ROBOTICS

Application manual

MultiMove

Trace back information:
Workspace R18-2 version a11
Checked in 2018-10-11
Skribenta version 5.3.008

Application manual

MultiMove

RobotWare 6.08

Document ID: 3HAC050961-001

Revision: E

Original instructions.

ABB AB, Robotics
Robotics and Motion
Se-721 68 Västerås
Sweden

Table of contents

Overview of this manual

About this manual

This manual contains information about the RobotWare optionsMultiMove Independent and MultiMove Coordinated. The latter includes some extended functionality. Unless something else is specified, MultiMove refers to both these options.

Usage

This manual can be used either as a brief description to find out if MultiMove is the right choice for solving a problem, or as a description of how to use it. This manual provides information about system parameters and RAPID components related to MultiMove, and many examples of how to use them. The details regarding syntax for RAPID components, and similar, are 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.
- be familiar with the option Multitasking (see *Application manual - Engineering tools*).

References

Reference	Document ID
Technical reference manual - RAPID Overview	*3HAC050947-001*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC050917-001*
Technical reference manual manual - RAPID kernel	*3HAC050946-001*
Operating manual - IRC5 with FlexPendant	*3HAC050941-001*
Operating manual - RobotStudio	*3HAC032104-001*
Product manual - IRC5	*3HAC021313-001*
Technical reference manual - System parameters	3HAC17076-1
Application manual - Engineering tools	3HAC020434-001
Application manual - Motion functions and events	3HAC18152-1
Application manual - Arc and Arc Sensor	*3HAC050988-001*
Application manual - Spot options	*3HAC050979-001*

Continues on next page

Revisions

Revision	Description
-	Released with RobotWare 6.0.
A	Released with RobotWare 6.01. • Added information about the Ethernet switch, see *Ethernet connections on page 21*.
B	Released with RobotWare 6.02. Section *Create a MultiMove system on page 27* updated to use Installation Manager.
C	Released with RobotWare 6.04. • Minor corrections.
D	Released with RobotWare 6.05. • Minor corrections.
E	Released with RobotWare 6.08. • Max number of motion tasks changed to seven.

Product documentation

Categories for user documentation from ABB Robotics

The user documentation from ABB Robotics 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 can be found via myABB Business Portal, *www.myportal.abb.com*.

Product manuals

Manipulators, controllers, DressPack/SpotPack, and most other hardware is 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 corresponding figures (or references to separate spare parts lists).
- References to circuit diagrams.

Technical reference manuals

The technical reference manuals describe reference information for robotics products, for example lubrication, the RAPID language, and system parameters.

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, software).
- How to install included or required hardware.
- How to use the application.
- Examples of how to use the application.

Continues on next page

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 troubleshooters.

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*.

This page is intentionally left blank

1 Introduction

1.1 About MultiMove

Purpose

The purpose of MultiMove is to let one controller handle several robots. This does not only save on hardware costs, it also allows advanced coordination between different robots and other mechanical units.

Here are some examples of applications:

- Several robots can work on the same moving work object.
- One robot can move a work object while other robots work on it.
- Several robots can cooperate to lift heavy objects.

Included functionality

MultiMove allows up to 7 tasks to be motion tasks (task that has move instructions). Since no more than 4 drive modules can be used, a controller can handle up to 4 robots. However, additional axes can be handled by separate tasks up to a total of 7 motion tasks.

Both MultiMove options allow you to implement:

- Independent movements (see *Independent movements on page 60*)
- Semi coordinated movements (see *Semi coordinated movements on page 63*)

In addition to what is mentioned above, the option MultiMove Coordinated allows you to implement:

- Coordinated synchronized movements (see *Coordinated synchronized movements on page 71*)

Included options

If you have MultiMove, you automatically have access to some options that are necessary in order to use MultiMove.

MultiMove always includes the option:

- Multitasking

In addition to what is mentioned above, MultiMove Coordinated includes the option:

- Multiple Axis Positioner

Basic approach

This is the general approach for setting up a MultiMove system.

1. Install hardware and software (see *Installation on page 19*).
2. Configure system parameters (see *Configuration on page 29*).
3. Calibrate coordinate systems (see *Calibration on page 41*).
4. Write RAPID program for each task (see *Programming on page 55*).

1.2 Terminology

About these terms

Some words have a specific meaning when used in this manual. It is important to understand exactly what is meant by these words. This manual's definition of these words are listed below.

Term list

Term	Explanation
Coordination	A robot that is coordinated to a work object will follow the movements of that work object.
Synchronization	Movements that are simultaneous. Synchronization refers to a similarity in time, not in room coordinates.
Positioner	A mechanical unit without TCP, which can only handle joint movements. A positioner is a mechanical unit, with one or several axes, that holds and moves a work object.
Robot	A mechanical unit with TCP, which can be programmed in Cartesian coordinates (x, y and z).
Task program	The same as a program. It is just a way of specifying that it is a program for one specific task.

1.3 Example applications

1.3.1 About the example applications

Three consistent examples

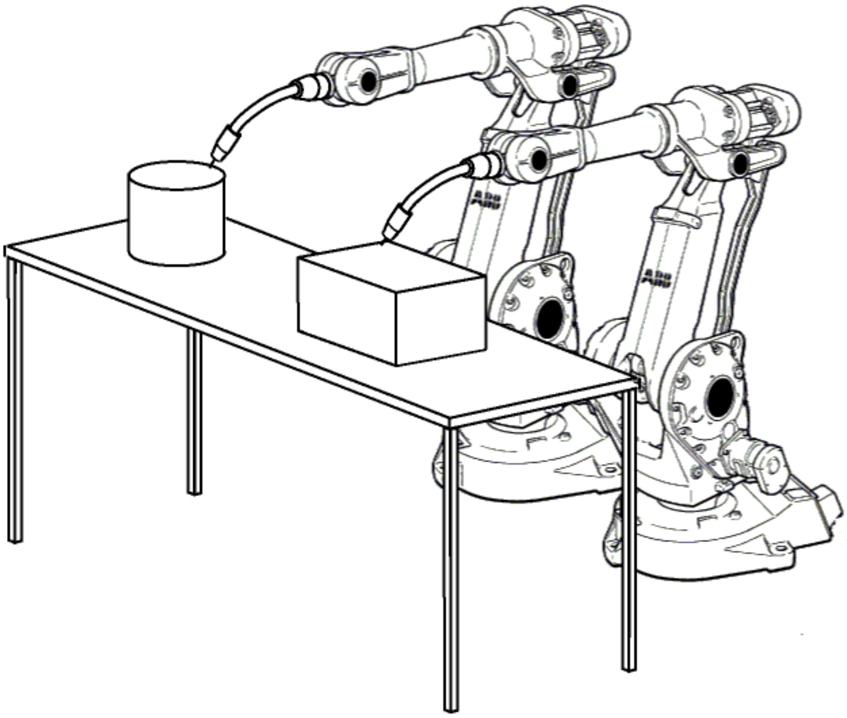
In this manual there are many examples (for configuration, RAPID code etc.). Every example is created for one of three physical robot systems. These example robot system setups are called "UnsyncArc", "SyncArc" and "SyncSpot" and will help you understand what kind of robot system an example is made for. The examples are also consistent, i.e. the RAPID code example for "SyncSpot" is made for a robot system configured as the configuration example for "SyncSpot".

1.3.2 Example "UnsyncArc"

About example "UnsyncArc"

In this example, two robots work independently on one work piece for each robot. They do not cooperate in any way and do not have to wait for each other.

Illustration



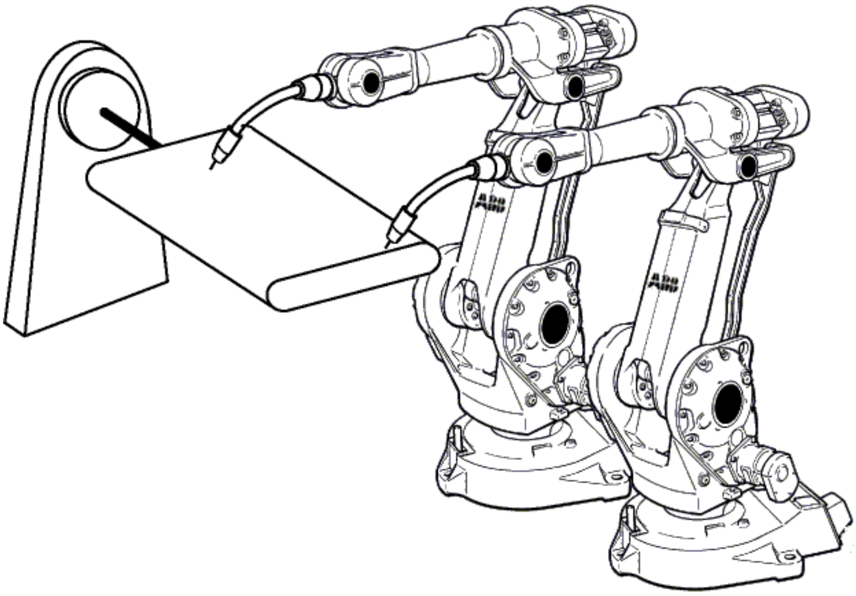
xx0300000590

1.3.3 Example "SyncArc"

About example "SyncArc"

In this example, two robots perform arc welding on the same work piece. The work object is rotated by a positioner.

Illustration



xx0300000594

This page is intentionally left blank

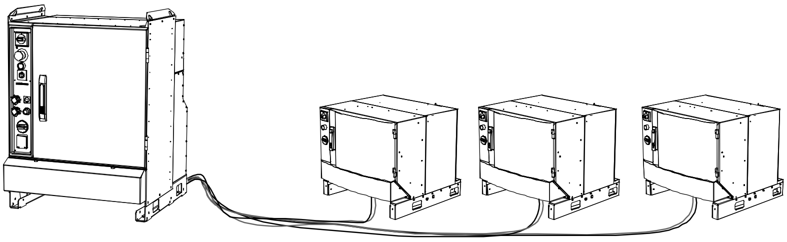
2 Installation

2.1 Hardware installation

2.1.1 About the hardware installation

Overview

A controller that handles several robots needs extra drive modules (one drive module per robot). Up to four drive modules can be used, including the one assembled with the control module.



xx0400001042

One Ethernet cable and one safety signal cable for each additional drive module must be connected to the control module. A MultiMove control module is equipped with an extra Ethernet switch to communicate with the additional drive modules.

This manual only describes what is specific for a MultiMove installation. For more information about installation and commissioning of the controller, see *Product manual - IRC5*.

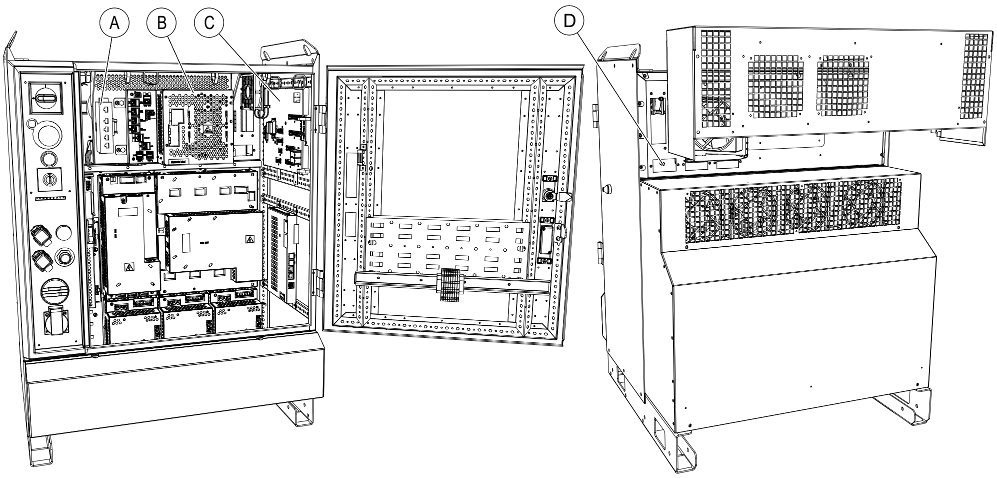
2.1.2 Connections on the control module

Connect drive modules to the control module

At delivery, both the Ethernet cable and the safety signal cable are connected to the drive module. They are also attached to a shield plate that fits in the slot of the control module.

Remove the cover from an empty slot and fit the shield plate of the communication cables in its place. Connect the Ethernet cable according to *Ethernet connections on page 21* and safety signal cable according to *Safety signal connections on page 23*.

DSQC1000

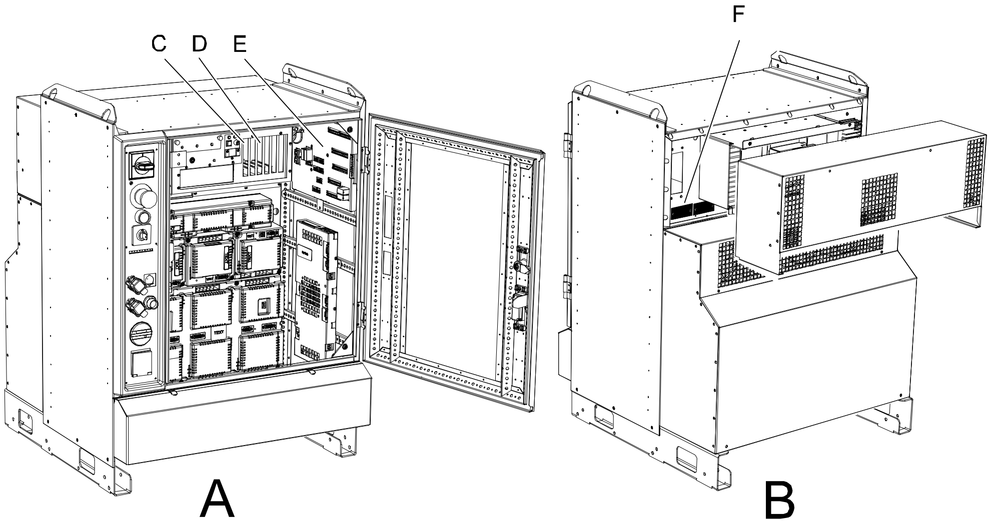


xx1400000408

A	MultiMove Ethernet switch DSQC1007 (3HAC045976-001)
B	Main computer DSQC1000
C	Panel board DSQC 643
D	Slots for inserting communication cables into the control cabinet

Continues on next page

DSQC 639



xx0600002780

A	Front view of single cabinet controller
B	Back view of single cabinet controller
C	Robot communication card
D	Ethernet card (only present if more than one drive module is used)
E	Panel board
F	Slots for inserting communication cables into the control cabinet

Ethernet connections

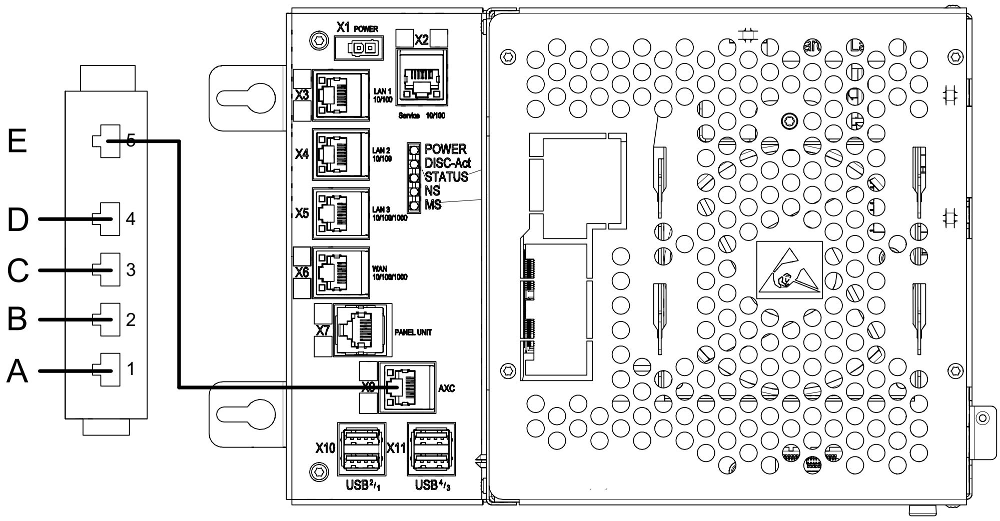
Connect the Ethernet cables according to the following figure:

Note

It is important that the right drive module is connected to the right Ethernet connection. If the order of the Ethernet connections do not match the selections made in Installation Manager (see *Create a MultiMove system on page 27*), the robot configuration will not correlate to the correct robot.

Continues on next page

DSQC1000



xx1400000409

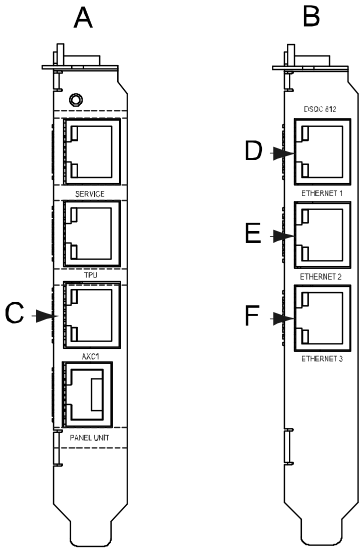
A	Ethernet connection to drive module #1 (already connected at delivery)
B	Ethernet connection to drive module #2
C	Ethernet connection to drive module #3
D	Ethernet connection to drive module #4
E	Ethernet connection between the MultiMove switch and the main computer (already connected at delivery)

Note

The Ethernet switch is mandatory for MultiMove, also when running MultiMove with only one axis computer in the system. For example, when running MultiMove on one robot together with a positioner or additional axes.

Continues on next page

DSQC 639

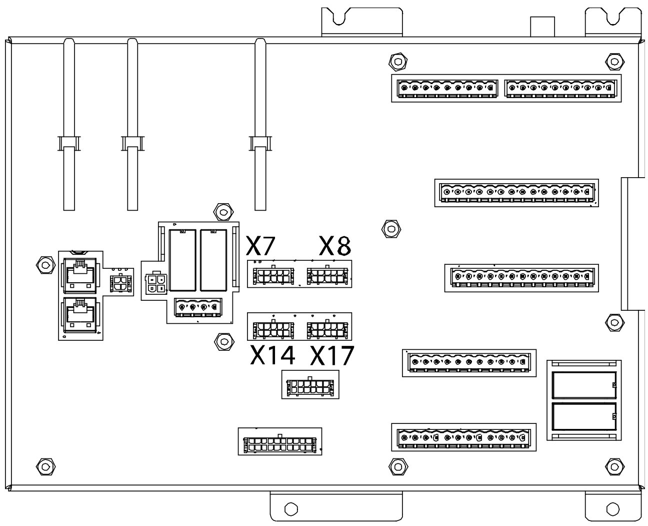


xx0400001141

A	Robot communication card
B	Ethernet card
C	Ethernet connection to drive module #1 (already connected at delivery)
D	Ethernet connection to drive module #2
E	Ethernet connection to drive module #3
F	Ethernet connection to drive module #4

Safety signal connections

The safety signal cable from a drive unit is connected to the Panel board according to the following figure:



xx0400001295

X7	Connector for safety signal cable to drive module #1(already connected at delivery)

Continues on next page

X8	Connector for safety signal cable to drive module #2
X14	Connector for safety signal cable to drive module #3
X17	Connector for safety signal cable to drive module #4

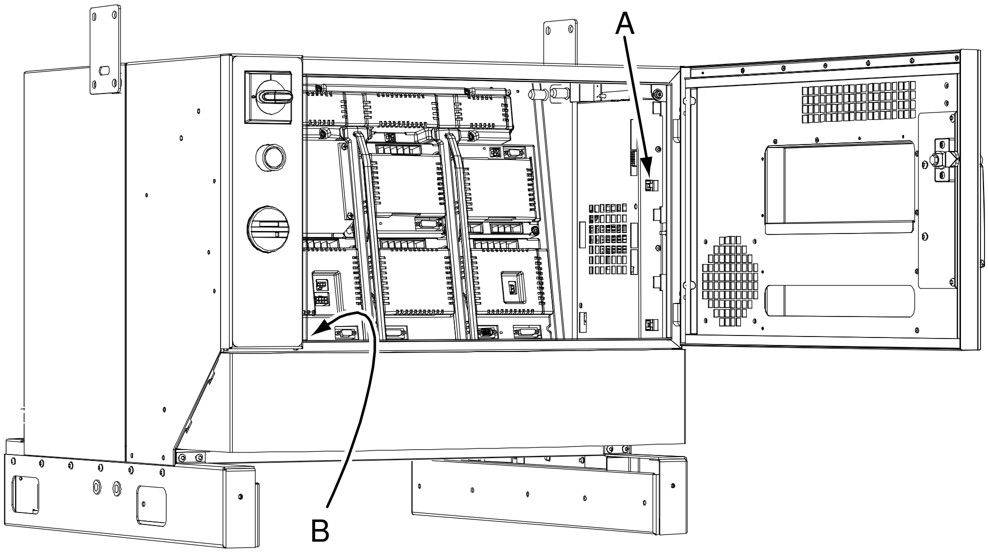
Remove the jumper connector and replace it with the safety signal cable.

2.1.3 Connections on the drive module

Already connected at delivery

When a MultiMove system is delivered from ABB, the Ethernet cable and the safety signal cable are already connected to the drive module. You only need to know how these cables are connected if you are going to change the hardware configuration or replace parts.

Connect the cables from the control module



xx0600002787

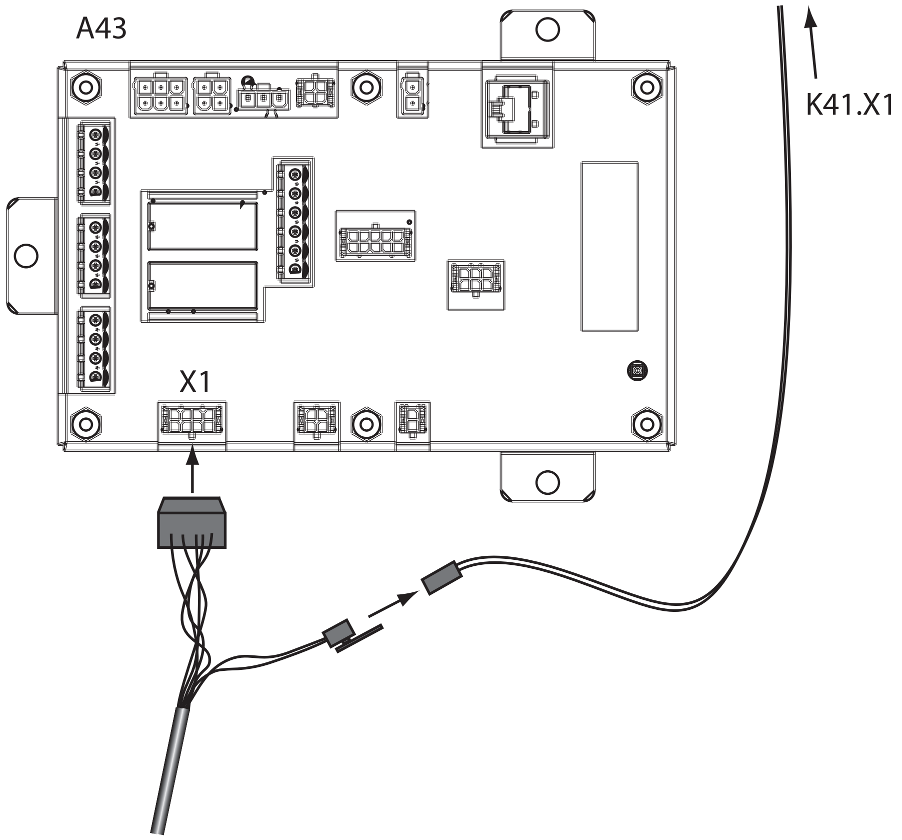
A	Ethernet connection
B	Contactor interface board

Connect the Ethernet cable to the Ethernet connection marked Computer module link. Connect the safety signal cable according to *Safety signal connection on page 26*.

Continues on next page

Safety signal connection

The safety signal cable is connected to the Contactor interface board (A43), connector X1. There is also a connector that needs to be connected to a cable with a connector marked K41.X1.



xx0600002786

2.2 Software installation

2.2.1 Software installation

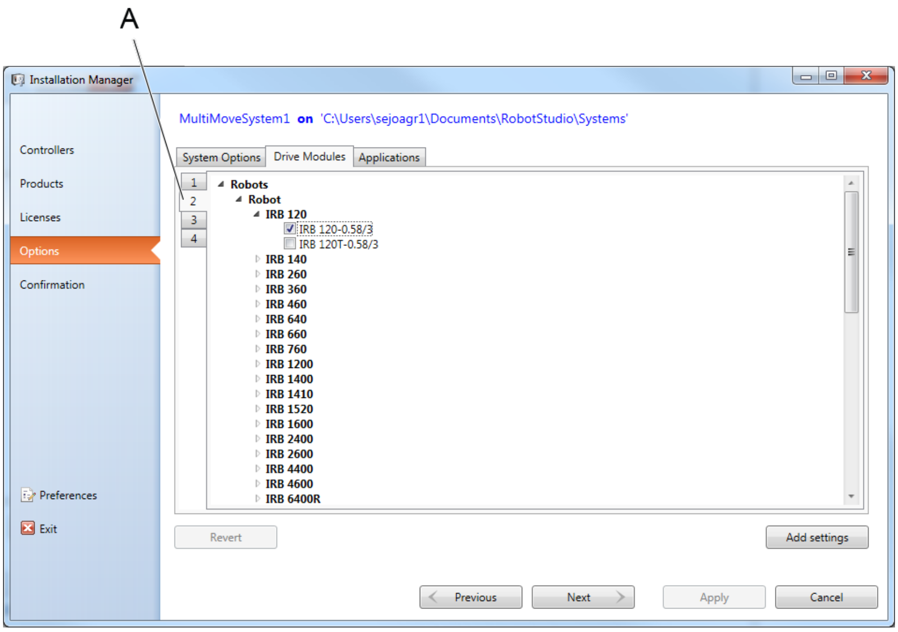
Install RobotStudio and RobotWare

Installation of RobotStudio and RobotWare on a PC is described in *Operating manual - Getting started, IRC5 and RobotStudio*.

Create a MultiMove system

Creating a new system is described in *Operating manual - RobotStudio*. Select the MultiMove option under **System Options**.

What is specific for a MultiMove system is that in the tab **Drive Modules**, one robot should be selected for each drive module.



xx1500000865

A	One tab for each drive module

Automatic configurations at installation

When creating a system, some configurations are automatically set up according to information from your license. For each robot, the following are created:

- Task
- Mechanical Unit Group
- Mechanical Unit
- Motion Planner

For more information about these system parameter types, see *System parameters on page 30*.

Continues on next page

CAUTION

A motion planner (type Motion Planner), created by the installation process, is configured to optimize the movement for its specific robot. If the default configuration is changed so that a robot uses the wrong motion planner, the robot motion will be affected.

3 Configuration

3.1 Configuration overview

About the system parameters

This chapter contains a brief description of each parameter that is specific for MultiMove. Parameters that are used the same way as for a single robot system are not mentioned here.

For more information about system parameters, see *Technical reference manual - System parameters*.

About the examples

The first three examples cover the topics *Controller* and *Motion*, since these are related to the physical constellation of the robot system. The last example covers the topic *I/O System*, which is handled similarly regardless of the robot system.

3.2 System parameters

3.2.1 Controller topic

Task

These parameters belong to the type *Task* in the topic *Controller*:

Parameter	Description
Task	The name of the task. Note that the name of the task must be unique. This means that it cannot have the same name as the mechanical unit, and no variable in the RAPID program can have the same name.
Type	Controls the start/stop and system restart behavior: • NORMAL - The task program is manually started and stopped (e.g. from the FlexPendant). The task stops at emergency stop. • STATIC - At a restart the task program continues from where it was. The task program cannot be stopped from the FlexPendant or by emergency stop. • SEMISTATIC - The task program starts from the beginning at re-start. The task program cannot be stopped from the FlexPendant or by emergency stop. A task that controls a mechanical unit must be of the type NORMAL.
MotionTask	Indicates whether the task program can control a mechanical unit with RAPID move instructions.
Use Mechanical Unit Group	Defines which mechanical unit group is used for the task. *Use Mechanical Unit Group* refers to the parameter *Name* for the type *Mechanical Unit Group*. A motion task (*MotionTask* set to Yes) controls the mechanical units in the mechanical unit group. A non-motion task (*MotionTask* set to No) will still be able to read values (e.g. the TCP position) for the active mechanical units in the mechanical unit group. Note that *Use Mechanical Unit Group* must be defined for all tasks, even if the task does not control any mechanical unit.

Mechanical Unit Group

A mechanical unit group must contain at least one mechanical unit, robot or other mechanical unit (i.e. both *Robot* and *Mech Unit 1* cannot be left empty).

These parameters belong to the type *Mechanical Unit Group* in the topic *Controller*:

Parameter	Description
Name	The name of the mechanical unit group.
Robot	Specifies the robot (with TCP), if there is any, in the mechanical unit group. *Robot* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Mech Unit 1	Specifies a mechanical unit without TCP, if there is any, in the mechanical unit group. *Mech Unit 1* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Mech Unit 2	Specifies the second mechanical unit without TCP, if there are more than one, in the mechanical unit group. *Mech Unit 2* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.

Continues on next page

Parameter	Description
Mech Unit 3	Specifies the third mechanical unit without TCP, if there are more than two, in the mechanical unit group. *Mech Unit 3* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Mech Unit 4	Specifies the forth mechanical unit without TCP, if there are more than three, in the mechanical unit group. *Mech Unit 4* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Mech Unit 5	Specifies the fifth mechanical unit without TCP, if there are more than four, in the mechanical unit group. *Mech Unit 5* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Mech Unit 6	Specifies the sixth mechanical unit without TCP, if there are more than five, in the mechanical unit group. *Mech Unit 6* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.
Use Motion Planner	Defines which motion planner is used for calculating the movements of this mechanical unit group. *Use Motion Planner* refers to the parameter *Name* for the type *Motion Planner* in the topic *Motion*.

3.2.2 Motion topic

Drive Module User Data

If a drive module should be disconnected without disturbing robots and additional axes connected to other drive modules in the robot system, then use the Drive Module Disconnect function.

This parameter belongs to the type *Drive Module User Data* in the topic *Motion*.

Parameter	Description
Allow Drive Module Disconnect	Set *Allow Drive Module Disconnect* to TRUE to disconnect the drive module.

Mechanical Unit

No parameters of type *Mechanical Unit* can be edited for a robot. They can only be edited for additional axes.

These parameters belong to the type *Mechanical Unit* in the topic *Motion*:

Parameter	Description
Name	The name of the mechanical unit.
Allow move of user frame	Indicates if the mechanical unit should be allowed to move user frames.
Activate at Start Up	Indicates if the mechanical unit should be active when the controller starts up. In a single robot system, the robot is always active. In a MultiMove system, any mechanical unit (including robots) can be inactive at start up and be activated later.
Deactivation Forbidden	Indicates if it should be possible to deactivate the mechanical unit. In a single robot system it is not possible to deactivate a robot. In a MultiMove system, one robot can be deactivated while another is still active.

Motion Planner

A motion planner calculates the movements of a mechanical unit group. When several tasks are in synchronized movement mode they use the same motion planner (the first of the involved motion planners), see pictures in the following examples.

At installation a *Motion Planner* is set up for each robot. The *Motion Planner* is configured to optimize the motion for that specific robot. Do not change connection between robot and *Motion Planner.*

These parameters belong to the type *Motion Planner* in the topic *Motion*:

Parameter	Description
Name	The name of the motion planner.

Continues on next page

Parameter	Description
Speed Control Warning	In synchronized movement mode, the speed of one robot can be slower than the programmed speed. This is because another robot might limit the speed (e.g. if the other robot has a longer path). If *Speed Control Warning* is set to Yes, a warning will be given when the robot moves slower than the programmed speed, in relation to the work object. *Speed Control Warning* is only used to supervise TCP speed, i.e. the speed of an additional axis is not supervised.
Speed Control Percent	If *Speed Control Warning* is set to Yes, a warning will be issued when the actual speed is slower than this percentage of the programmed speed.

3.2.3 I/O topic

Systems with several robots

Configuring *I/O* for a system with several robots is usually no different from a single robot system. However, for some system inputs and system outputs there is a need to specify which task or which robot it refers to. This is a description of the system parameters used to specify which task a system input is valid for, or which robot a system output is valid for. It is also explained when they have to be used.

For more information, see *Technical reference manual - System parameters*.

System Input

These parameters belong to the type *System Input* in the topic *I/O*.

Parameter	Description
Argument 2	Specifies which task this system input should affect. If the parameter *Action* is set to *Interrupt* or *Load and Start*, then *Argument 2* must specify a task. All other values for *Action* results in a system input that is valid for all tasks, and *Argument 2* is not required. *Argument 2* refers to the parameter *Task* for the type *Task*.

System Output

These parameters belong to the type *System Output* in the topic *I/O*.

Parameter	Description
Argument	Specifies which mechanical unit the system output refers to. If the parameter *Status* is set to *TCP Speed*, *TCP Speed Reference* or *Mechanical Unit Active*, then *Argument* must specify a mechanical unit. For all other values for *Status* the system output does not refer to a single robot, and *Argument* is not required. *Argument* refers to the parameter *Name* for the type *Mechanical Unit* in the topic *Motion*.

3.3 Configuration examples

3.3.1 Configuration example for "UnsyncArc"

About this example

This is an example of how to configure example "UnsyncArc", two independent robots. The robots are handled by one task each.

Task

Task	Type	MotionTask	Use Mechanical Unit Group
T_ROB1	NORMAL	Yes	rob1
T_ROB2	NORMAL	Yes	rob2

Mechanical Unit Group

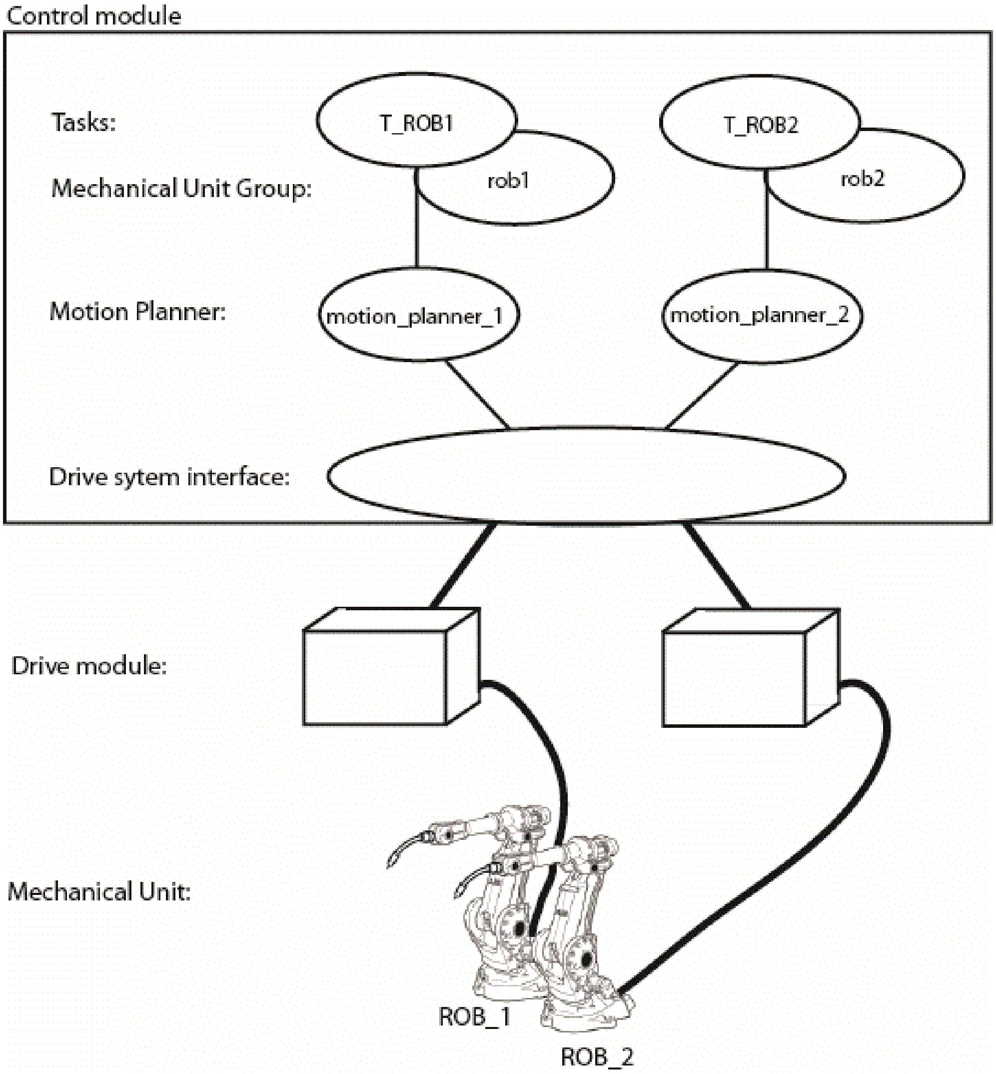
Name	Robot	Mech Unit 1	Use Motion Planner
rob1	ROB_1		motion_planner_1
rob2	ROB_2		motion_planner_2

Motion Planner

Name	Speed Control Warning
motion_planner_1	No
motion_planner_2	No

Continues on next page

Illustration



en0400000773

3.3.2 Configuration example for "SyncArc"

About this example

This is an example of how to configure example "SyncArc", two robots and a positioner. These three mechanical units are handled by one task each.

Task

Task	Type	MotionTask	Use Mechanical Unit Group
T_ROB1	NORMAL	Yes	rob1
T_ROB2	NORMAL	Yes	rob2
T_STN1	NORMAL	Yes	stn1

Mechanical Unit Group

Name	Robot	Mech Unit 1	Use Motion Planner
rob1	ROB_1		motion_planner_1
rob2	ROB_2		motion_planner_2
stn1		STN_1	motion_planner_3

Motion Planner

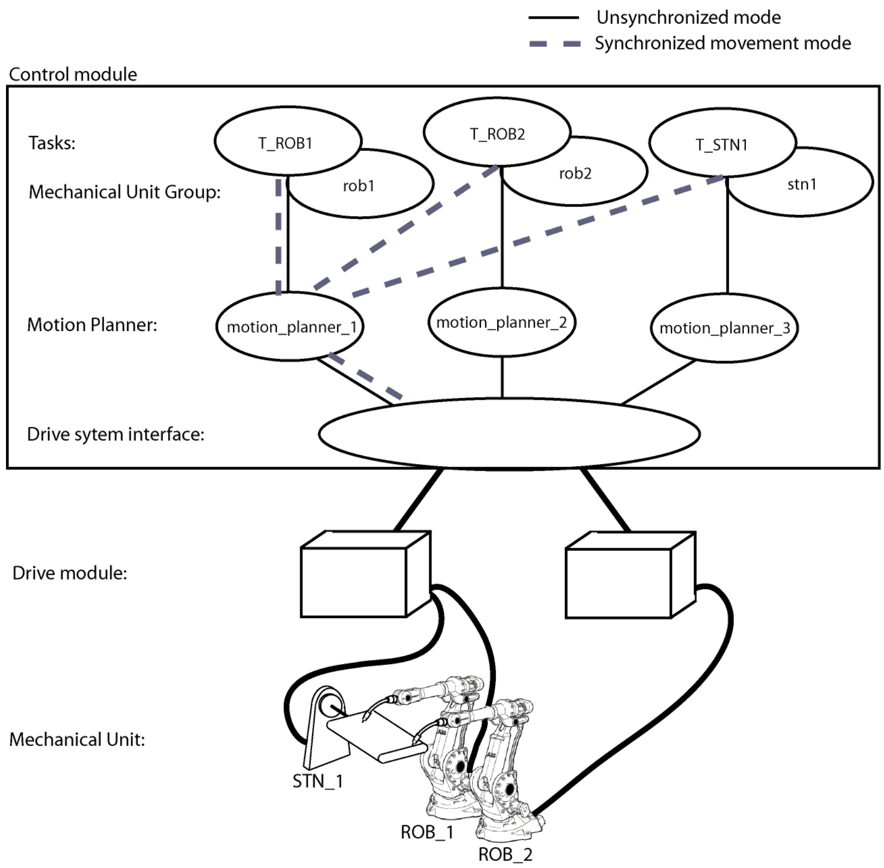
Name	Speed Control Warning	Speed Control Percent
motion_planner_1	Yes	90
motion_planner_2	Yes	90
motion_planner_3	No	

Mechanical Unit

Name	Allow move of user frame	Activate at Start Up	Deactivation Forbidden
ROB_1	Yes	Yes	Yes
ROB_2	Yes	Yes	Yes
STN_1	Yes	Yes	No

Continues on next page

Illustration



en0400000774

3.3.3 I/O configuration example

About this example

This is an example of how to configure some I/O signals that require an argument for task or robot. This example is based on the example "SyncArc".

The input signal `di_position` is set up to interrupt the program execution and call the routine `SetStartPosition` in the task T_STN1. The output signal `ao_speed1` is configured to indicate the speed of robot 1, and `ao_speed2` to indicate the speed of robot 2.

System Input

Signal Name	Action	Argument	Argument 2
di_position	Interrupt	SetStartPosition	T_STN1

System Output

Status	Signal Name	Argument
TCP Speed	ao_speed1	ROB_1
TCP Speed	ao_speed2	ROB_2

This page is intentionally left blank

4 Calibration

4.1 Calibration overview

Two types of calibration

There are two types of calibration that must be done for a robot system:

1. Joint calibration ensures that all axes are in correct position. Normally this is done before delivery of a new robot and only requires recalibration after repairing the robot. For more information, see the product manual for the respective robot.
2. Calibration of coordinate systems must be made when the robot is in place. A brief description of what coordinate systems to calibrate and in which order is presented below.

Calibrate coordinate systems

First of all you must decide what coordinate systems to use and how to place their origins and directions. For examples of suitable coordinate systems, see *Examples of coordinate systems on page 45*.

The coordinate systems are then calibrated in the following order:

	Action
1	Calibrate the tool. This includes calibration of TCP and load data. For description of how to calibrate the tool, see *Operating manual - IRC5 Integrator's guide*.
2	Calibrate the base coordinate system, relative to the world coordinate system, for all the robots. For description of how to calibrate the base coordinate system for a robot, see *Operating manual - IRC5 Integrator's guide*. If one robot already has a calibrated base coordinate system, the base coordinate system for another robot can be calibrated by letting the TCPs of the two robots meet at several points. This method is described in *Relative calibration on page 42*.
3	Calibrate the base coordinate systems, relative to the world coordinate system, for the positioners. For description of how to calibrate the base coordinate system for a positioner, see *Application manual - Additional axes and stand alone controller*.
4	Calibrate a user coordinate system, relative to the world coordinate system. For description of how to calibrate a user coordinate system, see *Operating manual - IRC5 Integrator's guide*.
5	Calibrate an object coordinate system, relative to the user coordinate system. For description of how to calibrate an object coordinate system, see *Operating manual - IRC5 Integrator's guide*.

4.2 Relative calibration

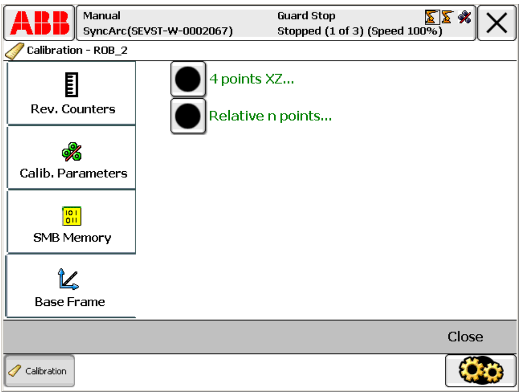
What is relative calibration

Relative calibration is used to calibrate the base coordinate system of one robot, using a robot that is already calibrated. This calibration method can only be used for a MultiMove system where two robots are placed close enough to have some part of their working areas in common.

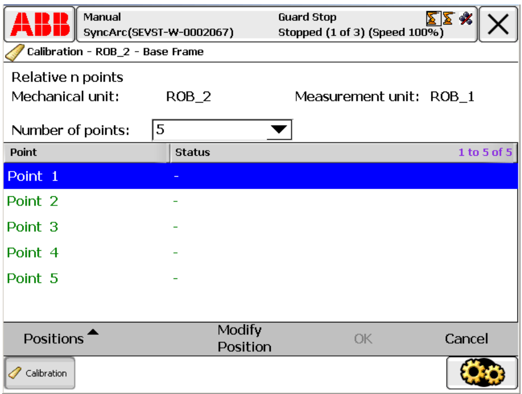
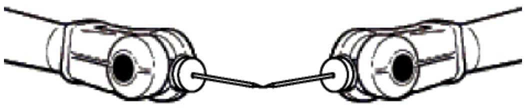
If one robot has a base coordinate system that is identical with the world coordinate system, this robot can be used as a reference for another robot. If no robot has a base coordinate system that is identical with the world coordinate system, the base coordinate system for one robot must be calibrated first. For information about other calibration methods, see *Operating manual - IRC5 Integrator's guide*.

How to perform relative calibration

The tools for both robots must be correctly calibrated before using relative calibration, and those tools must be active during calibration.

	Action	Info/illustration
1	In the **ABB** menu, select **Calibration**.	
2	Tap on the robot you want to calibrate.	
3	If present, tap **Manual Method (Advanced)**.	
4	Tap **Base Frame**.	 en0400000790
5	Tap **Relative n points.**	
6	If you have more than two robots, you must select which robot to use as reference. If you only have two robots, this step is skipped.	

Continues on next page

	Action	Info/illustration
7	The calibration can be performed with between 3 and 10 points. Select how many you want to use in **Number of Points**. To get adequate accuracy, at least 5 points is recommended.	 en0400000791
8	Select **Point 1**.	
9	Jog the robot you want to calibrate and the reference robot so that both TCPs are in the same point.	 xx0400000785
10	Tap on **Modify Position**.	
11	Repeat step 7-9 for all the points. Make sure that the points are spread out in both x, y and z coordinates. If, for example, all point are at the same height, the z coordinate will be poorly calibrated.	
12	Tap **OK**.	The calibration result is shown.
13	Tap **OK** to accept the calibration.	
14	Restart the controller.	

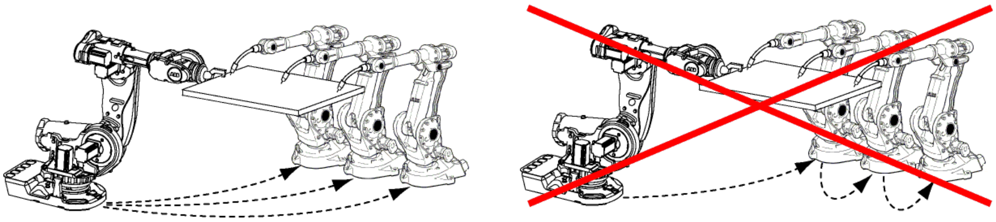
4.3 Calibration chains

Avoid long chains of calibrations

If a robot that is calibrated with relative calibration acts as reference in the next calibration, the inaccuracies in the calibrations are added for the last robot.

Example

You have four robots, where robot 1 holds a work piece that robots 2, 3 and 4 work on.



xx0400000901

Calibrate robot 2, 3 and 4 against robot 1.

If you would use robot 1 as reference for robot 2, robot 2 as reference for robot 3 and robot 3 as reference for robot 4, the accuracy for robot 4 would be poor.

4.4 Examples of coordinate systems

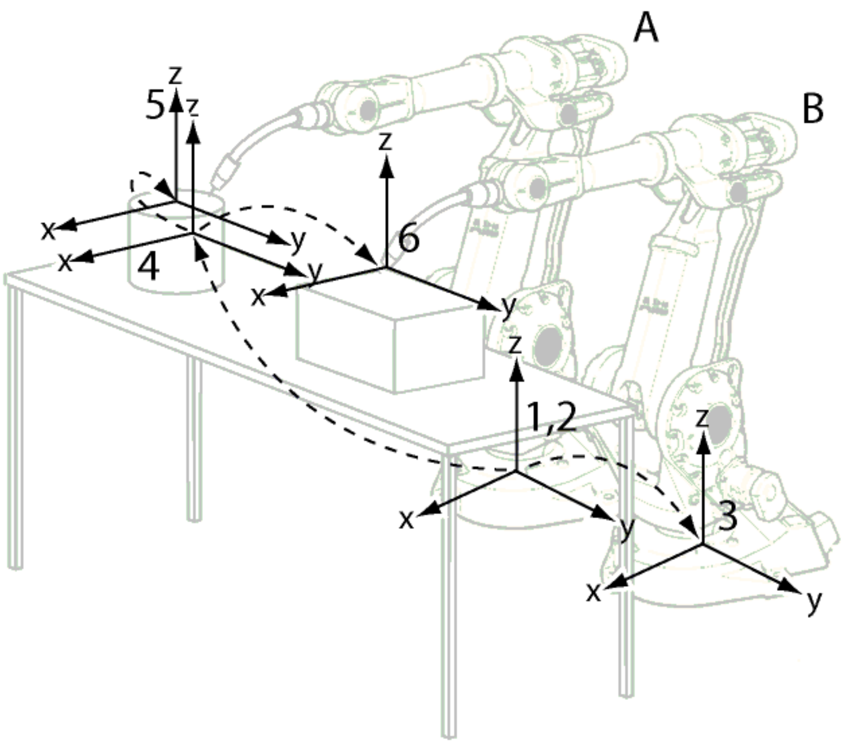
4.4.1 Example "UnsyncArc"

About this example

In this example, the world coordinate system and the base coordinate system for robot 1 (A) are identical.

The base coordinate system for robot 2 (B) is defined. Both robots have a user coordinate system with the origin in a table corner. An object coordinate system is defined for each robot's work object.

Illustration



xx0300000591

Coordinate systems

Item	Description
A	Robot 1
B	Robot 2
1	World coordinate system
2	Base coordinate system for robot 1
3	Base coordinate system for robot 2
4	User coordinate system for both robots
5	Object coordinate system for robot 1
6	Object coordinate system for robot 2

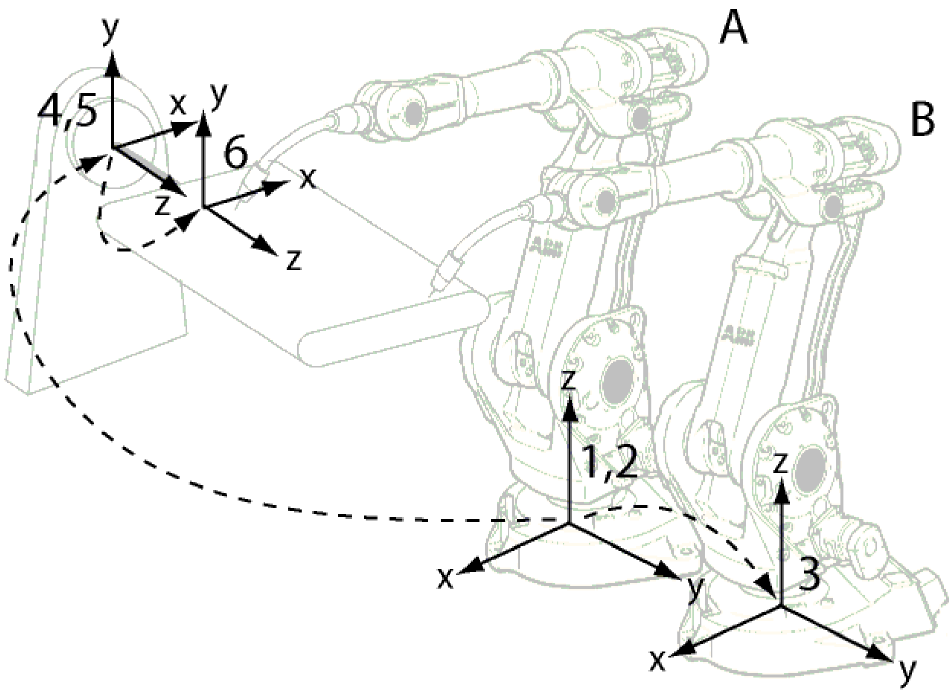
4.4.2 Example "SyncArc"

About this example

In this example, the world coordinate system and the base coordinate system for robot 1 (A) are identical.

The base coordinate system for robot 2 (B) is defined. A user coordinate system is defined to be connected to the rotating axis of the positioner. An object coordinate system is defined to be fixed to the work object held by the positioner.

Illustration



xx0300000595

Coordinate systems

Item	Description
A	Robot 1
B	Robot 2
1	World coordinate system
2	Base coordinate system for robot 1
3	Base coordinate system for robot 2
4	Base coordinate system for positioner
5	User coordinate system for both robots
6	Object coordinate system for both robots

5 User interface specific for MultiMove

5.1 FlexPendant for MultiMove system

About FlexPendant for MultiMove system

Working with the FlexPendant in a MultiMove system is not very different from a single robot system. This chapter will explain a few things that are specific for a MultiMove system. For general information about the FlexPendant, see *Operating manual - IRC5 with FlexPendant*.

What is specific for MultiMove?

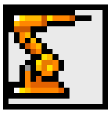
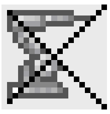
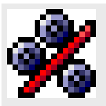
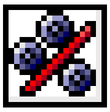
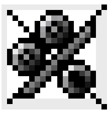
Some things that are specific for MultiMove are:

- The status bar shows which robots (and additional axes) are coordinated. See *Status bar indications on page 48*.
- When opening the program editor, you must select a task. See *Opening the Program Editor on page 50*.
- The Production window contains tabs for different tasks. See *Production Window on page 51*.
- The mechanical unit menu can contain several robots. See *Mechanical unit menu on page 52*.
- You can select which tasks to execute at start. See *Select which tasks to start with START button on page 53*.
- There is an additional method for calibrating a robot base frame, relative calibration. See *Relative calibration on page 42*.

5.2 Status bar indications

Symbol descriptions

On the right side of the status bar, at the top of the FlexPendant, there are symbols for the mechanical units in the system.

Symbol	Description
 xx0400001165	A robot that is not the selected mechanical unit, or coordinated with the selected mechanical unit. Jogging will not move this robot.
 xx0400001166	A robot that is the selected mechanical unit, or coordinated with the selected mechanical unit. Jogging will move this robot (together with any other coordinated mechanical units).
 xx0400001167	A robot belonging to an inactive task. For deactivation of tasks, see *Select which tasks to start with START button on page 53*.
 xx0400001168	An additional axis that is not the selected mechanical unit, or coordinated with the selected mechanical unit. Jogging will not move this additional axis.
 xx0400001169	An additional axis that is the selected mechanical unit, or coordinated with the selected mechanical unit. Jogging will move this additional axis (together with any other coordinated mechanical units).
 xx0400001170	An additional axis that is not active. Either the mechanical unit is deactivated, or the task inactive.

Example

en0400001158

This is an example of a MultiMove system with 4 robots and 2 additional axes, where...

- robot 1 belongs to a task that is inactive.
- robot 2 is not selected or coordinated with the selected unit (not affected by jogging).

Continues on next page

- additional axis 1 is selected mechanical unit and robot 3 and 4 are coordinated with additional axis 1. Any jogging at this point will move these three mechanical units.
- additional axis 2 is deactivated, or its task inactive.

5.3 Opening the Program Editor

Select task

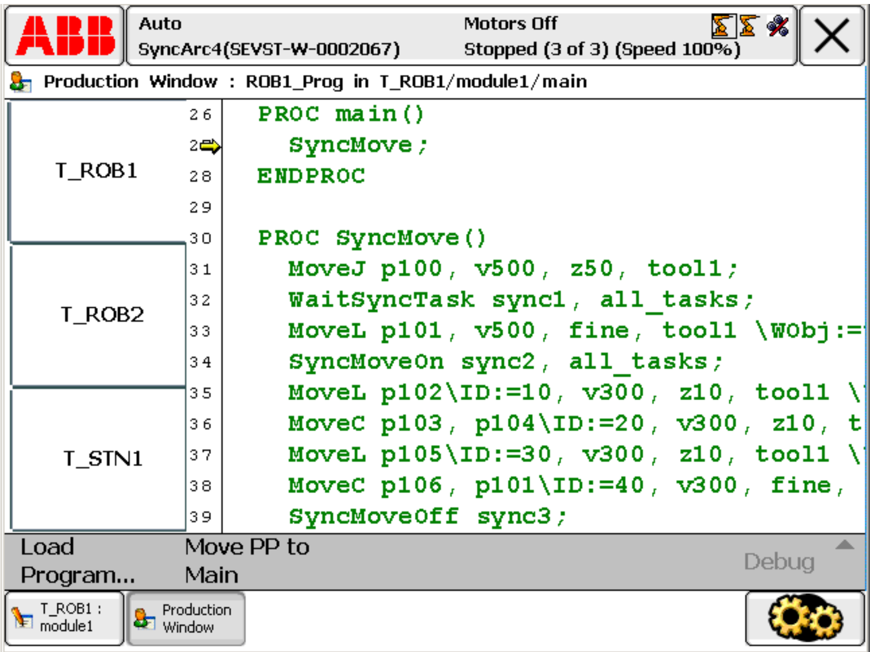
When opening the Program Editor for a system with more than one task, a list of all the tasks will be displayed. By tapping the task you want, the program code for that task is displayed.

For a system with only one task, this list is never shown. The program code is shown directly.

5.4 Production Window

The graphical display

In a system with more than one motion task there will be one tab for each motion task. By tapping a tab, you can see the program code for that task and where the program pointer and motion pointer are in that task.



en0400000796

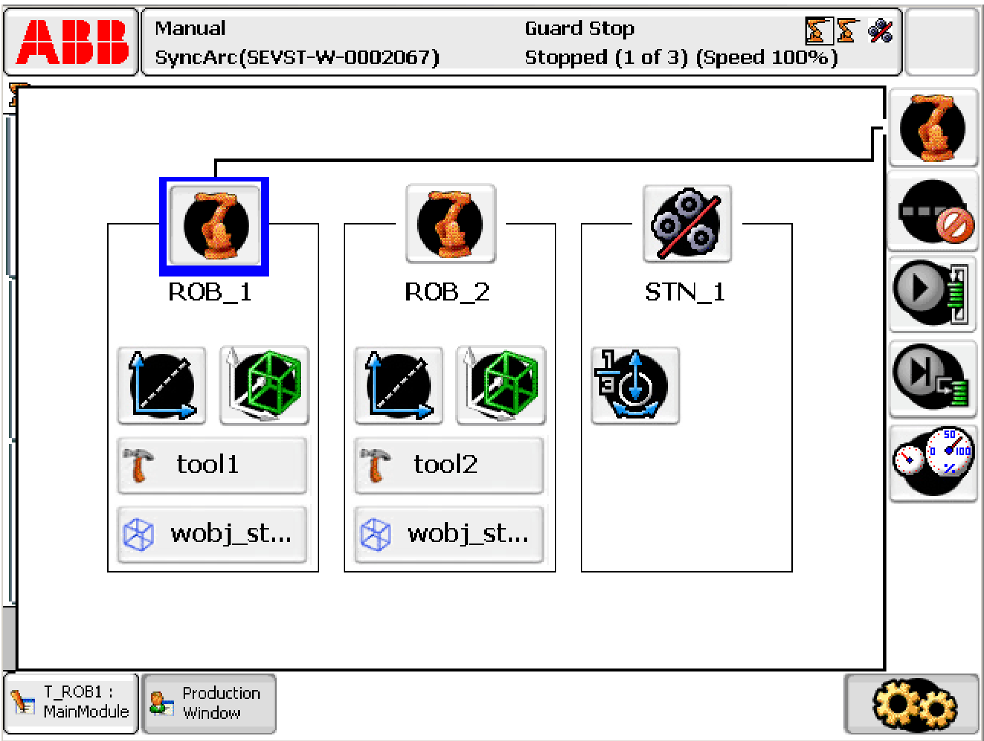
Move program pointer

If you tap **Move PP To Main**, the program pointer will be moved to main for all motion task programs.

5.5 Mechanical unit menu

The graphical display

In the QuickSet menu, tap the Mechanical unit menu button. All mechanical units will be shown.



en0400000789

The selected mechanical unit is highlighted with a frame around it.

Any mechanical unit that is coordinated with the selected unit will be indicated with a flashing frame and the text Coord.

Jogging coordinated or uncoordinated

Jogging a mechanical unit will automatically move all units that are coordinated with it.

In the example above, jogging STN_1 will move ROB_1 and ROB_2 as well, since they are coordinated with STN_1 (the work object wobj_stn1 is moved by STN_1). To be able to jog STN_1 without moving the robots, change the coordinate system for the robots to world coordinates or change the work object for the robots to wobj0. For more information about work objects being coordinated with a mechanical unit, see *Coordinated work objects on page 59*.

5.6 Select which tasks to start with START button

Background

The default behavior is that the programs of all NORMAL tasks are started simultaneously when pressing the START button. However, not all NORMAL task programs need to run at the same time. It is possible to select which of the NORMAL task programs will start when pressing the START button.

If **All Tasks** is selected in the **Task Panel Settings**, the programs of all STATIC and SEMISTATIC tasks with *TrustLevel* set to NoSafety can be selected to be started with the START button, forward stepped with the FWD button, backward stepped with the BWD button, and stopped with the STOP button.

If **Task Panel Settings** is set to **Only Normal tasks**, all STATIC and SEMISTATIC tasks are greyed out and cannot be selected in the task panel, Quickset menu (see *Operating manual - IRC5 with FlexPendant*, section *Quickset menu*). All STATIC and SEMISTATIC tasks will be started if the start button is pressed.

If **Task Panel Settings** is set to **All tasks**, STATIC and SEMISTATIC tasks with *TrustLevel*NoSafety can be selected in the task panel. All selected STATIC and SEMISTATIC tasks can be stopped, stepped, and started. .

A STATIC or SEMISTATIC task, not selected in the task panel, can still be executing. This is not possible for a NORMAL task.

Run Mode is always continuous for STATIC and SEMISTATIC tasks. The Run Mode setting in the Quickset menu is only applicable for NORMAL tasks (see *Operating manual - IRC5 with FlexPendant*, section *Quickset menu*).

This will only work in manual mode, no STATIC or SEMISTATIC task can be started, stepped, or stopped in auto mode.

Task Panel Settings

To start the **Task Panel Settings**, tap the ABB menu, and then **Control Panel**, **FlexPendant** and **Task Panel Settings.**

Selecting tasks

Use this procedure to select which of the tasks are to be started with the START button.

	Action
1	Set the controller to manual mode.
2	On the FlexPendant, tap the QuickSet button and then the tasks panel button to show all tasks. If **Task Panel Settings** is set to **Only Normal tasks**, all STATIC and SEMISTATIC tasks are greyed out and cannot be selected. If **Task Panel Settings** is set to **All tasks**, STATIC and SEMISTATIC tasks with *Trust-Level*NoSafety can be selected, while STATIC and SEMISTATIC tasks with TrustLevel set to other values are grayed out and cannot be selected.
3	Select the check boxes for the tasks whose program should be started by the START button.

Continues on next page

Resetting debug settings in manual mode

Use this procedure to resume normal execution manual mode.

	Action
1	Select **Only Normal tasks** in the **Task Panel Settings**.
2	Press START button. All STATIC and SEMISTATIC will run continuously and not be stopped by the STOP button or emergency stop.

Switching to auto mode

When switching to auto mode, all STATIC and SEMISTATIC tasks will be deselected from the tasks panel. The stopped STATIC and SEMISTATIC tasks will start next time any of the START, FWD or BWD button are pressed. These tasks will then run continuously forward and not be stopped by the STOP button or emergency stop.

What happens with NORMAL tasks that has been deselected in the tasks panel depends on the system parameter *Reset* in type *Auto Condition Reset* in topic *Controller*. If *Reset* is set to Yes, all NORMAL tasks will be selected in the tasks panel and be started with the START button. If *Reset* is set to No, only those NORMAL tasks selected in tasks panel will be started by the START button.

Note

Note that changing the value of the system parameter *Reset* will affect all the debug resettings (for example speed override and simulated I/O). For more information, see *Technical reference manual - System parameters*, section *Auto Condition Reset*.

Restarting the controller

If the controller is restarted, all NORMAL tasks will keep their status while all STATIC and SEMISTATIC tasks will be deselected from the tasks panel. As the controller starts up all STATIC and SEMISTATIC tasks will be started and then run continuously.

Deselect task in synchronized mode

If a task is in a synchronized mode, that is program pointer between `SyncMoveOn` and `SyncMoveOff`, the task can be deselected but not reselected. The task cannot be selected until the synchronization is terminated. If the execution continues, the synchronization will eventually be terminated for the other tasks, but not for the deselected task. The synchronization can be terminated for this task by moving the program pointer to main or to a routine.

If the system parameter *Reset* is set to Yes, any attempt to change to Auto mode will fail while a deselected task is in synchronized mode. Changing to Auto mode should make all NORMAL tasks selected, and when this is not possible it is not possible to change to Auto mode.

6 Programming

6.1 RAPID components

Data types

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

Data type	Description
syncident	A variable of the data type `syncident` is used to identify which `WaitSyncTask`, `SyncMoveOn` or `SyncMoveOff` instructions, in the different task programs, should be synchronized with each other. The name of the `syncident` variable must be the same in all task programs. Declare `syncident` variables globally in each task. Do not reuse a `syncident` variable (each `WaitSyncTask`, `SyncMoveOn` and `SyncMoveOff` in a task program should have a unique `syncident`).
tasks	A persistent variable of the data type `tasks` contains names of the tasks that will be synchronized with `WaitSyncTask` or `SyncMoveOn`. The `tasks` variable must be declared as system global (persistent) variable, with the same name and the same content in all task programs.
identno	A numeric value or a variable of type `identno` is used in the argument `ID` of any move instructions executed between the `SyncMoveOn` and `SyncMoveOff` instructions.

System data

System data is predefined, internal data of the robot. A system data can be read, but not changed, from a RAPID program. For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

System data	Description
ROB_ID	Reference to the robot (if any) controlled by the task. If used from a task that does not control a robot, an error will occur. Always use `TaskRunRob()` to check this before using `ROB_ID`.

Instructions

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

Instruction	Description
WaitSyncTask	`WaitSyncTask` is used to synchronize several task programs at a special point in the program. A `WaitSyncTask` instruction will wait for the other task programs. When all task programs have reached the `WaitSyncTask` instruction, they will continue their execution.

Continues on next page

Instruction	Description
SyncMoveOn	`SyncMoveOn` is used to start synchronized movement mode. A `SyncMoveOn` instruction will wait for the other task programs. When all task programs have reached the `SyncMoveOn`, they will continue their execution in synchronized movement mode. The move instructions in the different task programs are executed simultaneously, until the instruction `SyncMoveOff` is executed. A stop point must be programmed before the `SyncMoveOn` instruction.
SyncMoveOff	`SyncMoveOff` is used to end synchronized movement mode. A `SyncMoveOff` instruction will wait for the other task programs. When all task programs have reached the `SyncMoveOff`, they will continue their execution in unsynchronized mode. A stop point must be programmed before the `SyncMoveOff` instruction.
SyncMoveUndo	`SyncMoveUndo` is used to turn off synchronized movements, even if not all the other task programs execute the `SyncMoveUndo` instruction. `SyncMoveUndo` is intended for `UNDO` handlers. When the program pointer is moved from the procedure, `SyncMoveUndo` is used to turn off the synchronization.
MoveExtJ	`MoveExtJ` (Move External Joints) moves one or several mechanical units without TCP. `MoveExtJ` is used to move additional axes, in a task without any robot.

Functions

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

Function	Description
IsSyncMoveOn	`IsSyncMoveOn` is used to tell if the mechanical unit group is in synchronized movement mode. A task that does not control any mechanical unit can find out if the mechanical units defined in the parameter *Use Mechanical Unit Group* are in synchronized movement mode.
RobName	`RobName` is used to get the name of the robot controlled by the task. It returns the mechanical unit name as a string. If called from a task that does not control a robot, an empty string is returned.

Continues on next page

Synchronizing argument

This is a brief description of the arguments used by move instructions to facilitate the synchronization between tasks. For more information, see any move instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Argument	Description
ID	All move instructions executed between the `SyncMoveOn` and `SyncMoveOff` instructions must have the argument `ID` specified. The `ID` argument must be the same for all the move instructions (in each task program) that should execute simultaneously. The `ID` argument can be a numeric value or a `syncident` variable. The purpose of `ID` is to support the operator by making it easier to see which move instructions that are synchronized with each other. Make sure an `ID` value is not used for more than one move instruction, between the same `SyncMoveOn` and `SyncMoveOff` instructions. It is also helpful for the operator if the `ID` values are ascending for consecutive move instructions (e.g. 10, 20, 30, ...). Move instructions that are not between the `SyncMoveOn` and `SyncMoveOff` instructions must not have the argument `ID`.

6.2 Tasks and programming techniques

Different tasks

Each task program can handle the movements for one robot and up to 6 additional axes. Several tasks can be used, each containing a program quite similar to the program of the main task in a single robot application. For more information about the tasks, see the section about Multitasking in *Application manual - Controller software IRC5*.

One task program per robot

Each task program can only handle one TCP. This means that you must have one task for each robot.

Additional axes in separate tasks

Additional axes that move a work object can be handled by the same task program as one of the robots. However, if the additional axes should be able to move independent of the robots, it is best to let a separate task program handle the additional axes.

6.3 Coordinated work objects

About work objects

This section will only describe how to make a work object coordinated with a mechanical unit. For a detailed description of work objects, see *wobjdata - Work object data* in *Technical reference manual - RAPID Instructions, Functions and Data types*.

What determines coordination?

When declaring a work object, the second attribute (`ufprog`) and the third attribute (`ufmec`) determine if the work object is coordinated to any mechanical unit.

robhold

`robhold` defines if the work object is held by the robot in **this** task.

`robhold` is normally set to `FALSE`. The task of the robot that holds the work object (where `robhold` would be set to `TRUE`) does not have to declare it unless a stationary tool is used.

ufprog

If the work object is stationary, `ufprog` is set to `TRUE`.

If the work object can be moved by any mechanical unit, `ufprog` is set to `FALSE`.

ufmec

`ufmec` is set to the name of the mechanical unit that moves the work object.

If `ufprog` is set to `TRUE`, `ufmec` can be left as an empty string (no mechanical unit can move the work object).

Example 1

This is an example of a work object that can be moved by a mechanical unit with the name `STN_1`:

```
PERS wobjdata wobj_stn1 := [FALSE, FALSE, "STN_1",
     [[0,0,0],[1,0,0,0]], [[0,0,250],[1,0,0,0]]];
```

Example 2

Robot `ROB_1` is welding a part that is hold by robot `ROB_2`. The workobject is moved by robot `ROB_2`.

When declaring the work object in `ROB_1`, the `robhold` argument must be set to `FALSE`, since `robhold` `TRUE` is only used for stationary tools. For `ROB_2`, any work object can be active since it is only the joint angles of `ROB_2` that coordinates the work object for `ROB_1`.

```
PERS wobjdata wobj_rob1 := [FALSE, FALSE, "ROB_2",
     [[0,0,0],[1,0,0,0]], [[0,0,250],[1,0,0,0]]];
```

6.4 Independent movements

6.4.1 About independent movements

What is independent movements

If the different task programs, and their robots, work independently, no synchronization or coordination is needed. Each task program is then written as if it was the program for a single robot system.

Other dependencies than movements

Sometimes, even if the movements do not need to be coordinated, the task programs can have dependencies. For example, if one robot leaves an object that a second robot will pick up, the first robot must finish with the object before the second robot can grab it.

These interactions can be solved with:

- the instruction `WaitSyncTask`
- I/O signals
- persistent variables together with `WaitUntil`

See the section about Multitasking in *Application manual - Controller software IRC5*.

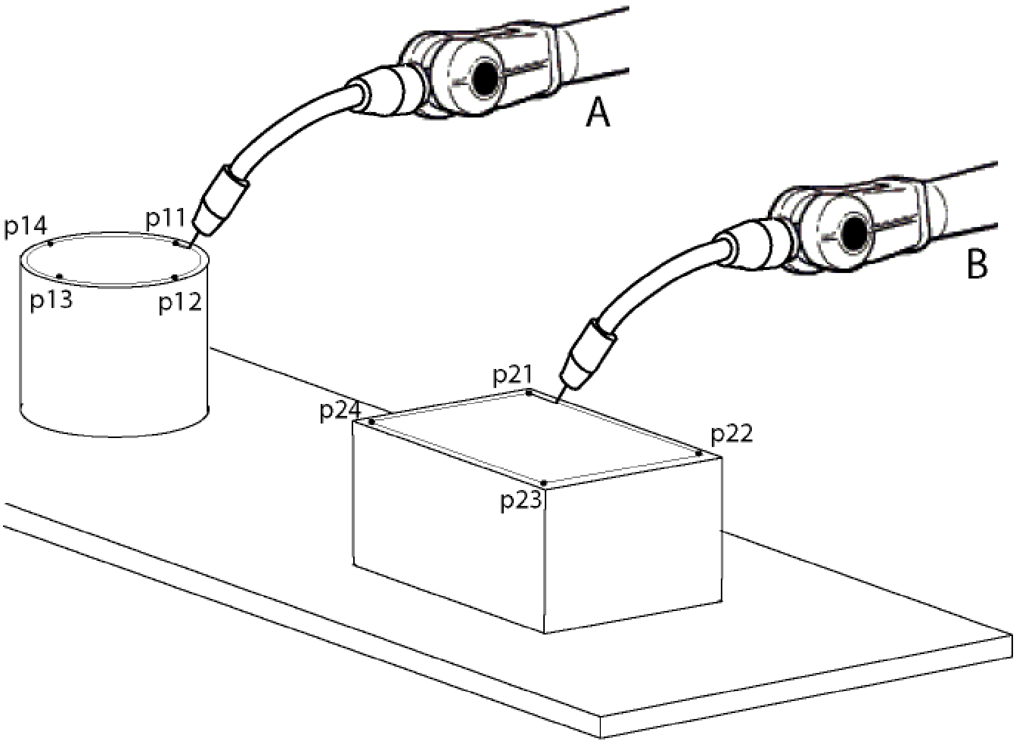
6.4.2 Example "UnsyncArc" with independent movements

Program description

In this example, one robot welds a circle on one object while the other robot welds a square on another object.

> **Note**
>
> To make the example simple and general, ordinary move instructions (e.g. `MoveL`) are used instead of weld instructions (e.g. `ArcL`). For more information about arc welding, see *Application manual - Arc and Arc Sensor*.

Illustration



xx0300000603

A	Robot 1
B	Robot 2

T_ROB1 task program

```
MODULE module1
  TASK PERS wobjdata wobj1 :=
    [ FALSE, TRUE, "",
    [ [500, -200, 1000], [1, 0, 0 ,0] ],
    [ [100, 200, 100], [1, 0, 0, 0] ] ];
  TASK PERS tooldata tool1 := ...
  CONST robtarget p11 := ...
  ...
```

Continues on next page

```
  CONST robtarget p14 := ...

  PROC main()
    ...
    IndependentMove;
    ...
  ENDPROC

  PROC IndependentMove()
    MoveL p11, v500, fine, tool1\WObj:=wobj1;
    MoveC p12, p13, v500, z10, tool1\WObj:=wobj1;
    MoveC p14, p11, v500, fine, tool1\WObj:=wobj1;
  ENDPROC
ENDMODULE
```

T_ROB2 task program

```
MODULE module2
  TASK PERS wobjdata wobj2 :=
    [ FALSE, TRUE, "",
    [ [500, -200, 1000], [1, 0, 0 ,0] ],
    [ [100, 1200, 100], [1, 0, 0, 0] ] ];
  TASK PERS tooldata tool2 := ...
  CONST robtarget p21 := ...
  ...
  CONST robtarget p24 := ...

  PROC main()
    ...
    IndependentMove;
    ...
  ENDPROC

  PROC IndependentMove()
    MoveL p21, v500, fine, tool2\WObj:=wobj2;
    MoveL p22, v500, z10, tool2\WObj:=wobj2;
    MoveL p23, v500, z10, tool2\WObj:=wobj2;
    MoveL p24, v500, z10, tool2\WObj:=wobj2;
    MoveL p21, v500, fine, tool2\WObj:=wobj2;
  ENDPROC
ENDMODULE
```

6.5 Semi coordinated movements

6.5.1 About semi coordinated movements

What is semi coordinated movements

Several robots can work with the same work object, without synchronized movements, as long as the work object is not moving.

A positioner can move the work object when the robots are not coordinated to it, and the robots can be coordinated to the work object when it is not moving. Switching between moving the object and coordinating the robots is called semi coordinated movements.

Implementation

Semi coordinated movements require some synchronization between the task programs (e.g. a *WaitSyncTask* instruction). The positioner must know when the work object can be moved, and the robots must know when they can work on the work object. However, it is not required that every move instruction is synchronized.

Advantages

The advantage is that each robot can work independently with the work object. If the different robots perform very different assignments, this may save cycle time compared to letting all the robot movements be synchronized.

6.5.2 Example "SyncArc" with semi coordinated movements

Program description

In this example, we want to accomplish the welding of a small square and a long line on one side of the object. On another side of the object we want to make a square and a circle.

The positioner will first position the work object with the first side up, while the robots wait. Robot 1 will then weld a line at the same time as robot 2 welds a square.

When the robots are done with the first welding operations, they wait while the positioner turns the work object so the second side is upwards. Robot 1 will then weld a circle at the same time as robot 2 welds a square.

WARNING

If the movement of the work object and the robot is not separated with `WaitSyncTask` and stop points the following can occur:

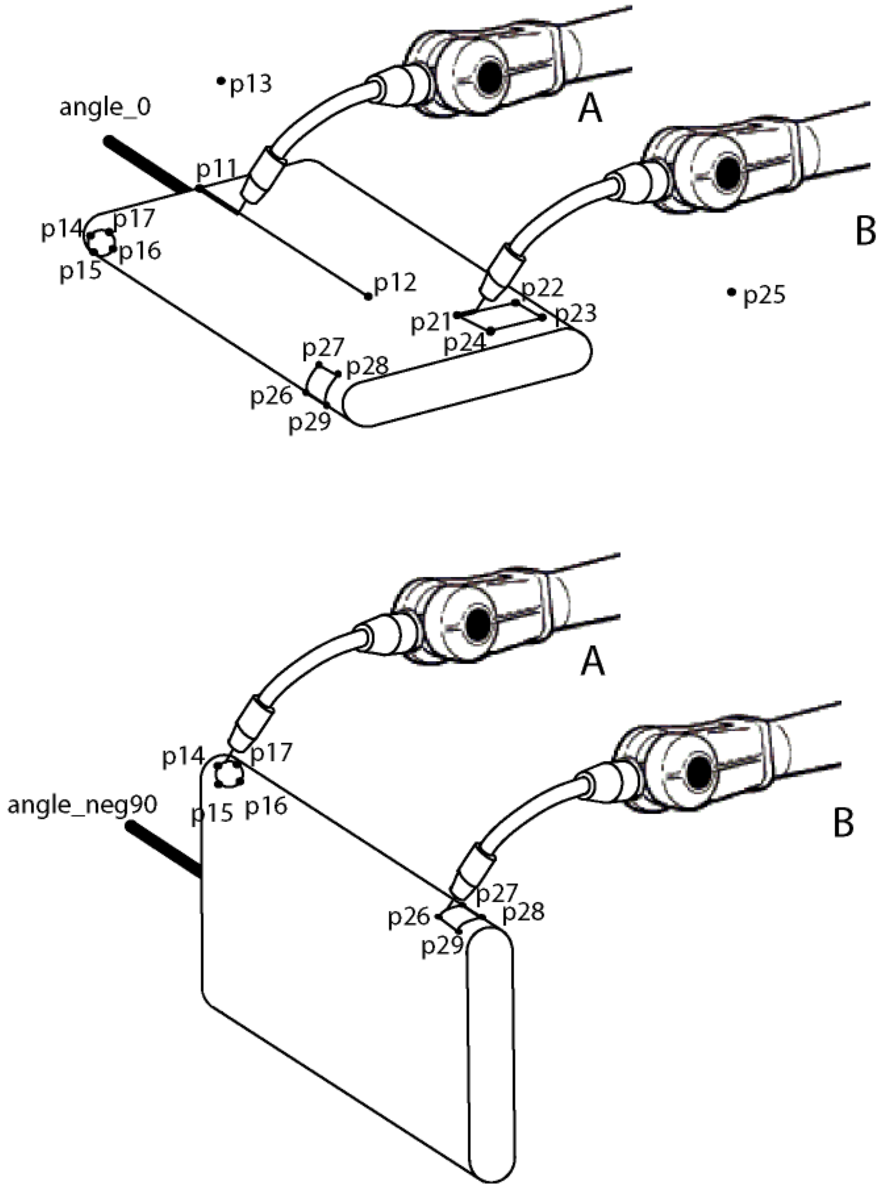
- the mechanical units controlled by the different tasks can collide
- the robot is stepping backwards in the wrong direction
- the movement or restart instruction can be blocked.

Note

To make the example simple and general, ordinary move instructions (e.g. `MoveL`) are used instead of weld instructions (e.g. `ArcL`). For more information about arc welding, see *Application manual - Arc and Arc Sensor*.

Continues on next page

Illustration



xx0300000596

A	Robot 1
B	Robot 2

T_ROB1 task program

```
MODULE module1
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  VAR syncident sync4;
  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 := ...
```

Continues on next page

```
    CONST robtarget p11 := ...
    ...
    CONST robtarget p17 := ...

    PROC main()
      ...
      SemiSyncMove;
      ...
    ENDPROC

    PROC SemiSyncMove()
      ! Wait for the positioner
      WaitSyncTask sync1, all_tasks;
      MoveL p11, v1000, fine, tool1 \WObj:=wobj_stn1;
      MoveL p12, v300, fine, tool1 \WObj:=wobj_stn1;
      ! Move away from the object
      MoveL p13, v1000, fine, tool1;
      ! Sync to let positioner move
      WaitSyncTask sync2, all_tasks;
      ! Wait for the positioner
      WaitSyncTask sync3, all_tasks;
      MoveL p14, v1000, fine, tool1 \WObj:=wobj_stn1;
      MoveC p15, p16, v300, z10, tool1 \WObj:=wobj_stn1;
      MoveC p17, p14, v300, fine, tool1 \WObj:=wobj_stn1;
      WaitSyncTask sync4, all_tasks;
      MoveL p13, v1000, fine, tool1;
    ENDPROC
  ENDMODULE
```

T_ROB2 task program

```
MODULE module2
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  VAR syncident sync4;
  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 tool2 := ...
  CONST robtarget p21 := ...
  ...
  CONST robtarget p29 := ...

  PROC main()
    ...
    SemiSyncMove;
    ...
  ENDPROC

  PROC SemiSyncMove()
    ! Wait for the positioner
```

Continues on next page

```
    WaitSyncTask sync1, all_tasks;
    MoveL p21, v1000, fine, tool2 \WObj:=wobj_stn1;
    MoveL p22, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p23, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p24, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p21, v300, fine, tool2 \WObj:=wobj_stn1;
    ! Move away from the object
    MoveL p25, v1000, fine, tool2;
    ! Sync to let positioner move
    WaitSyncTask sync2, all_tasks;
    ! Wait for the positioner
    WaitSyncTask sync3, all_tasks;
    MoveL p26, v1000, fine, tool2 \WObj:=wobj_stn1;
    MoveL p27, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p28, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p29, v300, z10, tool2 \WObj:=wobj_stn1;
    MoveL p26, v300, fine, tool2 \WObj:=wobj_stn1;
    WaitSyncTask sync4, all_tasks;
    MoveL p25, v1000, fine, tool2;
  ENDPROC
ENDMODULE
```

T_STN1 task program

```
MODULE module3
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  VAR syncident sync4;
  PERS tasks all_tasks{3} := [["T_ROB1"],["T_ROB2"],["T_STN1"]];
  CONST jointtarget angle_0 := [ [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9],
       [ 0, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  CONST jointtarget angle_neg90 := [ [ 9E9, 9E9, 9E9, 9E9, 9E9,
       9E9], [ -90, 9E9, 9E9, 9E9, 9E9, 9E9] ];

  PROC main()
    ...
    SemiSyncMove;
    ...
  ENDPROC

  PROC SemiSyncMove()
    ! Move to the wanted frame position. A movement of the
    ! positioner is always required before the first semi-
    ! coordinated movement.
    MoveExtJ angle_0, vrot50, fine;
    ! Sync to let the robots move
    WaitSyncTask sync1, all_tasks;
    ! Wait for the robots
    WaitSyncTask sync2, all_tasks;
    MoveExtJ angle_neg90, vrot50, fine;
    WaitSyncTask sync3, all_tasks;
```

Continues on next page

```
    WaitSyncTask sync4, all_tasks;
  ENDPROC
ENDMODULE
```

6.5.3 Considerations and limitations when using semi coordinated movements

Stand still in known position

The unit that controls the frame should stand still in a known position. To get a known position, order a movement to a finepoint.

Activate task

The unit that controls the frame should be activated in the task selection panel on the FlexPendant (see *Selecting tasks on page 53*).

Finepoints and WaitSyncTask before and after semi coordinated movement

The semi coordinated movement shall be separated with finepoints and `WaitSyncTask` instructions before and after the movement.

Dealing with a cleared path

When any of the instructions listed below is used, the path is removed and the position cannot be read by the other tasks.

- `ActUnit`
- `DeactUnit`
- `ClearPath`
- `SyncMoveOn`
- `SyncMoveoff`
- `SyncMoveSuspend`
- `SyncMoveResume`

After any of these instructions, order a movement to a wanted position for the unit that controls the frame and insert a `WaitSyncTask` instruction before the semicoordinated movement.

Before changing to synchronized movement with `SyncMoveOn` or `SyncMoveResume`, the semi coordinated movement must be ended with a finepoint and a `WaitSyncTask`.

Example with semi coordinated and coordinated movement

```
!Example with semicoordinated and synchronized movement
!Program example in task T_ROB1
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
PERS wobjdata rob2_obj:= [FALSE,FALSE,"ROB_2",
     [[0,0,0],[1,0,0,0]],[[155.241,-51.5938,57.6297],
     [0.493981,0.506191,-0.501597,0.49815]]];
VAR syncident sync0;
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
VAR syncident sync4;

PROC main()
  ...
  WaitSyncTask sync0, task_list;
```

Continues on next page

```
  MoveL p1_90, v100, fine, tcp1 \WObj:= rob2_obj;
  WaitSyncTask sync1, task_list;
  SyncMoveOn sync2, task_list;
  MoveL p1_100 \ID:=10, v100, fine, tcp1 \WObj:= rob2_obj;
  SyncMoveOff sync3;
  !Wait until the movement has been finished in T_ROB2
  WaitSyncTask sync3, task_list;
  !Now a semicoordinated movement can be performed
  MoveL p1_120, v100, z10, tcp1 \WObj:= rob2_obj;
  MoveL p1_130, v100, fine, tcp1 \WObj:= rob2_obj;
  WaitSyncTask sync4, task_list;
  ...
ENDPROC

!Program example in task T_ROB2
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
VAR syncident sync0;
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
VAR syncident sync4;

PROC main()
  ...
  MoveL p_fine, v1000, fine, tcp2;
  WaitSyncTask sync0, task_list;
  !Wait until the movement in T_ROB1 task is finished
  WaitSyncTask sync1, task_list;
  SyncMoveOn sync2, task_list;
  MoveL p2_100 \ID:=10, v100, fine, tcp2;
  SyncMoveOff sync3;
  !The path has been removed at SyncMoveOff
  !Perform a movement to wanted position for the object to
  !make the position available for other tasks
  MoveL p2_100, v100, fine, tcp2;
  WaitSyncTask sync3, task_list;
  WaitSyncTask sync4, task_list;
  MoveL p2_110, v100, z10, tcp2;
  ...
ENDPROC
```

When switching between semicoordinated to synchronized movement, a `WaitSyncTask` is needed (when using identity `sync1`).

When switching between synchronized to semicoordinated movement, the task that move the work object (`rob2_obj`) needs to move to the desired position. After that a `WaitSyncTask` is needed (identity `sync3`) before the semicoordinated movement can be performed.

6.6 Coordinated synchronized movements

6.6.1 About coordinated synchronized movements

What is coordinated synchronized movements

Several robots can work with the same moving work object.

The positioner or robot that holds the work object and the robots that work with the work object must have synchronized movements. This means that the RAPID task programs, that handle one mechanical unit each, execute their move instructions simultaneously.

Implementation

The synchronized movement mode is started by executing a `SyncMoveOn` instruction in each task program. The synchronized movement mode is ended by executing a `SyncMoveOff` instruction in each task program. The number of executed move instruction between `SyncMoveOn` and `SyncMoveOff` has to be the same for all task programs.

Advantages

Coordinated synchronized movements usually save cycle time since the robots do not have to wait while the work object is being moved. It also allows robots to cooperate in ways that would otherwise be difficult or impossible to achieve.

Limitations

Coordinated synchronized movements can only be used if you have the RobotWare option MultiMove Coordinated.

6.6.2 Example "SyncArc" with coordinated synchronized movement

Program description

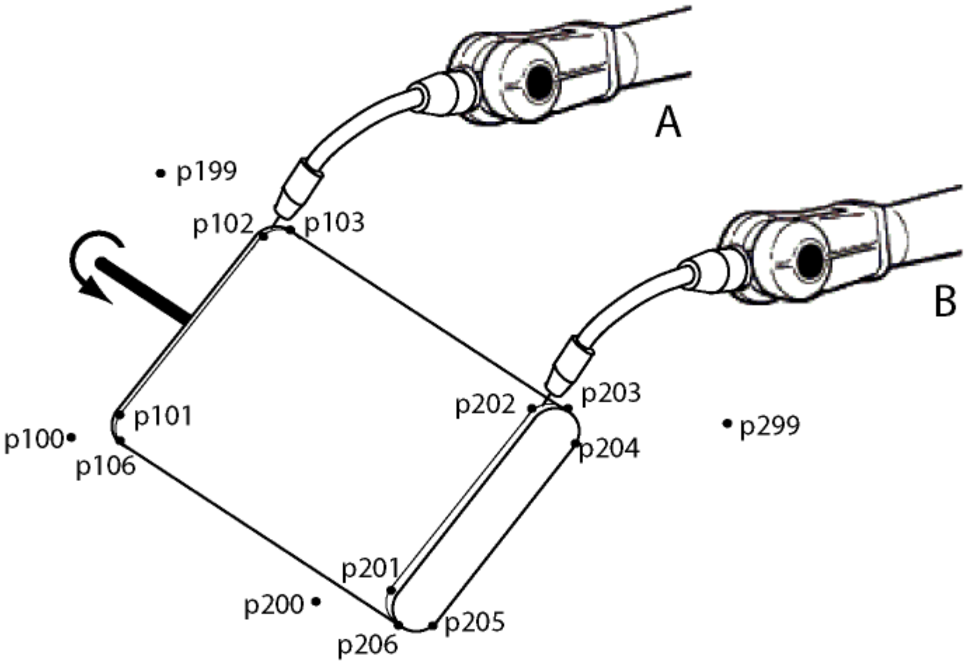
In this example, we want both robots to weld all the way around the object.

The robot TCPs are programmed to make circular paths relative to the work object. However, since the work object is rotating, the robots will almost stand still while the work object is turning.

Note

To make the example simple and general, ordinary move instructions (e.g. `MoveL`) are used instead of weld instructions (e.g. `ArcL`). For more information about arc welding, see *Application manual - Arc and Arc Sensor*.

Illustration



xx0300000597

A	Robot 1
B	Robot 2

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 := ...
```

Continues on next page

```
  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 \WObj:=wobj_stn1;
    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;
    SyncMoveOff sync3;
    MoveL p199, v1000, fine, tool1;
  UNDO
    SyncMoveUndo;
  ENDPROC
ENDMODULE
```

T_ROB2 task program

```
MODULE module2
  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 tool2 := ...
  CONST robtarget p200 := ...
  ...
  CONST robtarget p299 := ...

  PROC main()
    ...
    SyncMove;
    ...
  ENDPROC

  PROC SyncMove()
    MoveJ p200, v1000, z50, tool2;
    WaitSyncTask sync1, all_tasks;
    MoveL p201, v500, fine, tool2 \WObj:=wobj_stn1;
    SyncMoveOn sync2, all_tasks;
    MoveL p202\ID:=10, v300, z10, tool2 \WObj:=wobj_stn1;
```

Continues on next page

```
    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;
  UNDO
    SyncMoveUndo;
  ENDPROC
ENDMODULE
```

T_STN1 task program

```
MODULE module3
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3} := [["T_ROB1"],["T_ROB2"],["T_STN1"]];
  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()
    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;
  UNDO
    SyncMoveUndo;
  ENDPROC
ENDMODULE
```

6.7 Program execution

6.7.1 Corner zones

Corner zones and WaitSyncTask

Corner zones can be used when synchronizing several task programs with `WaitSyncTask`.

Corner zones and synchronized movements

Stop points must be used both before starting the synchronized movements with `SyncMoveOn` and before ending it with `SyncMoveOff`. All other move instructions between `SyncMoveOn` and `SyncMoveOff` can, on the other hand, use corner zones.

Dependences between synchronized instructions

In synchronized movements mode, all or none of the simultaneous move instructions must be programmed with corner zones. This means that the move instructions with the same `ID` must either all have corner zones, or all have stop points. If a move instruction with a corner zone and a move instruction with a stop point are synchronously executed in their respective task program, an error will occur.

Synchronously executed move instructions can have corner zones of different sizes (e.g. one use z10 and one use z50).

Corner zones converted to stop points

A corner zone will become a stop point if the task program has to wait for another task program. This can happen if `WaitSyncTask` is executed in a corner zone, but one task program reaches this instruction later than the others.

Example with corner zones

Given the RAPID code below, the following will happen:

- If robot1 reaches p11 at approximately the same time as robot2 reaches p21, both robots will be synchronized in corner zones (p11 and p21).
- If robot1 reaches p11 before robot2 reaches p21, p11 will become a stop point.
- If robot2 reaches p21 before robot1 reaches p11, p21 will become a stop point.

Note that both move instructions with corner zones and move instructions with stop points can be used in each task. You just have to make sure that the instructions with the same `ID` in both task programs are of the same type. The instructions before `SyncMoveOn` and `SyncMoveOff` must have stop points.

Part of T_ROB1 task program:

```
MoveL p11, v500, z50, tool1;
WaitSyncTask sync1, all_tasks;
MoveL p12, v500, fine, tool1;
SyncMoveOn sync2, all_tasks;
```

Continues on next page

```
MoveL p13\ID:=10, v500, z50, tool1 \WObj:=wobj_stn1;
MoveL p14\ID:=20, v500, fine, tool1 \WObj:=wobj_stn1;
SyncMoveOff sync3;
MoveL p15, v500, fine, tool1;
```

Part of T_ROB2 task program:

```
MoveL p21, v500, z50, tool2;
WaitSyncTask sync1, all_tasks;
MoveL p22, v500, fine, tool2;
SyncMoveOn sync2, all_tasks;
MoveL p23\ID:=10, v500, z10, tool2 \WObj:=wobj_stn1;
MoveL p24\ID:=20, v500, fine, tool2 \WObj:=wobj_stn1;
SyncMoveOff sync3;
MoveL p25, v500, fine, tool2;
```

6.7.2 Synchronization behavior

Synchronization point

When one task program reaches a synchronization point, it will wait until all task programs have reached the same synchronization point.

Synchronization points are:

- all `WaitSyncTask` instructions
- all `SyncMoveOn` instructions
- all `SyncMoveOff` instructions
- all move instructions between `SyncMoveOn` and `SyncMoveOff`

When one task program reaches a `WaitSyncTask`, `SyncMoveOn` or `SyncMoveOff` instruction, it will wait until all task programs have reached the instruction with the same `syncident` variable.

All move instructions between `SyncMoveOn` and `SyncMoveOff` must use the argument `ID`. When a task program reaches such a move instruction, it will wait until all task programs have reached the move instruction with the `ID` argument set to the same value.

Other instructions than movements

All synchronized task programs must execute the same number of move instructions between the `SyncMoveOn` and `SyncMoveOff` instructions. This does not affect functions or other instructions than move instructions. It is possible to have any number of functions and instructions that are not move instructions.

Example

In this example both task programs execute two move instructions, but one of the tasks executes other instructions and functions.

Robot 2 will wait and not move to p21 until robot 1 starts to move towards p11.

Since `SyncMoveOff` is a synchronization point, both tasks will wait for `di1` to become 1 before executing `SyncMoveOff`.

Part of T_ROB2 task program:

```
SyncMoveOn sync1, all_tasks;
time := CTime();
Write log, "Synchronization started "\NoNewLine;
Write log, time;
MoveL p11\ID:=10, v500, fine, tool1 \WObj:=wobj_stn1;
Set do1;
MoveC p12, p13\ID:=20, v500, fine, tool1 \WObj:=wobj_stn1;
WaitDI di1, 1;
SyncMoveOff sync2;
```

Part of T_ROB2 task program:

```
SyncMoveOn sync1, all_tasks;
MoveJ p21\ID:=10, v500, fine, tool2 \WObj:=wobj_stn1;
MoveL p22\ID:=20, v500, fine, tool2 \WObj:=wobj_stn1;
SyncMoveOff sync2;
```

6.7.3 Dummy instructions

About dummy instructions

The same number of move instructions must be executed between `SyncMoveOn` and `SyncMoveOff` in all task programs. If a move instruction is only executed under certain circumstances, the number of move instructions may differ from the other task programs. This can be solved by adding a move instruction to the point where the robot already is (a dummy instruction) for the case where the original move instruction is not executed.

Example with dummy move instructions

In this example, the task program needs to execute two move instructions if di1 is set to 1. If di1 is 0, two move instructions are executed that move the robot to the position where it already is (dummy instructions).

Part of a task program

```
SyncMoveOn sync1, all_tasks;
MoveL p1\ID:=10, v500, fine, tool1 \WObj:=wobj_stn1;
IF di1=1 THEN
  ! Instructions executed under certain conditions
  MoveL p2\ID:=20, v500, fine, tool1 \WObj:=wobj_stn1;
  MoveL p1\ID:=30, v500, fine, tool1 \WObj:=wobj_stn1;
ELSE
  ! Add dummy move instructions
  MoveL p1\ID:=20, v500, fine, tool1 \WObj:=wobj_stn1;
  MoveL p1\ID:=30, v500, fine, tool1 \WObj:=wobj_stn1;
ENDIF
SyncMoveOff sync2;
```

6.7.4 Motion principles

Robot speeds

When the movements of several robots are synchronized, all robots adjust their speed to finish their movements simultaneously. This means that the robot movement that takes the longest time will determine the speed of the other robots.

Example of robot speeds

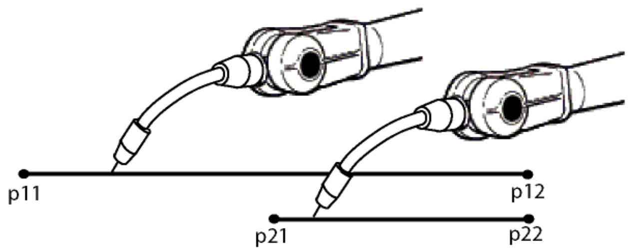
In this example, the distance between p11 and p12 is 1000 mm and the distance between p21 and p22 is 500 mm. When running the code below, robot1 will move 1000 mm at a speed of 100 mm/s. Since this will take 10 seconds, robot2 will move 500 mm in 10 seconds. The speed of robot2 will be 50 mm/s (and not 500 mm/s as programmed).

Part of T_ROB1 task program:

```
MoveJ p11, v1000, fine, tool1;
SyncMoveOn sync1, all_tasks;
MoveL p12\ID:=10, v100, fine, tool1;
```

Part of T_ROB2 task program:

```
MoveJ p21, v1000, fine, tool2;
SyncMoveOn sync1, all_tasks;
MoveL p22\ID:=10, v500, fine, tool2;
```



xx0400000907

6.7.5 Modify position

About modifying positions

A programmed position can be modified from the FlexPendant, see *Production Window on page 51*.

Modify position in unsynchronized mode

When the movements of the different tasks are unsynchronized, the position of each mechanical unit is modified individually.

Modify position in synchronized movement mode

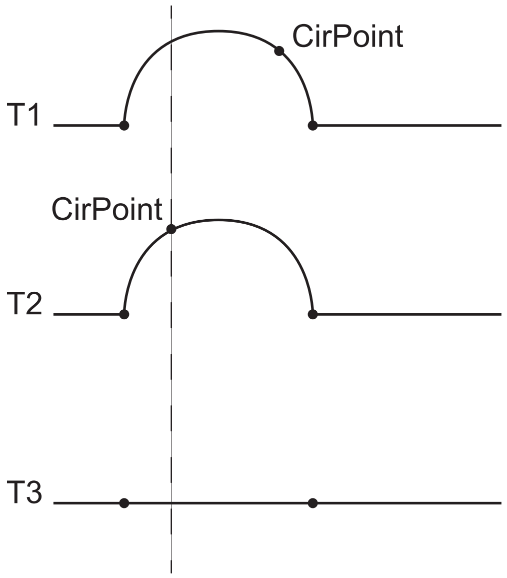
Modifying positions while in synchronized movement mode (when the execution is between a `SyncMoveOn` and `SyncMoveOff` instruction) behaves differently depending on if it is done from the Production Window or the Program Editor.

In the Production Window, the position will be modified for all tasks in synchronized movement mode. Circle points cannot be modified from the Production Window while in synchronized movement mode, thus if the marked point is a circle point, the function to modify position from the Production Window will not be enabled. In the Production Window, the position can only be modified for the current move instruction (where the motion pointer is).

In the Program Editor, the position will be modified only for the task program currently open in that editor window.

See also example on circular movement in the description of modifying positions in *Operating manual - IRC5 with FlexPendant*.

Modify circular position in synchronized movement mode



xx1400001119

6.7.6 Moving a program pointer

Moving PP in unsynchronized mode

When none of the tasks are in synchronized movement mode, a program pointer in one task can be moved without affecting the other tasks.

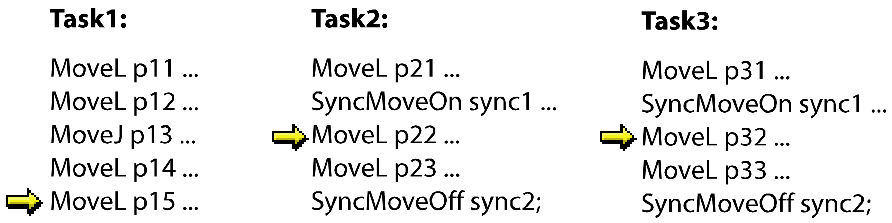
Moving PP in synchronized movement mode

If the program pointer is moved for one task, the program pointers for all tasks in synchronized movement mode are lost. This is the case even if the task where the program pointer is moved is not in synchronized movement mode. Even if a task is inactive, moving its program pointer will affect the program pointers of all tasks in synchronized movement mode.

Example

In this example, there are three tasks. Task2 and Task3 are in synchronized movement mode, while Task1 works independently. In this situation, the user taps **Move PP to Main** for Task1.

The program pointers for Task2 and Task3 will be lost.



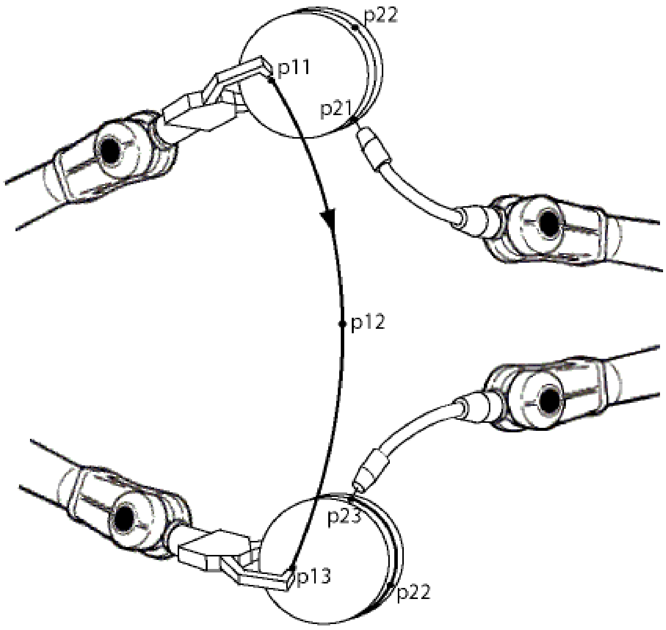
xx0500001444

6.7.7 Tool orientation at circular movements

Coordinated circular move instructions

There is a risk for incorrect tool orientation if two coordinated task programs both perform synchronized circular move instructions. If one robot holds a work object that another robot is working on, the circle interpolation affects both robots. The circle point should be reached at the same time for both circle paths to avoid incorrect orientation of the tool.

Example



xx0400000717

If p12 would be in the beginning of its circular path (closer to p11 than p13) and p22 would be in the end of its circular path (closer to p23 than p21) then the tool orientation could become wrong. If p12 and p22 are in the same relative position on the path (percentage of the path length) the tool orientation will remain correct.

Tip

By modifying the position for both robots circle point at the same time, you make sure the tool orientation stays correct. This means that, in the example, you should step through the program and then modify p12 and p22 at the same time.

6.7.8 Applications affected by MultiMove

Collision Detection

If collision is detected for one robot, all robots are stopped. Even if the robots are run individually, all robots behave as if it had collided.

One reason for this behavior is that when a collision is detected, there is a big risk that it was two robots that collided. Another reason is that if one robot stops and another continues, this might cause another collision.

World Zones

A world zone declared in one task program is only valid for the mechanical units that belong to that task. For a world zone to affect all mechanical units, it must be declared in all task programs.

6.8 Programming recommendations

6.8.1 Programming recommendations

Declare syncident globally in task

By declaring all variables of the data type `syncident` globally in the task program, there is no risk of having two `syncident` with the same name in the same task program.

Do not reuse syncident

A `syncident` variable is used as an argument for all `WaitSyncTask`, `SyncMoveOn` and `SyncMoveOff` instructions, so that the operator can distinguish which instructions are executed simultaneously in the different task programs. If one `syncident` variable would be used as argument for more than one instruction per task, that instruction would no longer be uniquely identified. To make sure your program code is understandable, never reuse a `syncident` variable.

Declaring tools, work objects and payloads

Declaring a variable as `TASK PERS` will make it persistent in the task program, but not shared between tasks. By declaring tools, work objects and payloads as task persistent, you do not have to keep track of whether the variable name is used in other tasks. If tools, work objects and payloads are declared as `TASK PERS`, the names do not have to be changed if the program is copied or mirrored to another task.

A work object that is used by several task programs is preferably declared as `PERS`. A tool can be declared as `PERS` if a background task needs to read the robot position.

Changing a PERS

A globally declared `PERS` will keep its value even if a new declaration of the same `PERS` is loaded. The value of the `PERS` that was first loaded will be preserved as long as there is any reference to that `PERS`.

If you want to replace all the task programs with new programs where the values of the `PERS` is different, remove all task programs first and then load all the new task programs. That way the old value of the `PERS` will be lost when all declarations of it are removed.

Changing the value of a `PERS` from the **Data Variable** view on the FlexPendant and saving the program, will update the `PERS` in a correct way.

Use SyncMoveUndo

Always use an `UNDO` handler with a `SyncMoveUndo` instruction in any procedure that has synchronized movements (i.e. that has a `SyncMoveOn` instruction).

After a `SyncMoveOn` instruction, the movements in the task program are synchronized with movements in other task programs. If the program pointer is then manually moved before the `SyncMoveOff` instruction is executed, the

Continues on next page

movements will still be synchronized. This can be avoided by having an `UNDO` handler that includes a `SyncMoveUndo` instruction.

When the program pointer is manually moved out of a procedure, the `UNDO` handler for that procedure is called. The `SyncMoveUndo` instruction will end the synchronization if the movements currently are synchronized. If the movements are not synchronized when the program pointer is moved, `SyncMoveUndo` will do nothing. It is, in other words, never any disadvantage in using `SyncMoveUndo`, but very useful if the program pointer is moved.

For more information about `UNDO` handlers, see *Technical reference manual - RAPID Overview*.

Coordinating against a work object

Coordinating against a work object moved by a mechanical unit in another task can be done in two ways:

- All move instructions coordinated with the work object must be executed when the work object is standing still. See *Semi coordinated movements on page 63*.
- The robot that is coordinated with the work object and the mechanical unit that moves the work object must be in synchronized movement mode. See *Coordinated synchronized movements on page 71*.

It is not possible to coordinate against a moving work object, controlled from another task, without being in synchronized movement mode.

Common work area

If two robots use the same work area, without being in synchronized movement mode, precautions must be taken to avoid collisions. Make sure that only one of the robots is in the common area at a time by using one of the following:

- WaitSyncTask
- World Zones
- I/O signal

This page is intentionally left blank

7 RAPID error recovery

7.1 Error recovery for MultiMove

Error in unsynchronized mode

If an error occurs during unsynchronized mode, it is handled just like in a single robot system. No other task program is affected by the error.

Error in synchronized movement mode

If an error occurs during synchronized movement mode, the task program with the error will stop with an error code, just like in a single robot system. Because of the synchronization, the other tasks will not continue to move. When the error has been resolved the movement can continue in all task programs.

7.2 Simple error recovery example

About this example

In this example, a division with zero causes an error during synchronized movement mode. Since the error handler can resolve the error without any motion instructions, the error handler does not have to consider the synchronization. The synchronized movement mode is active the whole time and the second move instruction is started for both robots as soon as the error handler has finished. If no other error can occur, the T_HANDLEROB task program does not need to have an error handler.

T_PROCROB task program

```
...
SyncMoveOn, sync1, motion_tasks;
MoveL p101\ID:=10, v100, z10, gun2 \WObj:=wobj_handlerob;
a:=3;
b:=0;
c:=a/b;
MoveL p102\ID:=20, v100, fine, gun2 \WObj:=wobj_handlerob;
SyncMoveOff sync2;
...
ERROR
  IF ERRNO = ERR_DIVZERO THEN
    b:=1;
    RETRY;
  ENDIF
```

T_HANDLEROB task program

```
...
SyncMoveOn, sync1, motion_tasks;
MoveL p201\ID:=10, v100, z10, grip1;
MoveL p202\ID:=20, v100, fine, grip1;
SyncMoveOff sync2;
...
```

7.3 Asynchronously raised errors

What is an asynchronously raised error

Asynchronously raised errors can be raised by another instruction than the instruction where the program pointer is. This means that an asynchronous error can be raised while the robot is in the middle of a path movement. For more information about asynchronously raised errors, see *Technical reference manual manual - RAPID kernel*.

The technique with asynchronously raised errors allows a failing instruction in one task program to raise an error in all other task programs with synchronized movements.

How to raise an asynchronous error

The instruction `ProcerrRecovery` will raise the error `ERR_PATH_STOP` and stop the movement for all task programs with synchronized movements.

Asynchronous errors can also be raised by process instructions (e.g. `ArcL`). These can raise one error code (describing the cause of the error) in the task program where the error occurred, and raise the error `ERR_PATH_STOP` in the other task programs with synchronized movements.

The task programs without errors

If two task programs run synchronized move instructions and one of them raise an asynchronous error, the movements will stop for both tasks. The task program where nothing went wrong will then get the error `ERR_PATH_STOP`. This error must be handled by an error handler. The error handler can handle `ERR_PATH_STOP` by just waiting for the other task to solve its problems and then resume the movements.

By using the instruction `StartMoveRetry`, the execution will continue when all tasks reach this instruction.

Independent movements in the error handler

If the error handler in one task program needs to execute a move instruction, the synchronization must be suspended first.

The synchronization is automatically suspended by the `StorePath` instruction. All tasks with synchronized movements must execute a `StorePath` instruction before the synchronization is turned off and the execution can continue.

The instruction `RestoPath` will restore synchronization to the mode it had before `StorePath`. All task programs with synchronized movements must execute the `RestoPath` instruction in their error handlers before the synchronization is resumed and the execution can continue.

Between the instructions `StorePath` and `RestoPath`, the failing task program can move independently to solve its problem. Since `RestoPath` works as a synchronization point, the other task programs will wait at this point until the problem has been resolved.

If the task program is not in synchronized movements mode, `StorePath` and `RestoPath` act just like they do in a single robot system. This means that the same

Continues on next page

error handler code can handle errors that occur both in synchronized movements mode and unsynchronized mode.

`StorePath` and `RestoPath` require the option Path Recovery. For more information about `StorePath` and `RestoPath`, see *Application manual - Controller software IRC5*.

7.4 Example of creating asynchronously raised error

About this example

In this example, a process is started by setting do_myproc to 1. The process is then supervised and the signal di_proc_sup is set to 1 if the process fails.

If a process failure occurs during a robot movement, an interrupt calls a trap routine. The instruction ProcerrRecovery will stop the movement and raise the error ERR_PATH_STOP in all task programs with synchronized movements.

The T_HANDLEROB task program must have an error handler that restarts the movement when the error has been resolved in the T_PROCROB task program. This only requires one instruction, StartMoveRetry.

T_PROCROB task program

```
VAR intnum proc_sup_int;

PROC main()
  ...
  SyncMoveOn, sync1, motion_tasks;
  my_proc_on;
  MoveL p101\ID:=10, v100, z10, gun1 \WObj:=wobj_handlerob;
  MoveL p102\ID:=20, v100, fine, gun1 \WObj:=wobj_handlerob;
  my_proc_off;
  SyncMoveOff sync2;
  ...

  ERROR
    IF ERRNO = ERR_PATH_STOP THEN
      my_proc_on;
      StartMoveRetry;
    ENDIF
ENDPROC

TRAP iprocfail
  my_proc_off;
  ProcerrRecovery \SyncLastMoveInst;
  RETURN;
ENDTRAP

PROC my_proc_on()
  SetDO do_myproc, 1;
  CONNECT proc_sup_int WITH iprocfail;
  ISignalDI di_proc_sup, 1, proc_sup_int;
ENDPROC

PROC my_proc_off()
  SetDO do_myproc, 0;
  IDelete proc_sup_int;
ENDPROC
```

Continues on next page

T_HANDLEROB task program

```
PROC main()
  ...
  SyncMoveOn, sync1, motion_tasks;
  MoveL p201\ID:=10, v100, z10, grip1;
  MoveL p202\ID:=20, v100, fine, grip1;
  SyncMoveOff sync2;
  ...

  ERROR
    IF ERRNO = ERR_PATH_STOP THEN
      StartMoveRetry;
    ENDIF
ENDPROC
```

7.5 Example with movements in error handler

About this example

In this example, an asynchronous error can occur that requires the robot to move to another position to resolve the error. The synchronization is suspended by using `StorePath` in all tasks with synchronized movements, and restored by using `RestoPath`.

The instruction `ArcL` is used in this example. This instruction handles the process for arc welding as well as acts as a move instruction. To understand this example, all you need to know is that it is a move instruction (similar to `MoveL`) which can result in asynchronous process errors. For more information about `ArcL`, see *Application manual - Arc and Arc Sensor* and *Technical reference manual - RAPID Instructions, Functions and Data types*.

> **Note**
>
> Note that the T_STN1 task program must have the instructions `StorePath` and `RestoPath`, even if there is no code between these instructions. No task program continues to execute its error handler until all task programs execute the `StorePath` instruction.

T_ROB1 task program

```
...
SyncMoveOn, sync1, all_tasks;
ArcL p101\ID:=10, v100, seam1, weld1, weave1, z10, gun1
     \WObj:=wobj_stn1;
...
ERROR
  IF ERRNO=AW_WELD_ERR OR ERRNO=ERR_PATH_STOP THEN
    StorePath;
    IF ERRNO=AW_WELD_ERR THEN
      gun_cleaning;
    ENDIF
    RestoPath;
    StartMoveRetry;
  ENDIF
...
PROC gun_cleaning()
  VAR robtarget p199;
  p199 := CRobT(\Tool:=gun1 \WObj:=wobj0);
  MoveL pclean, v100, fine, gun1;
  ...
  MoveL p199, v100, fine, gun1;
ENDPROC
```

T_ROB2 task program

```
...
SyncMoveOn, sync1, all_tasks;
```

Continues on next page

```
ArcL p201\ID:=10, v100, seam2, weld2, weave2, z10, gun2
     \WObj:=wobj_stn1;
...
ERROR
  IF ERRNO=AW_WELD_ERR OR ERRNO=ERR_PATH_STOP THEN
    StorePath;
    IF ERRNO=AW_WELD_ERR THEN
      gun_cleaning;
    ENDIF
    RestoPath;
    StartMoveRetry;
  ENDIF
...
PROC gun_cleaning()
  VAR robtarget p299;
  p299 := CRobT(\Tool:=gun2 \WObj:=wobj0);
  MoveL pclean, v100, fine, gun2;
  ...
  MoveL p299, v100, fine, gun2;
ENDPROC
```

T_STN1 task program

```
...
SyncMoveOn, sync1, all_tasks;
MoveExtJ angle_20\ID:=10, vrot50, z10;
...
ERROR
  IF ERRNO=ERR_PATH_STOP THEN
    StorePath;
    RestoPath;
    StartMoveRetry;
  ENDIF
...
```

8 Running a subset of a MultiMove system

8.1 How to continue with one or more drive units inactive.

Overview

It is possible to disconnect a drive module and continue working, for example:

- if a drive module is required elsewhere due to failure or similar.
- for tuning purposes during commissioning, for instance programming one robot at a time while the others are temporarily shut down.

Make sure the application allows work to continue without this drive module, by using the procedure *Continue with the Drive Module Disconnect function on page 95*.

But if one of the following conditions are fulfilled then *Continue with alternative configuration on page 96*.

- The first alternative fails.
- The limit switches are used on the robot.
- The drive module needs to be moved (e.g. for repair or installation in another cell).

Tip

Sometimes it is necessary to change the program and/or configuration so that the application will work with one less drive module.

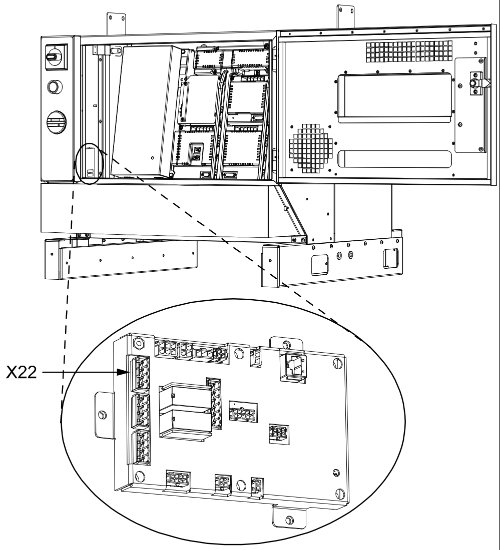
Continue with the Drive Module Disconnect function

This procedure shows how to let the functional robots continue with their applications without changing the configuration. This requires that the functional robots have no dependencies to the disconnected robot, or additional axes connected to the same drive module.

This is a brief description how to disconnect the drive module, for more detailed information see *Product manual - IRC5*, section *Connections - Connection of the Drive Module Disconnect*.

	Action	Info/illustration
1	Make sure that the system parameter `Allow_Drive_Module_Disconnect` is set to true.	
2	Switch to manual mode.	
3	Make sure the controller is in Motors Off state.	

Continues on next page

	Action	Info/illustration
4	Remove the connector from X22 in the drive module. When the connector is removed the following message appears on the Flex Pendant: Event Message 50320, Drive Module Disconnected.	 xx0500001599
5	The system now acts as if none of the mechanical units connected to this drive module exists.	

Note

Depending on the type of failure, this method may not always succeed. For example if there is an error in the axis computer. In that case, continue with the alternative configuration.

Continue with alternative configuration

This procedure shows how to let the functional robots continue with their applications but with changes in the standard configuration.

	Action	Info/illustration
1	Restart the controller using the restart mode **Start Boot Application**.	
2	Switch off the controller.	
3	Localize the Ethernet connection cable of the drive module you wish to disconnect. Remove it from the robot communication card in the control module. Note that the drive module's ethernet cables should be connected in the following order. So that there is no gap in this order: • AXC1 on the Robot communication card • ETHERNET 1 on the Ethernet card • ETHERNET 2 on the Ethernet card • ETHERNET 3 on the Ethernet card	See *Ethernet connections on page 21*.
4	Locate the safety signal connection cable of the drive module you wish to disconnect. Remove it from the panel board in the control module and replace it with a jumper connector. Move the safety signal connection cables so that there are no gaps in the following order X7, X8, X14 and X17.	See *Ethernet connections on page 21*.

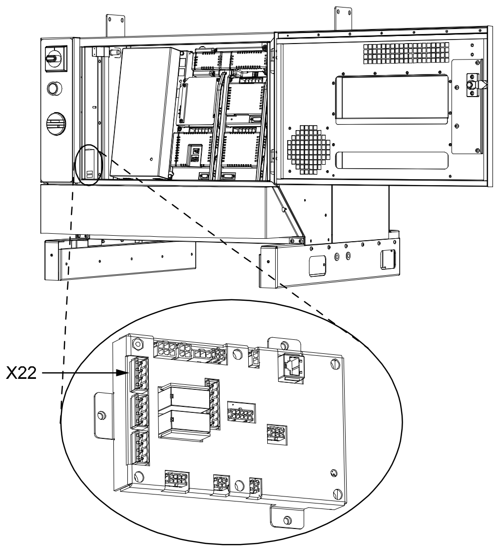
Continues on next page

	Action	Info/illustration
5	Switch on the power to the controller.	
6	Select a new robot system that is configured without the disconnected mechanical unit. Note that the configuration has to be in accordance with the connections in step 3.	

8.2 Running a subset in the “Unsync Arc” examples

Example with Drive Module Disconnect

In this example the configuration is according to “UnsyncArc”, and an error occurs on the process equipment on robot 1. The function Drive Module Disconnect is configured and there are no limit switches on the robots. Robot 2 should continue its work.

	Action	Info/illustration
1	Make sure that the system parameter `Allow_Drive_Module_Disconnect` is set to true.	
2	Switch to manual mode.	
3	Make sure the controller is in Motors Off state.	
4	Remove the connector from X22 in drive module 1.	 xx0500001599
5	The system now acts as if robot 1 does not exist. It is now safe to continue working with robot 2.	

Example without Drive Module Disconnect

In this example limit switches are used on the robots. The configuration is according to “UnsyncArc”, and an error occurs on robot 1. Robot 2 should continue its work.

	Action	Info/illustration
1	Restart the controller using the restart mode **Start Boot Application**.	
2	Switch off the controller.	
3	Remove the Ethernet connection of drive module 1 from the robot communication card in the control module and replace it with a jumper connector. Move the Ethernet connection of drive module 2 from the Ethernet card to the robot communication card.	See *Ethernet connections on page 21*.

Continues on next page

	Action	Info/illustration
4	Remove the safety signal connection of drive module 1 from X7 on the panel board in the control module. Move the safety signal connection of drive module 2 from X8 to X7. Then push the jumper connector in the X8 connector.	See *Ethernet connections on page 21*.
5	Switch on the power to the controller.	
6	Select a new robot system that is configured without robot 1. Note that the configuration has to be in accordance with the connections in step 3. I.e. the remaining robot will be referred to as robot 1.	

Tip

The same actions can be taken if the axis computer fails, or other failure that cannot be helped with Drive Module Disconnect.

This page is intentionally left blank

Index

ABB AB, Robotics
Robotics and Motion
S-721 68 VÄSTERÅS, Sweden
Telephone +46 (0) 21 344 400

ABB AS, Robotics
Robotics and Motion
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 22 87 2000

ABB Engineering (Shanghai) Ltd.
Robotics and Motion
No. 4528 Kangxin Highway
PuDong District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

ABB Inc.
Robotics and Motion
1250 Brown Road
Auburn Hills, MI 48326
USA
Telephone: +1 248 391 9000

abb.com/robotics

3HAC050961-001, Rev E, en