ABB Robotics

Application manual
Motion functions and events

Power and productivity
for a better world™

Application manual

Motion functions and events

RobotWare 5.14

Document ID: 3HAC 18152-1

Revision: F

The information in this manual is subject to change without notice and should not be construed as a commitment by ABB. ABB assumes no responsibility for any errors that may appear in this manual.

Except as may be expressly stated anywhere in this manual, nothing herein shall be construed as any kind of guarantee or warranty by ABB for losses, damages to persons or property, fitness for a specific purpose or the like.

In no event shall ABB be liable for incidental or consequential damages arising from use of this manual and products described herein.

Additional copies of this manual may be obtained from ABB at its then current charge.

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

Overview of this manual

About this manual

This manual explains the basics of when and how to use the following RobotWare base functionality and options:

- World Zones (608-1)
- Fixed Position Events
- Independent Axes (610-1)
- Path Recovery (611-1)
- Path Offset (612-1)

Usage

This manual can be used either as a reference to find out if a base functionality or option is the right choice for solving a problem, or as a description of how to use a base functionality or option. Detailed information regarding syntax for RAPID routines, and similar, is not described here, but can be found in the respective reference manual.

Who should read this manual?

This manual is mainly intended for robot programmers.

Prerequisites

The reader should...

- be familiar with industrial robots and their terminology
- be familiar with the RAPID programming language
- be familiar with system parameters and how to configure them.

Organization of chapters

The manual is organized in the following chapters:

Chapter	Contents
1.	Describes the option World Zones.
2.	Describes the base functionality Fixed Position Events.
3.	Describes the option Independent Axes.
4.	Describes the option Path Recovery.
5.	Describes the option Path Offset.

References

Reference	Document Id
Technical reference manual - RAPID overview	3HAC16580-1
Technical reference manual - RAPID Instructions, Functions and Data types	3HAC16581-1
Operating manual - IRC5 with FlexPendant	3HAC16590-1
Technical reference manual - System parameters	3HAC17076-1
Application manual - MultiMove	3HAC021272-001

Continues on next page

Continued

Revisions

Revision	Description
-	First edition
A	Minor corrections. Option changed name from Independent Movements to Independent Axes. Path recorder added to Path Recovery.
B	New instructions used for MultiMove systems.
C	Path Recovery: Corrected errors in code examples.
D	Fixed Position Events is now part of the RobotWare base functionality.
E	Path Recovery: `StorePath` and `RestoPath` is now included in RobotWare base.
F	Added instructions `TriggRampAO` and `TriggLIOs` and datatypes `triggios`, `triggiosdnum`, and `triggstrgo` to Fixed Position Events.

Product documentation, M2004

Categories for manipulator documentation

The manipulator documentation is divided into a number of categories. This listing is based on the type of information in the documents, regardless of whether the products are standard or optional.

All documents listed can be ordered from ABB on a DVD. The documents listed are valid for M2004 manipulator systems.

Product manuals

Manipulators, controllers, DressPack/SpotPack, and most other hardware will be delivered with a **Product manual** that generally contains:

- Safety information.
- Installation and commissioning (descriptions of mechanical installation or electrical connections).
- Maintenance (descriptions of all required preventive maintenance procedures including intervals and expected life time of parts).
- Repair (descriptions of all recommended repair procedures including spare parts).
- Calibration.
- Decommissioning.
- Reference information (safety standards, unit conversions, screw joints, lists of tools).
- Spare parts list with exploded views (or references to separate spare parts lists).
- Circuit diagrams (or references to circuit diagrams).

Technical reference manuals

The technical reference manuals describe the manipulator software in general and contain relevant reference information.

- **RAPID Overview**: An overview of the RAPID programming language.
- **RAPID Instructions, Functions and Data types**: Description and syntax for all RAPID instructions, functions, and data types.
- **RAPID Kernel**: A formal description of the RAPID programming language.
- **System parameters**: Description of system parameters and configuration workflows.

Continues on next page

Continued

Application manuals

Specific applications (for example software or hardware options) are described in **Application manuals**. An application manual can describe one or several applications.

An application manual generally contains information about:

- The purpose of the application (what it does and when it is useful).
- What is included (for example cables, I/O boards, RAPID instructions, system parameters, DVD with PC software).
- How to install included or required hardware.
- How to use the application.
- Examples of how to use the application.

Operating manuals

The operating manuals describe hands-on handling of the products. The manuals are aimed at those having first-hand operational contact with the product, that is production cell operators, programmers, and trouble shooters.

The group of manuals includes (among others):

- **Emergency safety information**
- **General safety information**
- **Getting started, IRC5 and RobotStudio**
- **Introduction to RAPID**
- **IRC5 with FlexPendant**
- **RobotStudio**
- **Trouble shooting**, for the controller and manipulator.

Safety

Safety of personnel

A robot is heavy and extremely powerful regardless of its speed. A pause or long stop in movement can be followed by a fast hazardous movement. Even if a pattern of movement is predicted, a change in operation can be triggered by an external signal resulting in an unexpected movement.

Therefore, it is important that all safety regulations are followed when entering safeguarded space.

Safety regulations

Before beginning work with the robot, make sure you are familiar with the safety regulations described in the manual *Operating manual - General safety information*.

1 World Zones (608-1)

1.1. Overview

Purpose

The purpose of World Zones is to stop the robot or set an output signal if the robot is inside a special user-defined zone. Here are some examples of applications:

- When two robots share a part of their respective work areas. The possibility of the two robots colliding can be safely eliminated by World Zones supervision.
- When a permanent obstacle or some temporary external equipment is located inside the robot's work area. A forbidden zone can be created to prevent the robot from colliding with this equipment.
- Indication that the robot is at a position where it is permissible to start program execution from a Programmable Logic Controller (PLC).

A world zone is supervised during robot movements both during program execution and jogging. If the robot's TCP reaches the world zone or if the axes reaches the world zone in joints, the movement is stopped or a digital output signal is set.

WARNING!

For safety reasons, this software shall not be used for protection of personnel. Use hardware protection equipment for that.

What is included

The RobotWare option World Zones gives you access to:

- instructions used to define volumes of various shapes
- instructions used to define joint zones in coordinates for axes
- instructions used to define and enable world zones

Basic approach

This is the general approach for setting up World Zones. For a more detailed example of how this is done, see *Code examples on page 14*.

1. Declare the world zone as stationary or temporary.
2. Declare the shape variable.
3. Define the shape that the world zone shall have.
4. Define the world zone (that the robot shall stop or that an output signal shall be set when reaching the volume).

Limitations

Supervision of a volume only works for the TCP. Any other part of the robot may pass through the volume undetected. To be certain to prevent this, you can supervise a joint world zone (defined by`WZLimJointDef` or `WZHomeJointDef`).

A variable of type `wzstationary` or `wztemporary` can not be redefined. They can only be defined once (with `WZLimSup` or `WZDOSet`).

1.2. RAPID components

Data types

This is a brief description of each data type in World Zones. For more information, see respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
wztemporary	`wztemporary` is used to identify a temporary world zone and can be used anywhere in the RAPID program. Temporary world zones can be disabled, enabled again, or erased via RAPID instructions. Temporary world zones are automatically erased when a new program is loaded or when program execution start from the beginning in the MAIN routine.
wzstationary	`wzstationary` is used to identify a stationary world zone and can only be used in an event routine connected to the event POWER ON. For information on defining event routines, see *Operating manual - IRC5 with FlexPendant*. A stationary world zone is always active and is reactivated by a warm start (switch power off then on, or change system parameters). It is not possible to disable, enable or erase a stationary world zone via RAPID instructions. Stationary world zones shall be used if security is involved.
shapedata	`shapedata` is used to describe the geometry of a world zone. World zones can be defined in 4 different geometrical shapes: • a straight box, with all sides parallel to the world coordinate system • a cylinder, parallel to the z axis of the world coordinate system • a sphere • a joint angle area for the robot axes and/or external axes

Instructions

This is a brief description of each instruction in World Zones. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
WZBoxDef	`WZBoxDef` is used to define a volume that has the shape of a straight box with all its sides parallel to the axes of the world coordinate system. The definition is stored in a variable of type `shapedata`. The volume can also be defined as the inverse of the box (all volume outside the box).
WZCylDef	`WZCylDef` is used to define a volume that has the shape of a cylinder with the cylinder axis parallel to the z-axis of the world coordinate system. The definition is stored in a variable of type `shapedata`. The volume can also be defined as the inverse of the cylinder (all volume outside the cylinder).
WZSphDef	`WZSphDef` is used to define a volume that has the shape of a sphere. The definition is stored in a variable of type `shapedata`. The volume can also be defined as the inverse of the sphere (all volume outside the sphere).

Continues on next page

Continued

Instruction	Description
WZLimJointDef	`WZLimJointDef` is used to define joint coordinate for axes, to be used for limitation of the working area. Coordinate limits can be set for both the robot axes and external axes. For each axis `WZLimJointDef` defines an upper and lower limit. For rotational axes the limits are given in degrees and for linear axes the limits are given in mm. The definition is stored in a variable of type `shapedata`.
WZHomeJointDef	`WZHomeJointDef` is used to define joint coordinates for axes, to be used to identify a position in the joint space. Coordinate limits can be set for both the robot axes and external axes. For each axis `WZHomeJointDef` defines a joint coordinate for the middle of the zone and the zones delta deviation from the middle. For rotational axes the coordinates are given in degrees and for linear axes the coordinates are given in mm. The definition is stored in a variable of type `shapedata`.
WZLimSup	`WZLimSup` is used to define, and enable, stopping the robot with an error message when the TCP reaches the world zone. This supervision is active both during program execution and when jogging. When calling `WZLimSup` you specify whether it is a stationary world zone, stored in a `wzstationary` variable, or a temporary world zone, stored in a `wztemporary` variable.
WZDOSet	`WZDOSet` is used to define, and enable, setting a digital output signal when the TCP reaches the world zone. When calling`WZDOSet` you specify whether it is a stationary world zone, stored in a `wzstationary` variable, or a temporary world zone, stored in a `wztemporary` variable.
WZDisable	`WZDisable` is used to disable the supervision of a temporary world zone.
WZEnable	`WZEnable` is used to re-enable the supervision of a temporary world zone. A world zone is automatically enabled on creation. Enabling is only necessary after it has been disabled with `WZDisable`.
WZFree	`WZFree` is used to disable and erase a temporary world zone.

Functions

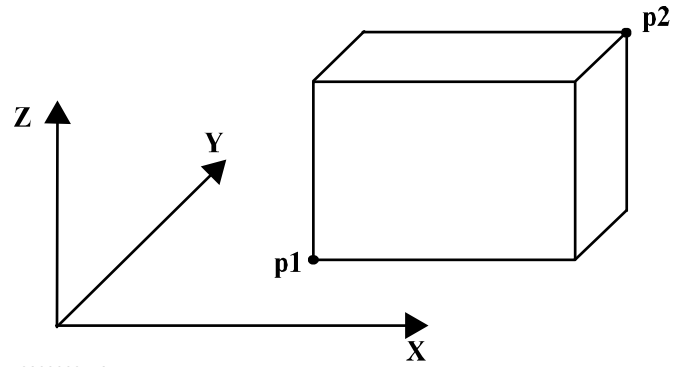
World Zones does not include any RAPID functions.

1.3. Code examples

Create protected box

To prevent the robot TCP from moving into stationary equipment, set up a stationary world zone around the equipment.

The routine `my_power_on` should then be connected to the event POWER ON. For information on how to do this, read about defining event routines in *Operating manual - IRC5 with FlexPendant*.



xx0300000178

```
VAR wzstationary obstacle;
PROC my_power_on()
  VAR shapedata volume;
  CONST pos p1 := [200, 100, 100];
  CONST pos p2 := [600, 400, 400];

  !Define a box between the corners p1 and p2
  WZBoxDef \Inside, volume, p1, p2;

  !Define and enable supervision of the box
  WZLimSup \Stat, obstacle, volume;
ENDPROC
```

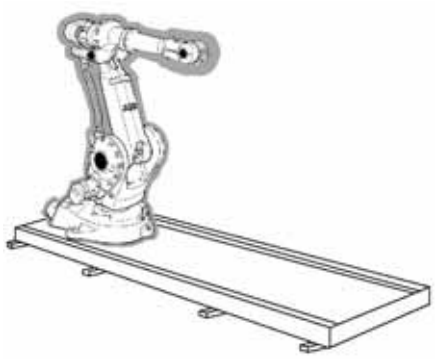

Continues on next page

Signal when robot is in position

When two robots share a work area it is important to know when a robot is out of the way, letting the other robot move freely.

This example defines a home position where the robot is in a safe position and sets an output signal when the robot is in its home position. The robot is standing on a travel track, handled as external axis 1. No other external axes are active.

The shadowed area in the illustration shows the world zone.



xx0300000206

```
VAR wztemporary home;
PROC zone_output()
  VAR shapedata joint_space;

  !Define the home position
  CONST jointtarget home_pos := [[0, -20, 0, 0, 0, 0], [0, 9E9,
       9E9, 9E9, 9E9, 9E9]];

  !Define accepted deviation from the home position
  CONST jointtarget delta_pos := [[2, 2, 2, 2, 2, 2], [10, 9E9,
       9E9, 9E9, 9E9, 9E9]];

  !Define the shape of the world zone
  WZHomeJointDef \Inside, joint_space, home_pos, delta_pos;

  !Define the world zone, setting  the
  !signal do_home to 1 when in zone
  WZDOSet \Temp, home \Inside, joint_space, do_home, 1;
ENDPROC
```

1.3. Code examples

2 Fixed Position Events

2.1. Overview

Purpose

The purpose of Fixed Position Events is to make sure a program routine is executed when the position of the TCP is well defined.

If a move instruction is called with the zone argument set to `fine`, the next routine is always executed once the TCP has reached its target. If a move instruction is called with the zone argument set to a distance (for example `z20`), the next routine may be executed before the TCP is even close to the target. This is because there is always a delay between the execution of RAPID instructions and the robot movements.

Calling the move instruction with zone set to `fine` will slow down the movements. With Fixed Position Events, a routine can be executed when the TCP is at a specified position anywhere on the TCP path without slowing down the movement.

What is included

The RobotWare base functionality Fixed Position Events gives you access to:

- instructions used to define a position event
- instructions for moving the robot and executing the position event at the same time
- instructions for moving the robot and calling a procedure while passing the target, without first defining a position event

Basic approach

Fixed Position Events can either be used with one simplified instruction calling a procedure or it can be set up following these general steps. For more detailed examples of how this is done, see *Code examples on page 21*.

1. Declare the position event.
2. Define the position event:
   - when it shall occur, compared to the target position
   - what it shall do
3. Call a move instruction that uses the position event. When the TCP is as close to the target as defined, the event will occur.

2.2. RAPID components and system parameters

Data types

This is a brief description of each data type in Fixed Position Events. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
triggdata	`triggdata` is used to store data about a position event. A position event can take the form of setting an output signal or running an interrupt routine at a specific position along the movement path of the robot. `triggdata` also contains information on when the action shall occur, for example when the TCP is at a defined distance from the target. `triggdata` is a non-value data type.
triggios	`triggios` is used to store data about a position event used by the instruction `TriggLIOs`. `triggios` sets the value of an output signal using a `num` value.
triggiosdnum	`triggiosdnum` is used to store data about a position event used by the instruction `TriggLIOs`. `triggiosdnum` sets the value of an output signal using a `dnum` value.
triggstrgo	`triggstrgo` is used to store data about a position event used by the instruction `TriggLIOs`. `triggstrgo` sets the value of an output signal using a `stringdig` value (string containing a number).

Instructions

This is a brief description of each instruction in Fixed Position Events. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
TriggIO	`TriggIO` defines the setting of an output signal and when to set that signal. The definition is stored in a variable of type `triggdata`. `TriggIO` can define the setting of the signal to occur at a certain distance (in mm) from the target, or a certain time from the target. It is also possible to set the signal at a defined distance or time from the starting position. By setting the distance to 0 (zero), the signal will be set when the TCP is as close to the target as it gets (the middle of the corner path).
TriggEquip	`TriggEquip` works like `TriggIO`, with the difference that `TriggEquip` can compensate for the internal delay of the external equipment. For example, the signal to a glue gun must be set a short time before the glue is pressed out and the gluing begins.
TriggInt	`TriggInt` defines when to run an interrupt routine. The definition is stored in a variable of type `triggdata`. `TriggInt` defines at what distance (in mm) from the target (or from the starting position) the interrupt routine shall be called. By setting the distance to 0 (zero), the interrupt will occur when the TCP is as close to the target as it gets (the middle of the corner path).

Continues on next page

Continued

Instruction	Description
TriggCheckIO	`TriggCheckIO` defines a test of an input or output signal, and when to perform that test. The definition is stored in a variable of type `triggdata`. `TriggCheckIO` defines a test, comparing an input or output signal with a value. If the test fails, an interrupt routine is called. As an option the robot movement can be stopped when the interrupt occurs. `TriggCheckIO` can define the test to occur at a certain distance (in mm) from the target, or a certain time from the target. It is also possible to perform the test at a defined distance or time from the starting position. By setting the distance to 0 (zero), the interrupt routine will be called when the TCP is as close to the target as it gets (the middle of the corner path).
TriggRampAO	TriggRampAO defines the ramping up or down of an analog output signal and when this ramping is performed. The definition is stored in a variable of type `triggdata`. `TriggRampIO` defines where the ramping of the signal is to start and the length of the ramping.
TriggL	`TriggL` is a move instruction, similar to `MoveL`. In addition to the movement the `TriggL` instruction can set output signals, run interrupt routines and check input or output signals at fixed positions. `TriggL` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggL`.
TriggC	`TriggC` is a move instruction, similar to `MoveC`. In addition to the movement the `TriggC` instruction can set output signals, run interrupt routines and check input or output signals at fixed positions. `TriggC` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggC`.
TriggJ	`TriggJ` is a move instruction, similar to `MoveJ`. In addition to the movement the `TriggJ` instruction can set output signals, run interrupt routines and check input or output signals at fixed positions. `TriggJ` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggJ`.
TriggLIOs	`TriggLIOs` is a move instruction, similar to `MoveL`. In addition to the movement the `TriggLIOs` instruction can set output signals at fixed positions. `TriggLIOs` is similar to the combination of `TriggEquip` and `TriggL`. The difference is that `TriggLIOs` can handle up to 50 position events stored as an array of datatype `triggios`, `triggiosdnum`, or `triggstrgo`.
MoveLSync	`MoveLSync` is a linear move instruction that calls a procedure in the middle of the corner path.
MoveCSync	`MoveCSync` is a circular move instruction that calls a procedure in the middle of the corner path.
MoveJSync	`MoveJSync` is a joint move instruction that calls a procedure in the middle of the corner path.

Functions

Fixed Position Events includes no RAPID functions.

Continues on next page

Continued

System parameters

This is a brief description of each parameter in Fixed Position Events. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Parameter	Description
Event Preset Time	`TriggEquip` takes advantage of the delay between the RAPID execution and the robot movement, which is about 70 ms. If the delay of the equipment is longer than 70 ms, then the delay of the robot movement can be increased by configuring *Event preset time*. *Event preset time* belongs to the type *Motion System* in the topic *Motion*.

2.3. Code examples

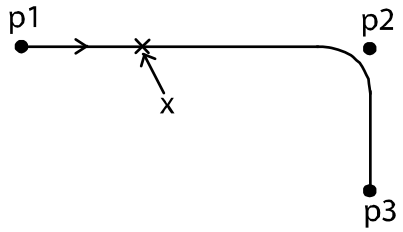
Example without Fixed Position Events

Without the use of Fixed Position Events, the code can look like this:

```
MoveJ p1, vmax, fine, tool1;
MoveL p2, v1000, z20, tool1;
SetDO do1, 1;
MoveL p3, v1000, fine, tool1;
```

Result

The code specifies that the TCP should reach `p2` before setting `do1`. Because the robot path is delayed compared to instruction execution, `do1` is set when the TCP is at the position marked with X (see illustration).



xx0300000151

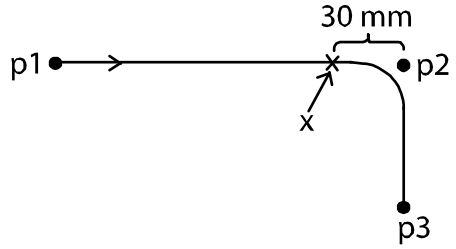
Example with TriggIO and TriggL instructions

Setting the output signal 30 mm from the target can be arranged by defining the position event and then moving the robot while the system is executing the position event.

```
VAR triggdata do_set;
!Define that  do1 shall be set  when 30 mm from target
TriggIO do_set, 30  \DOp:=do1, 1;
MoveJ p1, vmax, fine, tool1;
!Move to p2 and let system execute  do_set
TriggL p2, v1000, do_set, z20, tool1;
MoveL p3, v1000, fine, tool1;
```

Result

The signal `do1` will be set when the TCP is 30 mm from `p2`. `do1` is set when the TCP is at the position marked with X (see illustration).



xx0300000158

Continues on next page

2.3. Code examples

Continued

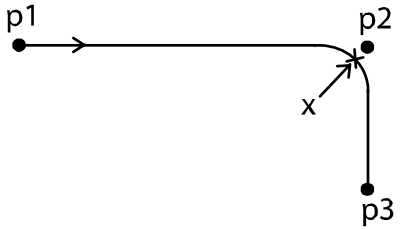
Example with MoveLSync instruction

Calling a procedure when the robot path is as close to the target as possible can be done with one instruction call.

```
MoveJ p1, vmax, fine, tool1;
!Move to p2 while calling a procedure
MoveLSync p2, v1000, z20, tool1, "proc1";
MoveL p3, v1000, fine, tool1;
```

Result

The procedure will be called when the TCP is at the position marked with X (see illustration).



xx0300000165

3 Independent Axes (610-1)

3.1. Overview

Purpose

The purpose of Independent Axes is to move an axis independently of other axes in the robot system. Some examples of applications are:

- Move an external axis holding an object (for example rotating an object while the robot is spray painting it).
- Save cycle time by performing a robot task at the same time as an external axis performs another.
- Continuously rotate robot axis 6 (for polishing or similar tasks).
- Reset the measurement system after an axis has rotated multiple revolutions in the same direction. Saves cycle time compared to physically winding back.

An axis can move independently if it is set to independent mode. An axis can be changed to independent mode and later back to normal mode again.

What is included

The RobotWare option Independent Axes gives you access to:

- instructions used to set independent mode and specify the movement for an axis
- an instruction for changing back to normal mode and/or reset the measurement system
- functions used to verify the status of an independent axis
- system parameters for configuration.

Basic approach

This is the general approach for moving an axis independently. For detailed examples of how this is done, see *Code examples on page 27*.

1. Call an independent move instruction to set the axis to independent mode and move it.
2. Let the robot execute another instruction at the same time as the independent axis moves.
3. When both robot and independent axis has stopped, reset the independent axis to normal mode.

Reset axis

Even without being in independent mode, an axis might rotate only in one direction and eventually loose precision. The measurement system can then be reset with the instruction `IndReset`.

The recommendation is to reset the measurement system for an axis before its motor has rotated 10 000 revolutions in the same direction.

Continues on next page

Continued

Limitations

A mechanical unit may not be deactivated when one of its axes is in independent mode.

Axes in independent mode cannot be jogged.

The only robot axis that can be used as an independent axis is axis number 6. On IRB 2400 and IRB 4400, the instruction `IndReset` can also be used for axis 4.

3.2. System parameters

About the system parameters

This is a brief description of each parameter in Independent Axes. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Arm

These parameters belongs to the type *Arm* in the topic *Motion*.

Parameter	Description
Independent Joint	Flag that determines if independent mode is allowed for the axis.
Independent Upper Joint Bound	Defines the upper limit of the working area for the joint when operating in independent mode.
Independent Lower Joint Bound	Defines the lower limit of the working area for the joint when operating in independent mode.

Transmission

These parameters belong to the type *Transmission* in the topic *Motion*.

Parameter	Description
Transmission Gear High	Independent Axes requires high resolution in transmission gear ratio, which is therefore defined as *Transmission Gear High* divided by *Transmission Gear Low*. If no smaller number can be used, the transmission gear ratio will be correct if *Transmission Gear High* is set to the number of cogs on the robot axis side, and *Transmission Gear Low* is set to the number of cogs on the motor side.
Transmission Gear Low	See *Transmission Gear High*.

3.3. RAPID components

Data types

There are no data types for Independent Axes.

Instructions

This is a brief description of each instruction in Independent Axes. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

An independent move instruction is executed immediately, even if the axis is being moved at the time. If a new independent move instruction is executed before the last one is finished, the new instruction immediately overrides the old one.

Instruction	Description
IndAMove	`IndAMove` (Independent Absolute position Movement) change an axis to independent mode and move the axis to a specified position.
IndCMove	`IndCMove` (Independent Continuous Movement) change an axis to independent mode and start moving the axis continuously at a specified speed.
IndDMove	`IndDMove` (Independent Delta position Movement) change an axis to independent mode and move the axis a specified distance.
IndRMove	`IndRMove` (Independent Relative position Movement) change a rotational axis to independent mode and move the axis to a specific position within one revolution. Because the revolution information in the position is omitted, `IndRMove` never rotates more than one axis revolution.
IndReset	`IndReset` is used to change an independent axis back to normal mode. `IndReset` can move the measurement system for a rotational axis a number of axis revolutions. The resolution of positions is decreased when moving away from logical position 0, and winding the axis back would take time. By moving the measurement system the resolution is maintained without physically winding the axis back. Both the independent axis and the robot must stand still when calling `IndReset`.

Functions

This is a brief description of each function in Independent Axes. For more information, see respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
IndInpos	`IndInpos`indicates whether an axis has reached the selected position.
IndSpeed	`IndSpeed` indicates whether an axis has reached the selected speed.

3.4. Code examples

Save cycle time

An object in station A needs welding in two places. The external axis for station A can turn the object in position for the second welding while the robot is welding on another object. This saves cycle time compared to letting the robot wait while the external axis moves.

```
!Perform first welding in station A
!Call subroutine for welding
weld_stationA_1;

!Move the object in station A, axis 1, with
!independent movement to position 90 degrees
!at the speed 20 degrees/second
IndAMove Station_A,1\ToAbsNum:=90,20;

!Let the robot perform another task while waiting
!Call subroutine for welding
weld_stationB_1;

!Wait until the independent axis is in position
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;

!Perform second welding in station A
!Call subroutine for welding
weld_stationA_2;
```


Continues on next page

Continued

Polish by rotating axis 6

To polish an object the robot axis 6 can be set to continuously rotate.

Set robot axis 6 to independent mode and continuously rotate it. Move the robot over the area you want to polish. Stop movement for both robot and independent axis before changing back to normal mode. After rotating the axis many revolutions, reset the measurement system to maintain the resolution.

Note that, for this example to work, the parameter *Independent Joint* for rob1_6 must be set to Yes.

```
PROC Polish()
  !Change axis 6 of ROB_1 to independent mode and
  !rotate it with 180 degrees/second
  IndCMove ROB_1, 6, 180;

  !Wait until axis 6 is up to speed
  WaitUntil IndSpeed(ROB_1,6\InSpeed);
  WaitTime 0.2;

  !Move robot where you want to polish
  MoveL p1,v10, z50, tool1;
  MoveL p2,v10, fine, tool1;

  !Stop axis 6 and wait until it's still
  IndCMove ROB_1, 6, 0;
  WaitUntil IndSpeed(ROB_1,6\ZeroSpeed);
  WaitTime 0.2;

  !Change axis 6 back to normal mode and
  !reset measurement system (close to 0)
  IndReset ROB_1, 6 \RefNum:=0 \Short;
ENDPROC
```

Reset an axis

This is an example of how to reset the measurement system for axis 1 in station A. The measurement system will change a whole number of revolutions, so it is close to zero (± 180°).

```
IndReset Station_A, 1 \RefNum:=0 \Short;
```

4 Path Recovery (611-1)

4.1. Overview

Purpose

Path Recovery is used to store the current movement path, perform some robot movements and then restore the interrupted path. This is useful when an error or interrupt occurs during the path movement. An error handler or interrupt routine can perform a task and then recreate the path.

For applications like arc welding and gluing, it is important to continue the work from the point where the robot left off. If the robot started over from the beginning, then the work piece would have to be scrapped.

If a process error occurs when the robot is inside a work piece, moving the robot straight out might cause a collision. By using the path recorder, the robot can instead move out along the same path it came in.

What is included

The RobotWare option Path Recovery gives you access to:

- instructions to suspend and resume the coordinated synchronized movement mode on the error or interrupt level.
- a path recorder, with the ability to move the TCP out from a position along the same path it came.

Limitations

The instructions `StorePath` and `RestoPath` only handles movement path data. The stop position must also be stored.

Movements using the path recorder has to be performed on trap-level, i.e. `StorePath` has to be executed prior to `PathRecMoveBwd`.

4.2. RAPID components

Data types

This is a brief description of each data type in Path Recovery. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
pathrecid	`pathrecid` is used to identify a breakpoint for the path recorder.

Instructions

This is a brief description of each instruction in Path Recovery. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
StorePath	`StorePath` is used to store the movement path being executed when an error or interrupt occurs. `StorePath` is included in RobotWare base.
RestoPath	`RestoPath` is used to restore the path that was stored by `StorePath`. `RestoPath` is included in RobotWare base.
PathRecStart	`PathRecStart` is used to start recording the robot's path. The path recorder will store path information during execution of the robot program.
PathRecStop	`PathRecStop` is used to stop recording the robot's path.
PathRecMoveBwd	`PathRecMoveBwd` is used to move the robot backwards along a recorded path.
PathRecMoveFwd	`PathRecMoveFwd` is used to move the robot back to the position where `PathRecMoveBwd` was executed. It is also possible to move the robot partly forward by supplying an identifier that has been passed during the backward movement.
SyncMoveSuspend	`SyncMoveSuspend` is used to suspend synchronized movements mode and set the system to independent movement mode.
SyncMoveResume	`SyncmoveResume` is used to go back to synchronized movements from independent movement mode.

Functions

This is a brief description of each function in Path Recovery. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
PathRecValidBwd	`PathRecValidBwd` is used to check if the path recorder is active and if a recorded backward path is available.
PathRecValidFwd	`PathRecValidFwd` is used to check if the path recorder can be used to move forward. The ability to move forward with the path recorder implies that the path recorder must have been ordered to move backwards earlier.

4.3. Store current path

Why store the path?

The simplest way to use Path Recovery is to only store the current path to be able to restore it after resolving an error or similar action.

Let's say that an error occur during arc welding. To resolve the error the robot might have to be moved away from the part. When the error is resolved, the welding should be continued from the point it left off. This is solved by storing the path information and the position of the robot before moving away from the path. The path can then be restored and the welding resumed after the error has been handled.

Basic approach

This is the general approach for storing the current path:

1. At the start of an error handler or interrupt routine:
   A. stop the movement
   B. store the movement path
   C. store the stop position
2. At the end of the error handler or interrupt routine:
   A. move to the stored stop position
   B. restore the movement path
   C. start the movement

Example

This is an example of how to use Path Recovery in error handling. First the path and position is stored, the error is corrected and then the robot is moved back in position and the path is restored.

```
MoveL p100, v100, z10, gun1;
...
ERROR
  IF ERRNO=MY_GUN_ERR THEN
     gun_cleaning();
  ENDIF
...
PROC gun_cleaning()
  VAR robtarget p1;

  !Stop the robot movement, if not already stopped.
  StopMove;

  !Store the movement path and current position
  StorePath;
  p1 := CRobT(\Tool:=gun1\WObj:=wobj0);
```

Continues on next page

Continued

```
  !Correct the error
  MoveL pclean, v100, fine, gun1;
  ...
  !Move the robot back to the stored position
  MoveL p1, v100, fine, gun1;

  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
```

Store path in a MultiMove system

In a MultiMove system the robots can keep the synchronized movement mode after `StorePath` with the argument `KeepSync`. However the robots can't switch from independent mode to synchronized mode, only the other way around.

After a Multimove system is set with the argument `KeepSync`, the system can change between synchronized, semi coordinated and independent mode on the `StorePath` level. The changes are made with the instructions `SyncMoveResume` and `SyncMoveSuspend`.

"SyncArc" example with coordinated synchronized movement

This is an example on how to use Path Recovery and keep synchronized mode in the error handler for a MultiMove system. Two robots perform arc welding on the same work piece. To make the example simple and general, we use move instructions instead of weld instructions. The work object is rotated by a positioner. For more information on the SyncArc example, see *Application manual - MultiMove*.

T_ROB1 task program

```
MODULE module1
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
PERS tasks all_tasks{3} := [["T_ROB1"],["T_ROB2"],["T_STN1"]];
PERS wobjdata wobj_stn1 := [ FALSE, FALSE, "STN_1", [ [0, 0, 0],
     [1, 0, 0 ,0] ], [ [0, 0, 250], [1, 0, 0, 0] ] ];
TASK PERS tooldata tool1 := ...
CONST robtarget p100 := ...
CONST robtarget p199 := ...
PROC main()
  ...
  SyncMove;
ENDPROC

PROC SyncMove()
  MoveJ p100, v1000, z50, tool1;
  WaitSyncTask sync1, all_tasks;
  MoveL p101, v500, fine, tool1;
  SyncMoveOn sync2, all_tasks;
```

Continues on next page

Continued

```
  MoveL p102\ID:=10, v300, z10, tool1 \WObj:=wobj_stn1;
  MoveC p103, p104\ID:=20, v300, z10, tool1 \WObj:=wobj_stn1;
  MoveL p105\ID:=30, v300, z10, tool1 \WObj:=wobj_stn1;
  MoveC p106, p101\ID:=40, v300, fine, tool1 \WObj:=wobj_stn1;
  SyncMoveOff sync3;
  MoveL p199, v1000, fine, tool1;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
    gun_cleaning();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC

PROC gun_cleaning()
  VAR robtarget p1;
  !Store the movement path and current position
  ! and keep syncronized mode.
  StorePath \KeepSync;
  p1 := CRobT(\Tool:=tool1 \WObj:=wobj_stn1);
  !Correct the error
  MoveL pclean1 \ID:=50, v100, fine, tool1 \WObj:=wobj_stn1;
  ...
  !Move the robot back to the stored position
  MoveL p1 \ID:=60, v100, fine, tool1 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE
```

T_ROB2 task program

```
MODULE module2
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
PERS tasks all_tasks{3};
PERS wobjdata wobj_stn1;
TASK PERS tooldata tool2 := ...
CONST robtarget p200 := ...
CONST robtarget p299 := ...
PROC main()
  ...
  SyncMove;
ENDPROC
PROC SyncMove()
  MoveJ p200, v1000, z50, tool2;
```

Continues on next page

Continued

```
  WaitSyncTask sync1, all_tasks;
  MoveL p201, v500, fine, tool2;
  SyncMoveOn sync2, all_tasks;
  MoveL p202\ID:=10, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveC p203, p204\ID:=20, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveL p205\ID:=30, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveC p206, p201\ID:=40, v300, fine, tool2 \WObj:=wobj_stn1;
  SyncMoveOff sync3;
  MoveL p299, v1000, fine, tool2;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
     gun_cleaning();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC
PROC gun_cleaning()
  VAR robtarget p2;
  !Store the movement path and current position.
  StorePath \KeepSync;
  p2 := CRobT(\Tool:=tool2 \WObj:=wobj_stn1);
  !Correct the error
  MoveL pclean2 \ID:=50, v100, fine, tool2 \WObj:=wobj_stn1;
  ...
  !Move the robot back to the stored position.
  MoveL p2 \ID:=60, v100, fine, tool2 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE
```

T_STN1 task program

```
MODULE module3
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3};
  CONST jointtarget angle_neg20 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9,
       9E9], [ -20, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  ...
  CONST jointtarget angle_340 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9],[
       340, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  PROC main()
  ...
     SyncMove;
  ...
```

Continues on next page

Continued

```
  ENDPROC
  PROC SyncMove()
    MoveExtJ angle_neg20, vrot50, fine;
    WaitSyncTask sync1, all_tasks;
    ! Wait for the robots
    SyncMoveOn sync2, all_tasks;
    MoveExtJ angle_20\ID:=10, vrot100, z10;
    MoveExtJ angle_160\ID:=20, vrot100, z10;
    MoveExtJ angle_200\ID:=30, vrot100, z10;
    MoveExtJ angle_340\ID:=40, vrot100, fine;
    SyncMoveOff sync3;
  ERROR
    IF ERRNO = ERR_PATH_STOP THEN
      gun_cleaning();
    ENDIF
  UNDO
    SyncMoveUndo;
  ENDPROC
  PROC gun_cleaning()
    VAR jointtarget resume_angle;
    !Store the movement path and current angle.
    StorePath \KeepSync;
    resume_angle := CJointT();
    !Correct the error
    MoveExtJ clean_angle \ID:=50, vrot100, fine;
    ...
    !Move the robot back to the stored position.
    MoveExtJ resume_angle \ID:=60, vrot100, fine;
    !Restore the path and start the movement
    RestoPath;
    StartMove;
    RETRY;
  ENDPROC
ENDMODULE
```


Continues on next page

Continued

Suspend and resume synchronized movements in the “SyncArc” example

`SyncMoveSuspend` is used to suspend synchronized movements mode and set the system to independent or semi coordinated movement mode.

`SyncMoveResume` is used to go back once more to synchronized movements.

These instructions can only be used after `StorePath\KeepSync` has been executed.

T_ROB1

```
PROC gun_cleaning()
  VAR robtarget p1;
  !Store the movement path and current position
  ! and keep syncronized mode.
  StorePath \KeepSync;
  p1 := CRobT(\Tool:=tool1 \WObj:=wobj_stn1);
  !Move in synchronized motion mode
  MoveL p104 \ID:=50, v100, fine, tool1 \WObj:=wobj_stn1;
  SyncMoveSuspend;
  !Move in independent mode
  MoveL pclean1, v100, fine, tool1;
  ...
  !Move the robot back to the stored position
  SyncMoveResume;
  MoveL p1 \ID:=60, v100, fine, tool1 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
```

T_ROB2

```
PROC gun_cleaning()
  VAR robtarget p2;
  !Store the movement path and current position.
  StorePath \KeepSync;
  p2 := CRobT(\Tool:=tool2 \WObj:=wobj_stn1);
  !Move in synchronized motion mode
  MoveL p104 \ID:=50, v100, fine, tool2 \WObj:=wobj_stn1;
  SyncMoveSuspend;
  !Move in independent mode
  MoveL pclean2 v100, fine, tool2;
  ...
  !Move the robot back to the stored position.
  SyncMoveResume;
  !Move in synchronized motion mode
  MoveL p2 \ID:=60, v100, fine, tool2 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
```

Continues on next page

Continued

```
ENDPROC
```

T_STN1

```
PROC gun_cleaning()
  VAR jointtarget resume_angle;
  !Store the movement path and current angle.
  StorePath \KeepSync;
  resume_angle := CJointT();
  !Move in synchronized motion mode
  MoveExtJ p1clean_angle \ID:=50, vrot100, fine;
  SyncMoveSuspend;
  ! Move in independent mode
  MoveExtJ p2clean_angle,vrot, fine;
  ...
  !Move the robot back to the stored position.
  SyncMoveResume;
  ! Move in synchronized motion mode
  MoveExtJ resume_angle \ID:=60, vrot100, fine;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
```

4.4. Path recorder

What is the path recorder

The path recorder can memorize a number of move instructions. This memory can then be used to move the robot backwards along that same path.

How to use the path recorder

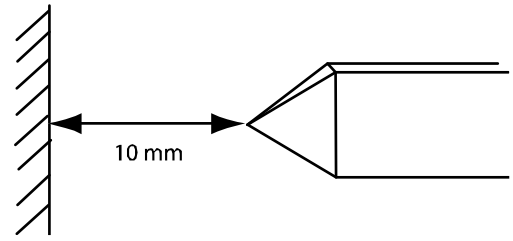
This is the general approach for using the path recorder:

1. Start the path recorder
2. Move the robot with regular move, or process, instructions
3. Store the current path
4. Move backwards along the recorded path
5. Resolve the error
6. Move forward along the recorded path
7. Restore the interrupted path

Lift the tool

When the robot moves backward in its own track, you may want to avoid scraping the tool against the work piece. For a process like arc welding, you want to stay clear of the welding seam.

By using the argument `ToolOffs` in the instructions `PathRecMoveBwd` and `PathRecMoveFwd`, you can set an offset for the TCP. This offset is set in tool coordinates, which means that if it is set to [0,0,10] the tool will be 10 mm from the work object when it moves back along the recorded path.



xx0400000828

NOTE!

When a MultiMove system is in synchronized mode all tasks must use `ToolOffs` if a tool is going to be lifted.

However if you only want to lift one tool, set `ToolOffs=[0,0,0]` in the other tasks.

Continues on next page

Continued

Simple example

If an error occurs between p1 and p4, the robot will return to p1 where the error can be resolved. When the error has been resolved, the robot continues from where the error occurred.

When p4 is reached without any error, the path recorder is switched off. The robot then moves from p4 to p5 without the path recorder.

```
...
VAR pathrecid start_id;
...
  MoveL p1, vmax, fine, tool1;
  PathRecStart start_id;
  MoveL p2, vmax, z50, tool1;
  MoveL p3, vmax, z50, tool1;
  MoveL p4, vmax, fine, tool1;
  PathRecStop \Clear;
  MoveL p5, vmax, fine, tool1;

ERROR
  StorePath;
  PathRecMoveBwd;
  ! Fix the problem
  PathRecMoveFwd;
  RestoPath;
  StartMove;
  RETRY;
ENDIF
...
```

Complex example

In this example, the path recorder is used for two purposes:

- If an error occurs, the operator can choose to back up to p1 or to p2. When the error has been resolved, the interrupted movement is resumed.
- Even if no error occurs, the path recorder is used to move the robot from p4 to p1. This technique is useful when the robot is in a narrow position that is difficult to move out of.

Note that if an error occurs during the first move instruction, between p1 and p2, it is not possible to go backwards to p2. If the operator choose to go back to p2, `PathRecValidBwd` is used to see if it is possible. Before the robot is moved forward to the position where it was interrupted, `PathRecValidFwd` is used to see if it is possible (if the robot never backed up it is already in position).

Continues on next page

Continued

```
...
VAR pathrecid origin_id;
VAR pathrecid corner_id;
VAR num choice;
...
  MoveJ p1, vmax, z50, tool1;
  PathRecStart origin_id;
  MoveJ p2, vmax, z50, tool1;
  PathRecStart corner_id;
  MoveL p3, vmax, z50, tool1;
  MoveL p4, vmax, fine, tool1;

  ! Use path record to move safely to p1
  StorePath;
  PathRecMoveBwd \ID:=origin_id
       \ToolOffs:=[0,0,10];
  RestoPath;
  PathRecStop \Clear;
  Clear Path;
  Start Move;

ERROR
  StorePath;

  ! Ask operator how far to back up
  TPReadFK choice,"Extract to:", stEmpty, stEmpty,
       stEmpty, "Origin", "Corner";

  IF choice=4 THEN
     ! Back up to p1
     PathRecMoveBwd \ID:=origin_id
          \ToolOffs:=[0,0,10];
  ELSEIF choice=5 THEN
     ! Verify that it is possible to back to p2,
     IF PathRecValidBwd(\ID:=corner_id) THEN
        ! Back up to p2
        PathRecMoveBwd \ID:=corner_id
             \ToolOffs:=[0,0,10];
     ENDIF
  ENDIF

  ! Fix the problem

  ! Verify that there is a path record forward
  IF PathRecValidFwd() THEN
     ! Return to where the path was interrupted
     PathRecMoveFwd \ToolOffs:=[0,0,10];
```

Continues on next page

Continued

```
ENDIF

! Restore the path and resume movement
RestoPath;
StartMove;
RETRY;
...
```

Resume path recorder

If the path recorder is stopped, it can be started again from the same position without loosing its history.

In the example below, the `PathRecMoveBwd` instruction will back the robot to p1. If the robot had been in any other position than p2 when the path recorder was restarted, this would not have been possible.

For more information, see the section about `PathRecStop` in *Technical reference manual - RAPID Instructions, Functions and Data types*.

```
...
MoveL p1, vmax, z50, tool1;
PathRecStart id1;
MoveL p2, vmax, z50, tool1;
PathRecStop;
MoveL p3, vmax, z50, tool1;
MoveL p4, vmax, z50, tool1;
MoveL p2, vmax, z50, tool1;
PathRecStart id2;
MoveL p5, vmax, z50, tool1;
StorePath;
PathRecMoveBwd \ID:=id1;
RestoPath;
...
```


Continues on next page

Continued

"SyncArc" example with coordinated synchronized movement

This is an example on how to use Path Recorder in error handling for a MultiMove system.

In this example two robots perform arc welding on the same work piece. To make the example simple and general, we use move instructions instead of weld instructions. The work object is rotated by a positioner.

For more information on the SyncArc example, see *Application manual - MultiMove*.

T_ROB1 task program

```
MODULE module1
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3} := [["T_ROB1"],["T_ROB2"],["T_STN1"]];
  PERS wobjdata wobj_stn1 := [ FALSE, FALSE, "STN_1",[ [0, 0, 0],
       [1, 0, 0 ,0] ], [ [0, 0,250], [1, 0, 0, 0] ] ];
  TASK PERS tooldata tool1 := ...
  CONST robtarget p100 := ...
  CONST robtarget p199 := ...
  PROC main()
     ...
     SyncMove;
  ENDPROC

  PROC SyncMove()
     WaitSyncTask sync1, all_tasks;
     MoveJ p100, v1000, z50, tool1;
     ! Start recording
     PathRecStart HomeROB1;
     MoveL p101, v500, fine, tool1;
     SyncMoveOn sync2, all_tasks;
     MoveL p102\ID:=10, v300, z10, tool1 \WObj:=wobj_stn1;
     MoveC p103, p104\ID:=20, v300, z10, tool1 \WObj:=wobj_stn1;
     MoveL p105\ID:=30, v300, z10, tool1 \WObj:=wobj_stn1;
     MoveC p106, p101\ID:=40, v300, fine, tool1 \WObj:=wobj_stn1;
     !Stop recording
     PathRecStop \Clear;
     SyncMoveOff sync3;
     MoveL p199, v1000, fine, tool1;
  ERROR
     ! Weld error in this program task
     IF ERRNO = AW_WELD_ERR THEN
        gun_cleaning();
     ENDIF
  UNDO
     SyncMoveUndo;
  ENDPROC
```

Continues on next page

Continued

```
  PROC gun_cleaning()
    VAR robtarget p1;
    !Store the movement path
    IF IsSyncMoveOn() THEN
      StorePath \KeepSync;
    ELSE
      StorePath;
    ENDIF
    !Move this robot backward to p100.
    PathRecMoveBwd \ID:=HomeROB1 \ToolOffs:=[0,0,10];
    !Correct the error
    MoveJ pclean1 ,v100, fine, tool1;
    ...
    !Move the robot back to p100
    MoveJ p100, v100, fine, tool1;
    PathRecMoveFwd \ToolOffs:=[0,0,10];
    !Restore the path and start the movement
    RestoPath;
    StartMove;
    RETRY;
  ENDPROC
ENDMODULE
```

T_ROB2 task program

```
MODULE module2
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3};
  PERS wobjdata wobj_stn1;
  TASK PERS tooldata tool2 := ...
  CONST robtarget p200 := ...
  CONST robtarget p299 := ...
  PROC main()
    ...
    SyncMove;
  ENDPROC
  PROC SyncMove()
    WaitSyncTask sync1, all_tasks;
    MoveJ p200, v1000, z50, tool2;
    PathRecStart HomeROB2;
    MoveL p201, v500, fine, tool2;
    SyncMoveOn sync2, all_tasks;
    MoveL p202\ID:=10, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveC p203, p204\ID:=20, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p205\ID:=30, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveC p206, p201\ID:=40, v300, fine, tool2 \WObj:=wobj_stn1;
    PathRecStop \Clear;
```

Continues on next page

Continued

```
      SyncMoveOff sync3;
      MoveL p299, v1000, fine, tool2;
    ERROR
      IF ERRNO = ERR_PATH_STOP THEN
        gun_move_out();
      ENDIF
    UNDO
      SyncMoveUndo;
    ENDPROC
    PROC gun_move_out()
      IF IsSyncMoveOn() THEN
        StorePath \KeepSync;
      ELSE
        StorePath;
      ENDIF
      ! Move this robot backward to p201
      PathRecMoveBwd \ToolOffs:=[0,0,10];
      ! Wait for the other gun to get clean
      PathRecMoveFwd \ToolOffs:=[0,0,10];
      !Restore the path and start the movement
      RestoPath;
      StartMove;
      RETRY;
    ENDPROC
  ENDMODULE
```

T_STN1 task program

```
  MODULE module3
    VAR syncident sync1;
    VAR syncident sync2;
    VAR syncident sync3;
    PERS tasks all_tasks{3};
    CONST jointtarget angle_neg20 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9,
          9E9], [ -20, 9E9, 9E9, 9E9, 9E9, 9E9] ];
    ...
    CONST jointtarget angle_340 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9],[
          340, 9E9, 9E9, 9E9,9E9, 9E9] ];
    PROC main()
      ...
      SyncMove;
      ...
    ENDPROC
    PROC SyncMove()
      WaitSyncTask sync1, all_tasks;
      MoveExtJ angle_neg20, vrot50, fine;
      PathRecStart HomeSTN1;
      SyncMoveOn sync2, all_tasks;
      MoveExtJ angle_20\ID:=10, vrot100, z10;
```

Continues on next page

Continued

```
    MoveExtJ angle_160\ID:=20, vrot100, z10;
    MoveExtJ angle_200\ID:=30, vrot100, z10;
    MoveExtJ angle_340\ID:=40, vrot100, fine;
    PathRecStop \Clear;
    SyncMoveOff sync3;
  ERROR
    IF ERRNO = ERR_PATH_STOP THEN
      gun_move_out();
    ENDIF
  UNDO
    SyncMoveUndo;
  ENDPROC
  PROC gun_move_out()
    !Store the movement
    IF IsSyncMoveOn() THEN
      StorePath \KeepSync;
    ELSE
      StorePath;
    ENDIF
    !Move the manipulator backward to angle_neg 20
    PathRecMoveBwd \ToolOffs:=[0,0,0];
    ...
    !Wait for the gun to get clean
    PathRecMoveFwd \ToolOffs:=[0,0,0];
    RestoPath;
    StartMove;
    RETRY;
ENDPROC
```

4.4. Path recorder

5 Path Offset (612-1)

5.1. Overview

Purpose

The purpose of Path Offset is to be able to make online adjustments of the robot path according to input from sensors. With the set of instructions that Path Offset offers, the robot path can be compared and adjusted with the input from sensors.

What is included

The RobotWare option Path Offset gives you access to:

- the data type `corrdescr`
- the instructions `CorrCon`, `CorrDiscon`, `CorrClear` and `CorrWrite`
- the function `CorrRead`

Basic approach

This is the general approach for setting up Path Offset. For a detailed example of how this is done, see *Code example on page 50*.

1. Declare the correction generator.
2. Connect the correction generator.
3. Define a trap routine that determines the offset and writes it to the correction generator.
4. Define an interrupt to frequently call the trap routine.
5. Call a move instruction using the correction. The path will be repeatedly corrected.

NOTE!

If two or more move instructions are called after each other with the`\Corr` switch it is important to know that all `\Corr` offsets are reset each time the robot starts from a finepoint. So ,when using finepoints, on the second move instruction the controller does not know that the path already has an offset. To avoid any strange behavior it is recommended only to use zones together with the `\Corr` switch and avoid finepoints.

Limitations

It is possible to connect several correction generators at the same time (for instance one for corrections along the Z axis and one for corrections along the Y axis). However, it is not possible to connect more than 5 correction generators at the same time.

After a controller restart, the correction generators have to be defined once again. The definitions and connections do not survive a controller restart.

The instructions can only be used in motion tasks.

5.2. RAPID components

Data types

This is a brief description of each data type in Path Offset. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
corrdescr	`corrdescr` is a correction generator descriptor that is used as the reference to the correction generator.

Instructions

This is a brief description of each instruction in Path Offset. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
CorrCon	`CorrCon` activates path correction. Calling `CorrCon` will connect a correction generator. Once this connection is made, the path can be continuously corrected with new offset inputs (for instance from a sensor).
CorrDiscon	`CorrDiscon` deactivates path correction. Calling `CorrDiscon` will disconnect a correction generator.
CorrClear	`CorrClear` deactivate path correction. Calling `CorrClear` will disconnect all correction generators.
CorrWrite	`CorrWrite` sets the path correction values. Calling `CorrWrite` will set the offset values to a correction generator.

Functions

This is a brief description of each function in Path Offset. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
CorrRead	`CorrRead` reads the total correction made by a correction generator.

5.3. Related RAPID functionality

The argument \Corr

The optional argument `\Corr` can be set for some move instructions. This will enable path corrections while the move instruction is executed.

The following instructions have the optional argument `\Corr`:

- MoveL
- MoveC
- SearchL
- SearchC
- TriggL (only if the controller is equipped with the base functionality Fixed Position Events)
- TriggC (only if the controller is equipped with the base functionality Fixed Position Events)
- CapL (only if the controller is equipped with the option Continuous Application Platform)
- CapC (only if the controller is equipped with the option Continuous Application Platform)
- ArcL (only if the controller is equipped with the option RobotWare Arc)
- ArcC (only if the controller is equipped with the option RobotWare Arc)

For more information on these instructions, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Interrupts

To create programs using Path Offset, you need to be able to handle interrupts. For more information on interrupts, see *Technical reference manual - RAPID overview*.

5.4. Code example

Linear movement with correction

This is a simple example of how to program a linear path with online path correction. This is done by having an interrupt 5 times per second, calling a trap routine which makes the offset correction.

Program code

```
VAR intnum int_no1;
VAR corrdescr id;
VAR pos sens_val;
PROC PathRoutine()
  !Connect to the correction generator
  CorrCon id;

  !Setup a 5 Hz timer interrupt.
  CONNECT int_no1 WITH UpdateCorr;
  ITimer\Single, 0.2, int_no1

  !Position for start of contour tracking
  MoveJ p10,v100,z10,tool1;

  !Run MoveL with correction.
  MoveL p20,v100,z10,tool1\Corr;

  !Remove the correction generator.
  CorrDiscon id;

  !Remove the timer interrupt.
  IDelete int_no1;
ENDPROC
TRAP UpdateCorr
  !Call a routine that read the sensor
  ReadSensor sens_val.x, sens_val.y, sens_val.z;

  !Execute correction
  CorrWrite id, sens_val;

  !Setup interrupt again
  IDelete int_no1;
  CONNECT int_no1 WITH UpdateCorr;
  ITimer\Single, 0.2, int_no1;
ENDTRAP
```

Contact us

ABB AB
Discrete Automation and Motion
Robotics
S-721 68 VÄSTERÅS
SWEDEN
Telephone +46 (0) 21 344 400

3HAC18152-1 Rev F, en