KUKA

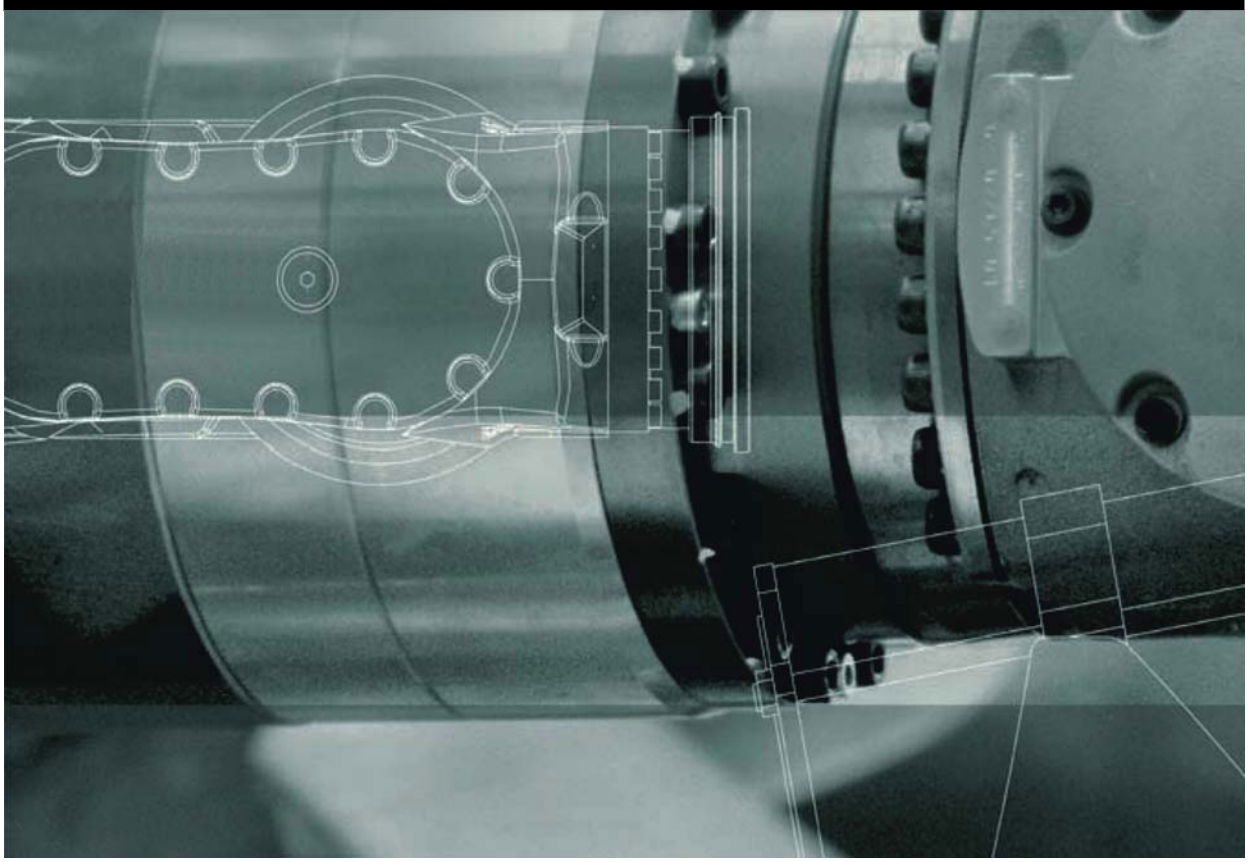
Training

KUKA Roboter GmbH

机器人编程 2

库卡系统软件 8

培训资料



发布日期 : 20.12.2011

版本 : P2KSS8 Roboterprogrammierung 2 V1 zh

KUKA Roboter GmbH
Zugspitzstraße 140
D-86165 Augsburg
德国

原版文件的翻译

KIM-PS5-DOC

Publication:	Pub COLLEGE P2KSS8 Roboterprogrammierung 2 (TG-COL) zh
Bookstructure:	P2KSS8 Roboterprogrammierung 2 V2.3
版本 :	P2KSS8 Roboterprogrammierung 2 V1 zh

目录

1 结构化编程

1.1 采用统一编程方法的目的

采用统一编程方法的目的

采用统一编程方法，以便：

- 通过严密的分段结构方便地解决复杂的问题
- 以清晰易懂的方式展示基本方法（无需深度编程知识）
- 提高维护、修改和扩展程序的效率

前瞻性程序规划可：

- 使复杂的任务得以分解成几个简单的分步任务
- 降低编程时的总耗时
- 使相同性能的组成部分得以更换
- 单独开发各组成部分

对一个机器人程序的 6 个要求：

1. 高效
2. 无误
3. 易懂
4. 维护简便
5. 清晰明了
6. 具有良好的经济效益

1.2 创建结构化机器人程序的辅助工具

注释有什么用处？

- 对程序内容或功能的说明
- 内容和用途可任意选择
- 改善程序的可读性
- 有利于程序结构化
- 注释的时效性由程序员负责
- KUKA 使用行注释
- 控制器不会将注释理解为句法

在什么时候和什么地方使用注释？

关于整个源程序的信息：

```
DEF PICK_CUBE()
; 该程序将方块从库中取出
; 作者： Max Mustermann
; 创建日期：2011.08.09
INI
...
END
```

源程序的分段：

```
DEF PALLETIZE()
;*****************************************************
;* 该程序将 16 个方块堆垛在工作台上 *
;* 作者： Max Mustermann-----------------------------*
;* 创建日期： 2011.08.09-----------------------*
;*****************************************************
INI
...
;------------ 位置的计算 ----------------
...
;------------16 个方块的堆垛 ---------------
...
;------------16 个方块的卸垛 ---------------
...
END
```

单行的说明：

```
DEF PICK_CUBE()

INI

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; 驶至抓取预备位置

LIN Grip_Pos ; 驶至方块抓取位置
...

END
```

对需执行的工作的说明：

```
DEF PICK_CUBE()

INI

; 此处还必须插入货盘位置的计算！

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; 驶至抓取预备位置

LIN Grip_Pos ; 驶至方块抓取位置

; 此处尚缺少抓爪的关闭

END
```

变为注释：

```
DEF Palletize()

INI

PICK_CUBE()

;CUBE_TO_TABLE()

CUBE_TO_MAGAZINE()

END
```

在机器人程序中使用 FOLD 有什么作用？

- 在 FOLD 里可以隐藏程序段
- FOLD 的内容对用户来说是不可见的
- FOLD 的内容完全如通常情况在程序运行流程中得到处理
- 通过使用 Fold 可改善程序的可读性

Fold 应用示例有哪些？

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

SET_EA            ; 由用户建立的 FOLD 关闭

PTP HOME Vel=100% DEFAULT  ; KUKA FOLD 关闭

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

END
```

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

SET_EA            ; 由用户建立的 FOLD 打开
$OUT[12]=TRUE
$OUT[102]=FALSE
PART=0
Position=0

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

PTP HOME Vel=100% Default

END
```

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

SET_EA            ; 由用户建立的 FOLD 关闭

PTP HOME Vel=100% DEFAULT  ; KUKA FOLD 打开
$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
```

为什么要使用子程序技术进行工作？

- 可以多次使用
- 避免程序码重复
- 节省存储空间
- 各组成部分可单独开发
- 随时可以更换具有相同性能的组成部分
- 使程序结构化
- 将总任务分解成分步任务
- 维护和排除程序错误更为方便

子程序的应用

```
DEF MAIN()

INI

LOOP

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

    IF $IN[1] THEN
        EXIT
    ENDIF

ENDLOOP

END
```

指令行的缩进有什么作用？

```
DEF INSERT()
INT PART, COUNTER
INI
PTP HOME Vel=100% DEFAULT
LOOP
     FOR COUNTER = 1 TO 20
         PART = PART+1
         ; 联机表格无法缩进！！！
PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table
     PTP XP5
   ENDFOR
...
ENDLOOP
```

合理命名的数据名称有什么作用？

为了能够正确解释机器人程序中的数据和信号函数，应在为其命名时使用意义明确的概念。其中包括：

- 输入和输出信号的长文本名称
- 工具与基坐标的名称
- 输入和输出的信号协定
- 点的名称

1.3 如何创建程序流程图

什么是程序流程图(PAP)？

- 用于程序流程结构化的工具
- 程序流程更加易读
- 结构错误更加易于识别
- 同时生成程序的文献

程序流程图图标

一个过程或程序的开始或结束

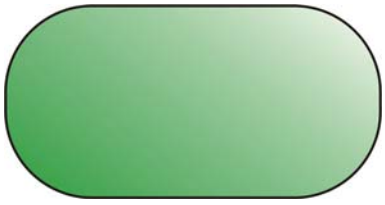


图 1-1

指令与运算的连接

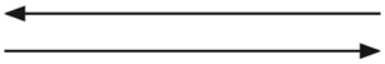


图 1-2

if 分支

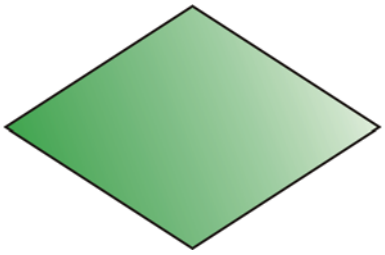


图 1-3

程序代码中的一般指令

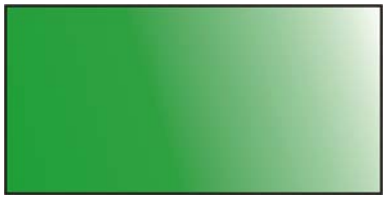


图 1-4

子程序调用

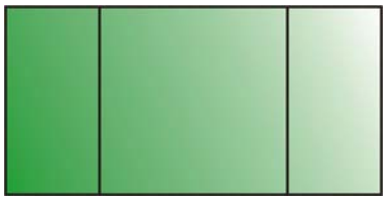


图 1-5

输入 / 输出指令

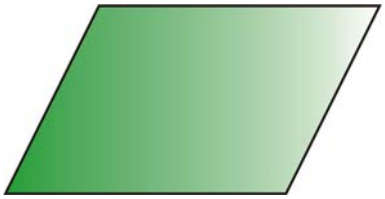


图 1-6

程序流程图示例

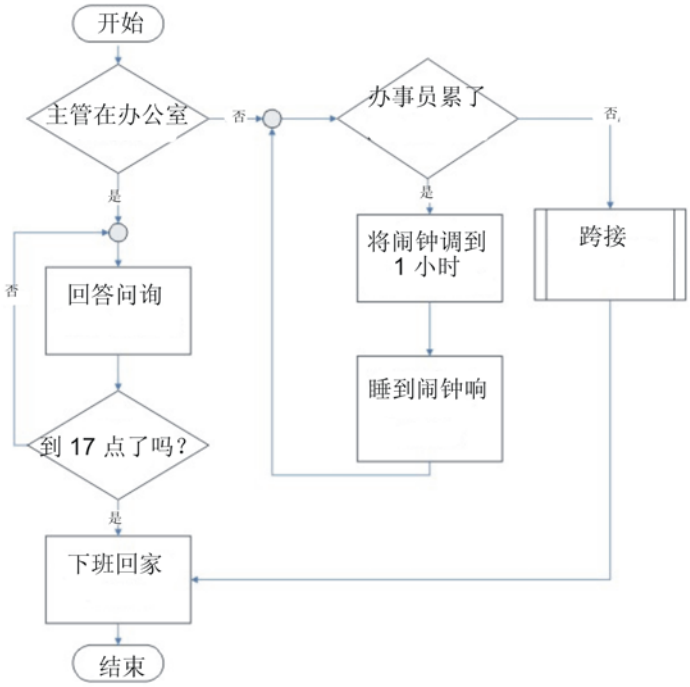


图 1-7

如何创建程序流程图

1. 在约 1 至 2 页的纸上将整个流程大致地划分
2. 将总任务划分成小的分步任务
3. 大致划分分步任务
4. 细分分步任务
5. 转换成 KRL 码

2 专家界面介绍

2.1 使用专家界面

说明

机器人控制器可向不同的**用户组**提供不同的功能。可以选择以下几个**用户组**：

- **操作人员**
  操作人员用户组。此为默认用户组。
- **应用人员**
  操作人员用户组。（在默认设置中操作人员和应用人员的目标群是一样的。）
- **专家**
  编程人员用户组。此用户组有密码保护。
- **管理员**
  功能与专家用户组一样。另外可以将插件（Plug-Ins）集成到机器人控制器中。此用户组有密码保护。
- **安全维护人员**
  该用户组可以激活和配置机器人的安全配置。此用户组有密码保护。
- **安全投入运行人员**
  只有当使用 KUKA.SafeOperation 或 KUKA.SafeRangeMonitoring 时，该用户组才相关。该用户组有密码保护。

专家用户组的扩展功能：

- 密码保护（默认：kuka）
- 可以借助 KRL 在编辑器中编程
- 模块的详细说明界面可供使用
- 显示 / 隐藏 DEF 行
- 展开和合拢折叠 (FOLD)
- 在程序中显示详细说明界面
- 创建程序时可从预定义的模板中选择
- 在下列情况下将自动退出专家用户组：
  - 当运行方式切换至 AUT（自动）或 AUT EXT（外部自动运行）时
  - 在一定的持续时间内未对操作界面进行任何操作时（300 秒）

功能

借助模板创建程序

- **Cell**：现有的 Cell 程序，只能被替换或者在删除 Cell 程序后重新创建。
- **Expert**：模块由只有程序头和程序结尾的 SRC 和 DAT 文件构成。
- **Expert Submit**：附加的 Submit 文件 (SUB) 由程序头和程序结尾构成
- **Function**：SRC 函数创建，在 SRC 中只创建带有 BOOL 变量的函数头。函数结尾已经存在，但必须对返回值进行编程。
- **Modul**：Modul（模块）由具有程序头、程序结尾以及基本框架 (INI 和 2 个 PTP HOME) 的 SRC 和 DAT 文件构成。
- **Submit**：附加的 Submit 文件 (SUB) 由程序头、程序结尾以及基本框架 (DECLARATION、INI、LOOP/ENDLOOP) 构成。

过滤器决定了在文件清单中如何显示程序。有以下**过滤器**可供选择：

- **详细信息**
  程序以 SRC 和 DAT 文件形式显示。（默认设置）
- **模块**
  程序以模块形式显示。

显示 / 隐藏 DEF 行

- 默认为不显示 **DEF 行**。当 **DEF 行**显示时才能在程序中进行声明。
- 对于那些被打开并选中了的程序来说，**DEF 行**将各自独立地显示或隐藏。如果详细说明界面打开，则 **DEF 行**将显示出来，无需专门进行显示操作。

打开 / 关闭 FOLD

- 对于应用人员，**FOLD** 始终关闭，但可以以专家身份打开
- 专家也可以编程设立自己的 **FOLD**
- Fold 的**句法**为：

```
;FOLD 名称
指令
;ENDFOLD <名称>
```

激活专家界面和纠错的操作步骤

激活专家界面

1. 在主菜单中选择 **配置 > 用户组**。
2. 作为**专家**登录：点击**登录**。选定用户组**专家**并用**登录**确认。
3. 此时需要：输入密码（默认：kuka）并用**登录**确认。

纠正程序中的错误

1. 在导航器中选择出错的模块

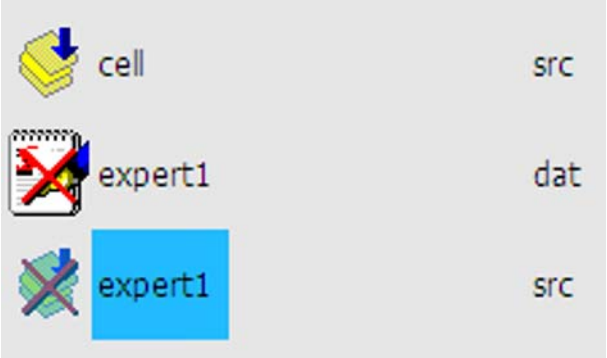


图 2-1: 有错误的程序

2. 选择菜单**错误列表**
3. 错误显示（*程序名* .ERR）随即打开
4. 选定错误，在下面的错误显示中将显出详细描述
5. 在错误显示窗口中按**显示**按键，跳到出错的程序中
6. 纠正错误
7. 退出编辑器并保存

3 变量和协定

3.1 KRL 中的数据保存

使用 KRL 以变量工作

变量概述

- 使用 KRL 对机器人进行编程时，从最普通的意义上来说，变量就是在机器人进程的运行过程中出现的计算值（“数值”）的容器。
- 每个变量都在计算机的存储器中有一个专门指定的地址。
- 每个变量都有一个非 KUKA 关键词的名称
- 每个变量都属于一个专门的数据类型
- 在使用前必须声明数据类型
- 在 KRL 中变量可划分为局部变量和全局变量

KRL 中变量的生存期

- 生存期是指变量预留存储空间的时间段
- 运行时间变量在退出程序或者函数时重新释放存储位置
- 数据列表中的变量持续获得存储位置中的当前值

KRL 中变量的有效性

- 声明为局部的变量只能在本程序中可用、可见
- 全局变量则在中央（全局）数据列表中创建
- 全局变量也可以在局部数据中创建，并在声明时配上关键词 **global（全局）**

KRL 的数据类型

- 数据类型是对某一集合中对象的统称
- 预定义的标准数据类型
- 自定义的标准数据类型
- 预定义的 KUKA 数据类型

使用 KRL 变量

命名规范

- KRL 中的名称长度最多允许 24 个字符。
- KRL 中的名称允许包含字母 (A-Z)、数字 (0-9) 以及特殊字符 “_” 和 “$”。
- KRL 中的名称不允许以数字开头。
- KRL 中的名称不允许为关键词。
- 不区分大小写

KRL 的数据类型

- **预定义的标准数据类型**

简单的数据类型	整数	实数	布尔数	单个字符
关键词	INT	REAL	BOOL	CHAR
数值范围	-2^{31} ... $(2^{31}-1)$	± 1.1 10^{-38}... ± 3.4 10^{+38}	TRUE / FALSE	ASCII 字符集
示例	-199 或 56	-0.0000123 或 3.1415	TRUE 或 FALSE	“A” 或 “q” 或 “7”

- **数组 /Array**

```
Voltage[10] = 12.75
Voltage[11] = 15.59
```

  - 借助下标保存相同数据类型的多个变量
  - 初始化或者更改数值均借助下标进行

- 最大数组的大小取决于数据类型所需的存储空间大小

- **枚举数据类型**

```
color = #red
```

  - 枚举类型的所有值在创建时会用名称（明文）进行定义
  - 系统也会规定顺序
  - 元素的最大数量取决于存储位置的大小

- **复合数据类型 / 结构**

```
Date = {day 14, month 12, year 1996}
```

  - 由不同数据类型的数据项组成的复合数据类型
  - 这些数据项可以由简单的数据类型组成，也可以由结构组成
  - 各个数据项均可以存取

生存期 / 有效性

- **在 SCR 文件中创建的变量被称为运行时间变量**
  - 不能被一直显示
  - 仅在声明的程序段中有效
  - 在到达程序的最后一行（END 行）时重新释放存储位置
- **局部 DAT 文件中的变量**
  - 在相关 SRC 文件的程序运行时可以一直被显示
  - 在完整的 SCR 文件中可用，因此在局部的子程序中也可用
  - 也可创建为全局变量
  - 获得 DAT 文件中的当前值，重新调用时以所保存的值开始
- **系统文件 $CONFIG.DAT 中的变量**
  - 在所有程序中都可用（全局）
  - 即使没有程序在运行，也始终可以被显示
  - 获得 $CONFIG.DAT 文件中的当前值

变量的双重声明

- 双重声明始终出现在使用相同的字符串（名称）时
- 如果在不同的 SRC 或 DAT 文件中使用相同的名称，则**不属**于双重声明
- 在同一个 SCR 和 DAT 文件中进行双重声明是不允许的，并且会生成错误信息
- 在 SRC 或 DAT 文件及 $CONFIG.DAT 中允许双重声明
  - 运行已定义好变量的程序时，只会更改局部值，而不会更改 $CONFIG.DAT 中的值
  - 运行“外部”程序时只会调用和修改 $CONFIG.DAT 中的值

KUKA 系统数据

- 系统数据类型有
  - 枚举数据类型，例如：运行方式 (mode_op)
  - 结构，例如：日期 / 时间 (date)
- 系统信息可从 KUKA 系统变量中获得
  - 读取当前的系统信息
  - 更改当前的系统配置
  - 已经预定义好并以“$”字符开始
    - $DATE（当前时间和日期）
    - $POS_ACT（当前机器人位置）
    - ...

3.2 涉及简单数据类型的工作

在下文中将介绍如何创建、初始化和改变变量。在此仅使用简单的数据类型。

使用 KRL 的简单数据类型

- 整数 (INT)
- 实数 (REAL)
- 布尔数 (BOOL)
- 单个字符 (CHAR)

3.2.1 变量声明

建立变量

变量声明

- 在使用前必须总是先进行声明
- 每一个变量均划归一种数据类型
- 命名时要遵守命名规范
- 声明的关键词为 **DECL**
- 对四种简单数据类型关键词 DECL 可省略
- 用预进指针赋值
- 变量声明可以不同形式进行，因为从中得出相应变量的生存期和有效性
  - 在 SRC 文件中声明
  - 在局部 DAT 文件中声明
  - 在 $CONFIG.DAT 中声明
  - 在局部 DAT 文件中配上关键词 “ 全局 ” 声明
- 创建常量
  - 常量用关键词 **CONST** 建立
  - 常量只允许在数据列表中建立

变量声明的原理

SRC 文件中的程序结构

- 在声明部分必须声明变量
- 初始化部分从第一个赋值开始，但通常都是从 “INI” 行开始
- 在指令部分会赋值或更改值

```
DEF main( )
; 声明部分
...
; 初始化部分
INI
...
; ????
PTP HOME Vel=100% DEFAULT
...
END
```

更改标准界面

- 只有作为专家才能使 DEF 行显示
- 为了在模块时于 “INI” 行前进入声明部分，这是必要的
- 在将变量传递到子程序中时能够看到 DEF 和 END 行也非常重要

计划变量声明

- 规定生存期
  - **SCR** 文件：程序运行结束时，运行时间变量 “ 死亡 ”
  - **DAT** 文件：在程序运行结束后变量还保持着

- 规定有效性 / 可用性
  - 在局部 SRC 文件中：仅在程序中被声明的地方可用。因此变量仅在局部 DEF 和 END 行之间可用（主程序或局部子程序）
  - 在局部 DAT 文件中：在整个程序中有效，即在所有的局部子程序中也有效
  - $CONFIG.DAT：全局可用，即在所有程序中都可以读写
  - 在局部 DAT 文件中作为全局变量：全局可用，只要为 DAT 文件指定关键词 PUBLIC 并在声明时再另外指定关键词 GOLBAL，就在所有程序中都可以读写
- 规定数据类型
  - BOOL：经典式 “ 是 ”/“ 否 ” 结果
  - REAL：为了避免四舍五入出错的运算结果
  - INT：用于计数循环或件数计数器的经典计数变量
  - CHAR：仅一个字符

    字符串或文本只能作为 CHAR 数组来实现
- 命名和声明
  - 使用 DECL，以使程序便于阅读
  - 使用可让人一目了然的合理变量名称
  - 请勿使用晦涩难懂的名称或缩写
  - 使用合理的名称长度，即不要每次都使用 24 个字符

在声明具有简单数据类型变量时的操作步骤

在 SCR 文件中创建变量

1. 专家用户组
2. 使 DEF 行显示出来
3. 在编辑器中打开 SCR 文件
4. 声明变量

```
DEF MY_PROG ( )
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
INI
...
END
```

5. 关闭并保存程序

在 DAT 文件中创建变量

1. 专家用户组
2. 在编辑器中打开 DAT 文件
3. 声明变量

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT
```

4. 关闭并保存数据列表

在 $CONFIG.DAT 中创建变量

1. 专家用户组
2. 在编辑器中打开 SYSTEM（系统）文件夹中的 $CONFIG.DAT

```
DEFDAT $CONFIG
BASISTECH GLOBALS
AUTOEXT GLOBALS
USER GLOBALS
ENDDAT
```

3. 选择 Fold“USER GLOBALS”，然后用软键 “ 打开 / 关闭 Fold” 将其打开
4. 声明变量

```
DEFDAT $CONFIG ( )
...
;=================================
; 用户自定义类型
;=================================
;=================================
; 外部用户自定义
;=================================
;=================================
; 用户自定义变量
;=================================
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT
```

5. 关闭并保存数据列表

在 DAT 文件中创建全局变量

1. 专家用户组
2. 在编辑器中打开 DAT 文件
3. 通过关键词 PULIC 扩展程序头中的数据列表

```
DEFDAT MY_PROG PUBLIC
```

4. 声明变量

```
DEFDAT MY_PROG PUBLIC
EXTERNAL DECLARATIONS
DECL GLOBAL INT counter
DECL GLOBAL REAL price
DECL GLOBAL BOOL error
DECL GLOBAL CHAR symbol
...
ENDDAT
```

5. 关闭并保存数据列表

3.2.2 简单数据类型变量的初始化

KRL 初始化说明

- 每次声明后变量都只预留了一个存储位置，值总是无效值
- 在 SRC 文件中声明和初始化始终在两个独立的行中进行
- 在 DAT 文件中声明和初始化始终在一行中进行
  常量必须在声明时立即初始化
- 初始化部分以第一次赋值开始

初始化的方法

整数的初始化

- 初始化为十进制数

```
value = 58
```

- 初始化为二进制数

```
value = 'B111010'
```

二进制	2^5	2^4	2^3	2^2	2^1	2^0
十进制	32	16	8	4	2	1

计算：1*32+1*16+1*8+0*4+1*2+0*1 = 58

- 初始化为十六进制数

```
value = 'H3A'
```

十六进制	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
十进制	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

计算：3*16 +10 = 58

使用 KRL 初始化时的操作步骤

在 SRC 文件中声明和初始化

1. 在编辑器中打开 SCR 文件
2. 已声明完毕
3. 执行初始化

```
DEF MY_PROG ( )
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
INI
counter = 10
price = 0.0
error = FALSE
symbol = "X"
...
END
```

4. 关闭并保存程序

在 DAT 文件中声明和初始化

1. 在编辑器中打开 DAT 文件
2. 已声明完毕
3. 执行初始化

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter = 10
DECL REAL price = 0.0
DECL BOOL error = FALSE
DECL CHAR symbol = "X"
...
ENDDAT
```

4. 关闭并保存数据列表

在 DAT 文件中声明和在 SRC 文件中初始化

1. 在编辑器中打开 DAT 文件
2. 进行声明

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT
```

3. 关闭并保存数据列表
4. 在编辑器中打开 SCR 文件
5. 执行初始化

```
DEF MY_PROG ( )
...
INI
counter = 10
price = 0.0
error = FALSE
symbol = "X"
...
END
```

6. 关闭并保存程序

常量的声明和初始化

1. 在编辑器中打开 DAT 文件
2. 进行声明和初始化

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL CONST INT max_size = 99
DECL CONST REAL PI = 3.1415
...
ENDDAT
```

3. 关闭并保存数据列表

3.2.3 用 KRL 对简单数据类型的变量值进行操纵

用 KRL 修改变量值的方法列表

根据具体任务，可以以不同方式在程序进程（SRC 文件）中改变变量值。以下介绍最常用的方法。也可借助于位运算和标准函数进行操纵，但不在此深入介绍。

通过以下途径进行数据操纵

- 基本运算类型
  - (+) 加法
  - (-) 减法
  - (*) 乘法
  - (/) 除法
- 比较运算
  - (==) 相同 / 等于
  - (<>) 不同
  - (>) 大于
  - (<) 小于
  - (>=) 大于等于
  - (<=) 小于等于
- 逻辑运算
  - (NOT) 反向
  - (AND) 逻辑“与”

- (OR) 逻辑 “ 或 ”
- (EXOR)“ 异或 ”

- 位运算
  - (B_NOT) 按位取反运算
  - (B_AND) 按位与
  - (B_OR) 按位或
  - (B_EXOR) 按位异或

标准函数

- 绝对函数
- 根函数
- 正弦和余弦函数
- 正切函数
- 反余弦函数
- 反正切函数
- 多种字符串处理函数

数据操纵时的关系

使用数据类型 REAL 和 INT 时的数值更改

- 四舍五入

```
; 声明
DECL INT A,B,C
DECL REAL R,S,T
; 初始化
A = 3      ; A=3
B = 5.5    ; B=6 ( x.5 起凑成整数 )
C = 2.25   ; C=2 ( 四舍五入 )
R = 4      ; R=4.0
S = 6.5    ; S=6.5
T = C      ; T=2.0 ( 应用四舍五入后的整数 )
```

- 数学运算结果 (+;-;*)

运算对象	INT	REAL
INT	INT	REAL
REAL	REAL	REAL

```
; 声明
DECL INT D,E
DECL REAL U,V
; 初始化
D = 2
E = 5
U = 0.5
V = 10.6
; 指令部分 ( 数据操纵 )
D = D*E ; D = 2 * 5 = 10
E = E+V ; E= 5 + 10.6 = 15.6 -> 四舍五入为 E=16
U = U*V ; U= 0.5 * 10.6 = 5.3
V = E+V ; V= 16 + 10.6 = 26.6
```

- 数学运算结果 (/)

  使用整数值运算时的特点 :
  - 纯整数运算的中间结果会去掉所有小数位
  - 给整数变量赋值时会根据一般计算规则对结果进行四舍五入

```
; 声明
DECL INT F
DECL REAL W
; 初始化
F = 10
W = 10.0
; 指令部分 （数据操纵）
; INT / INT -> INT
F = F/2 ; F=5
F = 10/4 ; F=2 （10/4 = 2.5 -> 省去小数点后面的尾数）
; REAL / INT -> REAL
F = W/4 ; F=3 （10.0/4=2.5 -> 四舍五入为整数）
W = W/4 ; W=2.5
```

比较运算

通过比较运算可以构成逻辑表达式。比较结果始终是 BOOL 数据类型

运算符 /KRL	说明	允许的数据类型
==	等于 / 相等	INT、REAL、CHAR、BOOL
<>	不等	INT、REAL、CHAR、BOOL
>	大于	INT、REAL、CHAR
<	小于	INT、REAL、CHAR
>=	大于等于	INT、REAL、CHAR
<=	小于等于	INT、REAL、CHAR

```
; 声明
DECL BOOL G,H
; 初始化 / 指令部分
G = 10>10.1 ; G=FALSE
H = 10/3 == 3 ; H=TRUE
G = G<>H ; G=TRUE
```

逻辑运算

通过逻辑运算可以构成逻辑表达式。这种运算的结果始终是 BOOL 数据类型

运算		NOT A	A AND B	A OR B	A EXOR B
A=TRUE	B=TRUE	FALSE	TRUE	TRUE	FALSE
A=TRUE	B=FALSE	FALSE	FALSE	TRUE	TRUE
A=FALSE	B=TRUE	TRUE	FALSE	TRUE	TRUE
A=FALSE	B=FALSE	TRUE	FALSE	FALSE	FALSE

```
; 声明
DECL BOOL K,L,M
; 初始化 / 指令部分
K = TRUE
L = NOT K ; L=FLASE
M = (K AND L) OR (K EXOR L) ; M=TRUE
L = NOT (NOT K) ; L=TRUE
```

运算将根据其优先级顺序进行

优先级	运算符
1	NOT (B_NOT)
2	乘 (*) ；除 (/)
3	加 (+)，减 (-)
4	AND (B_AND)
5	EXOR (B_EXOR)
6	OR (B_OR)
7	各种比较 (==; <>; ...)

```
; 声明
DECL BOOL X, Y
DECL INT Z
; 初始化 / 指令部分
X = TRUE
Z = 4
Y = (4*Z+16 <> 32) AND X ; Y=FALSE
```

数据操纵时的操作步骤

1. 确定一个或多个变量的数据类型
2. 确定变量的有效性和生存期
3. 进行变量声明
4. 初始化变量
5. 在程序运行中，即始终在 SCR 文件中对变量进行操纵
6. 关闭并保存 SRC 文件

3.3 KRL Arrays / 数组

KRL 数组的说明

数组，亦即 Arrays，可为具有相同数据类型并借助下标区分的多个变量提供存储位置

- 数组的存储位置是有限的，即最大数组的大小取决于数据类型所需的存储空间大小
- 声明时，数组大小和数据类型必须已知
- KRL 中的起始下标始终从 1 开始
- 初始化始终可以逐个进行
- 在 SRC 文件中的初始化也可以采用循环方式进行

数组维数

- 1 维数组

```
dimension1[4]= TRUE
```

- 2 维数组

```
dimension2[2,1]= 3.25
```

- 3 维数组

```
dimension1[3,4,1]= 21
```

- KRL 不支持 4 维及 4 维以上的数组

使用数组时的关联：

数组变量的生存期和有效性与使用简单数据类型的变量时相同

数组声明

- 在 SCR 文件中建立

```
DEF MY_PROG ( )
DECL BOOL error[10]
DECL REAL value[50,2]
DECL INT parts[10,10,10]
INI
...
END
```

- 在数据列表（亦 $CONFIG.DAT）中建立

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
DECL REAL value[50,2]
DECL INT parts[10,10,10]
...
ENDDAT
```

在 SRC 文件中对数组进行声明并初始化

- 通过调用索引单独对每个数组进行声明和初始化

```
DECL BOOL error[10]
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE
```

- 以合适的循环

```
DECL BOOL error[10]
DECL INT x
FOR x = 1 TO 10
error[x]=FALSE
ENDFOR
```

循环结束后 x 的值为 11

在数据列表中初始化数组

- 在每一个数组的数据列表中通过调用索引单独进行，接着将值显示在数据列表中

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE
```

- 在数据列表中不允许进行的声明和初始化

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
DECL INT size = 32
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE
```

将生成 10 条 “ 初始值语句不在初始化部分内 ” 的出错信息

在数据列表中对数组进行声明并在 SRC 文件中进行初始化

- 假如数组是如此建立在数据列表中的，则**不能**在数据列表中查看当前值；**只能**通过变量显示检查当前值。

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
```

```
DEF MY_PROG ( )
INI
Fehler[1]=FALSE
Fehler[2]=FALSE
Fehler[3]=FALSE
...
Fehler[10]=FALSE
```

或

```
DEF MY_PROG ( )
INI
FOR x = 1 TO 10
Fehler[x]=FALSE
ENDFOR
```

借助于循环进行初始化

- 1 维数组

```
DECL INT parts[15]
DECL INT x
FOR x = 1 TO 15
parts[x]= 4
ENDFOR
```

- 2 维数组

```
DECL INT parts_table[10,5]
DECL INT x, y
FOR x = 1 TO 10
    FOR y = 1 TO 5
       parts_table[x, y]= 6
    ENDFOR
ENDFOR
```

- 3 维数组

```
DECL INT parts_palette[5,4,3]
DECL INT x, y, z
FOR x = 1 TO 5
    FOR y = 1 TO 4
        FOR z = 1 TO 3
            parts_palette[x, y, z]= 12
        ENDFOR
    ENDFOR
ENDFOR
```

使用 Arrays 时的操作步骤

1. 确定数组的数据类型
2. 确定数组的有效性和生存期
3. 进行数组声明
4. 初始化数组元素
5. 在程序运行中，即始终在 SCR 文件中对数组进行操作
6. 关闭并保存 SRC 文件

```
DEF MY_PROG ( )
DECL REAL palette_size[10]
DECL INT counter
INI
; 初始化
FOR counter = 1 TO 10
  palette_size[counter] = counter * 1.5
ENDFOR
...
; 单个更改值
palette_size[8] = 13
...
; 值比较
IF palette_size[3] > 4.2 THEN
...
```

3.4 KRL 结构

包含多种单一信息的变量

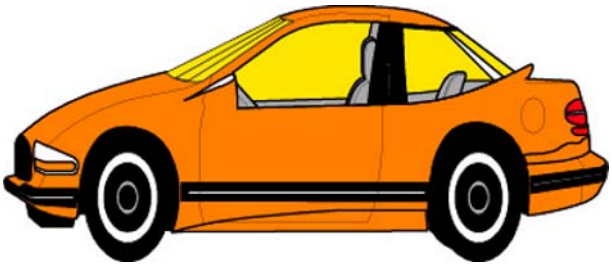


图 3-1

复合型数据类型：**结构**

- 用数组可将同种数据类型的变量汇总。但在现实中，大多数变量是由不同数据类型构成的。
- 例如，对一辆汽车而言，发动机功率或里程数为整数型。对价格而言，实数型最适用。而空调设备的存在则与此相反，更应为布尔型。
- 所有部分汇总起来可描述一辆汽车。
- 用关键词 STRUC 可自行定义一个结构
- 结构是不同数据类型的组合

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

- 一种结构必须首先经过定义，然后才能继续使用

结构的使用

结构的可用性 / 定义

- 在结构中可使用简单的数据类型 INT、REAL、BOOL 及 CHAR

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

- 在结构中可以嵌入 CHAR 数组

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, CHAR
car_model[15]
```

- 在结构中也可以使用诸如位置 POS 等已知结构

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, POS
car_pos
```

- 定义完结构后还必须对此声明工作变量

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
DECL CAR_TYPE my_car
```

结构的初始化 / 更改

- 初始化可通过括号进行
- 通过括号初始化时只允许使用常量（固定值）
- 赋值顺序可以不用理会

```
my_car = {motor 50, price 14999.95, air_condition = TRUE}
```

```
my_car = {price 14999.95, motor 50, air_condition = TRUE}
```

- 在结构中不必指定所有结构元素
- 一个结构将通过一个结构元素进行初始化
- 未初始化的值已被或将被设置为未知值

```
my_car = {motor 75} ; 价格未知
```

- 初始化也可以通过点号进行

```
my_car.price = 9999.0
```

- 通过点号进行初始化时也可以使用变量

```
my_car.price = value_car
```

- 结构元素可随时通过点号逐个进行重新更改

```
my_car.price = 12000.0
```

有效性 / 生存期

- 创建的局部结构在到达 END 行时便无效
- 在多个程序中使用的结构必须在 $CONFIG.DAT 中进行声明

命名

- 不允许使用关键词
- 为了便于辨认，自定义的结构应以 **TYPE** 结尾

KUKA 经常以保存在系统中的预设定结构工作。示例见位置和信息编程时

位置范围内预设定的 KUKA 结构

- **AXIS**：STRUC AXIS REAL A1, A2, A3, A4, A5, A6
- **E6AXIS**：STRUC E6AXIS REAL A1, A2, A3, A4, A5, A6, E1, E2, E3, E4, E5, E6
- **FRAME**：STRUC FRAME REAL X, Y, Z, A, B, C
- **POS**：STRUC FRAME REAL X, Y, Z, A, B, C
- **E6POS**：STRUC E6POS REAL X, Y, Z, A, B, C, E1, E2, E3, E4, E5, E6 INT S,T

带一个位置的结构的初始化

- 通过括号初始化时只允许使用常量（固定值）

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, POS
car_pos
DECL CAR_TYPE my_car
my_car = {price 14999.95, motor 50, air_condition = TRUE, car_pos {X
1000, Y 500, A 0}}
```

- 初始化也可以通过点号进行

```
my_car.price = 14999.95
my_car.car_pos = {X 1000, Y 500, A 0}}
```

- 通过点号进行初始化时也可以使用变量

```
my_car.price = 14999.95
my_car.car_pos.X = x_value
my_car.car_pos.Y = 750
```

创建结构

1. 结构的定义

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

2. 工作变量声明

```
DECL CAR_TYPE my_car
```

3. 工作变量的初始化

```
my_car = {motor 50, price 14999.95, air_condition = TRUE}
```

4. 值的更改和 / 或工作变量的值比较

```
my_car.price = 5000.0
```

```
my_car.price = value_car
```

```
IF my_car.price >= 20000.0 THEN
...
ENDIF
```

3.5 枚举数据类型 ENUM

明文用作变量值

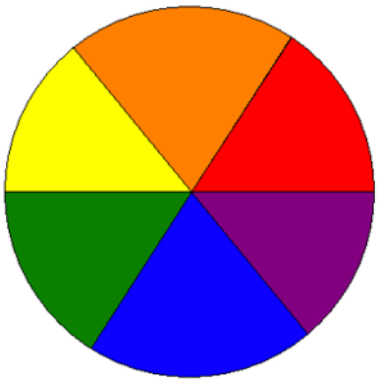


图 3-2

- 枚举数据类型由一定量的常量（例如红、黄、或蓝）组成

```
ENUM COLOR_TYPE green, blue, red, yellow
```

- 常量是可自由选择的名称
- 常量由编程员确定
- 一种枚举类型必须首先经过定义，然后才能继续使用

- 一个诸如 COLOR_TYPE 型箱体颜色的工作变量只能总是接受一个常量的一个值
- 一个常量的赋值始终以符号 # 进行

枚举数据类型的应用

可用性 / 应用

- 只能使用已知常量
- 枚举类型可扩展任意多次
- 枚举类型可单独使用

```
ENUM COLOR_TYPE green, blue, red, yellow
```

- 枚举类型可嵌入结构中

```
ENUM COLOR_TYPE green, blue, red, yellow
STRUC CAR_TYPE INT motor, REAL price, COLOR_TYPE car_color
```

有效性 / 生存期

- 创建的局部枚举类型在到达 END 行时便无效
- 在多个程序中使用的枚举类型必须在 $CONFIG.DAT 中进行声明

命名

- 枚举类型及其常量的名称应一目了然
- 不允许使用关键词
- 为了便于辨认，自定义的枚举类型应以 **TYPE** 结尾

生成枚举数据类型

1. 枚举变量和常量的定义

```
ENUM LAND_TYPE de, be, cn, fr, es, br, us, ch
```

2. 工作变量声明

```
DECL LAND_TYPE my_land
```

3. 工作变量的初始化

```
my_land = #be
```

4. 工作变量的值比较

```
IF my_land == #es THEN
...
ENDIF
```

4 子程序和函数

4.1 用局部子程序工作

局部子程序的定义

- 局部子程序位于主程序之后并以 DEF Name_Unterprogramm( ) 和 END 标明

```
DEF MY_PROG( )
; 此为主程序
...
END
_____________________________________
DEF LOCAL_PROG1( )
; 此为局部子程序 1
...
END
_____________________________________
DEF LOCAL_PROG2( )
; 此为局部子程序 2
...
END
_____________________________________
DEF LOCAL_PROG3( )
; 此为局部子程序 3
...
END
```

- SRC 文件中最多可由 255 个局部子程序组成
- 局部子程序允许多次调用
- 局部程序名称需要使用括号

用局部子程序工作时的关联

- 运行完毕局部子程序后，跳回到调出子程序后面的第一个指令

```
DEF MY_PROG( )
; 此为主程序
...
LOCAL_PROG1( )
...
END
_____________________________________
DEF LOCAL_PROG1( )
...
LOCAL_PROG2( )
...
END
_____________________________________
DEF LOCAL_PROG2( )
...
END
```

- 最多可相互嵌人 20 个子程序
- 点坐标保存在所属的 DAT 列表中，可用于整个文件

```
DEF MY_PROG( )
; 此为主程序
...
PTP P1 Vel=100% PDAT1
...
END
_____________________________________
DEF LOCAL_PROG1( )
...
; 与主程序中相同的位置
PTP P1 Vel=100% PDAT1
...
END
```

```
DEFDAT MY_PROG( )
...
DECL E6POS XP1={X 100, Z 200, Z 300 ... E6 0.0}
...
ENDDAT
```

- 用 RETURN 可结束子程序，并由此跳回到先前调用该子程序的程序模块中

```
DEF MY_PROG( )
; 此为主程序
...
LOCAL_PROG1( )
...
END
_____________________________________
DEF LOCAL_PROG1( )
...
IF $IN[12]==FALSE THEN
RETURN ; 跳回主程序
ENDIF
...
END
```

创建局部子程序的操作步骤

1. 专家用户组
2. 使 DEF 行显示出来
3. 在编辑器中打开 SCR 文件

```
DEF MY_PROG( )
...
END
```

4. 用光标跳到 END 行**下方**
5. 通过 DEF、程序名称和括号指定新的局部程序头

```
DEF MY_PROG( )
...
END
DEF PICK_PART( )
```

6. 通过 END 命令结束新的子程序

```
DEF MY_PROG( )
...
END
DEF PICK_PART( )
END
```

7. 用回车键确认后会在主程序和子程序之间插入一个横条

```
DEF MY_PROG( )
...
END

_____________________________________
DEF PICK_PART( )
END
```

8. 这时便可以继续编辑主程序和子程序
9. 关闭并保存程序

4.2 用全局子程序工作

全局子程序的定义

- 全局子程序有单独的 SRC 和 DAT 文件

```
DEF GLOBAL1( )
...
END
```

```
DEF GLOBAL2( )
...
END
```

- 全局子程序允许多次调用

用局部子程序工作时的关联

- 运行完毕局部子程序后，跳回到调出子程序后面的第一个指令

```
DEF GLOBAL1( )
...
GLOBAL2( )
...
END
```

```
DEF GLOBAL2( )
...
GLOBAL3( )
...
END
```

```
DEF GLOBAL3( )
...
END
```

- 最多可相互嵌入 20 个子程序
- 点坐标保存在各个所属的 DAT 列表中，并仅供相关程序使用

```
DEF GLOBAL1( )
...
PTP P1 Vel=100% PDAT1
END
```

```
DEFDAT GLOBAL1( )
DECL E6POS XP1={X 100, Z 200, Z 300 ... E6 0.0}
ENDDAT
```

Global2 ( ) 中 P1 的不同坐标

```
DEF GLOBAL2( )
...
PTP P1 Vel=100% PDAT1
END
```

```
DEFDAT GLOBAL2( )
DECL E6POS XP1={X 800, Z 775, Z 999 ... E6 0.0}
ENDDAT
```

- 用 RETURN 可结束子程序，并由此跳回到先前调用该子程序的程序模块中

```
DEF GLOBAL1( )
...
GLOBAL2( )
...
END
```

```
DEF GLOBAL2( )
...
IF $IN[12]==FALSE THEN
RETURN ; 返回 GLOBAL1( )
ENDIF
...
END
```

使用全局子程序编程时的操作步骤

1. 专家用户组
2. 新建程序

```
DEF MY_PROG( )
...
END
```

3. 新建第二个程序

```
DEF PICK_PART( )
...
END
```

4. 在编辑器中打开程序 MY_PROG 的 SCR 文件
5. 借助程序名和括号编程设定子程序的调用

```
DEF MY_PROG( )
...
PICK_PART( )
...
END
```

6. 关闭并保存程序

4.3 将参数传递给子程序

参数传递说明

- 句法

```
DEF MY_PROG( )
...
CALC (K, L)
...
END
```

```
DEF CALC(R:IN, S:OUT)
...
END
```

- 可通过两种方法将参数传递给子程序
  - 作为 IN 参数
  - 作为 OUT 参数
- 既可将参数传给局部的子程序，也可传给全局子程序

参数传递的原理

- 作为 IN 参数的参数传递 *(Call by value)*：
  - 变量值在主程序中保持不变，即变量以主程序原来的值继续工作
  - 子程序只能读取变量值，但不能写入
- 作为 OUT 参数的参数传递 *(Call by reference)*：
  - 变量值会在主程序中同时更改，即变量应用子程序的值
  - 子程序读取并更改该值，然后返回新的值
- 将参速传递给局部子程序

```
DEF MY_PROG( )
DECL REAL r,s
...
CALC_1(r)
...
CALC_2(s)
...
END

DEF CALC_1(num1:IN)
; 值 "r" 仅为只读传递至 num1
DECL REAL num1
...
END

DEF CALC_2(num2:OUT)
; 值 "s" 传递至 num2、更改并传回写入
DECL REAL num2
...
END
```

- 将参速传递给全局子程序

```
DEF MY_PROG( )
DECL REAL r, s
...
CALC_1(r)
...
CALC_2(s)
...
END
```

```
DEF CALC_1(num1:IN)
; 值 "r" 仅为只读传递至 num1
DECL REAL num1
...
END
```

```
DEF CALC_2(num2:OUT)
; 值 "s" 传递至 num2、更改并传回
DECL REAL num2
...
END
```

- 始终可以向相同的数据类型进行值传递
- 向其它数据类型进行值传递：

```
DEF MY_PROG( )
DECL DATATYPE1 value
CALC(value)
END

DEF CALC(num:IN)
DECL DATATYPE2 num
...
END
```

数据类型 1	数据类型 2	备注
BOOL	INT、REAL、CHAR	错误（... 参数不兼容）
INT	REAL	INT 值被用作 REAL 值
INT	CHAR	使用 ASCII 表中的字符
CHAR	INT	使用 ASCII 表中的 INT 值
CHAR	REAL	使用 ASCII 表中的 REAL 值
REAL	INT	REAL 值被四舍五入
REAL	CHAR	REAL 值被四舍五入，使用 ASCII 表中的字符

- 多参数传递

```
DEF MY_PROG( )
DECL REAL w
DECL INT a, b
...
CALC(w, b, a)
...
CALC(w, 30, a)
...
END

DEF CALC(ww:OUT, bb:IN, aa:OUT)
;1.) w <-> ww, b -> bb, a <-> aa
;2.) w <-> ww, 30 -> bb, a <-> aa
DECL REAL ww
DECL INT aa, bb
...
END
```

如果在子程序中也不计算这个值，则可以不传递该值。示例：
`RECHNE(w,, a)`

- 使用数组进行参数传递
  - 数组只能被整个传递到一个新的数组中

- 数组只允许以参数 OUT (Call by reference) 的方式进行传递

```
DEF MY_PROG( )
DECL CHAR name[10]
...
name="PETER"
RECHNE(name[])
...
END

_____________________________________
DEF RECHNE(my_name[]:OUT)
; 子程序中的数组应始终无数组大小创建
; 数组大小与输出端数组适配
DECL CHAR my_name[]
...
END
```

传递**整个**数组：FELD_1D[]（1 维），FELD_2D[,]（2 维），FELD_3D[,,]?3 ??

- 单个数组元素也可以被传递

```
DEF MY_PROG( )
DECL CHAR name[10]
...
name="PETER"
CALC(name[1])
...
END

_____________________________________
DEF RECHNE(symbol:IN)
; 仅传递一个字符
DECL CHAR symbol
...
END
```

在传递单个数组元素时，只允许变量作为目标，且不允许数组作为目标。此处仅将字母 “P” 传递到子程序中

参数传递时的操作步骤

预先考虑

1. 确定在子程序中需要哪些参数
2. 确定参数传递的种类（IN 或 OUT 参数）
3. 确定原始数据类型和目标数据类型（数据类型最好相同）
4. 确定参数传递的顺序

须注意：最先发送的参数被写到子程序中的第一个参数上，第二发送的参数被写到子程序中的第二个参数上，以此类推。

1. 将主程序载入编辑器
2. 在主程序中声明、初始化及可能需要操纵变量
3. 通过变量调用创建子程序调用
4. 关闭并保存主程序
5. 将子程序载入编辑器
6. 在 DEF 行中补充变量及 IN/OUT
7. 在子程序中声明、初始化及可能需要操纵变量
8. 关闭并保存子程序

完整的示例：

```
DEF MY_PROG( )
DECL REAL w
DECL INT a, Anzahl
w = 1.5
a = 3
b = 5
CALC(w, b, a)
; 当前值
; w = 3.8
; a = 13
; b = 5
END

DEF CALC(ww:OUT, bb:IN, aa:OUT)
; w <-> ww, b -> bb, a <-> aa

DECL REAL ww
DECL INT aa, bb
ww = ww + 2.3 ; ww = 1.5 + 2.3  =3.8 ->w
bb = bb + 5   ; bb = 5 + 5 = 10
aa = bb + aa  ; aa = 10 + 3= 13 -> a
END
```

4.4 函数编程

通过 KRL 定义函数

- 函数是一种向主程序返回某一值的子程序
- 通常需要输入一定的值才能计算返回值
- 在函数头中会规定返回到主程序中的数据类型
- 待传递的值通过指令 RETURN(return_value) 传递
- 有局部和全局函数两种
- 函数的句法

```
DEFFCT DATATYPE NAME_FUNCTION( )
...
RETURN(return_value)
ENDFCT
```

KRL 函数的原理

- 程序名同时也是一种特定数据类型的变量名称
- 调用全局函数

```
DEF MY_PROG( )
DECL REAL result, value
...
result = CALC(value)
...
END
```

```
DEFFCT REAL CALC(num:IN)
DECL REAL return_value, num
...
RETURN(return_value)
ENDFCT
```

指令 RETURN(return_value) 必须在指令 ENDFCT 之前。

- 调用局部函数

```
DEF MY_PROG( )
DECL REAL result, value
...
result = CALC(value)
...
END

_____________________________________
DEFFCT REAL CALC(num:IN)
DECL REAL return_value, num
...
RETURN(return_value)
ENDFCT
```

- 值传递时使用 IN / OUT 参数
  - 作为 IN 参数进行值传递

```
DEF MY_PROG( )
DECL REAL result, value
value = 2.0
result = CALC(value)
; value = 2.0
; result = 1000.0
END
```

```
DEFFCT REAL CALC(num:IN)
DECL REAL return_value, num
num = num + 8.0
return_value = num * 100.0
RETURN(return_value)
ENDFCT
```

传递的值 value 不改变。

  - 作为 OUT 参数进行值传递

```
DEF MY_PROG( )
DECL REAL result, value
value = 2.0
result = CALC(value)
; value = 10.0
; result = 1000.0
END
```

```
DEFFCT REAL CALC(num:OUT)
DECL REAL return_value, num
num = num + 8.0
return_value = num * 100.0
RETURN(return_value)
ENDFCT
```

传递的值 value 改变后返回。

函数编程时的操作步骤

1. 确定该函数应提供哪个值（返回数据类型）
2. 确定函数中需要哪些参数（传递数据类型）
3. 确定参数传递的种类（IN 或 OUT 参数）
4. 确定需要的是局部还是全局函数
5. 将主程序载入编辑器
6. 在主程序中声明、初始化及可能需要操纵变量

7. 创建函数调用
8. 关闭并保存主程序
9. 创建函数（全局或局部）
10. 将函数载入编辑器
11. 在 DEFFCT 行中补充数据类型、变量及 IN/OUT
12. 在函数中声明、初始化及操纵变量
13. 创建 RETRUN(return_value) 行
14. 关闭并保存函数

4.5 使用 KUKA 标准函数工作

KUKA 标准函数列表

数学函数：

说明	KRL 函数
绝对值	ABS(x)
平方根	SQRT(x)
正弦	SIN(x)
余弦	COS(x)
正切	TAN(x)
反余弦	ACOS(x)
反正切	ATAN2(y,x)

字符串变量的函数：

说明	KRL 函数
声明时确定字符串长度	StrDeclLen(x)
初始化后的字符串变量长度	StrLen(x)
删除字符串变量的内容	StrClear(x)
扩展字符串变量	StrAdd(x,y)
比较字符串变量的内容	StrComp(x,y,z)
复制字符串变量	StrCopy(x,y)

用于信息输出的函数：

说明	KRL 函数
设置信息	Set_KrlMsg(a,b,c,d)
设置对话	Set_KrlDLg(a,b,c,d)
检查信息	Exists_KrlMsg(a)
检查对话	Exists_KrlDlg(a,b)
删除信息	Clear_KrlMsg(a)
读取信息缓存器	Get_MsgBuffer(a)

使用 KUKA 标准函数时的原理

每一标准函数均用传递参数调出：

- 带固定值

```
result = SQRT(16)
```

- 简单数据类型变量

```
result = SQRT(x)
```

- 数组变量

```
result = StrClear(Name[])
```

- 枚举数据类型变量

- 结构变量
- 带多个不同变量

```
result = Set_KrlMsg(#QUIT, message_parameter, parameter[], option)
```

message_paramter、parameter[1...3] 和 **option** 是预定义的 KUKA 结构

每个函数均需要一个可将该函数的结果储存其中的合适变量：

- 数学函数返回一个实数 (REAL) 值
- 字符串函数返回布尔 (BOOL) 或 INT 值

```
; 删除字符串
result = StrClear(Name[])
```

- 信息函数返回布尔 (BOOL) 或 INT 值

```
; 删除信息提示 ( BOOL：已删除？ )
result = Clear_KrlMsg(Rueckwert)
```

5 用 KRL 进行运动编程

5.1 借助 KRL 给运动编程

运动的定义

对运动的必要说明

- 运动方式 PTP、LIN、CIRC
- 目标位置，必要时还有辅助位置
- 精确暂停或轨迹逼近
- 可能还有轨迹逼近距离
- 速度 - PTP (%) 和轨迹运动 (m/s)
- 加速度
- 工具 - TCP 和负载
- 工作基坐标
- 机器人引导型或外部工具
- 沿轨迹运动时的姿态引导
- 可能还有轨迹逼近距离
- 圆周运动 CIRC 时的圆心角

运动编程原理

运动方式 **PTP**

- PTP *目标点* <C_PTP < *轨迹逼近* >>
- 机器人运动到 DAT 文件中的一个位置；该位置已事先通过联机表单示教给机器人，机器人轨迹逼近 P3 点。

```
PTP XP3 C_PTP
```

- 机器人运动到输入的位置
  - 轴坐标（AXIS 或 E6AXIS）

```
PTP {A1 0, A2 -80, A3 75, A4 30, A5 30, A6 110}
```

  - 空间位置（以当前激活的工具和基坐标）

```
PTP {X 100, Y -50, Z 1500, A 0, B 0, C 90, S 3, T3 35}
```

- 机器人仅在输入一个或多个集合时运行

```
PTP {A1 30} ; 仅 A1 移动至 30°
```

```
PTP {X 200, A 30} ; 仅在 X 至 200mm，A 至 30°
```

运动方式 **LIN**

- LIN *目标点* < *轨迹逼近* >
- 机器人运行到一个算出的位置并轨迹逼近点 ABLAGE[4]

```
LIN ABLAGE[4] C_DIS
```

运行方式 **CIRC**

- CIRC *辅助点*，*目标点* <, CA *圆心角* > < *轨迹逼近* >
- 机器人运动到 DAT 文件中的一个位置；该位置已事先通过联机表单示教给机器人，机器人运行一段对应 190° 圆心角的弧段。

```
CIRC XP3, XP4, CA 190
```

- 圆心角 **CA**

在编程设定的目标点示教的姿态被应用于实际目标点处。

- **正圆心角 (CA>0)**：沿着编程设定的转向做圆周运动：起点 - 辅助点 - 目标点

图 5-1: 圆心角 CA = +290°

- **负圆心角 (CA<0)**：逆着编程设定的转向做圆周运动：起点 - 目标点 - 辅助点

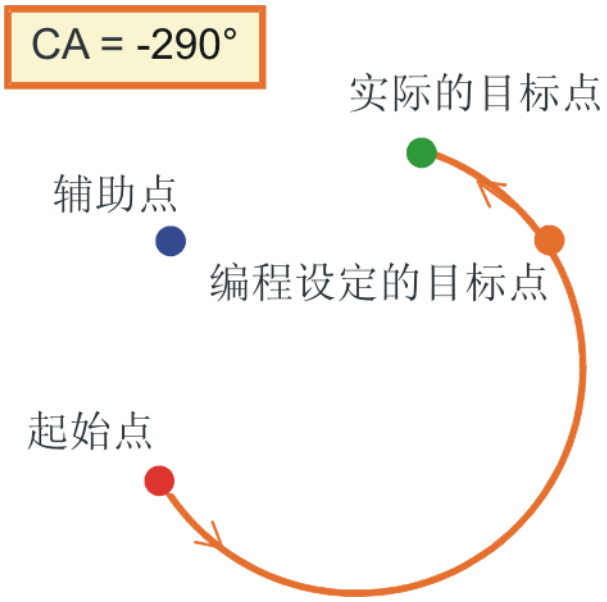


图 5-2: 圆心角 CA = -290°

运动参数的功能

运动编程的预设置

- 可以应用现有的设置：
  - 从 INI 行的运行中
  - 从最后一个联机表单中
  - 从相关系统变量的最后设置中
- 更改或初始化相关的系统变量

运动参数的系统变量

- **工具**：$TOOL 和 $LOAD
  - 激活所测量的 TCP

```
$TOOL = tool_data[x] ; x = 1...16
```

- 激活所属的负载数据

```
$LOAD = load_data[x] ; x = 1...16
```

- **参考基坐标 / 工作基坐标**：$BASE
  - 激活所测量的基坐标

```
$BASE = base_data[x] ; x = 1...16
```

- **机器人引导型或外部工具**：$IPO_MODE
  - 机器人引导型工具

```
$IPO_MODE = #BASE
```

  - 外部工具

```
$IPO_MODE = #TCP
```

- **速度**：
  - 进行 PTP 运动时

```
$VEL_AXIS[x] ; x=1...8，针对每根轴
```

  - 进行轨迹运动 LIN 或 CIRC 时

```
$VEL.CP = 2.0 ; [m/s] 轨迹速度
```

```
$VEL.ORI1 = 150 ; [°/s] 回转速度
```

```
$VEL.ORI2 = 200 ; [°/s] 转速
```

在大多数情况下，工具的作业方向是 X 轴方向。转速是指以角度 C 绕 X 轴旋转的速度。回转速度则是指绕其它两个角度 (A 和 B) 回转的速度。

- **加速**
  - 进行 PTP 运动时

```
$ACC_AXIS[x] ; x=1...8，针对每个轴
```

  - 进行轨迹运动 LIN 或 CIRC 时

```
$ACC.CP = 2.0 ; [m/s] 轨迹加速度
```

```
$ACC.ORI1 = 150 ; [°/s] 回转加速度
```

```
$ACC.ORI2 = 200 ; [°/s] 转动加速度
```

- **圆滑过渡距离**
  - 仅限进行 PTP 运动时：**C_PTP**

```
PTP XP3 C_PTP
$APO_CPTP = 50 ; C_PTP 的轨迹逼近大小，单位 [%]
```

  - 进行轨迹运动 LIN、CIRC 和 PTP 时：**C_DIS**
    与目标点的距离必须低于 $APO.CDIS 的值

```
PTP XP3 C_DIS
LIN XP4 C_DIS
$APO.CDIS = 250.0 ; [mm] 距离
```

  - 进行轨迹运动 LIN、CIRC 时：**C_ORI**
    主导姿态角必须低于 $APO.CORI 的值

```
LIN XP4 C_ORI
$APO.CORI = 50.0 ; [°] 角度
```

  - 进行轨迹运动 LIN、CIRC 时：**C_VEL**

在驶向目标点的减速阶段中速度必须低于 $APO.CVEL 的值

```
LIN XP4 C_VEL
$APO.CVEL = 75.0 ; [%] 百分数
```

- **姿态引导**：仅限进行 LIN 和 CIRC 时
  - 进行 LIN 和 CIRC 时：**$ORI_TPYE**
    -

```
$ORI_TYPE = #CONSTANT
```

在进行轨迹运动期间姿态保持不变。对于结束点来说，编程设定的姿态即被忽略

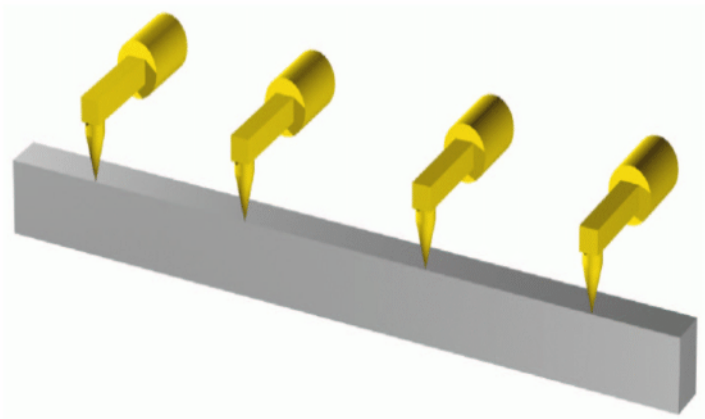


图 5-3: **稳定的方向导引**

-

```
$ORI_TYPE = #VAR
```

在进行轨迹运动期间，姿态会根据目标点的姿态不断地自动改变。

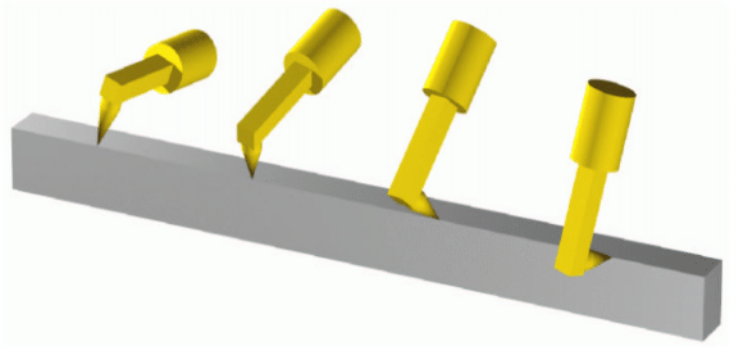


图 5-4: **标准或手动 PTP**

-

```
$ORI_TYPE = #JOINT
```

在进行轨迹运动期间，工具的姿态从起始位置至终点位置不断地被改变。这是通过手轴角度的线性超控引导来实现的。手轴奇点问题可通过该选项予以避免，因为绕工具作业方向旋转和回转不会进行姿态引导。

- 仅限于 CIRC：**$CIRC_TPYE**

如果通过 $ORI_TYPE = #JOINT **进行手轴角度的超控引导**，则变量 $CIRC_TYPE 就没有意义了。

```
$CIRC_TYPE = #PATH
```

圆周运动期间以轨迹为参照的姿态引导

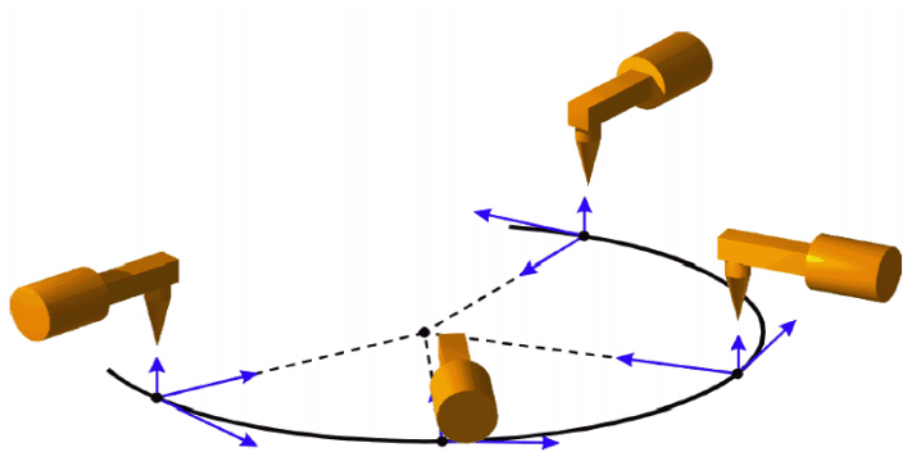


图 5-5: 恒定姿态，以轨迹为参照

```
$CIRC_TYPE = #BASE
```

圆周运动期间以空间为参照的姿态引导

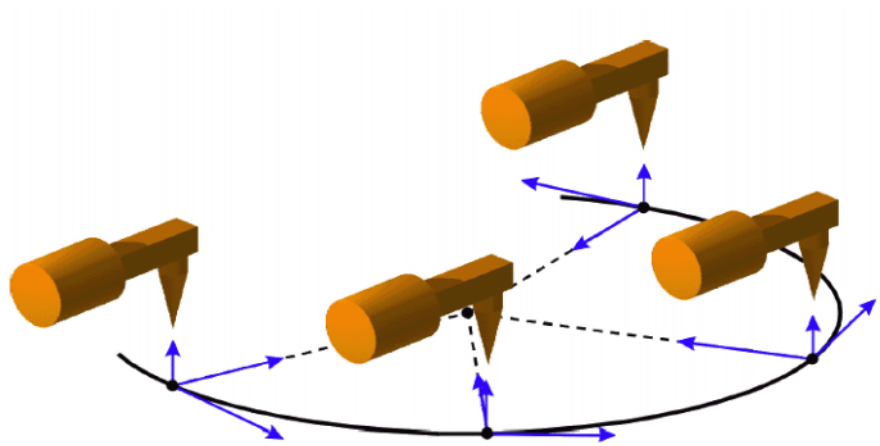


图 5-6: 恒定的姿态，以基坐标为参照

用 KRL 给运动编程时的操作步骤

1. 作为专家借助**打开**键将程序载入编辑器中
2. 检查、应用或重新初始化运动编程的预设定值：
   - 工具 ($TOOL 和 $LOAD)
   - 基坐标设置 ($BASE)
   - 机器人引导型或外部工具 ($IPO_MODE)
   - 速度
   - 加速度
   - 可能还有轨迹逼近距离
   - 可能还有姿态引导
3. 创建由以下部分组成的运动指令：
   - 运动方式 (PTP、LIN、CIRC)

- 目标点（采用 CIRC 时还有辅助点）
- 采用 CIRC 时可能还有圆心角 (CA)
- 激活轨迹逼近 (C_PTP、C_DIS、C_ORI、C_VEL)

4. 重新运动时返回点 3
5. 关闭编辑器并保存

5.2 借助 KRL 给相对运动编程

说明

- 绝对运动

```
PTP {A3 45}
```

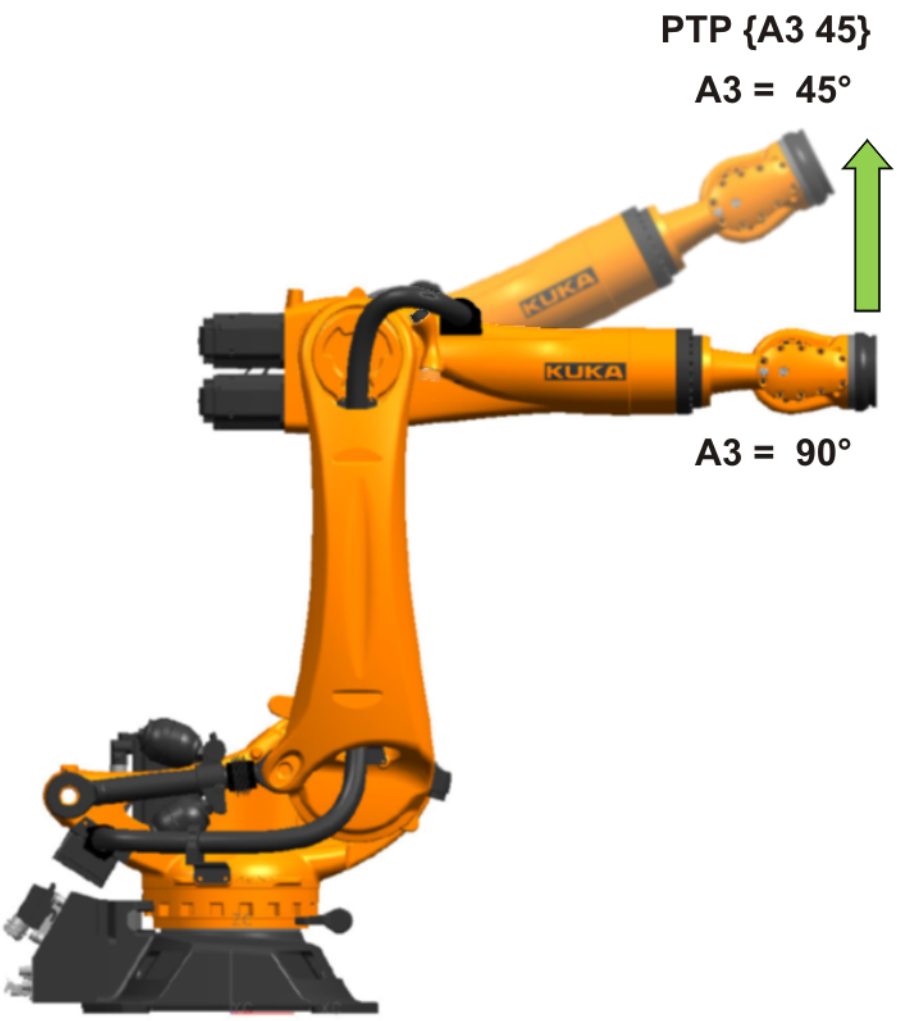


图 5-7: 轴 A3 的绝对运动

- 相对运动

```
PTP_REL {A3 45}
```

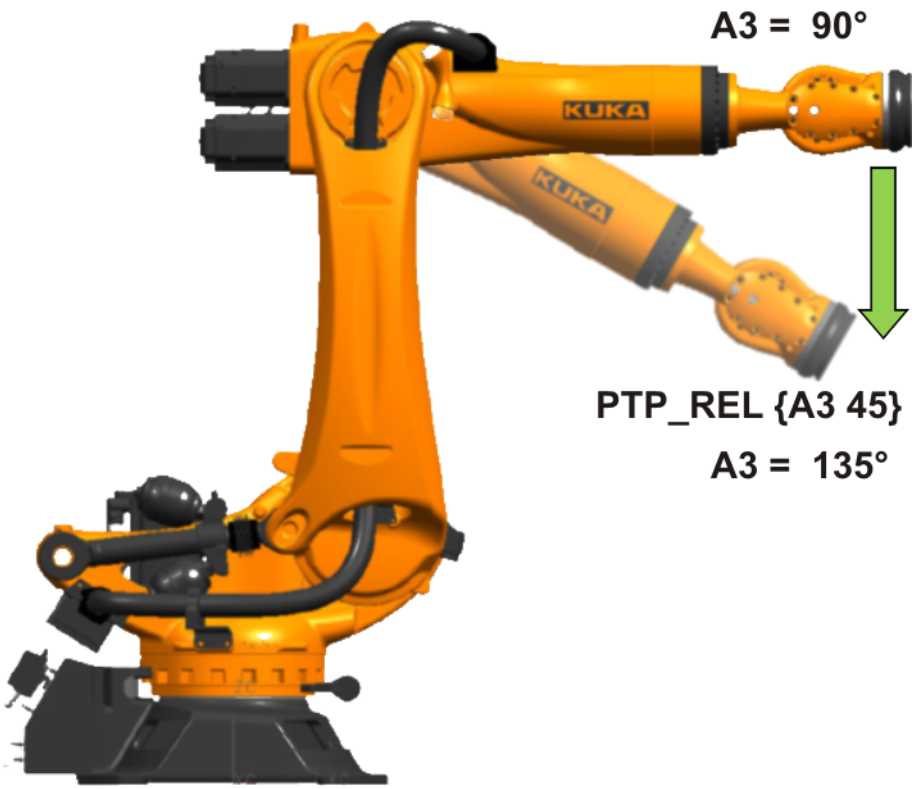


图 5-8: 轴 A3 的相对运动

以下运动为相对运动：

- PTP 运动
- LIN 运动
- CIRC 运动

相对运动的原理

REL 指令始终针对机器人的当前位置。因此，当一个 REL 运动中断时，机器人将从中断位置出发再进行一个完整的 REL 运动。

相对运动 PTP_REL

- `PTP_REL` *目标点* `<C_PTP` < *轨迹逼近* `>>`
- 轴 2 沿负方向移动 30 度。其它的轴都不动。

```
PTP_REL {A2 -30}
```

- 机器人从当前位置沿 X 轴方向移动 100 mm，沿 Z 轴负方向移动 200 mm。Y、A、B、C 和 S 保持不变。T 将根据最短路径加以计算。

```
PTP_REL {X 100,Z -200}
```

相对运动 LIN_REL

- `LIN_REL` *目标点* < *轨迹逼近* > <`#BASE|#TOOL`>
- TCP 从当前位置沿基坐标系中的 X 轴方向移动 100 mm，沿 Z 轴负方向移动 200 mm。Y、A、B、C 和 S 保持不变。T 则从运动中得出。

```
LIN_REL {X 100,Z -200} ; #BASE 为默认设置
```

- TCP 从当前位置沿工具坐标系中的 X 轴负方向移动 100 mm。Y、Z、A、B、C 和 S 保持不变。T 则从运动中得出。

  该示例适用于使工具沿作业方向的反向运动。前提是已经在 X 轴方向测量过工具作业方向。

```
LIN_REL {X -100} #TOOL
```

相对运动 CIRC_REL

- CIRC_REL *辅助点，目标点*<，CA *圆心角*> <*轨迹逼近*>
- 圆周运动的目标点通过 500° 的圆心角加以规定。目标点被轨迹逼近。

```
CIRC_REL {X 100,Y 30,Z -20},{Y 50},CA 500 C_VEL
```

用 KRL 给运动编程时的操作步骤

1. 作为专家借助**打开**键将程序载入编辑器中
2. 检查、应用或重新初始化运动编程的预设定值：
   - 工具 ($TOOL 和 $LOAD)
   - 基坐标设置 ($BASE)
   - 机器人引导型或外部工具 ($IPO_MODE)
   - 速度
   - 加速度
   - 可能还有轨迹逼近距离
   - 可能还有姿态引导
3. 创建由以下部分组成的运动指令：
   - 运动方式 (PTP_REL、LIN_REL、CIRC_REL)
   - 目标点（采用 CIRC 时还有辅助点）
   - 采用 LIN 时选择参照系 (#BASE 或 #TOOL)
   - 采用 CIRC 时可能还有圆心角 (CA)
   - 激活轨迹逼近 (C_PTP、C_DIS、C_ORI、C_VEL)
4. 重新运动时返回点 3
5. 关闭编辑器并保存

5.3 计算或操纵机器人位置

说明

机器人的目标位置

- 使用以下几种结构存储：
  - **AXIS / E6AXIS** - 轴角（A1...A6，也可能是 E1...E6）
  - **POS / E6POS** - 位置 (X, Y, Z)，姿态 (A, B, C) 以及状态和转角方向 (S, T)
  - **FRAME** - 仅位置 (X, Y, Z)，姿态 (A, B, C)
- 可以操纵 DAT 文件中的现有位置
- 现有位置上的单个集合可以通过点号有针对性地加以更改

原理

小心 计算时必须注意正确的工具和基坐标设置，然后在编程运动时加以激活。不注意这些设置可能导致运动异常和碰撞

重要的系统变量

- **$POS_ACT**：当前的机器人位置。变量 (E6POS) 指明 TCP 基于基坐标系的额定位置。
- **$AXIS_ACT**：基于轴坐标的当前机器人位置（额定值）。变量 (E6AXIS) 包含当前的轴角或轴位置

计算绝对目标位置

- 一次性更改 DAT 文件中的位置

```
XP1.x = 450 ; 新的 X 值 450mm
XP1.z = 30*distance ; 计算新的 Z 值
PTP XP1
```

- 每次循环时都更改 DAT 文件中的位置

```
; X 值每次推移 450mm
XP2.x = XP2.x + 450
PTP XP2
```

- 位置被应用，并被保存在一个变量中

```
myposition = XP3
myposition.x = myposition.x + 100 ; 给 x 值加上 100mm
myposition.z = 10*distance ; 计算新的 Z 值
myposition.t = 35 ; 设置转角方向值
PTP XP3 ; 位置未改变
PTP myposition ; 计算出的位置
```

操作步骤

1. 作为专家借助**打开**键将程序载入编辑器中
2. 计算 / 操纵位置。新计算得出的值可能要暂存在新的变量中
3. 检查、应用或重新初始化运动编程的预设定值：
   - 工具 ($TOOL 和 $LOAD)
   - 基坐标设置 ($BASE)
   - 机器人引导型或外部工具 ($IPO_MODE)
   - 速度
   - 加速度
   - 可能还有轨迹逼近距离
   - 可能还有姿态引导
4. 创建由以下部分组成的运动指令：
   - 运动方式 (PTP、LIN、CIRC)
   - 目标点（采用 CIRC 时还有辅助点）
   - 采用 CIRC 时可能还有圆心角 (CA)
   - 激活轨迹逼近 (C_PTP、C_DIS、C_ORI、C_VEL)
5. 重新运动时返回点 3
6. 关闭编辑器并保存

5.4 有针对性地更改状态和转角方向比特位

说明

- TCP 位置 (X, Y, Z) 和姿态 (A, B, C) 的数值不足于明确规定机器人的位置，因为虽然 TCP 相同，但轴的位置仍可能有多个。状态和转角方向用于从多个可能的轴位中确定一个唯一的位置。

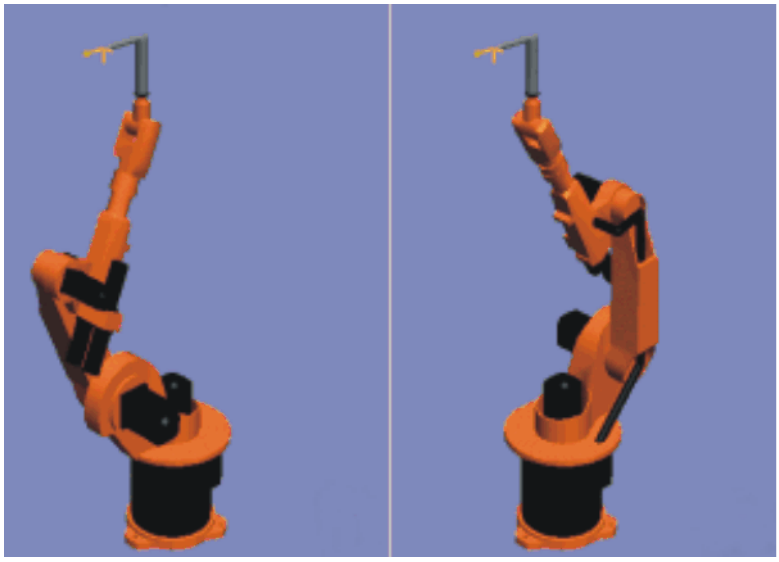


图 5-9: 例如：TCP 相同，轴位置不同

- 状态 (S) 和转角方向 (T) 是数据类型 POS 和 E6POS 的组成部分：

```
STRUC POS REAL X, Y, Z, A, B, C, INT S, T
```

```
STRUC E6POS REAL X, Y, Z, A, B, C, E1, E2, E3, E4, E5, E6, INT S, T
```

- 机器人控制器**仅在 PTP 运动**时才会考虑编程的状态和转角方向值。在 CP 运动时会将它们忽略。
- 因此，KRL 程序中的第一个运动指令必须是以下指令中的一种，以便为机器人定义唯一的起始位置：
  - POS 型或 E6POS 型的完整 PTP 指令
  - 或 AXIS 型或 E6AXIS 型的完整 PTP 指令

  “完整”表示必须输入目标点的所有组成内容。默认 HOME 位置始终是完整的 PTP 指令。
- 在其它指令中可以略去状态和转角方向：
  - 机器人控制器保留现有的状态值。
  - 转角方向值在 CP 运动时从轨迹中得出。
  - 进行 PTP 运动时，机器人控制器选择表明路径可能最短的转角方向值（即不会损坏软件限位开关，同时最接近起始角）。

功能

STATUS

- 用状态 (STATUS) 参数可防止轴的位置多义。
- **位 0**：给出手轴（A4、A5、A6）交点的位置。

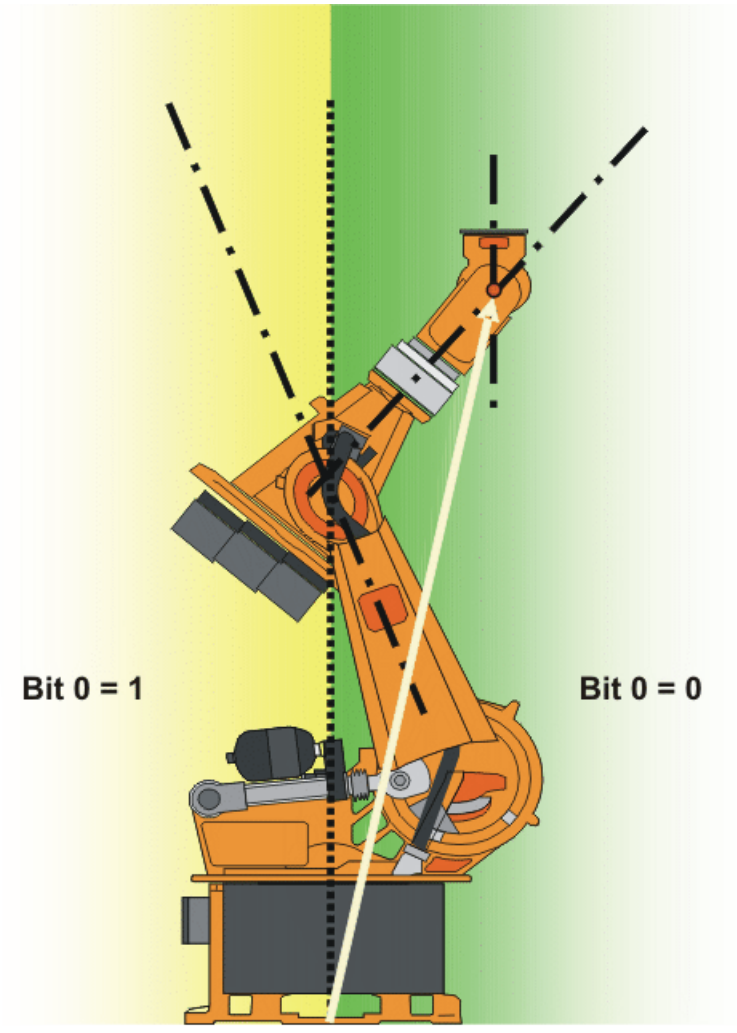


图 5-10: 例如：手轴的交点（红色的点）位于基本区域。

- **位 1**：给出轴 3 的位置。比特位 1 值改变时对应的角取决于机器人类型。

  对轴 3 和轴 4 相交的机器人，适用：

位置	值
A3 ≥ 0°	位 1 = 1
A3 < 0°	比特位 1 = 0

  对于轴 3 与轴 4 错开的机器人，比特位 1 改变时对应的角取决于偏移量的大小。

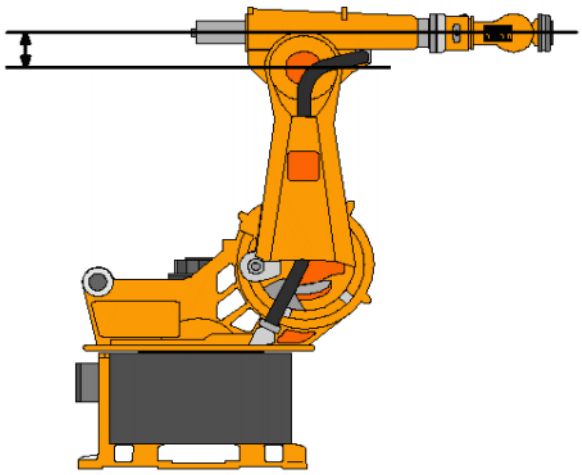


图 5-11: A3 和 A4 之间的偏量：例如 KR 30

- **位 2**：给出轴 5 的位置。

位置	值
A5 > 0	比特位 2 = 1
A5 ≤ 0	比特位 2 = 0

- **比特位 3 未用**，始终为 0。
- **位 4**：说明是否用一台绝对精确的机器人对点进行示教。

  该点既可用绝对精确机器人也可用非绝对精确机器人经过，不受制于比特位的值。比特位 4 仅用于给出信息，对机器人控制器如何计算该点无影响。这意味着，如果离线给一个机器人编程，则可忽略比特位 4。

说明	值
该点未用绝对精确的机器人示教。	比特位 4 = 0
该点已用一个绝对精确的机器人示教。	比特位 4 = 1

TURN

- 利用转角方向 (TURN) 参数可不用特别移动策略（例如中间点）就运动到大于 +180 ° 或小于 -180 ° 的轴角。对于旋转轴来说，各比特位如下决定轴值的正负号：

  比特位 = 0: 角度 ≥ 0 °

  比特位 = 1: 角度 < 0 °

- 各轴概览

值	比特位 5	比特位 4	比特位 3	比特位 2	比特位 1	比特位 0
0	A6 ≥ 0 °	A5 ≥ 0 °	A4 ≥ 0 °	A3 ≥ 0 °	A2 ≥ 0 °	A1 ≥ 0 °
1	A6 < 0 °	A5 < 0 °	A4 < 0 °	A3 < 0 °	A2 < 0 °	A1 < 0 °

- 示例

```
DECL POS XP1 = {X 900, Y 0, Z 800, A 0, B 0, C 0, S 6, T 19}
```

T 19 相当于 T ‘B010011’。即：

轴	角铁	二进制
A 1	负	1
A 2	负	2
A 3	正	4
A 4	正	8
A 5	负	16
A 6	正	32

操作步骤

1. 作为专家借助**打开**键将程序载入编辑器中
2. 操纵**状态**和**转角方向**。新计算得出的值可能要暂存在新的变量中
3. 检查、应用或重新初始化运动编程的预设定值：
   - 工具 ($TOOL 和 $LOAD)
   - 基坐标设置 ($BASE)
   - 机器人引导型或外部工具 ($IPO_MODE)
   - 速度
   - 加速度
   - 可能还有轨迹逼近距离
   - 可能还有姿态引导
4. 创建由以下部分组成的运动指令：
   - 运动方式 (PTP、LIN、CIRC)
   - 目标点（采用 CIRC 时还有辅助点）
   - 采用 CIRC 时可能还有圆心角 (CA)
   - 激活轨迹逼近 (C_PTP、C_DIS、C_ORI、C_VEL)
5. 重新运动时返回点 3
6. 关闭编辑器并保存

6 用系统变量工作

6.1 用计时器测量节拍时间

用 KUKA 系统计时器测量节拍时间的说明

图 6-1

- $TIMER[1]
- $TIMER[2]
- ...
- $TIMER[32]

系统变量 $TIMER[Nr] 用于测量时间进程。

> 计时器 $TIMER[Nr] 的数值输入 / 显示都以毫秒 **ms** 为单位。

通过 KRL 启动和停止计时器

- 启动：$TIMER_STOP[Nr] = FALSE
- 停止：$TIMER_STOP[Nr] = TRUE

> 计时器也可通过显示窗口手动初试化、启动和停止。

节拍时间测量的原理

计时器的预设

- 交货时计时器的预设为 0 ms
- 计时器保持其当前值
- 可将计时器往前或往后调到任意一个值

```
;计时器 5 预设为 0ms
$TIMER[5] = 0

; 计时器 12 设定为 1.5 秒
$TIMER[12] = 1500

;计时器 4 回调至 -8 秒
$TIMER[4] = -8000
```

- 计时器的复位和启动

```
; 计时器 7 复位至 0ms
$TIMER[7] = 0
; 启动计时器 7
$TIMER_STOP[7] = FALSE
```

- 停止计时器并接着进行比较

```
; 计时器 7 计时中
...
; 停止计时器 7
$TIMER_STOP[7] = TRUE

; 10 秒或更多时 ……
IF $TIMER[7] >= 10000 THEN
...
```

计时器的启动和停止始终通过**预进指针**实现

测量节拍时间的操作步骤

1. 从 32 种可能的计时器中选择一个 “ 空闲 ” 计时器
2. 计时器的预设 / 重置
3. 在注意预进指针的情况下启动计时器
4. 在注意预进指针的情况下停止计时器
5. 可能需要临时保存当前的节拍时间或者重新预设计时器

```
DEF MY_TIME( )
...
INI
$TIMER[1] = 0 ; 复位计时器 ?1
PTP HOME Vel=100% DEFAULT

WAIT SEC 0 ; 触发预进停止
$TIMER_STOP[1]=FALSE ; 开始节拍时间测量

PTP XP1
PTP XP2
LIN XP3
...
PTP X50
PTP HOME Vel=100% DEFAULT

WAIT SEC 0 ; 触发预进停止
$TIMER_STOP[1]=TRUE ; 结束节拍时间测量

; 当前节拍时间临时保存在计时器 12 中
$TIMER[12] = $TIMER[1]
END
```

7 使用程序流程控制

7.1 询问或 if 分支的编程

KRL 的询问和 if 分支说明

- if 分支用于将程序分为多个路径。
- IF 指令会对可能为真 (TRUE) 或为假 (FALSE) 的条件进行检查。籍此来判断是否执行指令。
- if 分支

```
IF ..... THEN
...
ELSE
...
ENDIF
```

使用 if 分支

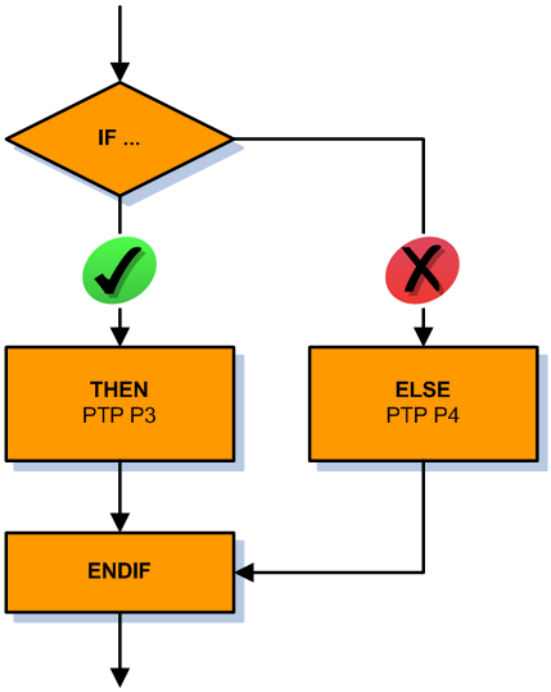


图 7-1: 程序流程图：IF 分支

if 分支

- 带选择分支语句

```
IF condition THEN
Anweisung
ELSE
; 指令
ENDIF
```

- 无选择分支语句（询问）

```
IF condition THEN
; 指令
ENDIF
```

if 分支示例

没有可选分支的 if 分支

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; 仅在 error_nr 5 时驶至 P21
IF error_nr == 5 THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END
```

有可选分支的 if 分支

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; 仅在 error_nr 5 时驶至 P21，否则 P22
IF error_nr == 5 THEN
PTP P21 Vel=100% PDAT21
ELSE
PTP P22 Vel=100% PDAT22
ENDIF
...
END
```

有复杂执行条件的 if 分支

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; 仅在 error_nr 1 或 10 或大于 99 时驶至 P21
IF ((error_nr == 1) OR (error_nr == 10) OR (error_nr > 99)) THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END
```

有布尔表达式的 if 分支

```
DEF MY_PROG( )
DECL BOOL no_error
...
INI
no_error = TRUE
...
; 仅在无故障 (no_error) 时驶至 P21
IF no_error == TRUE THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END
```

表达式 `IF no_error==TRUE THEN` 也可以简化为 `IF no_error THEN`。省略始终表示比较为真（`TRUE`）

7.2 给 switch-case 分支编程

KRL switch-case 分支说明

- 若需要区分多种情况并为每种情况执行不同的操作，则可用 switch case 指令达到目的。
- `switch` 指令 用于区分各种情况。

- `switch` 指令中传递的变量用作开关，在指令块中跳到预定义的 `case` 指令中。
- 如果 `switch` 指令未找到预定义的 `case`，则运行 `default`（默认）段
- switch-case 分支

```
SWITCH ...
CASE ...
...
CASE ...
...
CASE
...
...
DEFAULT
...
ENDSWITCH
```

switch-case 分支的应用

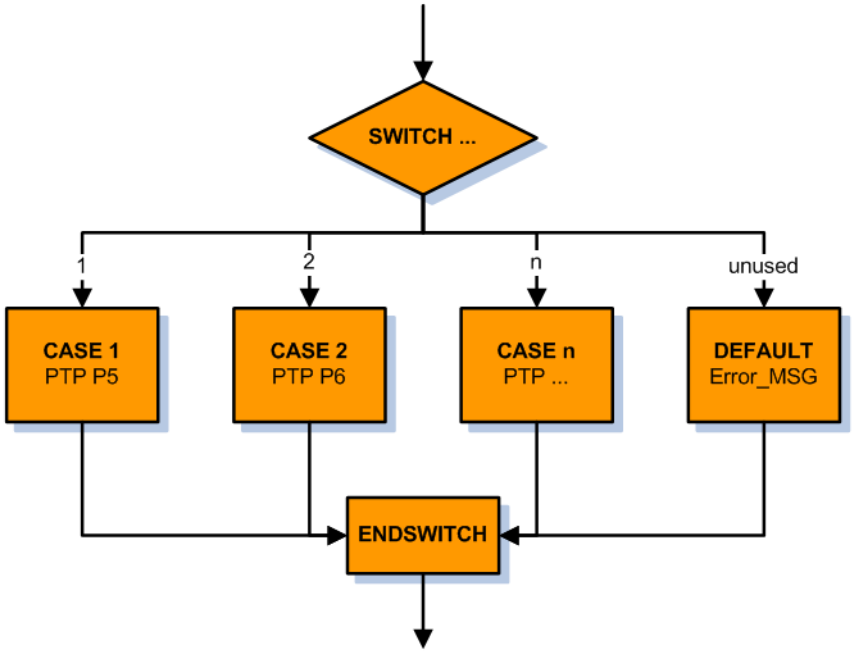


图 7-2: 程序流程图 : switch-case 分支

switch-case 分支

- 可使用有下列数据类型的 switch-case 分支
  - INT（整数）

```
SWITCH number
CASE 1
...
```

  - CHAR（单个字符）

```
SWITCH symbol
CASE "X"
...
```

  - ENUM（枚举数据类型）

```
SWITCH mode_op
CASE #T1
...
```

- 仅含定义的 switch-case 分支

```
SWITCH number
CASE 1
...
CASE 2
...
CASE 3
...
ENDSWITCH
```

当*编号* (nummer) 不等于 1 或 2 或 3 时，则直接跳至 ENDSWITCH，不执行指令。

- 仅含定义的 switch-case 分支和一种替代情况

```
SWITCH number
CASE 1
...
CASE 2
...
CASE 3
...
DEFAULT
...
ENDSWITCH
```

当*编号*不等于 1 或 2 或 3 时，则直接跳入 DEFAULT，以执行其指令。

- 一个 switch-case 分支中有多种解决方案

```
SWITCH number
CASE 1,2
...
CASE 3,4,5
...
CASE 6
...
DEFAULT
...
ENDSWITCH
```

switch-case 分支举例

无替代情况的 switch-case 分支

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; 仅在已存储情况下才可运行
SWITCH error_nr
CASE 1
PTP P21 Vel=100% PDAT21
CASE 2
PTP P22 Vel=100% PDAT22
CASE 3
PTP P23 Vel=100% PDAT23
CASE 4
PTP P24 Vel=100% PDAT24
ENDSWITCH
...
```

无替代情况的 switch-case 分支

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 99
...
; 在未定义的情况下，驶至起始位置 (HOME)
SWITCH error_nr
CASE 1
PTP P21 Vel=100% PDAT21
CASE 2
PTP P22 Vel=100% PDAT22
CASE 3
PTP P23 Vel=100% PDAT23
CASE 4
PTP P24 Vel=100% PDAT24
DEFAULT
PTP HOME Vel=100% DEFAULT
ENDSWITCH
...
```

带枚举数据类型的 switch-case 分支

```
DEF MY_PROG( )
ENUM COLOR_TYPE red, yellow, blue, green
DECL COLOR_TYPE my_color
...
INI
my_color = #red
...
SWITCH my_color
CASE #red
PTP P21 Vel=100% PDAT21
CASE #yellow
PTP P22 Vel=100% PDAT22
CASE #green
PTP P23 Vel=100% PDAT23
CASE #blue
PTP P24 Vel=100% PDAT24
ENDSWITCH
...
```

7.3 给循环编程

循环概述

- 循环用于重复程序指令
- 不允许从外部跳入循环结构中
- 循环可互相嵌套
- 有不同的循环类型
  - 无限循环
  - 计数循环
  - 条件循环
    - 当型循环
    - 直到型循环

7.3.1 给无限循环编程

无限循环的说明

- 无限循环是每次运行完之后都会重新运行的循环。
- 运行过程可通过外部控制而终止。
- 句法

```
LOOP
;    指令
...
; 指令
ENDLOOP
```

无限循环的原理

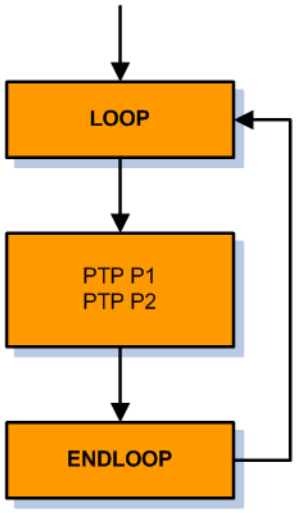


图 7-3: 程序流程图：无限循环

- 无限循环可直接用 EXIT 退出
- 用 EXIT 退出无限循环时必须注意避免碰撞
- 如果两个无限循环互相嵌套，则需要两个 EXIT 指令以退出两个循环

无限循环编程的示例

无中断的无限循环

```
DEF MY_PROG( )
INI
PTP HOME Vel=100% DEFAULT

LOOP
PTP P1 Vel=90% PDAT1
PTP P2 Vel=100% PDAT2
PTP P3 Vel=50% PDAT3
PTP P4 Vel=100% PDAT4
ENDLOOP

PTP P5 Vel=30% PDAT5
PTP HOME Vel=100% DEFAULT
END
```

因编程技术而从不驶至点 P5

带中断的无限循环

```
DEF MY_PROG( )
INI
PTP HOME Vel=100% DEFAULT

LOOP
PTP P1 Vel=90% PDAT1
PTP P2 Vel=100% PDAT2
IF $IN[3]==TRUE THEN ; 中断的操作
EXIT
ENDIF
PTP P3 Vel=50% PDAT3
PTP P4 Vel=100% PDAT4
ENDLOOP

PTP P5 Vel=30% PDAT5
PTP HOME Vel=100% DEFAULT
END
```

只要输入端 1 激活，则会驶至点 P5。
重要提示：对于 P2 和 P5 之间的运动必须检查是否会发生碰撞

7.3.2 给计数循环编程

计数循环的定义

- FOR 循环是一种可以通过规定重复次数执行一个或多个指令的控制结构。
- 步幅为 +1 时的句法

```
FOR counter = start TO last
;指令
ENDFOR
```

- 步幅 (increment) 也可通过关键词 STEP 指定为某个整数。

```
FOR counter = start TO last STEP increment
;指令
ENDFOR
```

计数循环的原理

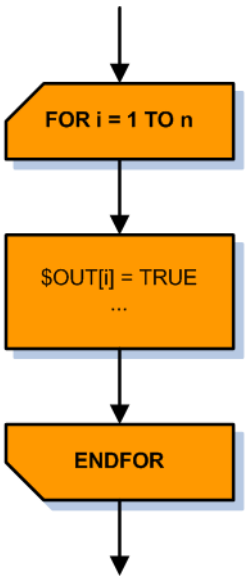


图 7-4: 程序流程图：计数循环

- 要进行计数循环则必须事先声明一个整数变量

- 该计数循环从值等于 start 时开始并最迟于值等于 last 时结束

```
FOR counter = start TO last
;指令
ENDFOR
```

- 该计数循环可借助 EXIT 立即退出

计数循环的原理

```
DECL INT counter

FOR counter = 1 TO 3 Step 1
; 指令
ENDFOR
```

1. 循环计数器被用起始值进行初始化：counter = 1
2. 循环计数器在 ENDFOR 时会以步幅 STEP 递增计数
3. 循环又从 FOR 行开始
4. 检查进入循环的条件：计数变量必须小于等于指定的终值，否则会结束循环
5. 根据检查结果的不同，循环计数器会再次递增计数或结束循环。结束循环后程序在 ENDFOR 行后继续运行

使用计数循环进行递减计数

```
DECL INT counter

FOR counter = 15 TO 1 Step -1
; 指令
ENDFOR
```

> 循环的初始值或者起始值必须大于等于终值，以便循环能够多次运行。

使用计数循环编程

计数循环编程的示例

- 没有指定步幅的单层计数循环

```
DECL INT counter

FOR counter = 1 TO 50
$OUT[counter] == FALSE
ENDFOR
```

> 没有借助 STEP 指定步幅时，会自动使用步幅 +1。

- 指定步幅的单层计数循环

```
DECL INT counter

FOR counter = 1 TO 4 STEP 2
$OUT[counter] == TRUE
ENDFOR
```

> 该循环只会运行两次。一次以起始数值 counter=1，另一次则以 counter=3。计数值为 5 时，循环立即终止。

- 指定步幅的双层计数循环

```
DECL INT counter1, counter2

FOR counter1 = 1 TO 21 STEP 2
    FOR counter2 = 20 TO 2 STEP -2
        ...
    ENDFOR
ENDFOR
```

每次都会先运行内部循环（此处以 counter1），然后运行外部循环 (counter2)。

7.3.3 当型循环的编程

当型循环的说明

- 当型循环也被称为前测试循环。
- 这种循环会一直重复过程，直至满足某一条件 (conition) 为止。
- 句法

```
WHILE condition
  ; 指令
ENDWHILE
```

- 当型循环可通过 EXIT 指令立即退出

当型循环的原理

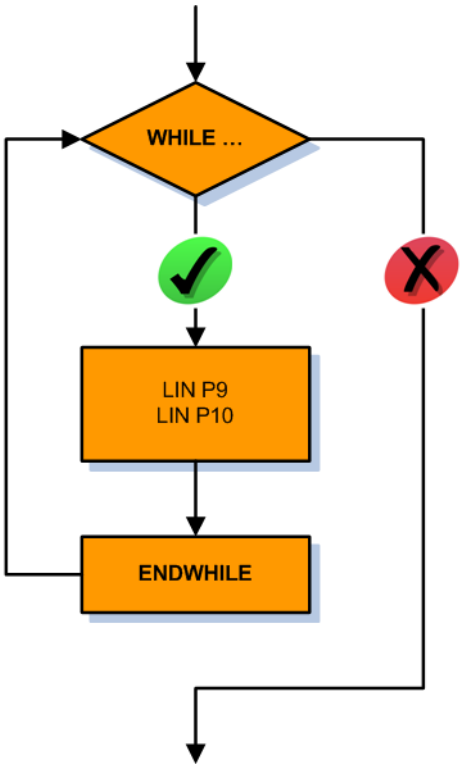


图 7-5: 程序流程图：当型循环

- 当型循环用于先检测是否开始某个重复过程
- 如要完成循环，必须满足执行条件
- 执行条件不满足时会导致立即结束循环，并执行 ENDWHILE 后的指令。

用一个当型循环编程

- 具有简单执行条件的当型循环

```
...
WHILE IN[41]==TRUE ; 部件备好在库中
PICK_PART( )
ENDWILE
...
```

表达式 WHILE IN[41]==TRUE 也可简化为 WHILE IN[41]。省略始终表示比较为真（TRUE）

- 具有简单否定型执行条件的当型循环

```
...
WHILE  NOT IN[42]==TRUE ; 输入端 42：库为空
PICK_PART( )
ENDWILE...
```

或

```
...
WHILE  IN[42]==FALSE ; 输入端 42：库为空
PICK_PART( )
ENDWILE...
```

- 具有复合执行条件的当型循环

```
...
WHILE  ((IN[40]==TRUE) AND (IN[41]==FALSE) OR (counter>20))
PALETTE( )
ENDWILE
...
```

7.3.4 直到型循环的编程

直到型循环的说明

- 直到型循环也称为后测试循环。
- 这种直到型循环先执行指令，在结束时测试退出循环的条件 (condition) 是否已经满足。
- 句法

```
REPEAT
  ; 指令
UNTIL condition
```

- 直到型循环可通过 EXIT 指令立即退出

直到型循环的原理

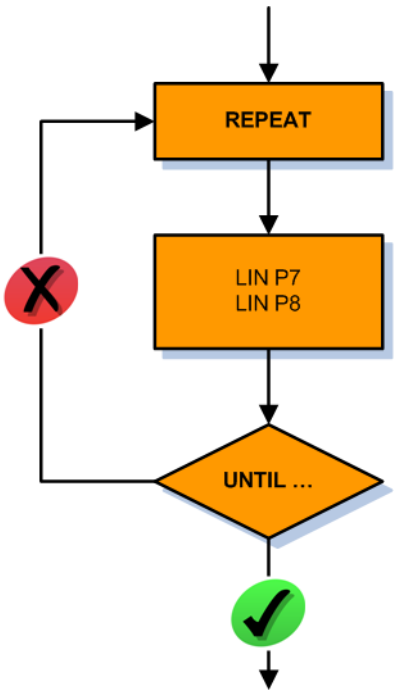


图 7-6: 程序流程图：直到型循环

- 条件满足时，退出循环，执行 UNTIL 后的指令。
- 条件不满足时，在 REPEAT 处重新开始循环。

直到型循环的编程

- 具有简单执行条件的直到型循环

```
...
REPEAT
PICK_PART( )
UNTIL IN[42]==TRUE ; 输入端 42：库为空
...
```

- 具有复杂执行条件的直到型循环

```
...
REPEAT
PALETTE( )
UNTIL ((IN[40]==TRUE) AND (IN[41]==FALSE) OR (counter>20))
...
```

7.4 等待函数编程

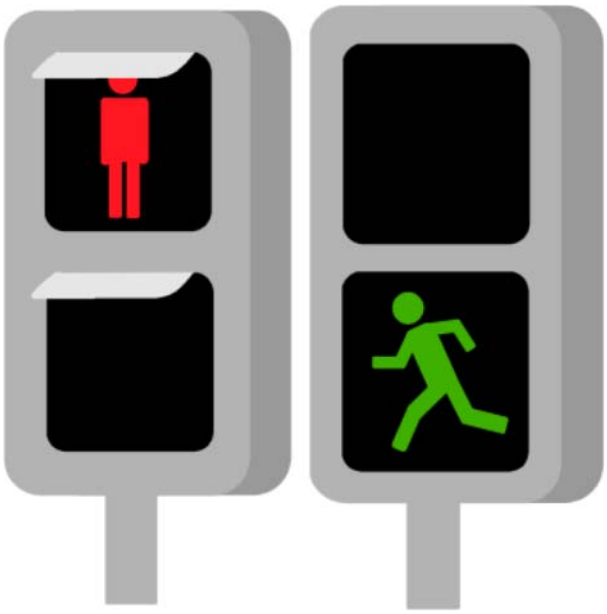


图 7-7

以 KRL 为以下函数编程：

- 时间等待函数
- 信号等待函数

7.4.1 时间等待函数

KRL 中时间等待函数的说明

- 在过程可以继续运行前，时间等待函数等待指定的时间 (time)。
- 句法

```
WAIT SEC time
```

时间等待函数原理

- 时间等待函数的单位为秒 (s)
- 最长时间为 2147484 秒，相当于 24 天多
- 时间值也可用一个合适的变量来确定
- 最短的有意义的时间单元是 0.012 秒（IPO 节拍）
- 如果给出的时间为负值，则不等待
- 时间等待函数触发预进停止，因此无法轨迹逼近
- 为了直接生成预进停止，可使用指令 WAIT SEC 0

给时间等待函数编程

- 具有固定时间的时间等待函数

```
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
WAIT SEC 5.25
PTP P3 Vel=100% PDAT3
```

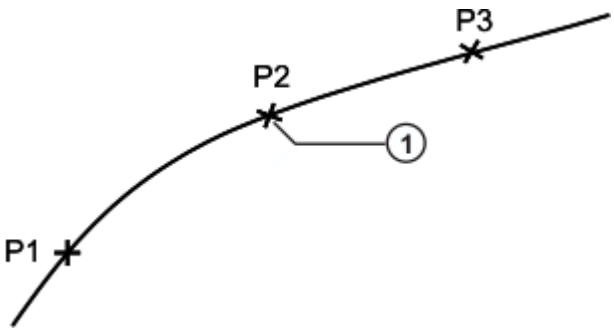


图 7-8: 逻辑运动示例

- 具有计算出时间的时间等待函数

```
WAIT SEC 3*0.25
```

- 具有变量的时间等待函数

```
DECL REAL time
time = 12.75
WAIT SEC time
```

7.4.2 信号等待函数

信号等待函数说明

- 信号等待函数在满足条件 (condition) 时才切换到继续进程，使过程得以继续
- 句法

```
WAIT FOR condition
```

信号等待函数原理

- 信号等待函数触发预进停止，因此无法轨迹逼近
- 尽管已满足了条件，仍生成预进停止
- 用指令 CONTINUE 可阻止预进停止

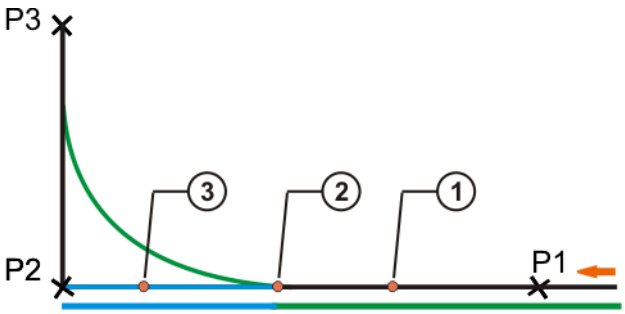


图 7-9: 带预进的逻辑运动示例

信号等待函数的编程

- 带预进停止的 WAIT FOR

```
PTP P1 Vel=100% PDAT1
PTP P2 CONT Vel=100% PDAT2
WAIT FOR $IN[20]
PTP P3 Vel=100% PDAT3
```

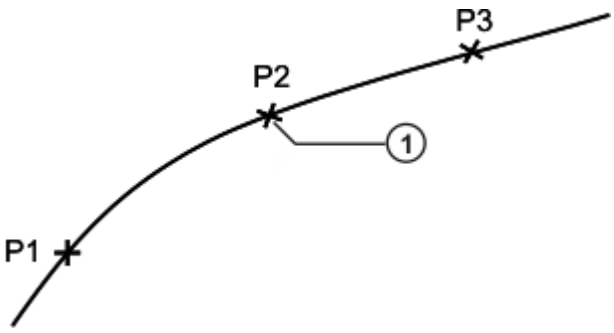


图 7-10: 逻辑运动示例

- 在预进过程中加工的 WAIT FOR （使用 CONTINUE）

```
PTP P1 Vel=100% PDAT1
PTP P2 CONT Vel=100% PDAT2
CONTINUE
WAIT FOR ($IN[10] OR $IN[20])
PTP P3 Vel=100% PDAT3
```

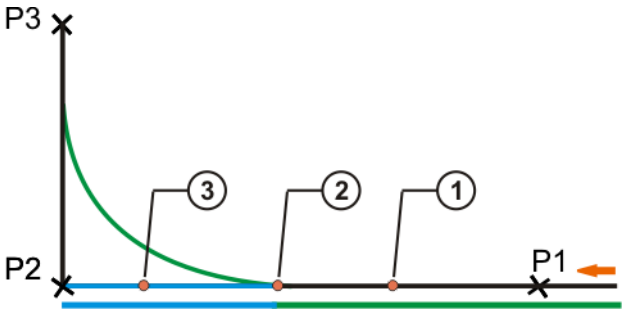


图 7-11: 带预进的逻辑运动示例

8 KRL 的切换函数

8.1 简单切换函数的编程

简单切换函数说明

概述

- 机器人控制系统最多可以管理 4096 个数字输入端和 4096 个数字输出端
- 输入 / 输出端通过可作为选项配备的现场总线系统实现
- 根据用户要求进行专门配置
- 项目设计通过 WorkVisual 进行

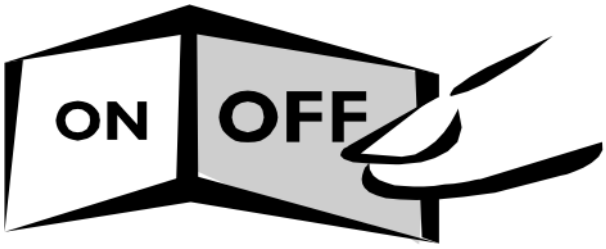


图 8-1

简单切换函数的用途

- 简单开 / 关一个输出端（含预进 / 预进停止）
- 给输出端加上脉冲
- 用主进指针来切换输出端（无预进停止）

简单切换函数的功能

简单开 / 关一个输出端

- 接通一个输出端

```
$OUT[10]=TRUE
```

- 关闭一个输出端

```
$OUT[10]=FALSE
```

- 通过输出端的切换将生成一个预进停止，因此不能进行轨迹逼近运动

```
...
PTP P20 CONT Vel=100% PDAT20
$OUT[30]=TRUE
PTP P21 CONT Vel=100%PDAT21
```

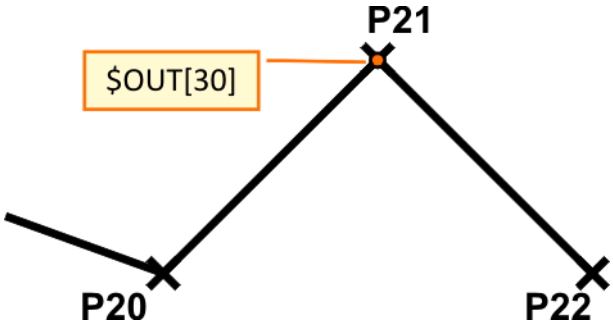


图 8-2: 带预进停止的切换

- 用指令 CONTINUE 可取消预进停止
- 用指令 CONTINUE 也同时切换到预进
- 可用 CONTINUE 轨迹逼近
- CONTINUE 仅涉及下一行（包括空行）

```
...
PTP P20 CONT Vel=100% PDAT20
CONTINUE
$OUT[30]=TRUE
PTP P21 CONT Vel=100%PDAT21
```

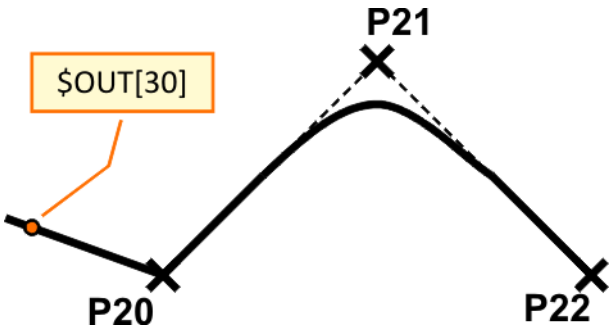


图 8-3: 在预进中切换

用主进指针来切换

- 最多可有 8 个输出端根据主进切换，不会引起预进停止
- 如果编程设定了精确停止，则达到目标点时切换
- 如果编程设定了轨迹逼近，则将在向目标点进行轨迹逼近运动的中点切换

```
...
LIN P20 CONT Vel=100% PDAT20
$OUT_C[30]=TRUE
LIN P21 CONT Vel=100%PDAT21
```

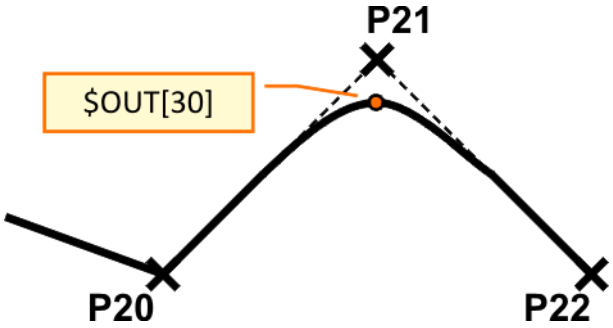


图 8-4: 带主进的切换

给输出端加上脉冲

- 设定一个脉冲
- 在此过程中，输出端在特定时间内设置为定义的电平
- 此后输出端由系统自动复位
- PULSE（脉冲）指令触发一次预进停止

注意	在紧急停止、操作停机或故障停机时该脉冲不中断！

- 句法

  PULSE（*信号，电平，脉冲宽度*）

```
PULSE ($OUT[30], TRUE, 20); 正脉冲

PULSE ($OUT[31], FALSE, 20); 负脉冲
```

- 如果在结束指令之前编程设定了一个脉冲，则程序处理时间将相应延长

```
...
PULSE($OUT[50],TRUE,2)
END
```

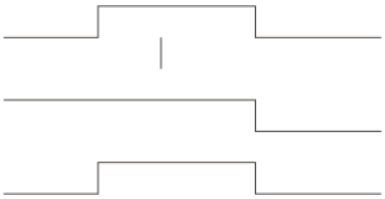


图 8-5: PULSE+END 示例

- 如果在脉冲激活状态将程序处理复位 (**RESET**) 或中断 (**CANCEL**)，则脉冲将立即复位

```
...
PULSE($OUT[50],TRUE,2)
; 现在程序复位或取消选择
```

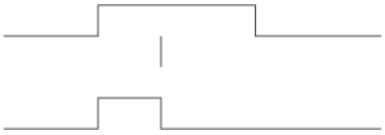


图 8-6: PULSE+RESET 示例

简单切换函数的编程

- **带预进停止**的输出端切换

```
...
LIN P20 CONT Vel=100% PDAT20
$OUT[50]=TRUE ; 接通
LIN P21 CONT Vel=100%PDAT21
$OUT[50]=FALSE ; 关断
LIN P22 CONT Vel=100%PDAT22
```

- 借助脉冲功能**带预进停止**的输出端切换

```
...
LIN P20 CONT Vel=100% PDAT20
PULSE ($OUT[50], TRUE, 1.5) ; 正脉冲
PULSE ($OUT[51], FALSE, 1.5) ; 负脉冲
LIN P21 CONT Vel=100%PDAT21
```

- **在预进过程中**的输出端切换

```
...
LIN P20 CONT Vel=100% PDAT20
CONTINUE
$OUT[50]=TRUE ; 接通
LIN P21 CONT Vel=100%PDAT21
CONTINUE
$OUT[50]=FALSE ; 关断
LIN P22 CONT Vel=100%PDAT22
```

- 借助脉冲功能**在预进过程中**的输出端切换

```
...
LIN P20 CONT Vel=100% PDAT20
CONTINUE
PULSE ($OUT[50], TRUE, 1.5) ; 正脉冲
CONTINUE
PULSE ($OUT[51], FALSE, 1.5) ; 负脉冲
LIN P21 CONT Vel=100%PDAT21
```

- **带主进**的输出端切换

```
...
LIN P20 CONT Vel=100% PDAT20
$OUT_C[50]=TRUE
LIN P21 CONT Vel=100%PDAT21
```

8.2 使用 TRIGGER WHEN DISTANCE 语句并以轨迹为参照的切换函数编程

关于使用 TRIGGER WHEN DISTANCE 句法并以轨迹为参照的切换函数的说明

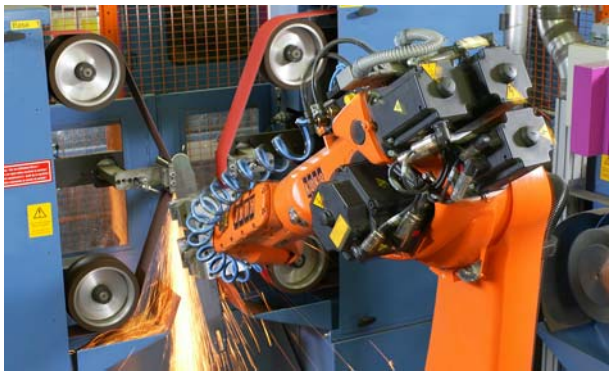


图 8-7: 循环应用程序

- 轨迹切换指令 TRIGGER 可以触发一个定义的指令。
- 指令与运动语句的起点或目标点有关
- 指令与机器人运动同时执行。
- 可以有切换点延迟

使用 TRIGGER WHEN DISTANCE 句法并以轨迹为参照的切换函数的功能

句法

- TRIGGER WHEN DISTANCE= *位置* DELAY= *时间* DO *指令* <PRIO= *优先级*>
- **位置**：规定在哪个点触发指令。可能的值：
  - **0**：指令在动作语句的起点处被触发。
  - **1**：指令在目标点处被触发。如果目标点是轨迹逼近形式，则指令将在该轨迹逼近弧形的中点处被触发。
- **时间**：以此可确定所选位置的延迟时间
  - 可应用正值和负值
  - 时间以毫秒 (ms) 为单位表示
  - 可毫无问题地应用 10,000,000 ms 及以下的时间值
  - 时间值过大或过小时最迟或最早将于切换极限处切换
- **指令**：可行的方式有：
  - 给一个变量赋值

不能对运行时间变量赋值。

  - OUT 指令
  - PULSE 指令
  - 调出一个子程序。在此情况下，**必须给明** *优先级*。
- **优先级**（仅当调出一个子程序时）：
  - 有优先级 1、2、4 - 39 以及 81 - 128 可供选择。
  - 优先级 40 - 80 预留给优先级由系统自动分配的情况。如果优先级应由系统自动给出，则应如下进行编程：PRIO = -1

使用 TRIGGER WHEN DISTANCE 语句并以轨迹为参照的切换函数编程

用 TRIGGER WHEN DISTANCE 进行切换的途径

- 起点和目标点均为精确停止点

```
LIN XP1
LIN XP2
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3
LIN XP4
```

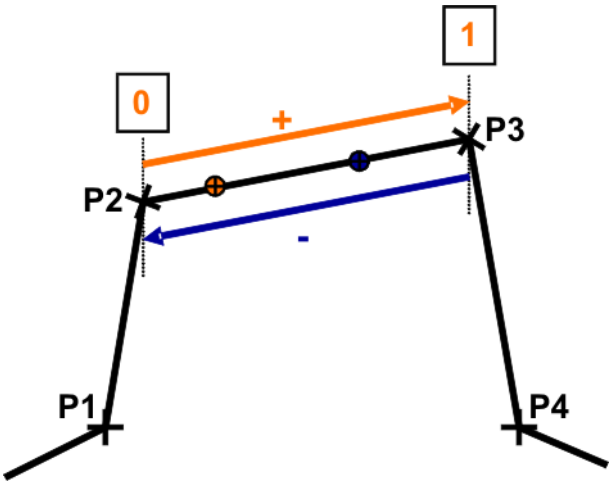


图 8-8: 带精确暂停 / 精确暂停的 TRIGGER WHEN DISTANCE 示例

- 起点是轨迹逼近点，目标点是精确停止点

```
LIN XP1
LIN XP2 C_DIS
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3
LIN XP4
```

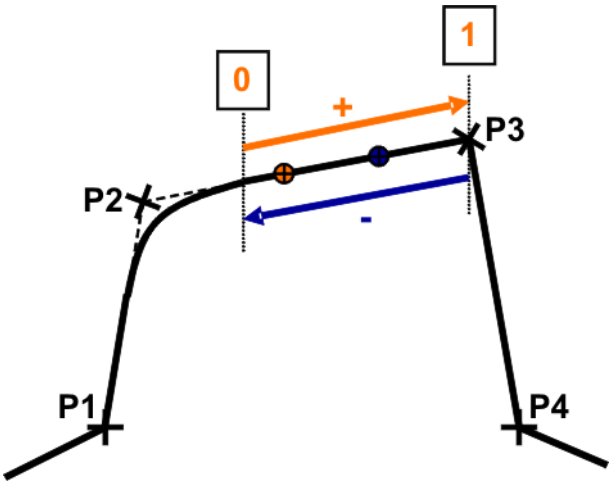


图 8-9: 带轨迹逼近 / 精确暂停的 TRIGGER WHEN DISTANCE 示例

- 起点是精确停止点，目标点是轨迹逼近点

```
LIN XP1
LIN XP2
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3 C_DIS
LIN XP4
```

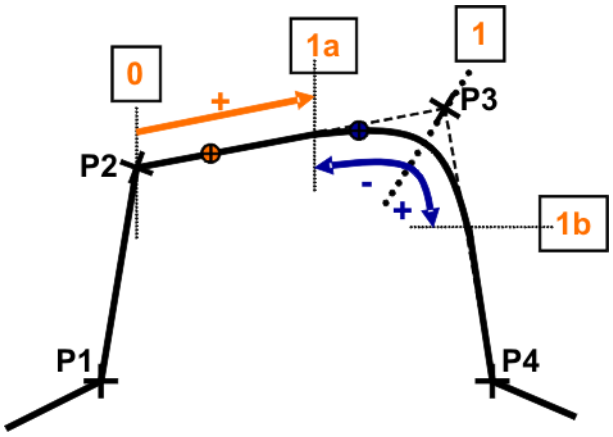


图 8-10: 带精确暂停 / 轨迹逼近的 TRIGGER WHEN DISTANCE 示例

- 起点和目标点均为轨迹逼近点

```
LIN XP1
LIN XP2 C_DIS
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3 C_DIS
LIN XP4
```

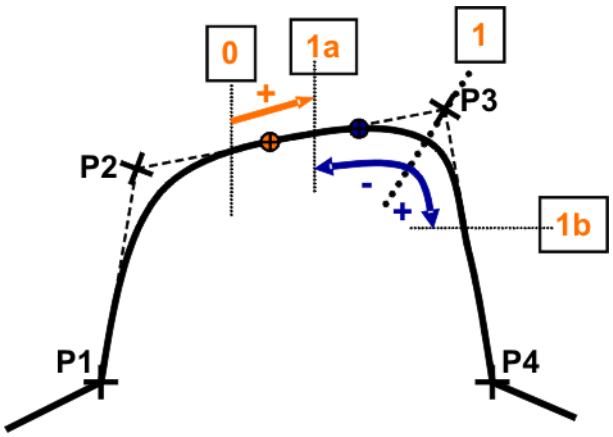


图 8-11: 带轨迹逼近 / 轨迹逼近的 TRIGGER WHEN DISTANCE 示例

8.3 使用 TRIGGER WHEN PATH 语句并以轨迹为参照的切换函数编程

关于使用 TRIGGER WHEN PATH 句法并以轨迹为参照的切换函数的说明

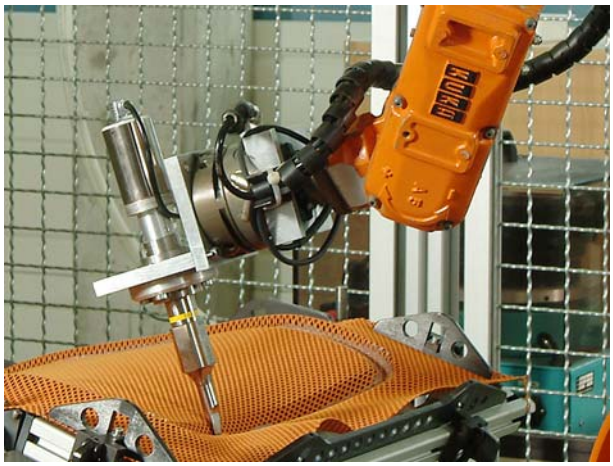


图 8-12: 粘接

- 轨迹切换指令 TRIGGER 可以触发一个定义的指令。

- 指令 PATH 与运动语句的目标点有关
- 指令与机器人运动同时执行。
- 可以与切换点有空间和 / 或时间上的位移 / 延时

必须通过轨迹运动 (LIN 或 CIRC) 驶至目标点。该运动不允许是 PTP。

使用 TRIGGER WHEN PATH 句法并以轨迹为参照的切换函数的功能

句法

- TRIGGER WHEN PATH= *行程段* DELAY= *时间* DO *指令* <PRIO= *优先级* >
- **行程段**：确定相对目标点的位移。
  - 正值：向运动结束方向推送该指令
  - 负值：向运动开始方向推送该指令
  - 行程段以毫米 (mm) 为单位表示
  - 可给出 +/- 10,000,000 mm 范围内的位移值
  - 值过大或过小时最迟或最早将于切换极限处切换
- **时间**：在此通过 PATH 值确定至选定位置的位移时间。
  - 可应用正值和负值
  - 时间以毫秒 (ms) 为单位表示
  - 可毫无问题地使用 10,000,000 ms 以下（包括 10,000,000 ms 本身）的时间值
  - 时间值过大或过小时最迟或最早将于切换极限处切换
- **指令**：
  - 给一个变量赋值

不能对运行时间变量赋值。

  - OUT 指令
  - PULSE 指令
  - 调用一个子程序。在此情况下，**必须给明** *优先级*。
- **优先级**（仅当调用一个子程序时）：
  - 有优先级 1、2、4 - 39 以及 81 - 128 可供选择。
  - 优先级 40 - 80 预留给优先级由系统自动分配的情况。如果优先级应由系统自动给出，则应如下进行编程：PRIO = -1

切换区域

- 朝运动结束方向位移：

  指令可**最多**执行**至 TRIGGER WHEN PATH 后的第一个精确停止点**（经过所有轨迹逼近点）。

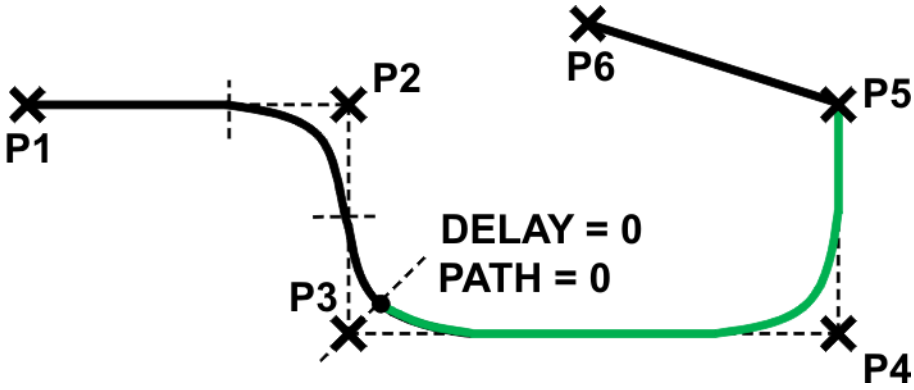


图 8-13: TRIGGER WHEN PATH 运动结束的切换极限

即：如果目标点是一个精确停止点，则该指令不能超出该目标点。

- 朝运动起始方向位移：
  指令可**最多**执行**至运动语句的起点**（即至 TRIGGER WHEN PATH 前的最后一个点）。
  - 如果起点是一个精确停止点，则指令可最多执行至起点。

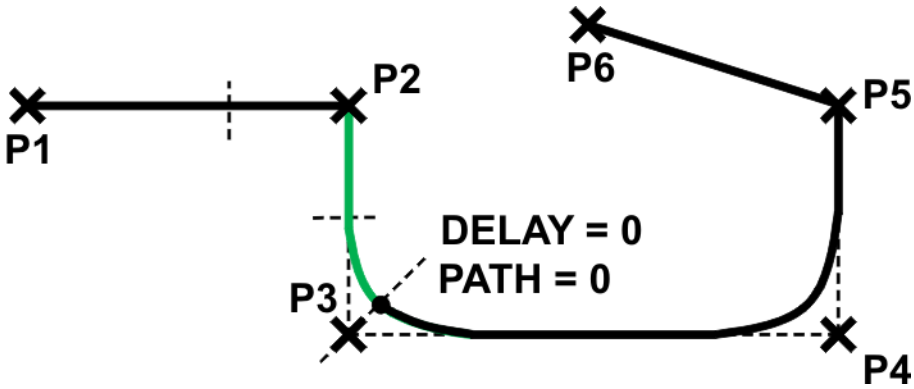


图 8-14: TRIGGER WHEN PATH 切换点的切换极限（精确暂停）

  - 如果起点是一个轨迹逼近的 PTP 点，则指令最多可执行至其轨迹逼近圆弧末端。

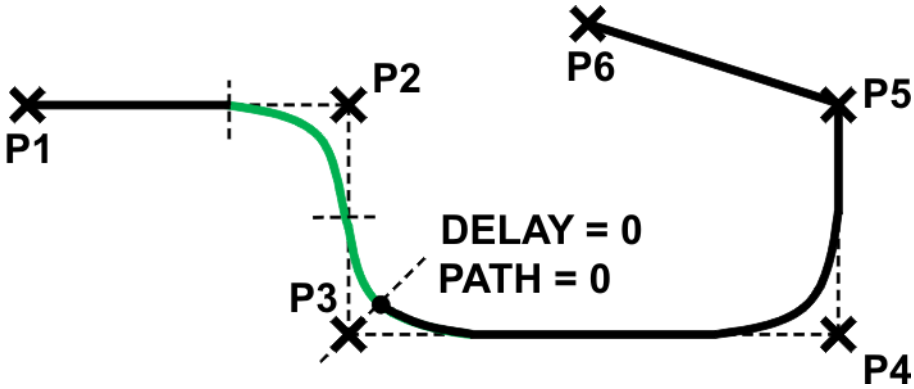


图 8-15: TRIGGER WHEN PATH 切换点的切换极限（轨迹逼近）

使用 TRIGGER WHEN PATH 句法并以轨迹为参照的切换函数编程

- 向运动结束方向切换

```
LIN XP2 C_DIS
TRIGGER WHEN PATH = Y DELAY = X DO $OUT[2] = TRUE
LIN XP3 C_DIS
LIN XP4 C_DIS
LIN XP5
LIN XP6
```

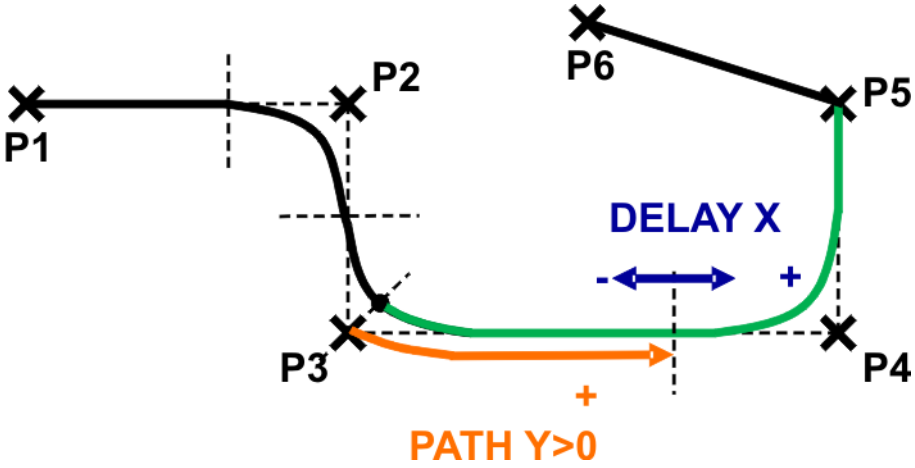


图 8-16: TRIGGER WHEN PATH 沿运动结束方向切换

- 向运动起始方向切换

```
LIN XP2 C_DIS
TRIGGER WHEN PATH = Y DELAY = X DO $OUT[2] = TRUE
LIN XP3 C_DIS
LIN XP4 C_DIS
LIN XP5
LIN XP6
```

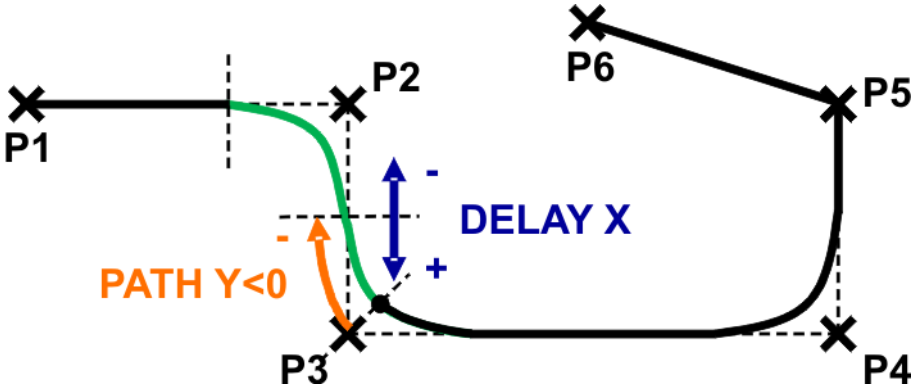


图 8-17: TRIGGER WHEN PATH 沿运动起始方向切换

9 使用 WorkVisual 编程

9.1 用 WorkVisual 进行项目管理

项目流程

1. 将项目从机器人控制系统载入 WorkVisual
   (>>> 9.1.1 " 用 WorkVisual 打开项目 " 页码 79)
2. 修改项目，例如 . KRL 程序
   (>>> 9.2 " 用 WorkVisual 编辑 KRL 程序 " 页码 91)
3. 比较项目（合并）
   (>>> 9.1.2 " 用 WorkVisual 比较项目 " 页码 82)
4. 将项目从 WorkVisual 载入机器人控制系统（deployen，即调度）
   (>>> 9.1.3 " 将项目传输给机器人控制系统（安装）" 页码 86)
5. 激活项目（确认）
   (>>> 9.1.4 " 在机器人控制系统中激活项目 " 页码 89)

9.1.1 用 WorkVisual 打开项目

WorkVisual 的缩写

软件包 **WorkVisual** 是受控于 KR C4 的机器人工作单元的工程环境。它具有以下功能：

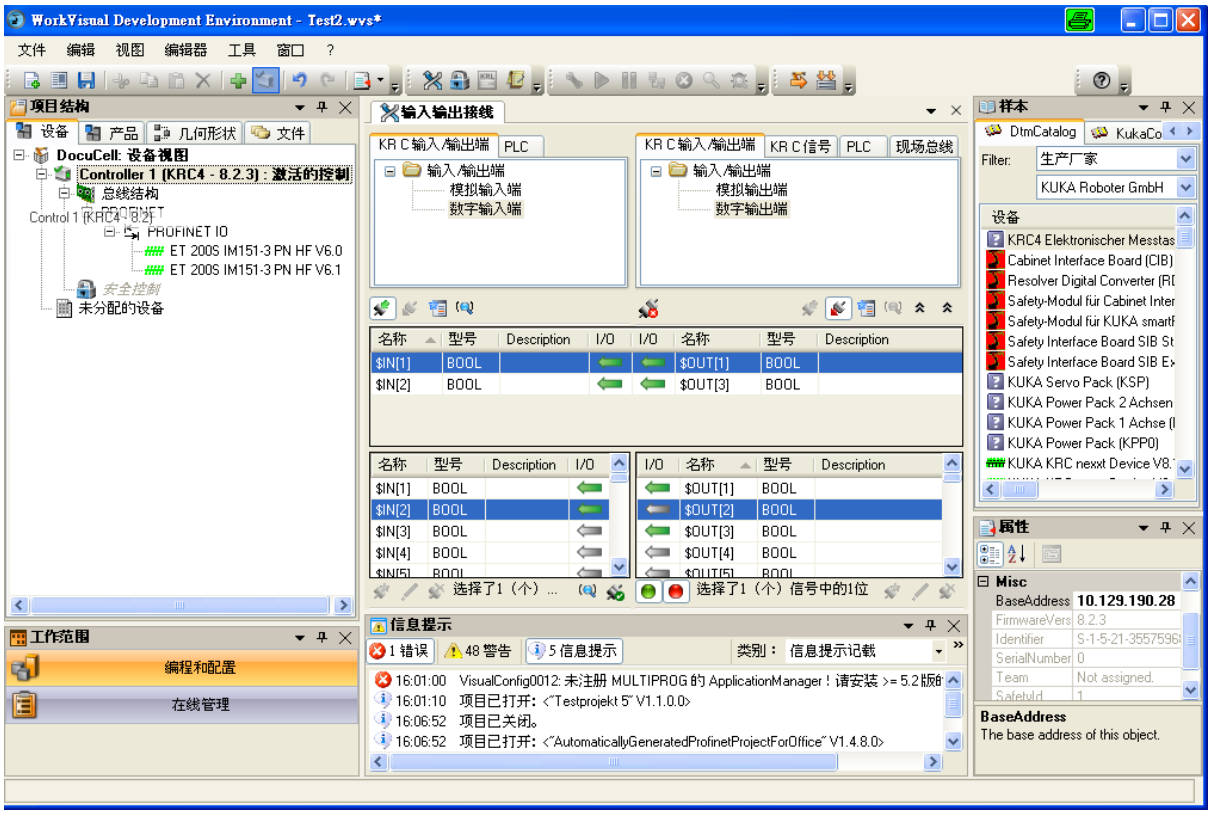


图 9-1: WorkVisual 操作界面

- 将项目从机器人控制系统传输到 WorkVisual
- 将项目与其它项目进行比较，如果需要则应用差值
- 将项目传送给机器人控制系统
- 架构并连接现场总线
- 编辑安全配置
- **对机器人离线编程**

- 管理长文本
- 诊断功能
- 在线显示机器人控制系统的系统信息
- 配置测量记录、启动测量记录、分析测量记录（用示波器）

WorkVisual 操作界面的结构和功能

在默认状态下，并非所有单元都显示在操作界面上，而是可根据需要显示或隐藏。

除了此处图示的窗口和编辑器之外，还有更多可供选用。这些可通过菜单项 **【窗口】** 和 **【编辑器】** 显示。

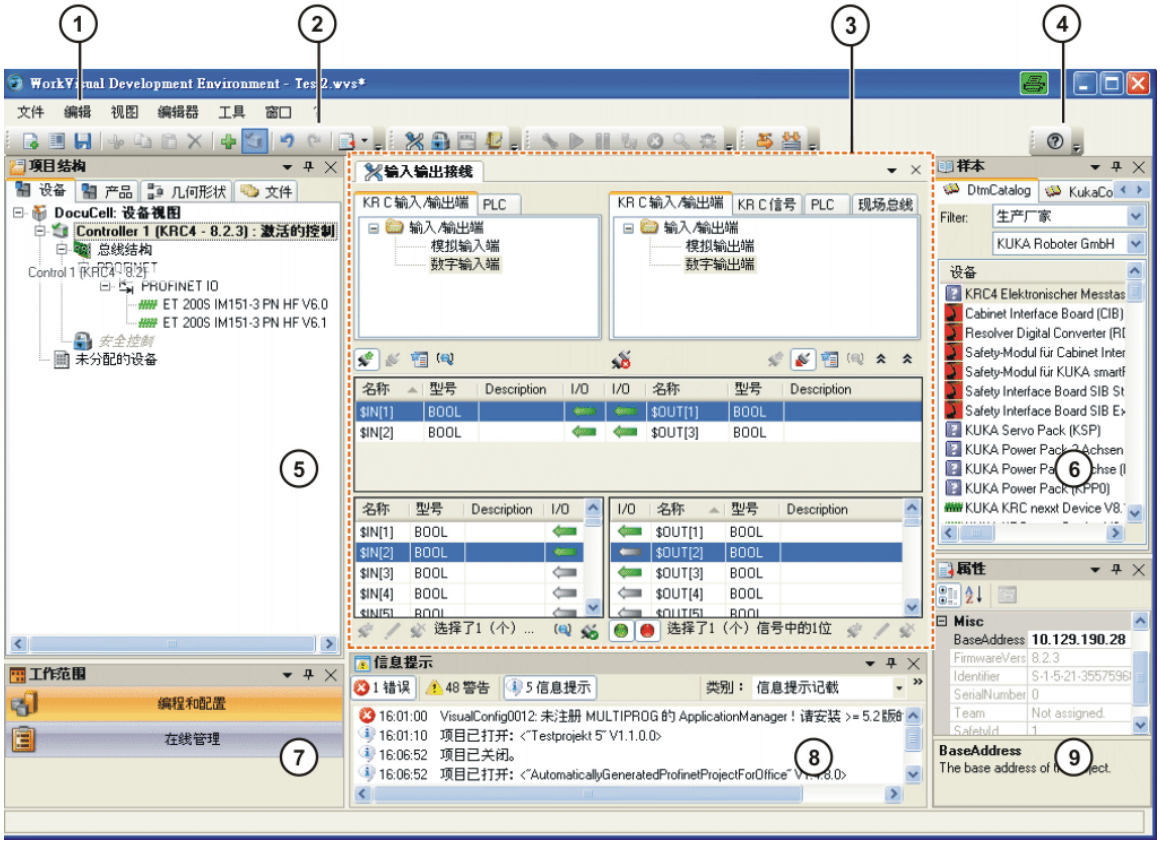


图 9-2: 操作界面概览

【项目结构】窗口

-

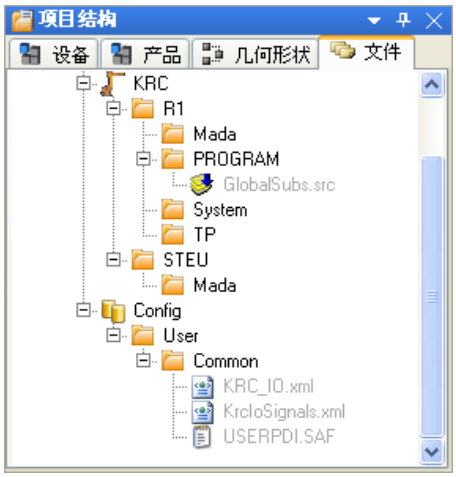


图 9-3: 例如：自动生成的文件为灰色

- **设备：**
  在选项卡 **【设备】**中显示设备的关联性。此处可将单个设备分配给一个机器人控制系统。
- **产品：**
  选项卡 **【产品】**主要用于 WorkVisual Process，较少用于 WorkVisual。此处将一个产品所需的所有任务均显示在一个树形结构中。
- **几何形状：**
  选项卡 **【几何形状】**主要用于 WorkVisual Process，较少用于 WorkVisual。此处将项目中的所有三维对象均显示在一个树形结构中。
- **文件：**
  选项卡 **【文件】**包含属于项目的程序和配置文件。
  彩色显示文件名：
  - 自动生成文件（用功能 **【生成代码】**）：灰色
  - 在 WorkVisual 中手动贴入的文件：蓝色
  - 从机器人控制系统传输到 WorkVisual 的文件：黑色

项目浏览器

-

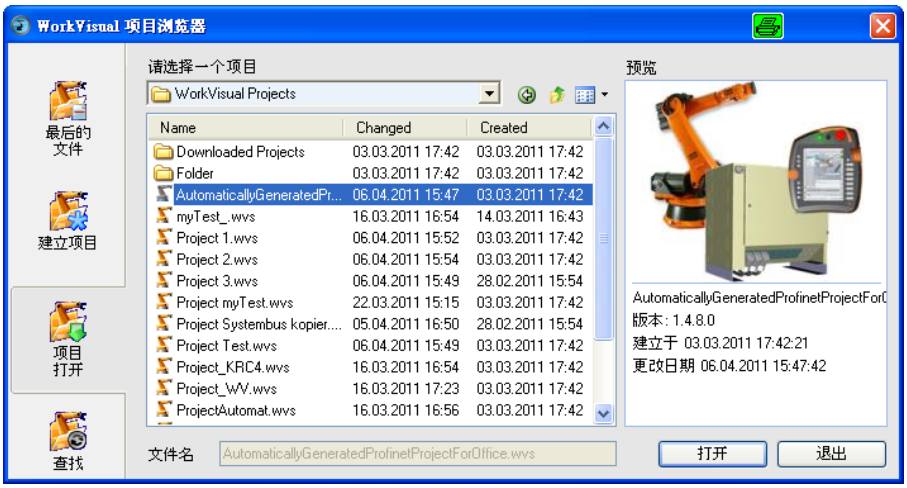


图 9-4: 项目浏览器

- **【最后的文件】**显示最后使用的文件
- **【建立项目】**用于生成：
  - 一个新的空项目
  - 根据模板建立一个新项目
  - 在现有项目基础上创建新项目
- **【项目打开】**用于打开现有项目
- **【查找】**用于从机器人控制系统加载一个项目

用 WorkVisual 加载项目的方法

在每个**具有网络连接**的机器人控制系统中，都可选出一个项目并传送给 WorkVisual 。

即使该电脑里尚没有该项目时也能实现。

该项目保存在目录：Eigene Dateien（我的文档）\WorkVisual Projects\Downloaded Projects 之下。

1. 按序选择菜单项：**文件 > 查找项目。** **【项目浏览器】**随即打开。左侧已选中选项卡 **【查找】**。
2. 在 **【可用工作单元】**栏展开所需工作单元的节点。该工作单元的所有机器人控制系统均显示出来。
3. 展开所需机器人控制系统的节点。所有项目均将显示。

4. 选中所需项目，并点击 **【打开】**键。项目将在 WorkVisual 里打开。

9.1.2 用 WorkVisual 比较项目

说明

- 一个 WorkVisual 中的项目可以与另一个项目比较
- 这可以是机器人控制系统上的一个项目或一个本机保存的项目
- 一目了然地列出不同之处并可以显示详细信息。
- 用户可以针对每个不同之处分别决定
  - 是否要保持当前项目中的状态
  - 或者要接受另一个项目中的状态

项目比较的原理

合并项目

-

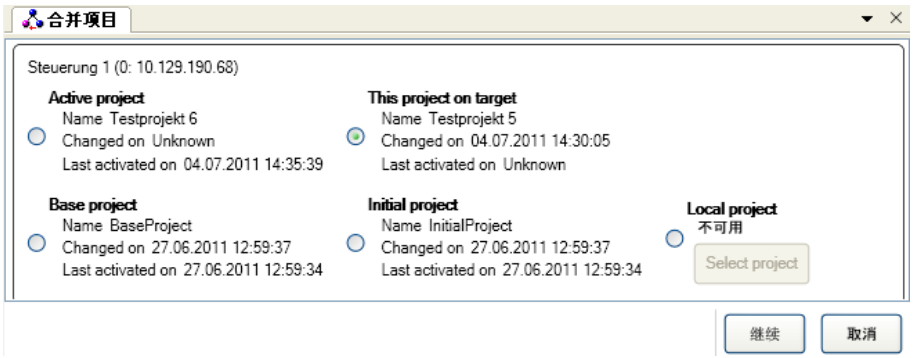


图 9-5: 选择 “ 合并 ” 项目

- 激活的项目
- 控制系统里的同名项目（只限于通过网络连接）
- 主项目
- 初始项目
- 本机项目（来自笔记本电脑）

比较

项目之间的差异即以一览表的形式显示出来。对于每项区别，都可选择要应用哪种状态。

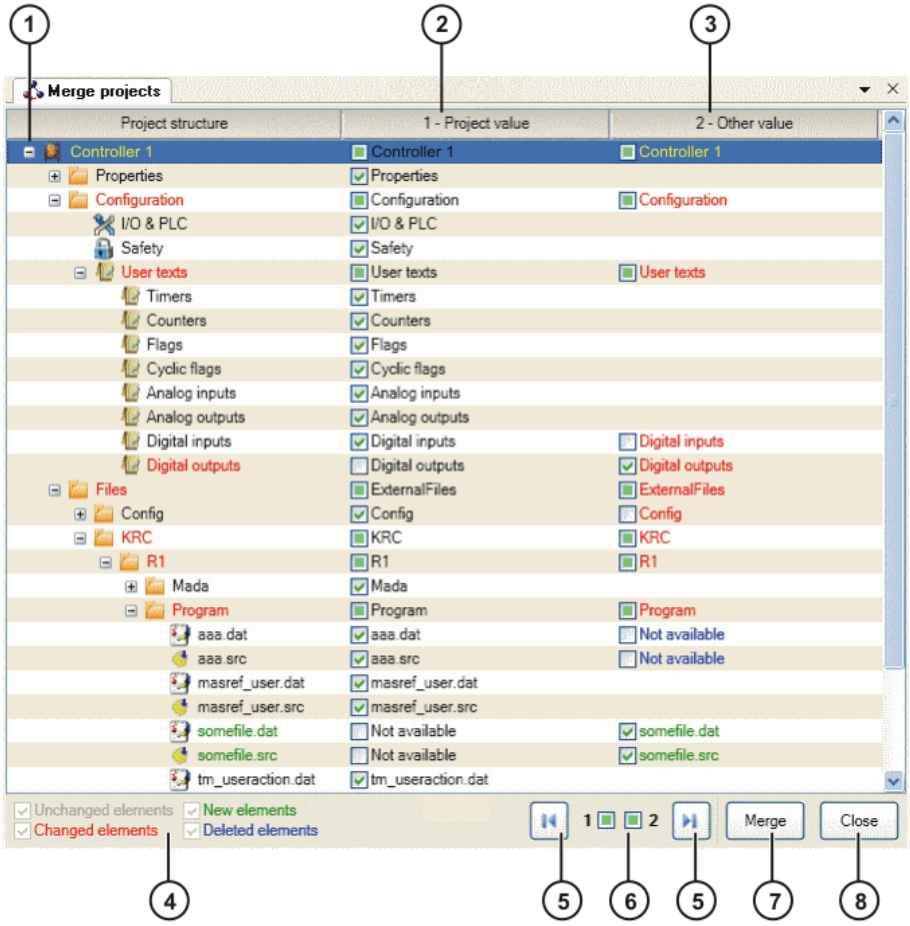


图 9-6: 例如：区别概览

颜色说明：

列	说明
项目结构	每个单元都由其被选定的列中的颜色显示。
当前项目值 (1)	所有单元都以黑色显示。
基准值 (2)	■ 绿色：在打开的项目中不存在但在比较项目中存在的单元。 ■ 蓝色：在打开的项目中存在但在比较项目中不存在的单元。 ■ 红色：所有其它单元。这也包括上级单元，这些上级单元有多种颜色。

项目比较的处置方法

1. 在 WorkVisual 中按序选择菜单项：**工具 > 比较项目**。窗口 **【比较项目】** 打开。

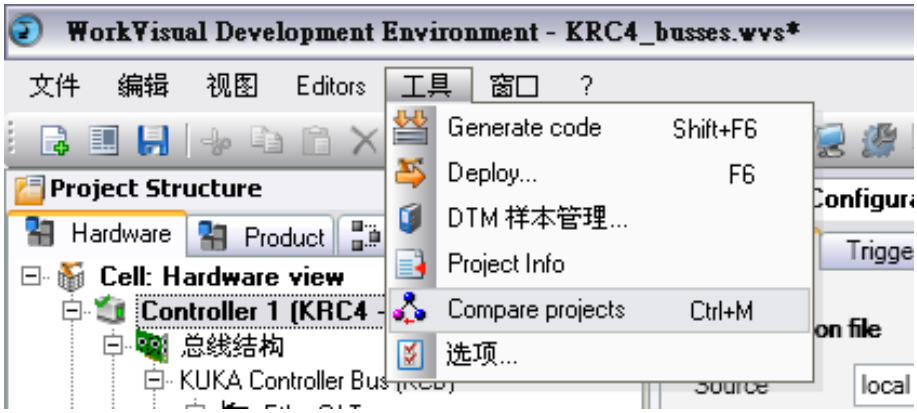


图 9-7

2. 选择需与当前 WorkVisual 项目作比较的项目，例如实际机器人控制系统上的同名项目。

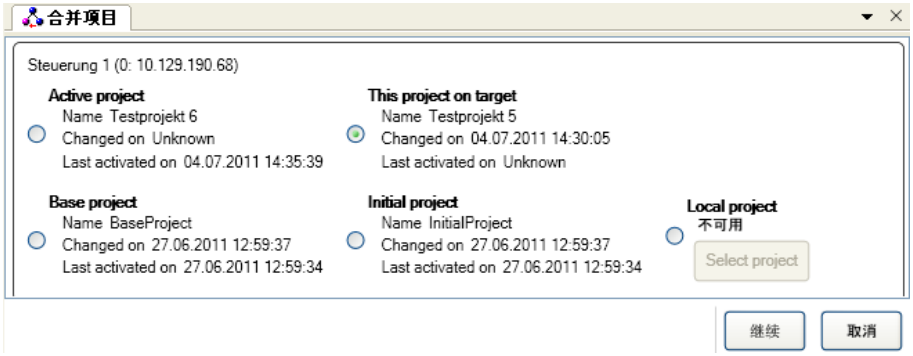


图 9-8: 选择 “ 合并 ” 项目

3. 点击**继续**。显示一个进度条。（如果项目包含多个机器人控制系统，则显示每个系统的进度条）。
4. 当进度条已填满且出现状态显示 “ **合并准备就绪** ” 时，点击 **【显示区别】** 键。项目之间的差异即以一览表的形式显示出来。

   若未发现差异，则在信息窗口中显示与此相关的讯息。继续执行第 8 步。随后无需执行后续步骤。
5. 对每种差异均应选择需采用的状态。不必一次过完成所有差异的这种选择。

   如果合适的话，也可保留默认选择。
6. 按动 **合并**，以将更改传给 WorkVisual 应用。
7. 重复步骤 5 和 6 任意多次。这样，可逐步编辑各个区域。

   若不再有其他区别，则显示以下信息：**无其它区别。**
8. 关闭窗口 **【比较项目】**。
9. 若在机器人控制系统的项目中改变了附加轴的参数，则必须在 WorkVisual 中将其更新：
   - 为这些附加轴打开窗口**机器参数配置**。
   - 在**一般轴相关机器参数**区域内按用于导入机器参数的按键。

     数据即被更新。
10. 保存项目。

项目比较示例

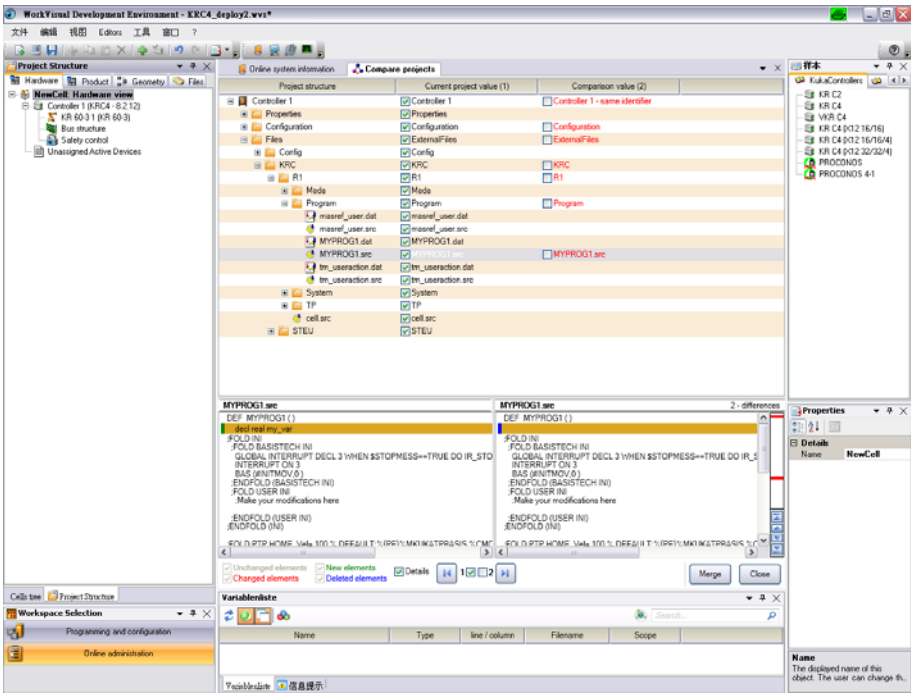


图 9-9: 示例：项目比较概览

决定应该应用文件的哪种状态

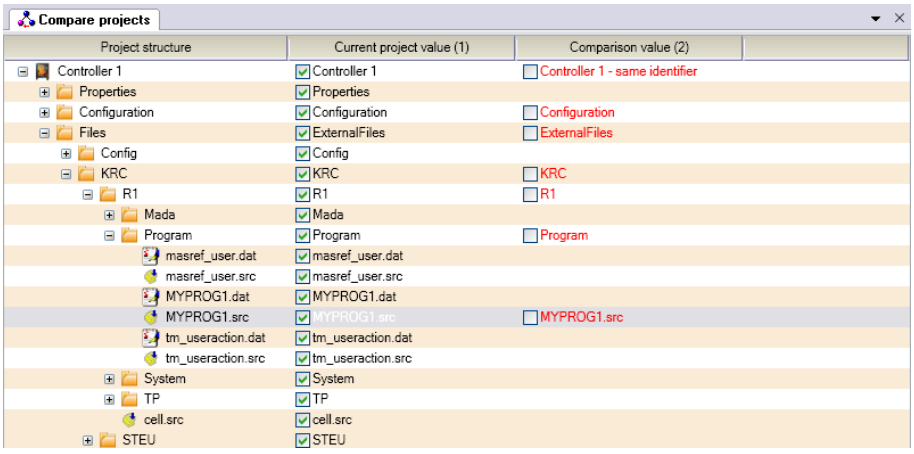


图 9-10: 示例：合并项目

【详细信息】功能键激活时，可显示不同文件之间的差异

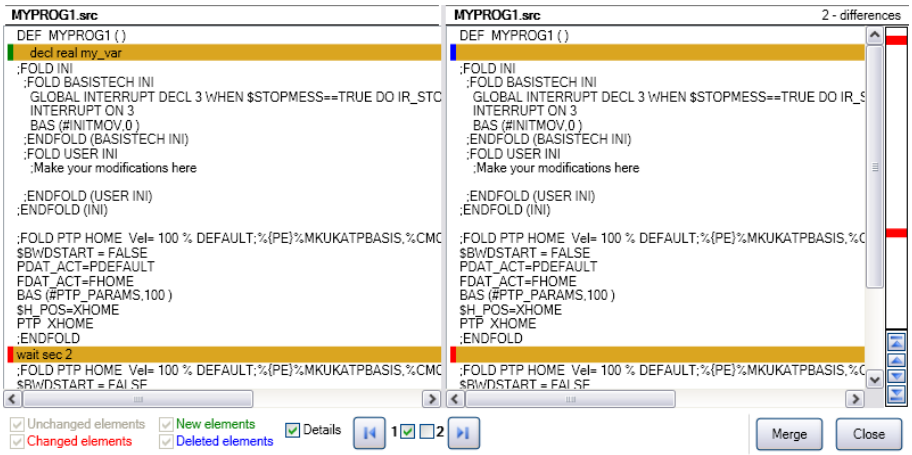


图 9-11: 示例：细节已更新

9.1.3 将项目传输给机器人控制系统（安装）

说明

- 如果项目里已发生变化，WorkVisual 必须将变化情况发送给控制系统
- 库卡将该做法叫做 “**Deployen**”（调度）
- 在将一个项目传输到机器人控制系统时，总是先生成代码。
- 与实际机器人控制系统之间的网络连接是 “**Deployen**” 的首要条件

> 如果在实际应用的机器人控制系统上有一个项目，在先前某时已被传输但从未被激活，则会在传输另一个项目时被盖写。
> 通过某个项目的传输和激活，一个在实际应用的机器人控制系统上的同名项目被盖写（在安全询问之后）。

功能

生成代码

- 以此步骤可单独生成代码，这样可事先检验生成过程是否无错。
- 调用可通过：
  - 按序打开菜单：**工具 > 生成代码**
  - 或者按动按钮 
  - 代码在窗口 **【项目结构】**的选项卡 **【文件】**中显示。
    自动生成的代码显示为浅灰色。

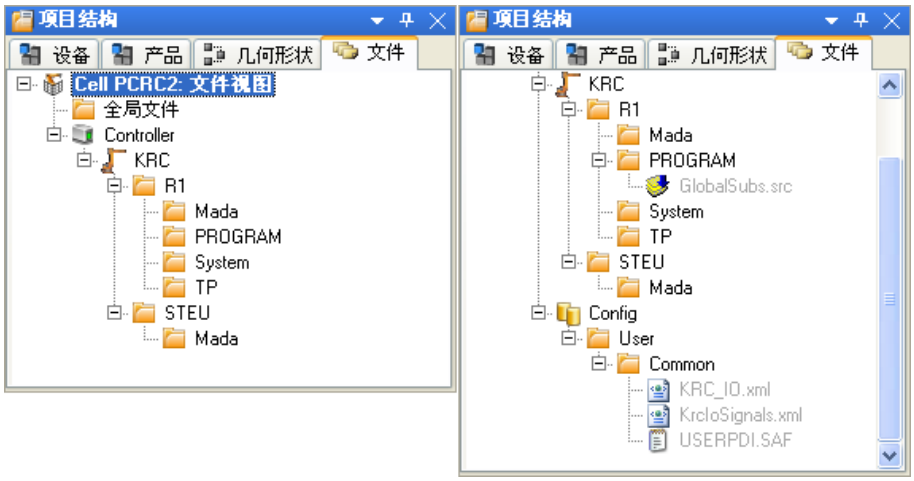


图 9-12: 生成代码示例：之前 - 之后

- 代码即生成。当过程结束时，信息窗口中显示以下信息提示：**编译了项目 <“{0}” V{1}>。结果见文件树。**

操作步骤

1. 在菜单栏中点击 **【安装 ...】**键。窗口 **【项目传输】**打开。

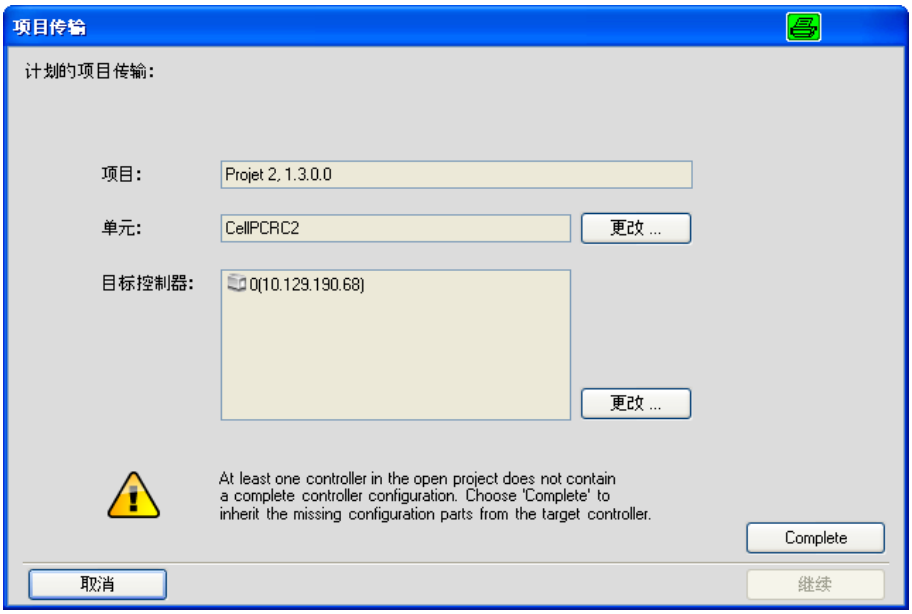


图 9-13: 不完整配置的概览及提示

2. 如果所涉及的项目还从来未从机器人控制系统回传至 WorkVisual，则它还不包含所有配置文件。这通过一个提示显示出来。（配置文件包括机器参数文件、安全配置文件和很多其它的文件）。
   - 如果未显示该提示：继续执行第 13 步。
   - 如果显示该提示：继续执行第 3 步。
3. 点击 **【完整化】**。显示以下安全询问：**项目必须保存，并重置激活的控制系统！ 您想继续吗？**
4. 点击 **【是】** 应答。窗口 **【合并项目】** 随即打开。

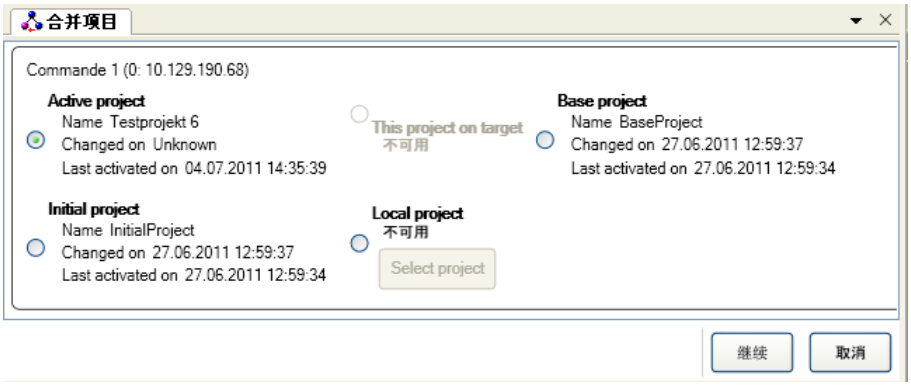


图 9-14: 选择 “ 完整化 ” 项目

5. 选择一个要应用其配置数据的项目，例如一个在实际存在的机器人控制系统上的激活项目。
6. 点击 **【继续】**。显示一个进度条。（如果项目包含多个机器人控制系统，则显示每个系统的进度条）。
7. 当进度条已填满且出现状态显示 “ **合并准备就绪** ” 时，点击 **【显示区别】** 键。
   项目之间的差异即以一览表的形式显示出来。
8. 对每种差异均应选择需采用的状态。不必一次完成所有差异的这种选择。
   如果合适的话，也可保留默认选择。
9. 点击 **合并**，以应用更改。
10. 重复步骤 8 和 9 任意多次。这样，可逐步编辑各个区域。

若不再有其他区别，则显示以下信息：**无其它区别。**

11. 关闭窗口 **【比较项目】**。
12. 在菜单栏中点击按键 **【安装 ...】**。重新显示单元归类概览。有关不完整配置的提示不再显示。

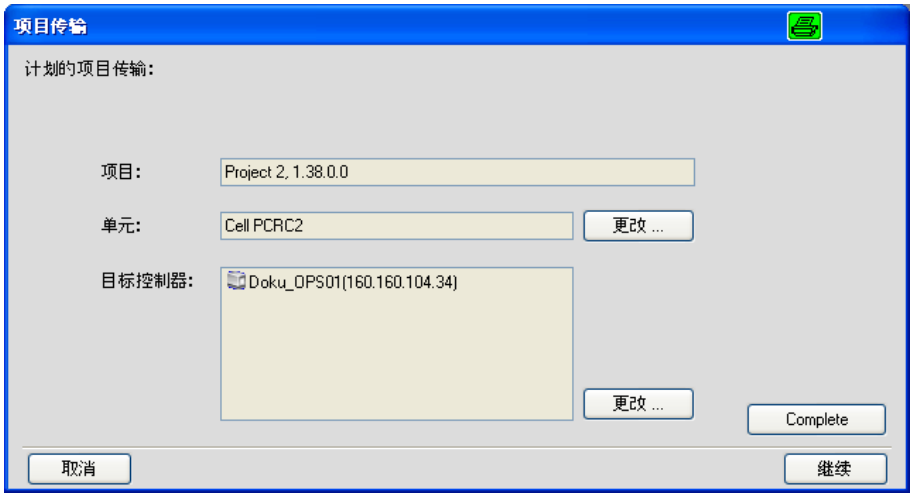


图 9-15: 概览

13. 点击 **【继续】**。启动生成程序。当进度条显示 100 % 时，程序即已生成，项目被传输。
14. 点击 **【激活】**。

> **警告** 在运行方式 AUT 和 AUT EXT 状态下，只涉及程序变动的项目无需经过安全问答即可激活

15. 仅限于运行方式 T1 及 T2：KUKA smartHMI 显示安全询问 "*允许激活项目 [...] 吗？*"。另外还显示通过激活是否会盖写一个项目；如果是的话，是哪一个？

    如果没有重要的项目被盖写，则需在 30 分钟之内点击 **【是】**键确认。
16. 显示相对于机器人控制系统仍激活项目而进行的更改的概览。通过 **【详细信息】**复选框可以显示相关更改的详情。
17. 概览显示安全询问 "*您想继续吗？*"。用 **【是】**回答。该项目即在机器人控制系统中激活。对此 WorkVisual 将显示一条确认信息。

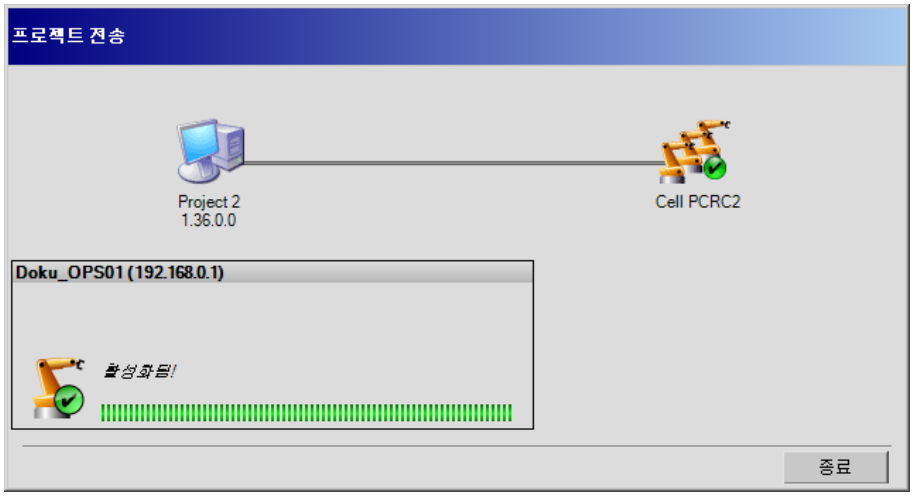


图 9-16: WorkVisual 中的确认

18. 点击 **【结束】**键关闭窗口 **【项目传输】**。
19. 若未在 30 分钟内回答机器人控制系统的询问，则项目仍将传输，但在机器人控制系统中不激活。该项目可独立激活。

⚠ 警告 在机器人控制系统中激活一个项目后必须在控制系统中检查安全配置！ 如果不检查，则机器人可能以错误的数据运行。可能会造成人员死亡、严重身体伤害或巨大的财产损失。

⚠ 警告 如果一个项目的激活失败，则会在 WorkVisual 显示一则出错信息。在此情况下，必须执行以下措施：

- 重新激活项目。（同一个或另一个）。
- 或以冷启动重新启动机器人控制系统。

9.1.4 在机器人控制系统中激活项目

说明

- 项目可直接在机器人控制系统中激活。
- 项目也可在机器人控制系统上从 WorkVisual 激活。
  （这里不再赘述，可从 WorkVisual 在线读取相关资料）

项目管理功能

一般性说明

- 机器人控制系统可对系统内的多个项目实行管理
- 与此相关的所有功能只在用户组 **【专家】** 里才可处理
- 调用可通过：
  - 按序打开菜单：**文件 > 项目管理**；
  - 点击操作界面上的 **WorkVisual 符号**键，然后点击 **【打开】**

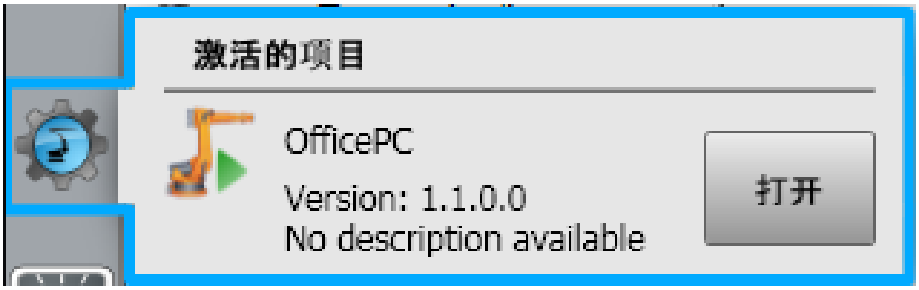


图 9-17: 操作界面里的项目显示

处置 / 操作

-

图 9-18: 窗口“项目管理”

- 除了通常的项目，窗口**项目管理**还包含以下特别项目：

项目	说明
初始项目	初始项目总是存在。用户无法更改。它包含供货时机器人控制系统的状态。
主项目	用户可将激活的项目作为主项目来保存。该功能一般用于确保一个有效可靠的项目状态。 主项目不能激活，但可以复制。用户无法更改主项目。但它可以通过保存一个新的主项目被覆盖（在安全询问之后）。 如果激活了一个未包含所有配置文件的项目，则从主项目里提取和应用所缺失的信息。例如当某个项目由 WorkVisual 旧版本激活时，便可能是该情况。（配置文件包括机器数据文件、安全配置文件和其它很多文件）。

- 软键说明

软键	说明
激活	激活选定的项目。 如果所选定的项目被钉住：则建立一份所选定项目的副本。（被钉住的项目本身不能激活，只能激活其副本）。然后，用户可以决定是应立即激活拷贝，还是要使当前项目保持激活。
固定	被钉住的项目不可被更改、激活或删除。然而可被拷贝或松开。您可将项目钉住，以防止其被意外删除。
松开	松开项目。
拷贝	拷贝选定的项目。
删除	删除选定的项目。
编辑	打开一个可更改所选定项目的名称和 / 或说明的窗口。
更新	更新项目列表。例如：如此可显示自打开显示以来传输到机器人控制系统中的项目。

操作步骤

限制：如果激活时会影响到 **通讯参数** 的范围更改，则必须选择 【安全维护员】或更高级别的用户组。
如果选择了工作模式 AUT 或 AUT EXT：当只会引起 KRL 程序变化时才能激活项目。如果项目中含有会造成其它变化的设置，则不能将其激活。

1. 按序选择菜单项：**文件 > 项目管理**。窗口 **【项目管理】**打开。
2. 选中所需项目并以按键 **【激活】**来激活。
3. KUKA smartHMI 显示安全询问 “ *允许激活项目 [...] 吗？* ”。另外还显示是否通过激活以覆盖一个项目；如果是的话，是哪一个？
   如果没有相关的项目要覆盖，在 30 分钟内用 **【是】**键确认该查询。
4. 显示相对于机器人控制系统仍激活项目而进行的更改的概览。通过 **【详细信息】**复选框可以显示相关更改的详情。
5. 概览显示安全询问 “ *您想继续吗？* ”。用 **【是】**回答。该项目即在机器人控制系统中激活。

⚠ 警告 在机器人控制系统中激活一个项目后必须在控制系统中检查安全配置！ 如果不检查，则机器人可能以错误的数据运行。可能会造成人员死亡、严重身体伤害或巨大的财产损失。

9.2 用 WorkVisual 编辑 KRL 程序

- 文件处理
  (>>> 9.2.1 " 文件处理 " 页码 91)
- KRL 编辑器的使用
  (>>> 9.2.2 "KRL 编辑器的使用 " 页码 97)

9.2.1 文件处理

说明

- 将现有文件载入 KRL 编辑器中
- 从样本中添加新文件
- 添加外部文件

样本模板的原理

- 在使用模板前必须载入相应的样本
- 通过**文件 > 添加样本**载入样本，激活相应的模板
  - KRL 模板：*KRL Templates.afc*
  - VW 模板：*VW Templates.afc*
- KRL 模板的样本

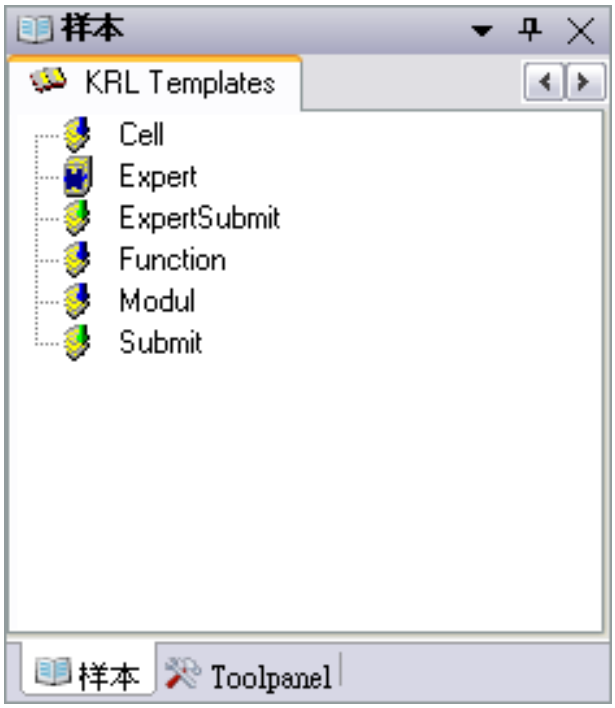


图 9-19: KRL 模板的样本

操作步骤

在 KRL 编辑器中打开文件 (SRC/DAT) 的操作步骤

1. 切换文件的项目树

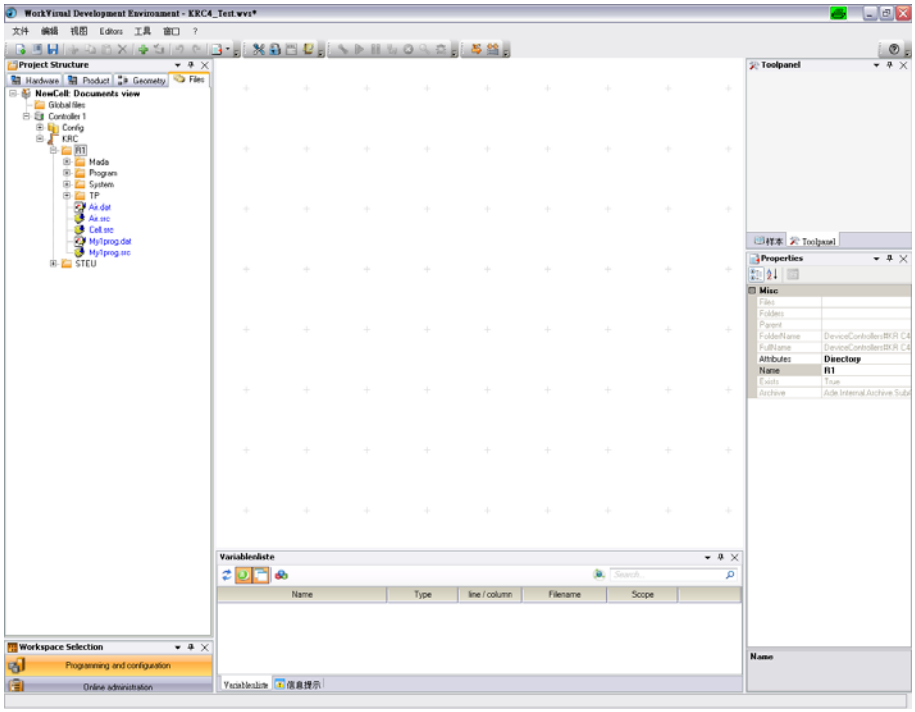


图 9-20: WorkVisual 项目树

2. 将文件夹展开至 R1 文件夹

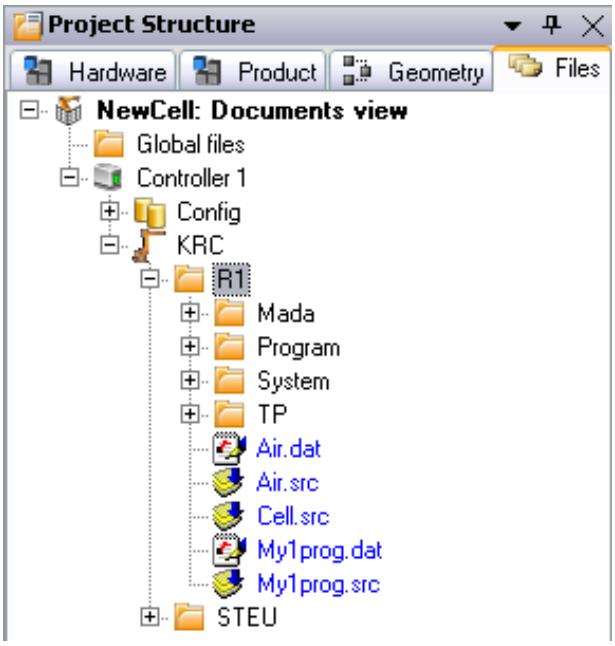


图 9-21: WorkVisual 文件结构 (R1) 项目树

3. 选择文件或者
   - 通过双击
   - 工具栏按钮 
   - 右击并在相关菜单中选择  Krl 编辑器

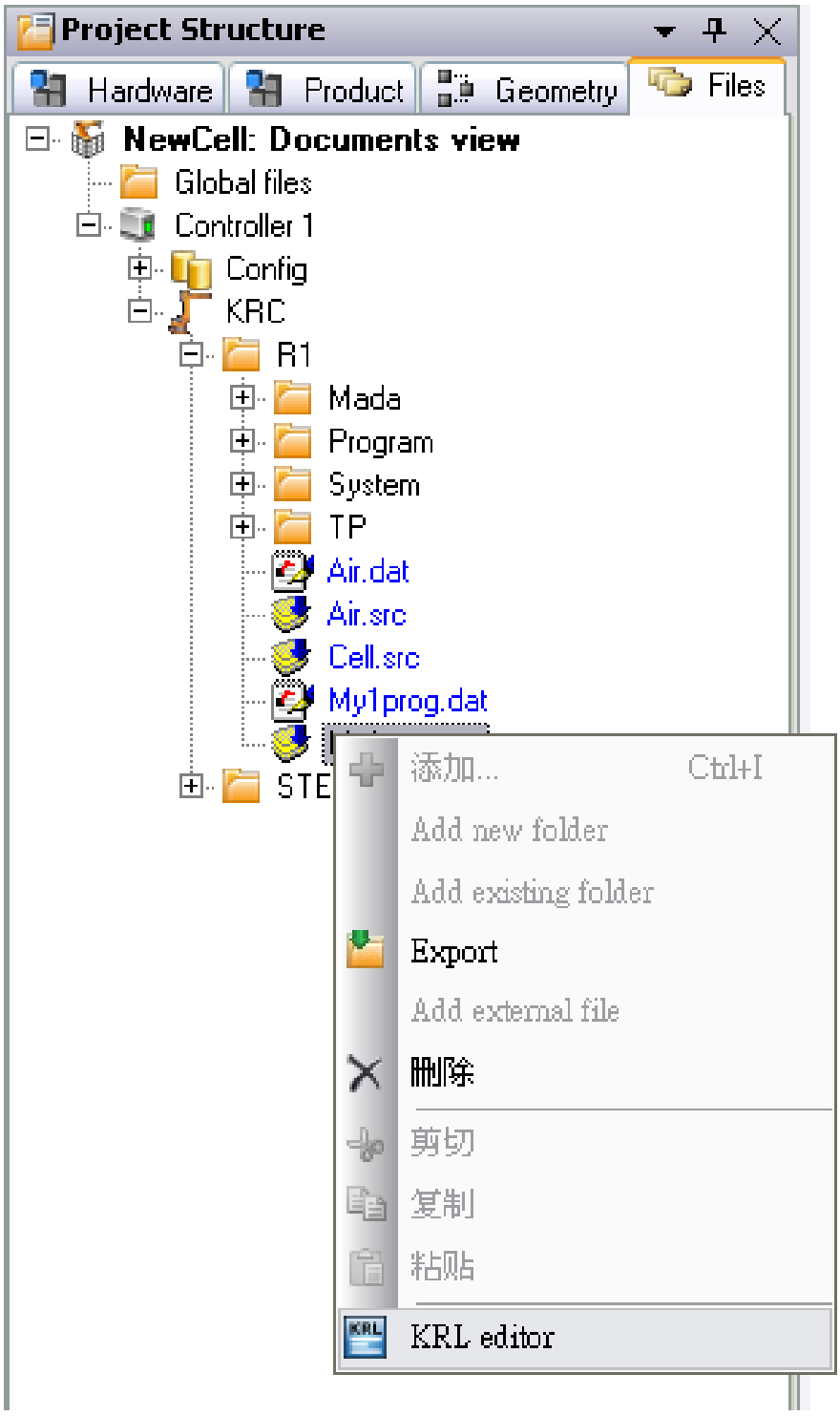


图 9-22: WorkVisual 相关菜单（KRL 编辑器）

借助 KRL 模板添加文件的操作步骤

1. 切换文件的项目树
2. 将文件夹展开至 R1 文件夹
3. 选择需要创建新文件的文件夹
4. 右击并在相关菜单中选择  添加

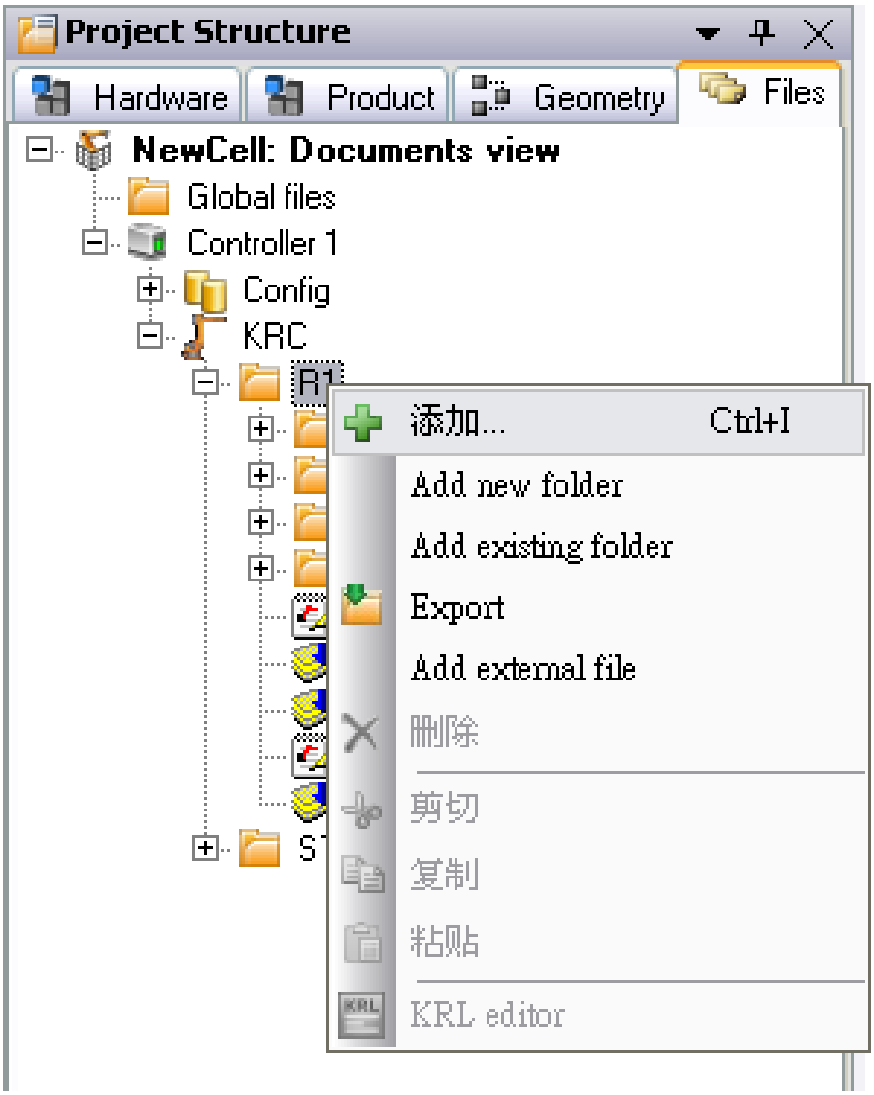


图 9-23: WorkVisual 相关菜单 （添加）

5. 选择模板

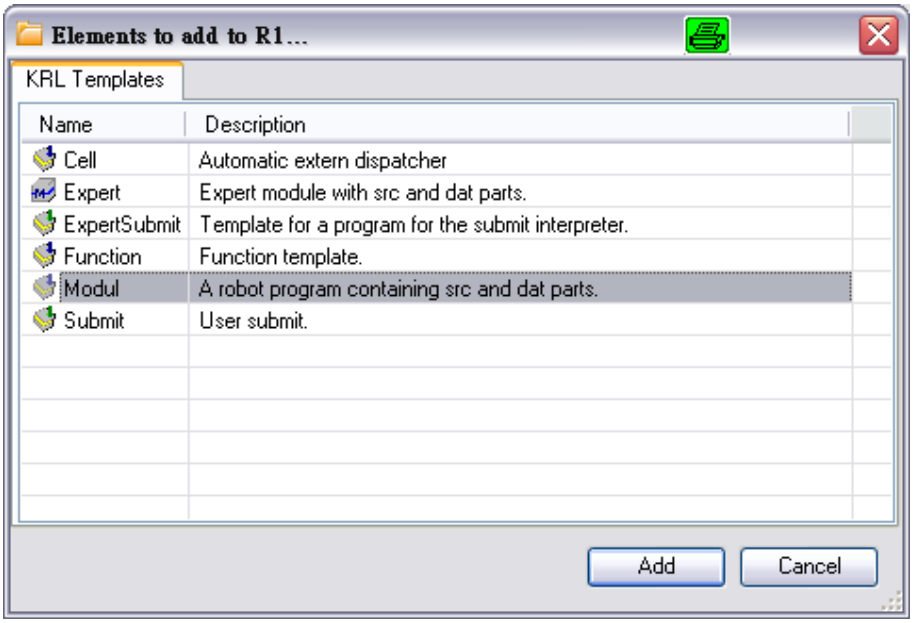


图 9-24: WorkVisual KRL 模板

6. 指定程序名称

添加外部文件的操作步骤

1. 切换文件的项目树
2. 将文件夹展开至 R1 文件夹
3. 选择需要创建新文件的文件夹
4. 右击并在相关菜单中选择“添加外部文件”

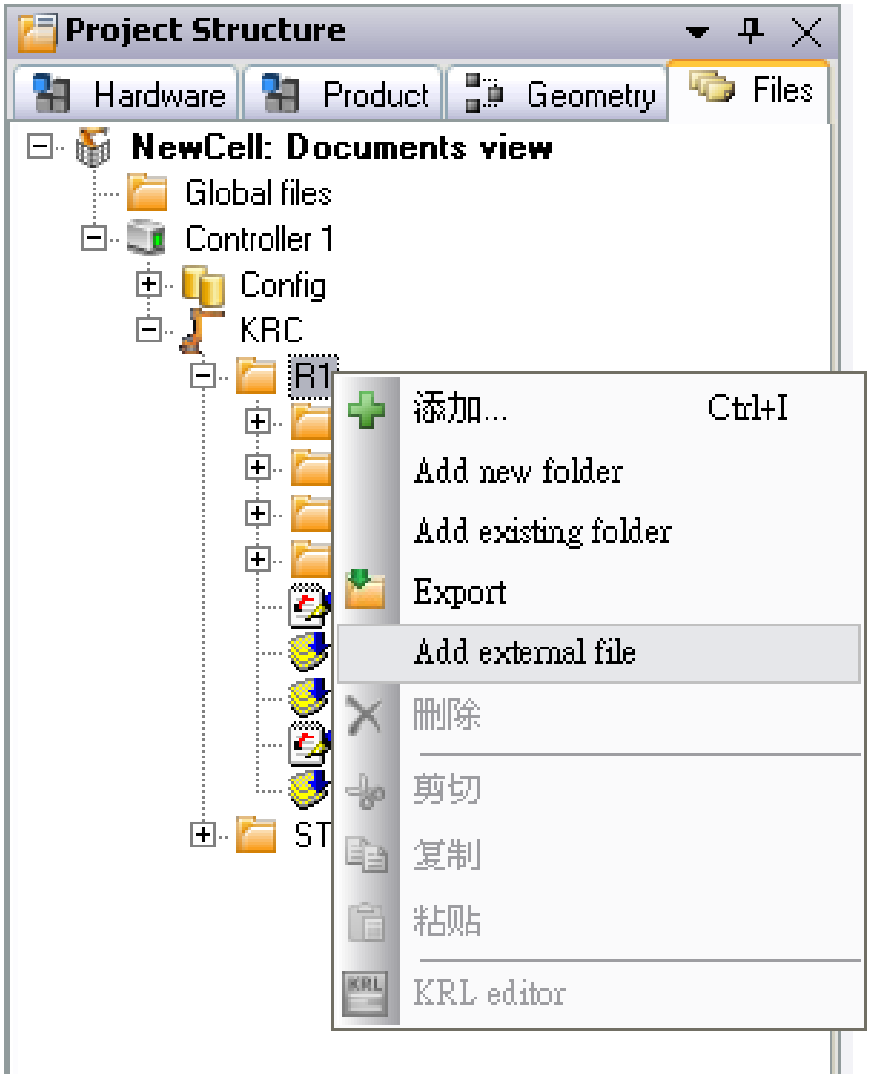


图 9-25: WorkVisual 相关菜单（添加外部文件）

5. 选择文件，然后按下“打开”

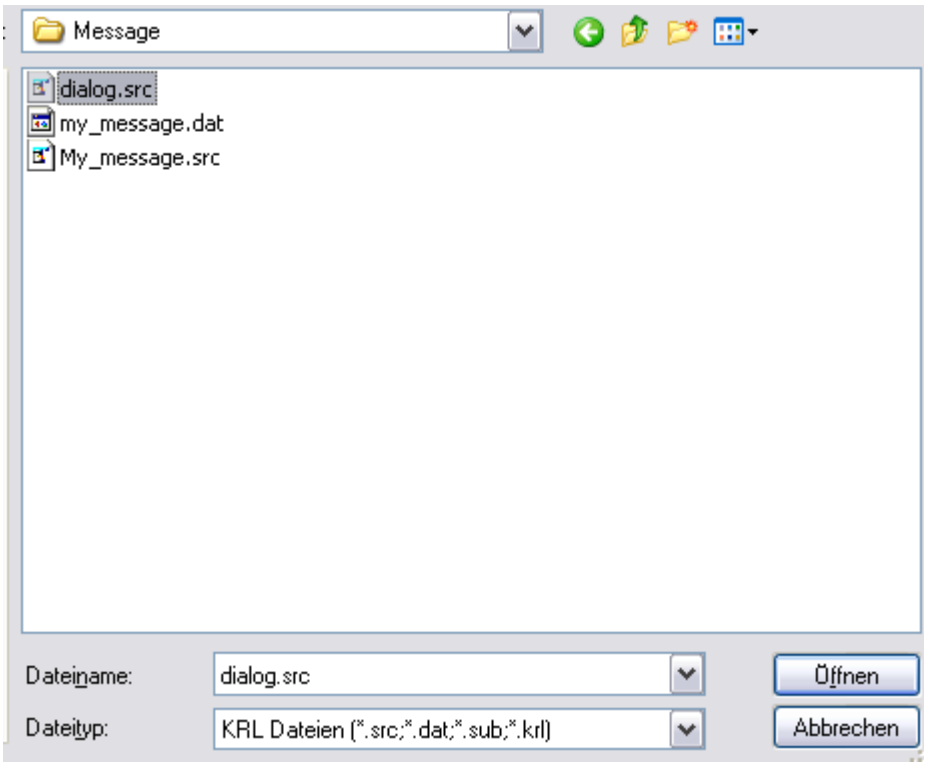


图 9-26: 添加 WorkVisual 外部文件

9.2.2 KRL 编辑器的使用

KRL 编辑器的说明

程序编辑 (SRC/DAT)

- 通过直接 KRL 输入
- 借助 KRL 指令的快速输入（KRL 片断）
- 借助工具箱中的联机表单

KRL 编辑器的属性

- 编辑器的配置
- KRL 编辑器中的颜色说明
- 错误识别 (KRL Parser)
- 变量表
- KRL 编辑器中的附加编辑功能

配置 KRL 编辑器

- 选择菜单顺序**其它 > 选项**。窗口**选项**打开。
- 文件夹**文本编辑器**包括子项**外观**和**行为**。
- 外观

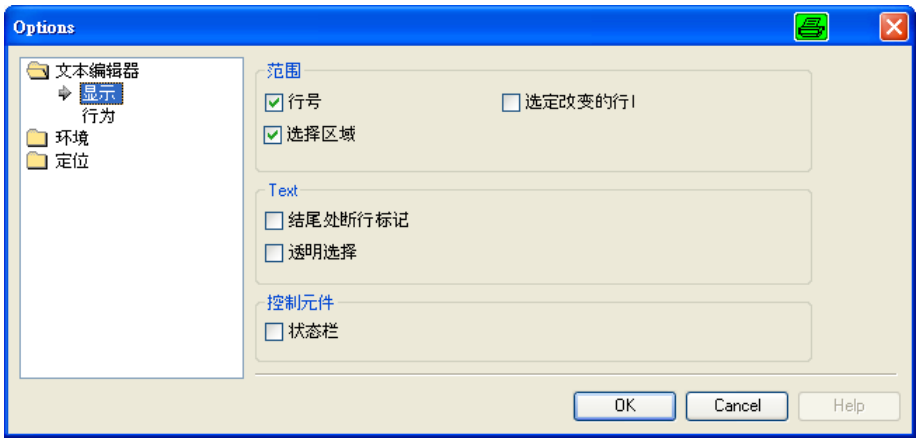


图 9-27: 外观

外观：

栏位	说明
行号	激活：显示行号。
选择区域	激活：选定的代码另外还在左侧用一条红色竖条标出。
选定改变的行	激活：在修改过的行的行首用黄色标记。
结尾处断行标记	仅当在**行为**下复选框**自动换行**激活时才相关。 ■ 激活：换行以一个绿色的小箭头标示。 ■ 关闭：换行不标示。
透明选择	显示选定的代码： ■ 激活：浅色背景上的橘色文字 ■ 关闭：深色背景上的白色文字
状态栏	激活：KRL 编辑器下部显示一条状态栏。例如：显示程序名和光标所在行的编号。

- 行为

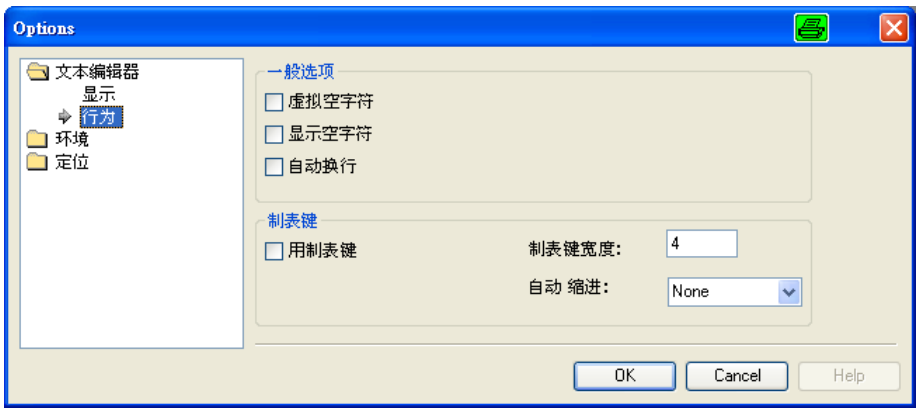


图 9-28: 行为

行为：

栏位	说明
虚拟空字符	■ 激活：在空行中光标可置于一个任意位置。 ■ 关闭：在空行中光标只能置于行首。
显示空字符	激活：显示控制符。（空格符、跳格符等等）

栏位	说明
自动换行	■ 激活：达到窗口宽度时换行。 ■ 关闭：不换行。如果有宽于窗口的行，将自动显示一个滚动条。
用跳格	■ 激活：用制表键插入跳格。 ■ 关闭：制表键插入在**跳格宽度**下定义的空格数量。
跳格宽度	一个跳格的宽度应等于 *x* 个空格。
自动缩进	用一个回车键生成一个新行时的行为： ■ **无**：新行不缩进。 ■ **两端对齐**：新行与上一行缩进量相同。 ■ **Smart**：Fold 内的行为 若上一行缩进，则缩进量应用于新行。 若上一行未缩进，则新行将缩进。

KRL 编辑器的颜色说明

■ 颜色说明

KRL 编辑器会识别输入代码的组成部分并自动用不同的颜色来表示。

代码组成部分	颜色
KRL 关键词 （除 ;FOLD 和 ;ENDFOLD 之外）	中蓝
;FOLD 和 ;ENDFOLD	灰色
数字	深蓝
字符串（文字在双引号 “...” 内）	红色
注释	绿色
特殊字符	蓝绿色
其它代码	黑色

■ 颜色应用示例

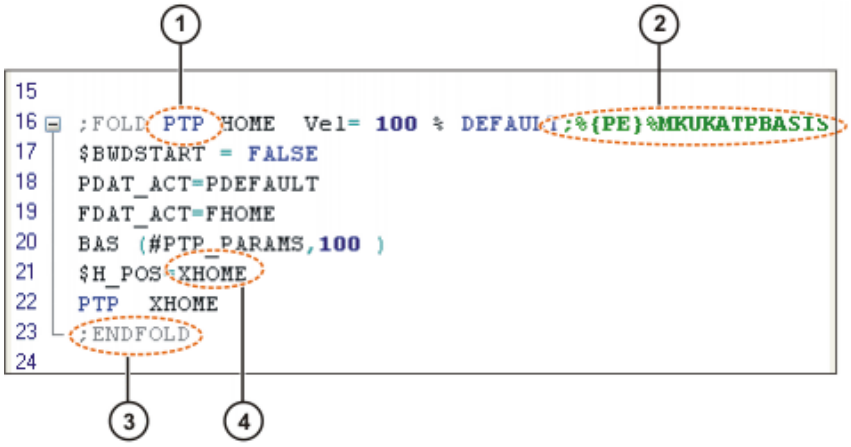


图 9-29: KRL 编辑器中的颜色举例

1	KRL 关键词：蓝色
2	注释：绿色
3	FOLD：灰色
4	其它代码：黑色

KRL 编辑器错误识别

■ KRL 编辑器具有自动识别错误的功能

■ 编程编码中识别出的错误用红色波纹下划线标出

- 只有当选择了类别 **KRL-Parser** 后，才能在信息窗口看到错误
- 可识别出 KRL 错误以及部分结构错误（在程序的错误位置声明）
- 而变量中的打字错误则无法识别

> 错误识别不能识别所有错误。即使没有用波纹下划线标出之处，也不能保证程序无错。

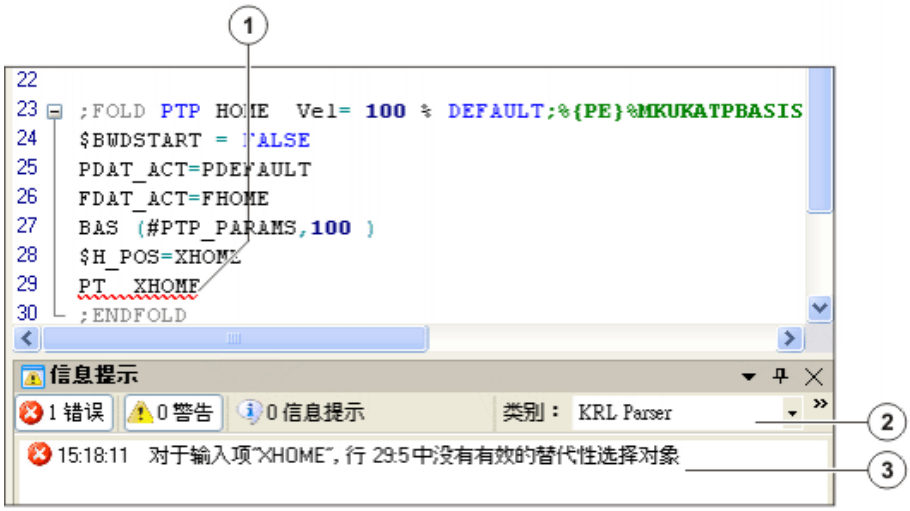


图 9-30: 错误识别示例

1	KRL 编辑器：错误用红色波纹下划线标出。
2	信息窗口：已选择了类别 **KRL Parser**。
3	信息窗口：含有行号和列号的错误描述

变量列表的功能

- 在某个特定文件中所声明的所有 KRL 变量都能清晰明了地显示在一个列表中。

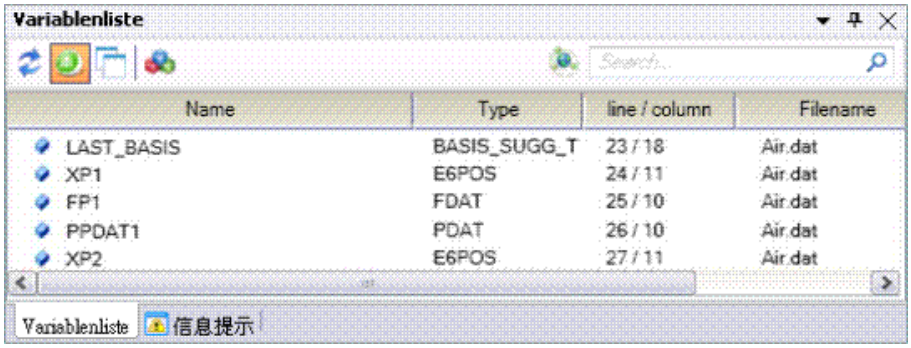


图 9-31: 变量表示例

- 对于 SRC 文件，也总是显示相关 DAT 文件的变量，反之亦然

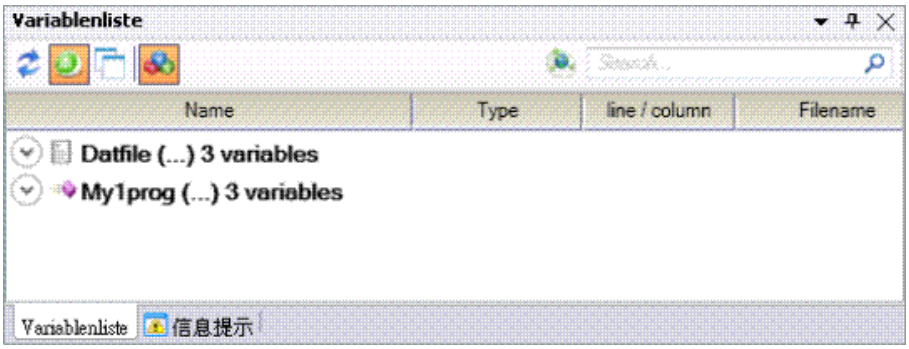


图 9-32: 变量表示例（SRC 和 DAT 变量）

- 在搜索栏内输入变量名或名称的一部分。查找结果将立即显示。

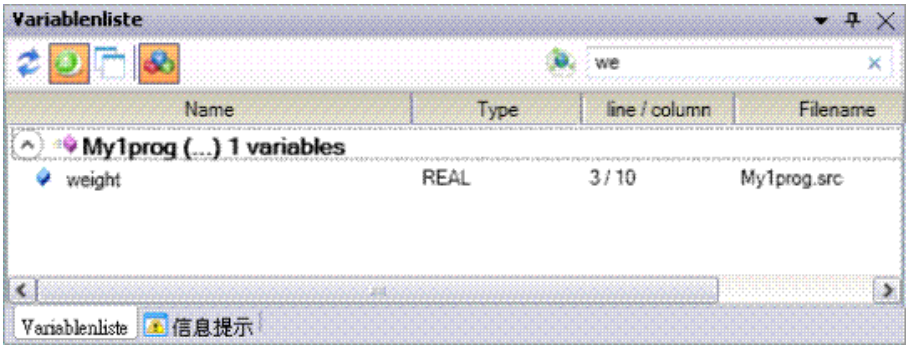


图 9-33: 变量表示例（搜索功能）

如果搜索栏内已输入了一些内容，则即使光标被放到其它文件上，这仍然适用。如果搜索栏是空的，则原则上只显示一个文件的所有变量。

- 可能的设置

图 9-34: 变量列表窗口

点击某一列时，列表可按该列排序。

按键	说明
	**重新读取**
	一旦结构树中选定的文件变动时，便更新列表
	当前编辑器变动时更新列表
	将变量根据局部子功能编组 (SRC/DAT)
	按键被按下：则查找涉及所有全局变量。

附加的编辑功能

常用编辑功能可在相关菜单中通过**编辑**调出。包括：

- **剪切、粘贴、复制、删除**
- **撤销、还原**
- **查找 …、替换 …**
- **定位 …**
- **全选**

在相关菜单的**扩展**下还有其它编辑功能可供选用：

扩展 > …	说明
跳格代替空格	在选定的范围以跳格代替空格。 前提条件：配置设定**用跳格**处于激活状态。
空格代替跳格	在选定的范围以空格代替跳格。
增大缩进	在光标位置插入空格或跳格。
减小缩进	删除光标位置左侧的跳格或空格。
添加注释	在选定行的行首打一个分号。
删除注释	删除选定行行首处的分号。
合上所有 Fold	关闭当前显示文件的所有 Fold。
展开所有 Fold	打开当前显示文件的所有 Fold。

操作步骤

KRL 指令的快速输入（KRL 片断）

必须编程设定一个中断声明。为了不必完整输入句法 `INTERRUPT DECL … WHEN … DO`，人们使用 KRL 片断。这样就只须在句法的变量位置手动填写。

使用代码片断编程时的操作步骤

1. 将光标置于所需位置。
   - 用右键点击并在相关菜单中选择选项**插入代码片断**。一个列表栏即显示出来。双击所需指令。

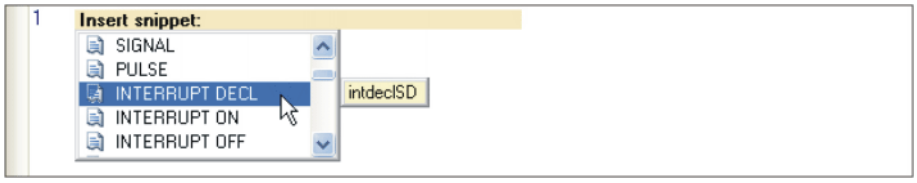


图 9-35: 双击指令

   - 或：输入缩写并按 TAB 键。

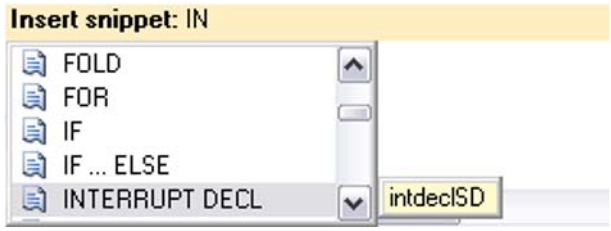


图 9-36: 带有搜索功能的 KRL 片断

2. KRL 句法自动贴入。第一个变量位具有红色背景。输入所需数值。

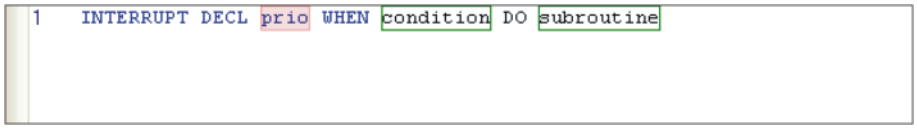


图 9-37: 第一个变量位具有红色背景

3. 用回车键跳到下一个变量位置。输入所需数值。

4. 对所有变量位置重复步骤 3。

KRL 编辑器：使用工具箱编程

激活工具箱

1. 菜单 **窗口 > 工具箱**
2. 定位或固定工具箱
3. 将程序载入 KRL 编辑器，工具箱自动添加功能

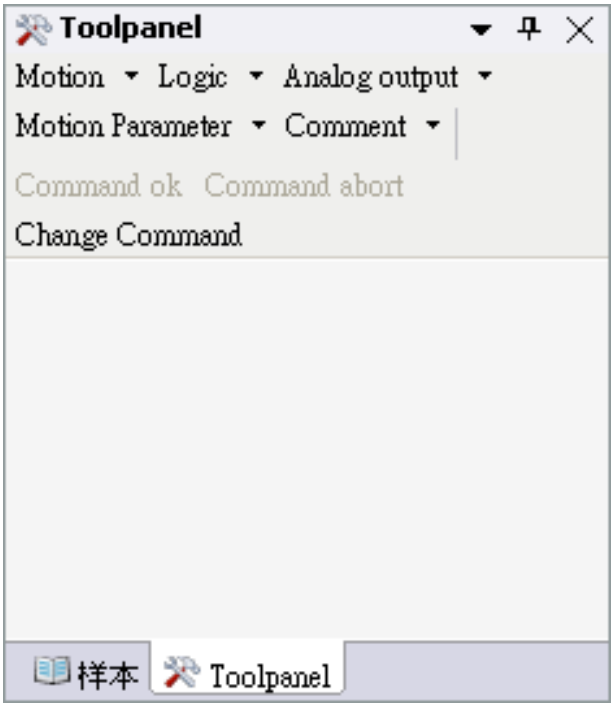


图 9-38: 填满的工具箱

使用工具箱编程时的操作步骤

1. 将光标置于所需位置。
2. 在工具箱中选择所要的联机表单

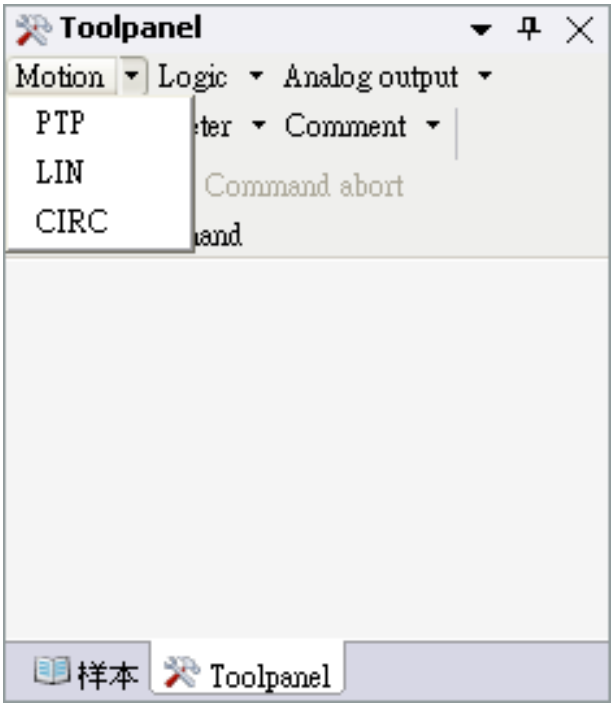


图 9-39: 运动工具箱

3. 编辑联机表单 / 设置参数

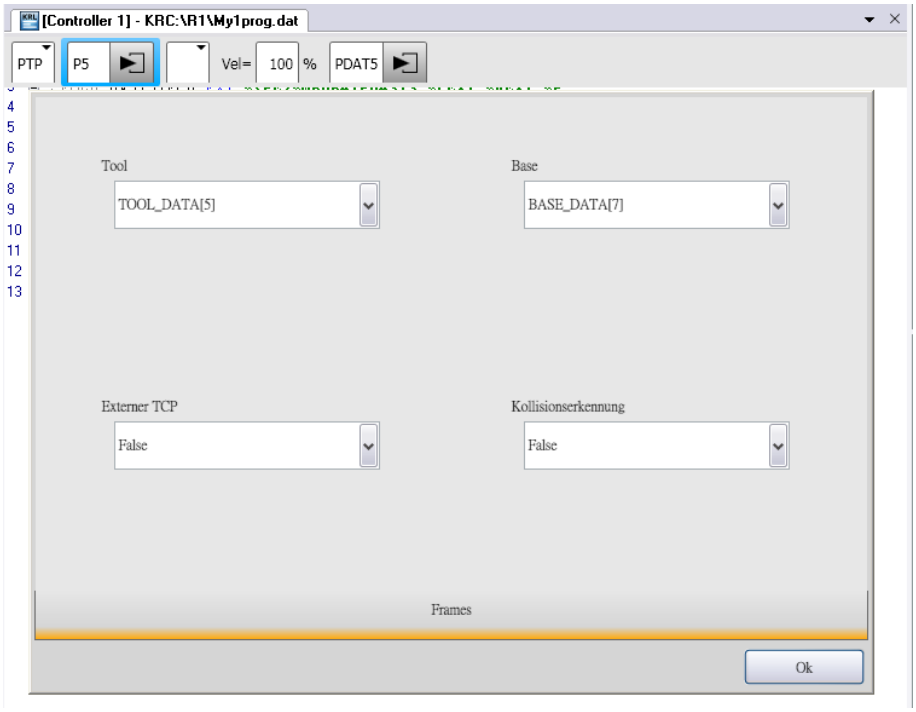


图 9-40: KRL 编辑器联机表单

4. 用工具箱上的 **OK 指令**键结束联机表单

索引