axial. Por lo tanto, no es posible realizar una detección de colisiones en este caso.

3. Finalizar el movimiento.

Cálculo del rango de tolerancia y activación de la detección de colisiones

1. Seleccionar en el menú principal la secuencia **Configuración** > **Extras** > **Detección de colisiones**.
   (>>> Fig. 9-2)

2. En el campo **KCP** debe figurar la entrada **MonOff**. Si no fuera así, pulsar **Desactivar**.
3. Iniciar el programa y ejecutarlo varias veces. Al cabo de 2 o 3 ejecuciones del programa, la unidad de control del robot calcula un rango de tolerancia adecuado para la práctica.
4. Pulsar **Activar**. Ahora en la ventana **Detección de colisiones**, en el campo **KCP** figura la entrada **MonOn**.

   Guardar la configuración con **Cerrar**.

1. Seleccionar un programa.
2. Seleccionar en el menú principal la secuencia **Configuración** > **Extras** > **Detección de colisiones**.
3. El offset para un movimiento se puede modificar mientras se ejecuta un programa: cuando el movimiento deseado se muestre en la ventana **Detección de colisiones**, pulsar las teclas junto al momento de fuerza y el momento de impacto. La ventana se queda en este movimiento. Modificar el offset mediante estas teclas.

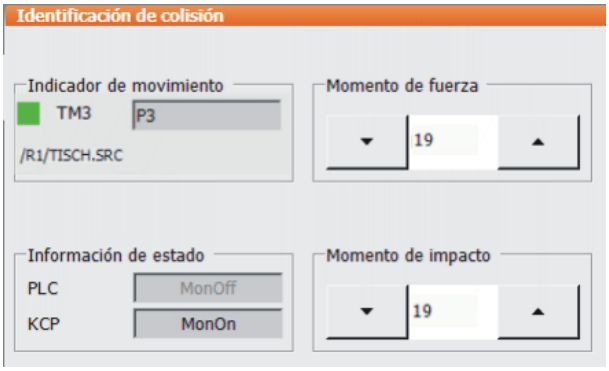

Fig. 9-6: Valores modificados de detección de colisiones

   Como alternativa se puede seleccionar un paso en el movimiento deseado.

4. Aceptar el cambio con **Guardar**.
5. Guardar la configuración con **Cerrar**.
6. Ajustar el modo de servicio original y el modo de ejecución del programa.

Indice